INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Cloud DReAM - Dynamic Resource Allocation Management for Large Scale MMOGs

Miguel António Moreira de Sousa Adaixo

Dissertation submitted to obtain the Master Degree in

Information Systems and Computer Engineering

Jury

President:	Professor Pedro Manuel Moreira Vaz Antunes de Sousa
Supervisor:	Professor Paulo Jorge Pires Ferreira
Co-Supervisor:	Professor Luís Manuel Antunes Veiga
Members:	Professor João Carlos Serrenho Dias Pereira

November 2012

Agradecimentos

Quero agradecer aos meus orientadores Paulo Ferreira e Luís Veiga por todas as opiniões, conhecimento e disponibilidade, sem o qual este trabalho não teria sido possivel.
Agradeço também à minha familia, especialmente aos meus pais Manuel Adaixo e Olga Adaixo, e à minha irmã Margarida, por todo o apoio e paciência que me deram não só durante a realização deste trabalho, mas durante toda a vida.
Ao André Pessoa, pelo seu contributo para o resultado final desta tese e pelo apoio indispensável que me prestou na fase final deste trabalho.
Um agradecimento final a todos os meus colegas e amigos, com quem partilhei bons e maus momentos, e que ajudaram a tornar este trabalho possivel.

Resumo

Nos ultimos anos os massively multiplayer online games (MMOG) têm vindo a ganhar um número crescente de adeptos. Este tipo de sistemas colocam uma série de dificuldades aos criadores do jogo. Dificuldades tais como garantir escalabilidade do sistema, ao mesmo tempo que é mantido o desempenho do jogo aos olhos do jogador, de modo a que este disfrute de uma boa experiência. As implementações mais comuns destes sistemas baseiam-se numa arquitectura cliente-servidor, usando estratégias diferentes para distribuir a carga pelos vários servidores. Esta abordagem levanta questões relevantes no que toca a escalabilidade dos seus recursos. Tem sido realizada muita investigação nesta área, quer seja a tentar optimizar as abordagens actuais ou usando outras novas. Peer-to-peer e mais recentemente o paradigma de cloud computing são exemplos de novas abordagens. Neste trabalho propomos uma abordagem baseada em cloud computing, de modo a resolver os principais problemas com os quais um MMOG tem de lidar, tendo em mente que a utilização de recursos deve ser optimizada.

Abstract

In the last few years massively multiplayer online games (MMOG) have been gaining an ever increasing number of adepts. This type of systems pose a series of difficulties to designers. Difficulties such as guaranteeing scalability of the system, while maintaining the overall game performance to the users in order to make the game an enjoyable experience. The most common implementations of these systems rely on a client-server organization, using different types of approaches to distribute the load among the various servers. This approach raises relevant issues when it comes to scalability of its resources. There has been a lot of research on the area, either trying to optimize this approach or trying to use other types of structures. Peer-to-peer and more recently the cloud computing paradigm are examples of such structures. In this work we propose an approach that uses a cloud computing platform in order to solve the major issues that a MMOG has to deal with, having in mind that resource waste should be optimized.

Palavras Chave

Keywords

Palavras Chave

Computação na Nuvem
Elasticidade
Gestão de Recursos
Jogos Online Multijogador

Keywords

Cloud Computing
Elasticity
Resource Management
Multiplayer Online Games

Contents

List of Figures

List of Tables

List of Abbreviations

AOI	Area of Interest
API	Application programming interface
FPS	First person shooter
IaaS	Infrastructure as a service
MMOG	Massively multiplayer online game
NPC	Non playable character
P2P	Peer-to-Peer
PaaS	Platform as a service
RPC	Remote procedure call
SaaS	Software as a service
VFC	Vector Field Consistency

1 Introduction

1.1 Motivation

A Massively multiplayer online game (MMOG) is a genre of game in which a very large number of players interact with one another within the game's virtual world. This type of interaction uses the internet as a support medium, and requires a persistent game world in which the player's actions take place.
This type of system is a clear example of a distributed system. Taking this into account, most of the common problems with distributed systems apply in this type of game, and in some cases can be even harder to solve than on regular distributed systems. The two main distributed systems concerns that are dealt with in this case are scalability and usability of the application.
Typically such games run on top of a client-server architecture, with many clients connecting to one server where the game world is hosted. Users want to play the game as smoothly as possible no matter how big the number of players currently connected to the server. This can pose a serious challenge to the server, which has to process many clients at the same time, and respond to them in a timely manner. It becomes clear that this process can be very heavy on resource consumption on peak hours, where most players are connected.
Thinking about the costs of all this operation, it is easy to realise that it will be very high, especially on those hours where the number of clients connected is lower. During these periods of lower resource demand, it is not justified to have resources being wasted on potential clients that will not connect within that time periods. In order to solve this problem, we want to develop a system that manages this resource management automatically with a minimal impact on the game performance, preventing the waste of resources.

1.2 Objectives

The main objective of this project is to create a distributed system, that has the capability to dynamically adjust the resources it uses to support the MMOG. This means increasing or decreasing the number of resources used, in order to support an ever changing number of players. This adaptation performed by the system, should be done in an automatic and timely fashion to serve the needs of the users. It also has to be done in such a way that it allows for the usability of the game to be kept unchanged as much as possible. It is also necessary for the cost of the resource usage to be kept to the minimum value possible while maintaining the service quality on par with what is currently done. The last aspect being

considered is the security of the application. Avoiding cheating or other unauthorized actions that might hinder legitimate players experience is an important concern that is being taken into account.

1.3 Difficulties/Problems

There are a series of difficulties and problems related to what we are trying to achieve. One of them is the problem of threshold. When is it time to add or remove resources? In order to minimize resource usage and the corresponding cost, this has to be taken into account. Furthermore, it is important to find what is the best metric to use as threshold. We can think about using the number of players connected, the load of the servers, the communications taking place between servers among some other options.
The next problem we have to deal with is game consistency. When we add resources, in this case more server machines to support the increasing number of players, we have to ensure that the game world remains consistent in both the new machines and the older ones. This is also relevant when we remove machines that are no longer needed. It is mandatory to ensure that any information present on the disconnecting machines is kept so that players are not affected by any loss of data.
The performance of the game is another difficulty that we face in this system. It is necessary that the users do not realise what is happening in the background while they play. What this means is that gameplay should be kept smooth independently of the addition or removal of machines that might be taking place.
Security of the game state and information is also to be considered. MMOG's deal very often with some sort of ingame currency or other information that can not be tampered with. If an attacker or malicious user manages to compromise the application's security, he will also compromise the enjoyment of the game by the legitimate users.
Finally, it is very important to consider scalability, since it is a main concern in this kind of systems. The typical client-server architecture that is in use nowadays is not the most scalable approach and game companies often have to find complex solutions to make it scale.

1.4 Existing solutions advantages/disadvantages

There are some already existent approaches that try to solve the same problem that we want to solve. We describe them briefly here and discuss their advantages and disadvantages; On a later section we go deeper in how they work.
To begin with, there are two common architectures being used for MMOG, which are the peer-to-peer (commonly known as P2P), and the client-server architecture [11]. P2P has some advantages related with transmission latency and the absence of a single point of failure. It is however hard to prevent cheating and to manage the game state using this architecture. For this reason most companies rely on the classic client-server architecture which solves the disadvantages of P2P at the expense of the number of servers and bandwidth consumption.

Project Darkstar [37] is described as a platform that proposes to solve all the hard issues related with the development of MMOG, one of them being the addition or subtraction of resources as needed, and leaves to the programmers only the task of creating a fun game. Darkstar is a server-side infrastructure designed to exploit the multithreaded, multicore chips that exist today. It also aims to scale to a large group of machines while giving the programmer the illusion of being developing in a single-threaded, single-machine environment. The project has since lost its funding due to restructuring in the company where it was being developed, and was not concluded and tested enough.
A different solution based on P2P [28] was also proposed, which uses the advantages of this type of architecture in order to achieve the resource efficiency that we are looking for. It does however need to have some special nodes or trusted peers that control the security aspect of the game, which makes the system not able to fully exploit a P2P architecture. Another proposed approach was a MMOG support system [3], that presented a solution for most of the MMOG problems. However, scalability is a bit neglected and relieved to a second plane of focus. The test setting for this approach is also somewhat limited which does not make it clear how the system would behave on a larger scale.

1.5 Proposed solution

We propose a solution based on a cloud computing approach [2]. Cloud computing is a fairly recent approach to resource usage, that seems very interesting in the context of a MMOG. It provides the resources that a game of this type requires, while managing them in an efficient way that helps to minimize resource waste in times of low usage by the users. This contrasts with the other two commonly taken approaches. Client-server is limited when it comes to scalability. Peer-to-peer raises problems such as how will the game work when there are not enough nodes connected and how to effectively store the game data among the peers. Security is also an important concern that is hard to ensure in a peer-to-peer architecture.
In our approach we use Eucalyptus [1], an open-source platform that supports cloud computing. We chose this platform because it fits the requirements of our system. This platform allows us to virtualize a set of resources and then use them for our game. The game servers are hosted in this cloud. New instances of the server is launched when the need for it arises. An image of the game server has been previously uploaded to the cloud system, so that it can be launched when necessary.
Our system mediates the interaction between the clients and the cloud system. It acts as middleware and has direct control over the cloud actions. The Eucalyptus platform provides tools that allow for this control and monitoring. The middleware controls how the cloud scales and deal with the increase or decrease in the number of players. It also deals with the clients state and respective updates, as well as the persistence of the servers game data.

[1] http://open.eucalyptus.com/

1.6 Thesis Outline

The rest of this document is organized in the following manner. Chapter 2 addresses the related work, analysing other systems with a similar goal. Chapter 3 describes the proposed architecture and how we solve the problems that have been identified. On Chapter 4, we describe how we implemented our system and detail its components. In Chapter 5 we describe how we evaluate the system and present our results. To conclude, Chapter 6 summarizes the work done so far and reason about the overall success of the work, as well as present our ideas for future work.

2 Related Work

This chapter addresses the related work already mentioned in the introduction with more detail, and focus on the points that are important to our current solution. We begin by discussing in more detail the most common network architectures for MMOG. We then address some already existent solutions to the problems we previously identified, as well as other topics that are relevant to the goal we are trying to achieve.

2.1 Architectures

2.1.1 Client-Server

Client-server [32] is a distributed systems classic architecture. The main idea is to split the tasks of the system between two types of entities. The first entity is the server, which provides a given service, and the second entity is the client, which connects to the server and requests the service that it provides. The most common scenario for this architecture is having clients and servers in different machines, communicating with each other through some network setup.

The servers are said to be sharing their resources with the clients that connect to them, while clients do not share anything neither between themselves, nor with the server to which they connect. In this scenario, the communication is started with the clients initiative to ask the server for a connection, while the servers simply wait for incoming requests to which they reply.

The vast majority of MMOG employ some variant of this approach, mainly because of the advantages it gives, such as an easy to manage game state, and the prevention of cheating, since all the game logic runs on the server side, and only updates are sent to client. However, this approach has some downsides that derive from the advantages it provides. In order to achieve the advantages stated, this approach puts a higher resource cost on the server side, and also consumes a lot of bandwidth. There is also the issue of reduced interactivity between the clients, since every action has to go through the server, before it can reach the other clients.

The other main problem with this approach is the single point of failure located in the server. If the server fails then the game becomes unplayable for every client. Finally, it becomes clear that this approach is limited concerning scalability, since it is very dependent on the server resources.

2.1.2 P2P

Peer-to-peer computing [32] is a distributed application architecture that partitions tasks or workloads among peers. Peers are equally privileged, equipotent participants in the application. They are said to form a peer-to-peer network of nodes. In this architecture, every node accumulates the functions of client and server, and shares its resources, such as processor power, disk space and so on, with the other nodes. This architecture exists in two different forms: pure P2P where all nodes are equal, and hybrid P2P where there is at least one central server node that manages the others as we can see in Figure 2.1.

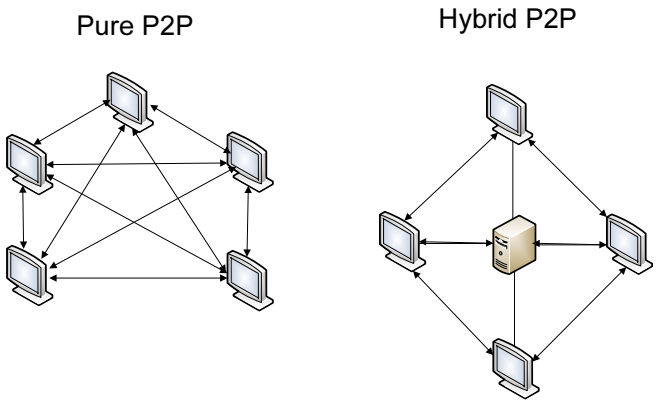


Figure 2.1: The two topologies of P2P networks

P2P seems attractive to the MMOG market because it minimizes the transmissions latency, it has no single point of failure, and it also reduces the costs to the game companies, since they do not have to invest so much money on server hardware and bandwidth.

On the other hand, P2P makes it harder to monitor possible cheating attempts from users with bad intentions. Since we have a part of the game logic running on the client machine, as there is no server, it is fairly easy for the cheats to be introduced. For the same reason it is also harder to manage the game state using this approach.

Some authors have listed a series of six aspects [12] they consider of utmost importance when considering to use P2P in a MMOG. Those issues are interest management, game event dissemination, NPC (non-playable character) host allocation, game state persistency, cheating mitigation and incentive mechanisms.

For all that was previously mentioned, P2P has not been used in many commercial solutions and the prevailing architecture is client-server that was described in the previous section. However, there are many studies being developed in this area, and some of them present interesting ideas and concepts which are discussed in Section 2.4.

2.1.3 Cloud

There is a third alternative solution, that has been gaining many adepts lately. The cloud computing paradigm [2] consists in the delivery of computing power as a service instead of as product. Clouds can be grouped into three major styles, the infrastructure as a service (IaaS), platform as a service (PaaS) and software as a service (SaaS). As the designation indicates, this classification refers to clouds in terms of service they provide, depending on the portion of the stack that is being delivered as a service. A cloud works as a pool of resources that are provided to the user. The users of the cloud only need to be concerned with the computing service being asked for, as the underlying details of how it is achieved are hidden. This method of distributed computing is done through pooling all computer resources together and being managed by software rather than a human. There are already some existing solutions using cloud computing such as the Amazon EC2[1] or the Google app engine[2]

This type of resource distribution seems attractive because of the way it allows an application to scale by requesting more resources as the need for them arises. Clouds also deal with the problem of machine failure without letting the user know it has happened.

On the other hand, using a cloud for a MMOG to host its servers also rises some important issues. Since clouds are supposed to be pools of shared resources (according to the architecture of a cloud), that are made available to the user by some service provider, might not seems very attractive to game developing companies that would obviously have to get their code running on the providers machines. Also if we think of the current way clouds are provided, every time the game needs to add another server to deal with the number of players, all the game server logic would have to be sent to the cloud in order for the server to work which is surely a time consuming task. This last concern is not relevant in some systems, that allow the creation of an image that is kept by the cloud and is launched when necessary. It is always possible for the company to create its own cloud dedicated to the game, which would solve these two problems, but that might destroy the advantage of reducing costs on the resource consumption and maintenance that normal clouds are supposed to offer its users.

From a theoretical point of view, clouds allow for a good scalability and resource management, while keeping the costs reduced to a minimum. In practice the real cost reduction is dependent on the way the cloud is to be deployed. It also allows for a proper security measures enforcement, since the servers are always under the control of the game provider.

2.2 Interest Management

When a player takes any action inside the virtual world of a MMOG, it changes is own state. In the same way, all the other players with which he interacts, also perform actions that cause them to change state. In order for a player to have a consistent view of what is happening around him, each player must maintain a copy of the relevant game state on his machine [8]. The simplest way of achieving this would

[1] http://aws.amazon.com/pt/ec2//192-7629682-8045352/
[2] http://code.google.com/intl/pt-PT/appengine/

be for every player to keep a full copy of the game state, and all players would broadcast updates to all other players. It is very easy to realise that this approach is no feasible, as it consumes a lot of resources and it requires an enormous available bandwidth, which makes this solution simply not scalable enough for this type of system.

It is important to clarify what is the information that is relevant to the player, such as events that are happening on is immediate vicinity up until a certain distance. For this reason interest management is normally modelled as a sphere around the player avatar [8].

Interest management schemes can be grouped into two major categories, space-based and class-based, or extrinsic and intrinsic respectively [23]. Space-based interest is determined based on the relative position of objects in the virtual world, while class-based is determined from an object's attributes [8].

2.2.1 Auras

The space-based approach is normally based on proximity, so it is modelled using an aura-nimbus model, where the aura represents the area that bounds the presence of an object in space and the nimbus represents the area where the avatar can perceive other avatars. Both aura and nimbus can be modelled as circles around the players avatar, being the nimbus a circle with a larger radius than the aura. A player can then see and receive updates from actions of other players located inside the area covered by his nimbus. This approach when used in its pure form allows for a fine-grained interest management in which only the relevant updates are sent to subscribers. On the other hand it is not very scalable due to the cost of computing the intersections between players auras and nimbus [8] [21].

To mitigate the problems of the pure aura-nimbus approach, region-based approaches can be used. These approaches divide the game into regions that are static and are used to calculate the area of interest in a cheaper way. The quality of this solution is strongly related with the way the game world is divided into areas. The most common approach is dividing into square tiles, but there are other shapes that have been shown to be more efficient such as hexagons [8].

2.2.2 Vector Field Consistency

A different approach to interest management that unifies several forms of consistency enforcement and multi-dimensional criteria to limit replica divergence with techniques based on locality-awareness. This technique is called vector field consistency(VFC) [34].

VFC splits the world into zones around a pivot (the player avatar). This creates consistency zones based on the distance between the pivot and the target object. Consistency zones are iso-surfaces, ring shaped and concentric in the pivot. This separation in consistency zones, guarantees that objects in the same zone have the same degree of consistency. Each zone has a degree of consistency assigned to it that specifies the degree of consistency to be used for that region, and obviously this degree gets weaker as the distance gets higher.

The consistency degrees are defined as a 3 dimensional vector, where each dimension is a numerical scalar,

representing time, sequence and value. Time specifies the maximum time a replicated object can spend without a refresh; sequence specifies the maximum number of lost replica updates; and value represents the maximum relative difference between replica contents or against a constant. When any of these values is exceeded, it means that the current replica needs to be updated. It is possible to ignore any of these values, by setting them to an infinite value.
An application of this algorithm to a first person shooter game [21] was also proposed, which was implemented for the game Cube 2: Sauerbraten [3], and showed good results, with a considerable decrease in communication and an overall increase of the game performance and scalability.
The results of this approach seem interesting from a point of view of resource consumption. With this approach it is possible to limit interactions between players to the ones that are in fact relevant. This allows for a reduction of resource consumption and the overhead of communication. This effectively increases the system performance while keeping the resource consumption under control, which is a fundamental aspect that we are considering.

2.3 Load Distribution Algorithms

Architecture is not the only concern when it comes to MMOG. It is also important to have a load distribution algorithm, to distribute the load among the available servers in an efficient way. This section presents some existing approaches.

2.3.1 Dynamic Load Sharing Algorithm

A dynamic load sharing algorithm [11], divides the game world into a grid of equal cells. Cells are an abstraction that represent the different zones of the world. The number of servers can be less or equal to the number of cells. Each server manages a group of neighbouring cells, which is called a partition. A server also talks to his neighbour servers when it is necessary to perform load sharing. There are various levels of neighbours being considered. Level one are the ones adjacent to the current partition; level two are the ones that are adjacent to level one neighbours and so on.
Each avatar receives updates from within the partition where it is currently located, and also from avatars in other partitions that are located on the border between the two partitions. This introduces the concept of subscription region, which are the cells that compose the border of each partition. An avatar that is in partition A and close enough to the border to be able to see avatar in partition B, should be subscribed into B's subscription region which is composed by the border cells of the partition.
The system works with a fixed threshold of clients, above which the server is considered overloaded. They also consider another variable that is called Extra load, which represents the capability that the current server has to accept load from other servers.
This approach seems to perform well even with skewed workloads, and effectively decreases the workload on the servers, successfully balancing the overall load among the available servers, and also minimizes

[3] http://sauerbraten.org/

the migration ratio of clients between servers while balancing.

2.3.2 Dynamic Load balancing Algorithm

A dynamic load balancing algorithm [22] divides the game server architecture into two types of servers, a master server and a number of game processing servers. The master server acts as the controller of the system, and also as a sort of gateway for connecting clients, since it distributes this connections among the game processing servers. The master server also controls the number of processing servers active at the moment.

The game processing servers are the major game servers that process actions of game units such as the player avatar and NPCs. These servers are composed of four manager components, which are the unit action manager, the player connection manager, the load balancing manager and boundary interaction manager. The unit processing manager as the name implies, is responsible for managing all the action taken either by player units or by non-player units that work under autonomous routines. The player connection manager is responsible for checking the status of the clients connection and also of sending him notifications. The load balancing manager is where the load balancing algorithm performs his job in order to decrease performance degradation which comes as a natural consequence of the movement of the players inside the game universe. Finally, the boundary interaction manager, is responsible for dealing with the problem that we described previously, which occurs when game units are located in the border between two partitions.

The first step of the proposed algorithm is the spatial partition of the game regions. The authors propose a vertical partition against the x axis. Each of these partitions is managed by a processor which means that every game unit present in that partition, is assigned to the corresponding processor. They also consider the border areas of the partitions, and call them export areas or import areas. Import areas are located in the neighbour processors and contain the game units that can be transferred to the current processor, while export areas are located in the current processor and represent the opposite.

Each node has one of five possible states: normal, overloaded, locked, underloaded and unknown. The nodes check periodically their current load and compare it to a fixed threshold value and sends a request for help in case of an overload. These requests are only sent to its two direct neighbours. If the neighbour nodes can not accept the extra load, they notify that fact and the requesting node is forced to send a remote help request, which is forwarded by the neighbour nodes to their direct neighbours. When a node is found the message goes back through the neighbour chain until the original requester and the load is shared.

In the tests performed, the system was compared with static partitioning and revealed some better results as expected.

2.3.3 Locality Aware Dynamic Load Management

One common event in MMOG is the gathering of a large number of players in a common place called hotspot. This event which is known as flocking is the main motivation for the locality aware dynamic

load management algorithm [10]. The authors propose an algorithm that uses dynamic partition of the game world among the available servers in order to solve this problem. There are two main aspects being considered: The balance of the server load in terms of the number of players, by splitting the game world partitions across more servers; Decreasing inter-server communication by maintaining the locality of adjacent regions or aggregating adjacent regions into large partitions to be assigned fewer servers[10]. The two main concepts of this approach are locality aware load shedding and locality aware load aggregation. When a server detects that it is taking more than a certain predefined time to send updates to 90% of its clients, the server is considered overloaded and load shedding is started. Load shedding attempts to migrate some load to its neighbour servers (neighbours in this case are actual pyshical servers that are close to the loaded server) as long as these servers are below a given safe load threshold, which means the servers are still available to take extra load. This first attempt to split to closer servers is due to its objective to maintain locality. If it is not possible to find a neighbour to send the extra load, the server floods the network with a message to find a server that is capable of receiving its load. In this case, the other server also checks its threshold before accepting any extra load.

As already mentioned, the other aspect of this algorithm is aggregation. After an hotspot is formed and players rush into it, its interest eventually disappears and players start to move out of that area. When the hotspot was formed it most certainly triggered a load shed and so that area is now split among various servers. As players move out, this splitting of areas between servers causes an increase in inter-server communications, which is not a desirable situation. What the algorithm does in this case is attempting to group closer regions back into the same server, in order to reduce not only the number of servers used, but also the communication performed between servers, thus improving the game performance.

In the tests performed by the authors, the algorithm showed very encouraging results when compared with other existing approaches. This type of algorithm improved the performance by a factor of 8 when compared to a static partitioning algorithm, and by a factor of 6 when compared with other load balancing algorithms like [11] [22], that do not consider spacial locality.

2.3.4 Hybrid Load Balancing

Most of the load balancing algorithms fall into one of two categories, global load balancing or local load balancing. Global load balancing takes the entire state of the system into account when performing load balancing, while local load balancing considers only the state of the current server and its adjacent peers. The global load balancing approach gives a better view of the system, but it can be very slow to generate it. The local load balancing option is in fact faster, but has the disadvantage of creating load distribution solutions that do not last long, since they did not consider the global view of the system and decided how to split the load based only on the neighbour servers. The Hybrid load balancing algorithm [20], proposes a solution to this problem, by combining both approaches.

The Hybrid load balancing algorithm starts with the usual split of the game world into regions, and creates a set that represents the load of each server. It also creates the opposite set to this last one, which is the capacity a server has to receive extra load without being overloaded. Based on this information, it

creates clusters of servers, which are groups of servers with an overall reduced load and that are capable of accepting load from other more heavily loaded servers. The clusters have an abstract value called the center of mass, which represents the mean location in the game world of all users that are present in the regions managed by that cluster.

When a server is overloaded, it checks the neighbours load information and based on that tries to find a potential server to share the load. Using the information just described, the server tries to find a set of potential target clusters. To select which server cluster receives the load, the server evaluates the values of load in that cluster as well as the center of mass, to choose the one that is closest to the current server. If for any reason that cluster can not accept the extra load, the process then checks the next cluster present in the set.

This method was compared with the algorithm that served as a basis for the authors proposed approach. It was found that it takes more time for it to execute the load balancing, but it produces better results in terms of overloaded servers during the test time.

Another approach that can be considered an hybrid approach is detailed in [19], where information about local load and global load are used to achieve a better load balance. The world is divided into hexagonal areas, as the authors argue that it reduces the communication between neighbours. When a server is loaded, the algorithm starts with a local search for possible candidates to share the load, and creates candidate lists with this information. The search is expanded until all the overloaded servers are identified as well as the servers that possibly will receive the extra load. This approach was tested against local and global load balancing approaches, and showed an average of loaded servers close to the global approach, and much better than the local one. The ratio of migrated users is much lower than the global approach and slightly above the local approach which was to be expected. When it comes to the global system operation cost, it was shown that the algorithm manages to stabilize that operation cost faster than global and local approaches, and keep it stable as time increases, while the other approaches fluctuate much more.

2.3.5 Microcell Oriented Load Balancing

Some solutions use some form of P2P to solve the load balancing problem, even if not in a pure form, but using the hybrid version of P2P. In [1], the authors argue that a pure P2P approach is not feasible for an MMOG, and that the hybrid version is a more suited architecture to use. In this approach, the authors propose the use of hexagonal shapes to divide the game world into different areas known as microcells. Special nodes, called masters are responsible for one microcell, and coordinating in a collaborative manner the interactions between the members active in each microcell. There is a pool of master nodes, that are responsible for controlling the game world.

One of the main concerns of this approach is the definition of hotspots in the game. It is argued that simply counting the number of players in a zone is not a correct approach, as the real thing that generates load is the amount of messages exchanged due to the players interactions either between themselves or with the environment. In order to address this, the rate message generation rate, is being used as a

measure to estimate the load on the master nodes.

When a server is loaded, it is necessary to move some of the microcells it is responsible for to a less loaded server. In order to do this, each microcell defines a border area, similar to what is done in [22], where information is shared between adjacent microcells, concerning avatars present in those border areas. Only microcells that are managed by different peers, even though they are adjacent, have this border area, since it is not necessary in the case where both regions are managed by the same peer. This allows for a sharing of a partial state of the microcell, rather than a full state, which allows for the reduction of the communication needed, and an overall reduction of system load.

When a server is loaded, the algorithm looks for all the clusters of microcells that are being managed by the overloaded server, and then it selects the smallest cluster, and chooses the cell that currently has less connections with other cells, which means it has less avatars that are connected to avatars in other areas (such as border cases). The selected cell is then migrated to a less loaded server. The process is repeated until the server is no longer overloaded.

An alternative approach to this idea is taken in [35], where the authors also propose a microcell type division for the game world, but in this case apply it to a client server architecture, having one server deal with clusters of microcells. The approach is very similar to the one described in [1]; The main difference resides in the various algorithms proposed to make an efficient distribution of this microcells in the available servers. Four algorithms are proposed and tested for this purpose. The results show that the best of them is a deployment algorithm that combines simulated annealing that allows for a reduction of up to 30% in the individual servers load. The test setting for this approach is somewhat limited in its scale, thus it is not clear if this approach would scale well with a large scale MMOG.

2.3.6 Load Balancing with Kd-Tree Partition

A Kd-tree, which is short for K dimensional tree, is a data structure used in computing. It is used to organize points in k-dimensional spaces. Their most common use includes searches that include multiple dimensional keys. One example of this is nearest neighbour searches. Fig. 2.2 shows an example of a 2 dimensional kd-tree.

The first two assumptions that are made for the load balancing with Kd-tree algorithm [5], are the fact

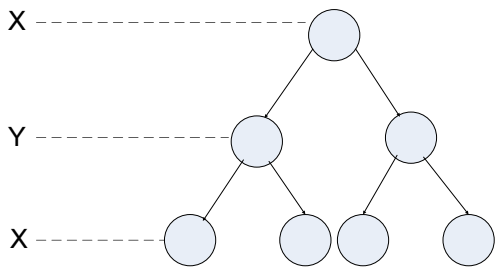


Figure 2.2: 2 dimensional Kd-tree structure

that the system is heterogeneous, since every server might have different resources, and that load is not proportional to the number of players. The load is considered to be the amount of bandwidth required to send state update messages between clients and servers.

In this approach it is proposed the use of a Kd-tree with k=2, since the authors claim that most of the current games use a two-dimensional representation of space, even though they are three-dimensional. We are not sure if this claim is still valid nowadays as some of the games stated as examples for this have evolved and might now use a three-dimensional representation of space (for instance, World of Warcraft[4] allows players to fly in a three dimensional world), but it is always possible to use a tree with a higher branching level.

The representation of world regions for this approach is done in the following manner: each node of the tree represents a region of the space. In each node it is also stored a value the authors call a split coordinate, that refers to the division of this region in sub-regions. This subdivision is represented by the child nodes, which represent the sub-region before it was split and the sub-region that contains points whose coordinates are greater than or equal to the split coordinate. Each level of the tree represents the x and y axes of a two-dimensional world; that is, if level one represents the x axis, then level two represents the y, and then level three represents x again and so on. On the leaf nodes no coordinates are stored, only the list of avatars present in that region. Leaf nodes are mapped to game servers. Each node also stores two more values that refer to the capacity and load of the subtree. Each non-leaf node's load is equal to the sum of the loads of its children nodes, and the same happens to the capacity value, where the value is also the sum of the capacities of its children nodes.

One of the key factors of this algorithm is the distribution of player avatars. The authors argue that taking into account the number of players is an unrealistic measure of load, and propose the use of the amount of communication needed between them as a better suited measure. To do this they assume that players that are farther apart in the world map, have less need to communicate between themselves, so they create a sorted list of avatars based on their coordinates. After having sorted the avatars, it is possible to calculate the load they generate and thus define the load of the node to which they are attributed. Each node uses a dynamic load balancing algorithm based on the KD-tree structure that allows for the splitting of the load when needed.

This approach was compared with other similar systems such as the Progrega and BFBCT [6]. It was shown that the KD-tree approach achieves better results when it comes to the deviation of the ideal balance of a system, and also has a better migration ratio in a situation with hotspots in the game world as well as a lower communication between servers, than both the approaches proposed in [6]. This approach has some encouraging results concerning communication, which definitely impacts the game performance. Although, this approach might consume more resources than what would be desirable. When the game world becomes too subdivided, and the number of players is high enough, the processing of the structure might be complex. The idea of using communication has one of the load metrics, is definitely interesting and seems to fit well in our dynamic resource allocation scenario.

[4] www.wow-europe.com

2.4 P2P Further Discussion

There is a lot of research on the subject of using P2P for MMOG. As it was previously stated in this document, P2P seems like a very natural solution to scalability for a MMOG environment. In this section we discuss some further aspects of P2P that do not fit our previous discussion of architectures. These are, however, just as relevant to our discussion as the architecture part. They present possible solutions to some of the problems that we described and that are mentioned in [12] as being essential for any system that attempts to use P2P for a MMOG.

There are some proposed approaches to using P2P for MMOG. In one of these approaches [14], the authors claim that it is possible to take advantage of P2P by creating an overlay network that uses a series of components to achieve its goals. The first component is Pastry [30], a DHT-based Overlay Network, which provides the organization and structure that allows the system to tolerate network failures as well as a high number of connecting peers.

To manage the game objects, the authors are using an extension to Pastry called Past which allows the persistence of objects. The last component is Scribe, another extension of Pastry that is meant to deal with multicast, which is used to disseminate the game events to all the peers. This approach has been tested with an MMOG that is not too time critical. The authors claimed their approach suffices in that case, but they have not conducted tests on a more typical MMOG. They are in fact time-critical, and latency is a serious issue. The approach taken in [28], seems like an interesting approach that actually solves most of the issues with P2P that could be problematic for an MMOG. It does require some special nodes to control the security aspect of the game, so it can be argued that it is not pure P2P, which might bring some problems like points of failure that P2P normally does not have. These special nodes have to monitor in some way the actions of other peers, which introduces an additional layer of complexity and possible overhead to the application that is not desirable.

2.4.1 Consistency

One of the referenced problems with P2P for MMOG is the difficulty of keeping state and by consequence of keeping consistency of the various servers. On the approach taken in [31], the authors state that a game does not always need the same consistency level on every situation. This is not a new idea, it is directly related to the already discussed concept of area of interest that was previously described. What the authors propose is a consistency management infrastructure based on P2P, which allows for different consistency models to be applied. For each situation, different consistency models are available, and it is decided at runtime which one of them is to be used. This requires the use of a reflective API, that can provide information such as current delay on responses, system load and other important informations that allow the selection of an appropriate consistency model. It is possible for a user of this infrastructure to extend it to suit his own needs, by using plugins that allow the creation of new consistency models to add to the ones that already exist.

Finally, in order to fully exploit the P2P network, the authors propose that objects can be dynamically relocated between peers in order to provide a better gameplay. A simple example provided is a player fighting a NPC while the latency between the peer of the player and the peer hosting the NPC is high, it is possible to relocate the NPC to the player peer, in order to provide a higher consistency and synchronization that does not hinder the gameplay. Nevertheless, this relocation is only performed in cases where the interaction is expected to be long lasting and not only of mere seconds, since the cost of relocations can be high.

There are some other approaches to consistency in P2P like the ones proposed in [16] [9], where Voronoi diagrams are used to address consistency. A Voronoi diagram is obtained by dividing a given area, where a set of points exist. This partition is done in such a way that for a given point p, a region is associated with it that contains all the points that are closer to p than to any other point in the space. This creates a more dynamic division of space which according to the authors suits better the needs of P2P load distribution, and helps to create a better model for consistency and area of interest around the player.

2.4.2 Player Distribution and Load Balancing

Some authors argue that the world division into areas is unavoidable in any algorithm that tries to solve the load balancing problem for MMOG. Their approach [18] uses P2P, and they have created an algorithm that tries to effectively distribute zones of the game world among all the peers, in such a way that inter-zone communications cost can be kept to a minimum. It was presented proof that this is a NP-hard problem, and an heuristic to solve this problem was provided and tested.

Some would disagree with the fact that every area of the game needs a server attached to it; An algorithm for player clustering [29] argues that conceptually areas with no players in them, do not need servers to be assigned. This approach can be achieved by grouping the players into what is called player clusters. To create such clusters, each player's distance is evaluated, and if they are close enough to each other then they belong in the same group. These groups have arbitrary shapes because the distance is being computed with relation to the players, instead of to the center of the cluster.

Each cluster is assigned to a Virtual Server (VS) that is responsible for that cluster and follows the group as it moves. VS's are also responsible for deciding which VS's are its neighbours and which of them are not; since there is no central authority the VS's have to decide this by themselves.

Since a cluster of players is equivalent to a VS, when the load on this VS exceeds a certain threshold it is necessary to take some measures. For load balancing purposes, the authors consider three possible approaches, moving clusters from one node to the other, moving one or some players from one cluster to another, or splitting one cluster into two parts and move them to different servers, as described in [29]. This approach was tested against a CAN-based approach [27], and revealed advantages in some situations, but performed clearly worse when it was tested with an uneven distribution of players from a real game, having to transfer almost the double of the players than the CAN approach.

2.4.3 Game Security

One of the main reasons P2P is not more widely used in MMOGs is cheating. As stated before, it is hard to monitor user's actions in a P2P environment. To address this issue there have been some proposals [17] that aim to create a reputation system similar to that of auctioning websites such as eBay[5]. The basic concept is that every member has a rating based on his interactions with other members. These ratings can only be given when two members enter each others area of interest, and rate each other based on their interaction.

In order to avoid users tampering with their own rating, each users rating is stored by one or more nodes known as trusted peers, which are chosen among the peers based on their rating. When a query for a users rating is made, its reputation value is given based on the reply of the majority of trusted peers, in order to keep malicious trusted peer from tampering with the reputation.

This approach has many positive aspects, and solves some problems related with the P2P security, but it is still exploitable; nothing prevents two peers from crossing each other many times and interacting, in order to elevate each others rating. Also, the mechanism for choosing the trusted peers is also not perfect, mainly due to the fact that was just mentioned, but this problem is very much mitigated by the way the reputation queries work, by using a type of quorum approach that rules out malicious users.

Another important aspect in P2P environments is authentication. In a MMOG it is essential to have the players authenticated in order to track their actions, and to guarantee that they are who they claim to be. Based on this fact, there was a proposal for an authentication mechanism for P2P MMOG environments [36], that tries to deal with the security problem.

To begin with, the authors assume two types of virtual environments, open and moderated, where moderated ones are operated by a specific system operator that is responsible for managing the environment and the players that participate. An open environment has no operator and it is operated cooperatively by all the participants.

Two approaches to authentication are proposed, one for each type of environment. The first one is based on the well known concept of certificate authority [15] and is applied in the moderate environments. The operator issues certificates to the other users, with which they sign and validate their and the other player's messages, while the operator acts as a certificate authority. For the open environments, the authors propose a scheme based also on public keys exchange. A user creates its key pair and then stores it in several peers. Since peers can not be trusted, when it is necessary to retrieve a key, the valid key is chosen based on a majority mechanism.

The approach used for moderate environments implies the use of an entity that regulates the entire process. This creates a possible point of failure. On the second approach the solution relies on the peers not being compromised. This is something that is hard to achieve in a P2P environment, thus this approach might have some issues there.

[5] http://www.ebay.com/

2.5 Cloud Approaches

Cloud computing is a relatively new paradigm. Only in recent years have we started to see applications of this idea for commercial purposes. Systems such as the Amazon EC2 or the Google AppEngine are examples of this. Several companies have started providing their cloud platforms for others to use. MMOGs are not using this approach yet. Game companies rely on huge data centers to deal with the game servers. This approach requires a lot of maintenance and specialized personnel to install and maintain the game servers. There is some research on the topic of cloud MMOGs, but its use in actual services is still purely academic and very limited.

MMOGs are applications that can require a very large amount of resources at one time, and very small amounts in other periods of time. Observations of user behaviour in applications such as Runescape[6] or the Steam platform[7], show that player connections have a large variation during different hours of the day [24]. A cloud approach seems like a natural solution to solve the problem of client connection variation. The resource elasticity it gives, can solve the resource adaptation problem.

Different studies have addressed the problem of virtualization overhead [25], that rises from the use of clouds. It was found that virtualization has some costs when it comes to performance of the game specially during hours of heavy client usage and high load. Nevertheless, it was found that if such costs are taken into account when developing the system, it is possible to achieve good results.

The OnLive [8] platform is one of the first to deliver a gaming platform in the cloud. Although it operates as a service that provides games to clients independently of their personal machine configuration, some of its concepts are interest to our current discussion. Based on what is done on this platform, a new system that mixes cloud computing with p2p [33] was proposed. This approach tries to explore clients and server location in order to minimize the delays from communication. On this platform, the client installed on the user's computer is a small application, and the vast majority of the computation that occurs, is taken care of by the cloud. The cloud hosts both the clients and the servers, even if in different physical locations. This creates a P2P overlay that can operate on the cloud while having the illusion of being in a normal distributed environment. Updates are sent directly between peers, and the cloud is being used as a host to provide the resources. It seems like this approach suffers from the same problems of the other P2P approaches that we have previously mentioned. Problems such as the fact that the number of connected nodes might not be enough to support the game at all times, are to be considered in this case. On the other hand, the security issues that normally rise on P2P, are more contained in this case, since the game provider has more control over the peers.

2.6 Proxys

The network aspect of a multiplayer game is also an important aspect that must be considered. Some authors compare the growth of the MMOG to that of Web a few years back, which led to the creation

[6] http://www.runescape.com/
[7] http://store.steampowered.com/
[8] http://www.onlive.com/

of Web caches, URL-based switches and other solutions to support that technology [4]. They argue that the enormous growth of MMOG systems, more than justifies a similar approach, and in order to achieve this, they propose a concept of booster box.

The observation that led to the attempt to address the several MMOG problems in the network was the fact that servers need to share state, either from the game world or the users, and most of the times servers are exchanging information among themselves that is not relevant to all of them, for instance because a player is not in the area of interest of another player. This is where booster boxes come into play; they are network aware devices, through monitoring traffic, measuring network parameters such as delay, by participating in routing exchanges, or by acting as an application layer proxy server [4]. The booster box combines network and application awareness in a single entity that serves one or more clients in its network vicinity.

The main operations of a booster box are caching, aggregation of events, intelligent filtering and application-level routing. The caching operation is common to other types of systems, and here is no different: it stores non-real-time events, and acts on behalf of the server in order to reduce its load. Aggregation of events derives from the idea that clients might send similar requests to the server, so the booster box groups them in only one relevant event and sends it to the server. Intelligent filtering uses the application awareness of the booster box in order to decide if a given event is still relevant, and in a negative scenario, those events are effectively dropped by the booster box without ever reaching the server. Finally the application-level routing operation makes the booster box act as a router and only send the events to the servers that are responsible for handling them.

This approach is thinking ahead into the future, where the data structures that support current MMOG will not be able to deal with larger numbers of users. This would require ISP's to install the boosters boxes, and allow third parties such as the development company to have access to the box and configure it for their particular application, which in the current way things are done is still very hard to achieve, and would most likely bring more problems than it would solve. This solution does not seem very scalable either, which is a major requirement for our system. Moreover it can not really achieve the elasticity of resources that the system requires, as the booster boxes are not fully under the control of the game provider. Security also seems to be an issue in this case, since booster boxes are somewhat public and are dealing with game logic. This aspect makes them vulnerable.

2.7 Academic Systems

We now discuss some academic systems, that even though their final goal might not be the same as the one we are proposing, have some similarities. The solutions proposed are close to what we intend to do in some important aspects. We briefly analyse them and point out where they differ from our goal.

2.7.1 Kosmos

Kosmos [3] is not a system by itself, it is a game that was created with the purpose of showing how the concept designed by the authors would work in a real world environment. The authors propose a solution for an MMOG that attempts to deal with all the common issues related with this type of system. The structure proposed is a common client-server implementation, where all the game servers are connected with each other, and accept client connections. The game world is divided into regions, that can assume the form of any convex polygon. Each region is assigned to a different server.

Kosmos introduces two concepts of locking: the region lock and the object lock. Region refers to the act of locking a certain region of the game world. The authority to do this lies solely with the server that is responsible for that region, and it accepts requests from other servers that might be executing events that somehow involve that region. As usual, when a server acquires the lock, subsequent requests are queued and have to wait for the lock to be released. The object lock is similar to this one, servers request locks for objects, and the lock is granted by the server that is responsible for managing the state of that object. Both this locks are used in order to maintain the consistency of the game, even if not in a very efficient manner.

Another important concept in this approach, are the events. Events are atomic transactions that happen in the world and change one or more object states. In order to announce these events, the publish/subscribe pattern is being used. As usual in this type of game, problems always arise in the border between region. Interest management between players in neighbouring areas is an example of such a problem. In order to solve this the authors use publish/subscribe, where every server is subscribed to receive events from neighbouring servers, if these events occur in the border areas between the two. Servers also subscribe clients to receive relevant events, when their avatar is in a border area that belongs to the players area of interest, and unsubscribe them as well when they leave this border area. Clients may receive events from various servers, but they only communicate with one server at a time.

In order to deal with the appearance of hotspots in the game world, the authors propose an algorithm that tries to predict the formation of an hotspot, and gradually starts to migrate data to another server that is less loaded over an extended period of time, in order to maintain seamless player transfer between servers. Unfortunately, this algorithm relies on predicting something that the authors themselves claim that is unpredictable, which is the spontaneous formation of hotspots. It is very likely that the goal of seamless player transfer is not be fully achieved.

Another issue is scalability, which the authors only address briefly and do not really provide a solid algorithm to deal with the fluctuation of the number of clients connecting at any given time.

The solution was tested using a simple game called Kosmos, and was tested for a relatively small number of clients, showing good results in handling the border problems. Nevertheless, while some of the approaches proposed here are interesting and can probably be developed further, this scenario is not the most common one for today's applications of this type, where the number of players is way higher than the 500 used in the tests.

2.7.2 Solipsis

Solipsis [13] uses P2P, which theoretically is a self-scalable architecture. There are three types of entities being considered: avatars, objects and sites. Avatars in this context, represent the users which are the main actors of the system, objects can represent everyday objects in the virtual world such as books, furniture etc. Sites represent parts of the virtual world that can be populated by users and occupied by objects.

In a more technical aspect, Solipsis is divided into hosts, which maintain the information about every entity in the virtual world, having one host for each instance of the virtual world. Since an host can have multiple entities under its responsibility at any given time, the concept of node is introduced. A node is a set of resources that is allocated to the management of a given entity, it maintains the information about the entity.

As previously stated, this system works over a P2P overlay based on Voronoi tesselation to map the relations between the nodes in the virtual world. The nodes are self-suficient, and compute most of the data that they need, such as position in the world with relation to physical concepts. The system employs some form of interest management, based on concentric circular areas with different consistency levels. This is done in order to limit the updates that are being exchanged. This management is achieved for every site and concerning entities that are in that site at the time.

Solipsis proposes a general system for P2P use for a virtual environment However it does not take into account factors that are fundamental for a typical MMOG, such as security. MMOG normally deal with some form of virtual currency or items that are prone to exploitation, and Solipsis does not propose any solution to that, mainly because it is not its main purpose. It might constitute a good foundation, but it would need a big refinement in order to be usable for an application of MMOG type.

2.7.3 HyperVerse

HyperVerse [7] is a system for supporting distributed environments, that relies on P2P as a base structure for operation. It does not use a pure P2P approach, but rather a two-tier one, with a federated backbone and a loosely structured P2P client overlay. The system implements an interest management algorithm, based on a circular are around the player. This is done in order to determine what information is relevant to the avatars. It briefly suggests an interest management scheme based on social bias between players, where avatars could choose who they interact with and get updates only from those avatars. They do describe this approach as almost impossible to attain with current structures and methods.

The world is subdivided into regions as in many other approaches that we have seen. These regions are used to enforce different levels of consistency in cooperation with the interest management techniques also in use.

The distribution of information between the peers is being done using a torrent protocol, that the authors argue deals well with big amounts of information specially in crowded areas.

One of the problems with this proposal, besides the fact that it was not thoroughly tested yet, is the fact that much like the previous system presented [13], it does not consider the security aspect that has to

be considered when using P2P for a commercial MMOG.

2.7.4 Darkstar

The MMOG market is an ever growing phenomenon. That was the conclusion drawn by the responsible for project Darkstar [37]. He also acknowledges that the current state of game development is not directed to exploiting the today's multi-core processors architecture. He argues it could be very useful to this type of game, since most of the events that happen in-game can be parallelized. This classifies MMOG's as embarrassingly parallel problems. This type of problem is one in which its tasks can be almost trivially parallelized, but as the author explains, games have always been seen as single threaded programs, and their developers are not familiar with the nuances of multi-core programming, so they can not effectively exploit this inherent parallelism.
In order to deal with these issues project Darkstar was created. It is an attempt to create a server-side infrastructure that deals with the multithreaded and multicored aspects of the current computer chips, relieving the programmer of such task, and giving him the illusion of being working in a single threaded, single machine environment. The system works in an event based fashion, that creates a server task to respond to user input events, and has the capability to change the game world.
Darkstar provides a container for the server to run and benefit from interfaces for persistence and communication purposes. All the data is kept in data stores that can be accessed by any cluster of machines that is running the Darkstar stack and game logic, which allows for any piece of data to be moved between all the machines. The same happens with communications, which allows the movement of tasks between machines which helps to deal with the load balancing problems by allowing to move information and players around, instead of partitioning the world at compile time.
This project had some interesting concepts but was not totally concluded before it lost its funding due to restructuring on the company where it originated. This makes it hard to know if it would achieve its purpose efficiently.

2.8 Summary

In this chapter we discussed the most relevant research related to our work. We analysed and discussed the infrastructure aspect of an MMOG. On this subject we presented and discussed the advantages and disadvantages of client-server, P2P and cloud infrastructures. We have discussed interest management approaches and evaluated their impact when it comes to the game network and overall performance. We have also discussed several load balancing and player distribution techniques, analysing their advantages and disadvantages. To conclude we presented several academic and commercial systems, that have a similar goal or that contribute in any way to our work. Table 2.1 compares the analysed systems against our requirements.

	Scalability	Usability	Performance	Resource Optimization	Security
Client-Server	not enough	yes	yes	no	yes
P2P	yes	yes	yes	not completely	no
Cloud	yes	implementation dependent	implementation dependent	yes	yes
Auras	it helps	not related	yes	no	not related
Vector Field Consistency	it helps	not related	yes	yes	not related
Dynamic Load Balance	no	yes	not clear	not clear	yes
Locality Aware	not clear	yes	yes	yes	yes
Hybrid Load Balance	not clear	yes	not clear	yes	yes
Kosmos	no	not clear	not clear	yes	yes
Solipsis	yes	not clear	not clear	yes	no
Hyperverse	yes	not clear	not clear	yes	no
Darkstar	yes	not tested	not tested	not tested	yes

Table 2.1: Comparative table

3 Architecture

3.1 Introduction

In this work we decided to use the cloud computing approach. Cloud computing has a series of characteristics that we already mentioned early in this document which greatly contributed to our decision. This characteristics are in line with our requirements of scalability, elasticity of resources, performance and usability. The main characteristic that weighted on our decision was the elasticity aspect of clouds. Furthermore, this is also an approach that has not been widely explored for MMOG, in contrast with the other two possible approaches that we have discussed, and that have had extensive research on this subject.

The cloud approach seems very natural for Cloud DReAM. We want a very scalable system, that is also capable of adjusting its resource usage as the need arises, and clouds are capable of performing both roles. It might be argued that P2P would present itself as a more scalable and effective solution; however, as we have discussed in previous sections, despite its many positive aspects, P2P lacks in the security, game state persistence and availability aspects, which are critical for an MMOG.

As for client-server, it is simply not scalable enough for the type of system that we are aiming at. Even though most of the actual MMOG in the market use this approach, it is still not elastic enough to deal with client fluctuation. It also has a higher monetary cost not only on the type of hardware that is needed, but also on the personnel required to maintain it. Furthermore, the possible overhead of managing the addition and subtraction of servers could severely hinder the game performance which is something that can not happen in a MMOG.

3.2 System Overview

On a very high level view our system is represented in Fig. 3.1, where a number of game clients connects to the game servers that are being hosted by our cloud platform. Each of the servers present in the cloud is running a previously loaded image of the game server. Clients connect to one of the servers according to the cloud's load balancing policies. The resources (in this case the servers) being used to support the game are managed dynamically. The game can start with one single server, and due to the interactions happening inside the game world, it might be necessary to add more resources to support the increasing demand on the servers. The opposite is also true, the number of resources being used can be decreased, if the current demand does not justify their presence. This management is performed automatically by

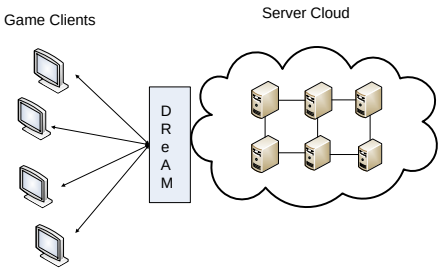


Figure 3.1: General view of the system using a cloud platform

the system and the player is not aware of what is happening in the background.

The Cloud DReAM middleware system acts between the client and the cloud server. It is split between the clients, the servers and the cloud manager. On the client side, it is responsible for dealing with the client status updates having into account its area of interest (AOI). The client's status is kept by the middleware as well as the status of other players that might be relevant. This information is then used to decided what is to be presented to the player. The communication between client and the middleware platform is possible through an API.

On the server side, it is necessary to take into account the load balancing issues. Cloud DReAM is responsible for dealing with this aspect. To fulfil this task, it is important to consider how the players are distributed among the servers. Cloud DReAM uses an algorithm which divides the game world into different areas that are managed by different servers. The consequence of this is that players connect to the server that is responsible for the game area where their avatar is currently located. Based on this information, the system takes the appropriate measures. These measures are the migration of clients between servers based on their position inside the game world.

The servers are capable of keeping some state locally, but since they are volatile machines that can be removed when they are not needed, it is necessary to guarantee that any state they had is kept. Cloud DReAM can use the tools provided by the cloud platform in order to coordinate the state transfer between the virtual machines and, if needed, the persistent storage of the game information.

The third component of the system is the cloud manager. This component is the direct responsible for the operations performed on the cloud platform. The decisions related with scaling and load balancing are managed by this component. In order to perform its function, the cloud manager receives information from clients and servers. It also provides them with information they need to operate. Fig. 3.2 illustrates the role of the cloud manager as a component that oversees what is happening on the system.

All the actions of our system are performed in the background, and need to be as efficient as possible, in order to maintain the performance and enjoyment of the game. Any effort to scale and optimize game resources is not useful if the playability is compromised by that process.

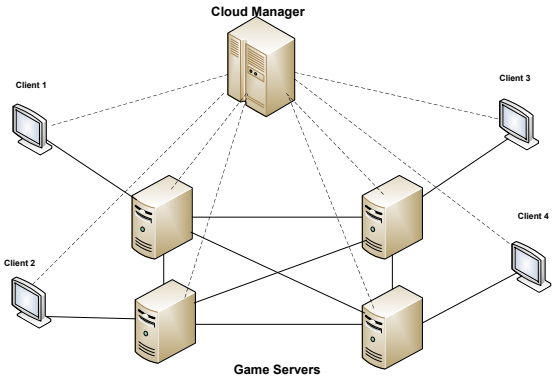


Figure 3.2: Representation of the interaction between the 3 components of the system

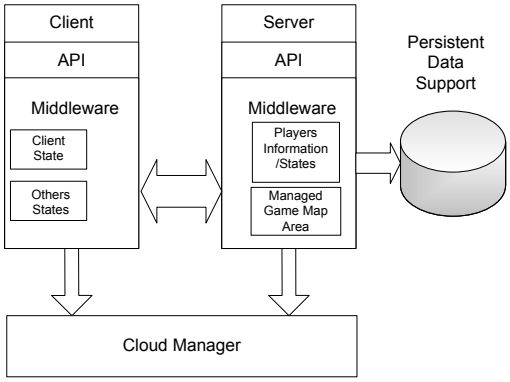


Figure 3.3: System components architectural view

3.3 Cloud DReAM

The Cloud DReAM is intended to be deployed into a cloud environment. It uses concepts such as interest management and load balancing to perform its functions. It is divided into three major components, the clients, the servers and the cloud manager. This section describes all the architectural details of the system.

3.3.1 System Components

In Fig. 3.3 we present a more detailed view of our components. This view provides a better insight into the details of each component and to how they work, as well as to the relations that exist between them. We now describe each one of these components in detail.

Clients

Clients are connected to a single server at each moment. When they join the network they are assigned to a given server and remain connected to that server throughout the execution of the game, until their

avatar moves to a position located on an area of the map managed by a different server, in which case the player is transferred to the new server. The management and execution of the interactions between a server and its clients is performed by the server.
Each client keeps a local copy of its state, and of the states of other avatars on its AOI. Clients exchange messages with their designated server in order to receive and send updates. The other possible type of message exchanged is the command to change the current designated server to a different one.
The clients simply send any state updates from themselves to the server, and receive updates related to other players states when the server sends them.

Servers

Servers are responsible for running the game logic, as well as performing all the operations related with consistency and load balance. Each server has an area of the game map over which it is responsible. Players located inside this area are responsibility of the server. The management of which area belongs to each server is responsibility of the cloud manager. When a client connects to the game, it is connected to the server that is responsible for the area where the avatar is in that moment. The server is then responsible for managing the client connection, redirecting it to a different server when it is necessary, as described in section 3.3.7
The servers are organized in a topology similar to P2P. What this means is that servers act as peers between themselves, and exchange information. They do not need to be aware of every other server, but they need to be aware of the servers managing the neighbouring areas of the map. Servers can be added in two different moments, at the beginning of the game or during the execution of the game, in order to share the load of overloaded servers.
Servers enforce the consistency of the system. They are the ones enforcing the interest management techniques that are explained in section 3.3.2. They also need to monitor themselves in order to determine when they are overloaded. A threshold is defined for the processor usage. When this threshold is reached the server is considered overloaded and requires assistance. To do this, an overloaded server notifies the cloud manager to that fact, so that it can take the appropriate measures. An analogous process applies when a server finds itself underloaded.

Cloud manager

The cloud manager, as the name implies, is responsible for the management of the cloud infrastructure. It is responsible for monitoring the server status and decide when it is time to add more servers. Servers contact the cloud manager when they are overloaded in order to request the creation of new instances to share the load. The cloud manager has the tools to create the new instances and to designate the area of the game map over which the new instances are responsible.
The cloud manager knows which server is responsible for each area. When a client connects to the game,

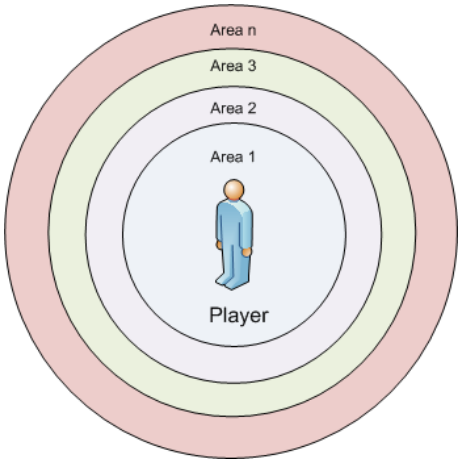


Figure 3.4: Consistency areas around a player's avatar

it first connects to the cloud manager. Since the cloud manager knows the region over which each server is responsible, it redirects the client to the server who currently manages the position of the player's avatar. This means that the cloud manager is also effectively working as a sort of gateway into the system. This approach might not be desirable when it comes to scalability as the cloud manager can become a bottleneck, but for now we only use one of these components.

When an existing server finds itself to be overloaded, it notifies the cloud manager. The cloud manager then commands the cloud infrastructure to launch new instances of the server in order to address the excess of load on the requesting server. The new server is given an area over which it is responsible. Any necessary data such as players states and other relevant information is replicated to this new instance. Finally the player's clients that are on the area of this new server are redirected in order to communicate with the correct server.

The opposite happens when a server finds itself to be underloaded. After an area being split between two servers, there can be the case where the load in those two servers does not justify the existence of both of them. The cloud manager is responsible for the termination of the excessive resources, by commanding the cloud to terminate those instances.

3.3.2 Interest Management

To help us with keeping game performance despite the operations performed by Cloud DReAM, we are using an interest management method on the system. As it was discussed in section 2.2, interest management allows us to reduce the amount of bandwidth that is required for the several game components to communicate. The interest management approach that Cloud DReAM uses is VFC [34]. As previously described, VFC provides several concentric consistency zones centered on the pivot. The consistency gets monotonously smaller as we move away from the pivot. This means that the closer an object is to the pivot, the higher its consistency will be when compared to objects that are located farther away from the pivot. Fig. 3.4 illustrates the VFC concept. Each zone is defined by a consistency array. This array is three-dimensional and each of its dimensions represents a different constraint to the replica divergence:

- **Time**: Specifies the maximum time in seconds, that a replica can spend without being updated.
- **Sequence**: Specifies the maximum number of updates that a replica can ignore before it needs to be updated.
- **Value**: Specifies the maximum difference between the content of the replica and the original object. This value is specified as a percentage.

Any of the previous values can be ignored by setting it to an infinite value. For example an array defined by $k = [1, \infty, 50]$, stipulates that an object replica in this area is at most 1 second behind the master object that it represents, and differs from it no more than 50%. The value of sequence is being ignored in this case.

The VFC system used on the Cloud DReAM relies on a Client-Server architecture, where the server is the most consistent point of the system. It is also the server responsibility to manage the consistency of the clients.

Each client has a local pool of replicated objects. These objects are replicas from the main objects that are hosted on the server. Clients can freely read and change their local replicas. Any change to the local replicas is propagated to the server. The server is responsible for the management and propagation of the updates, having into account each clients view.

The updates are periodic, although they are independent from each other, since there is no synchronism between clients and server. Any update received from clients is not immediately applied to the main replica. It is only applied when the consistency round is over. Each round, the server performs the consistency management and determine which of the clients need to receive updates. After determining which clients need updates, and what are the updates that should be sent, the server sends them the information.

3.3.3 Load Balancing / Player Distribution

For load balancing we use an algorithm [26] that divides the game world among the available servers. Since bandwidth usage is always a concern in this type of game, this algorithm is already considering the usage of the VFC interest management approach to minimize this concern.

The world is divided into rectangular shapes, and each shape is assigned to a different server among the available ones, as illustrated in Fig. 3.5. This division is not static and is bound to change as the game is in progress. When a client connects to the game, it is assigned to one of the existing servers, depending on the positions of his avatar on the game world. The user is not aware of this division. When the avatar changes its position to an area managed by a different server, the client is redirected to the corresponding server, and the user does not notice this transition.

As mentioned in section 3.3.1, servers are organized in a structure where communication is analogous to P2P. A server is only aware of a subset of all the existing servers at any given time. This partial view contains the server's direct neighbours (in terms of the area of the map that each server is responsible for). To support the VFC functionalities, servers need to perform two actions. The first one is the subscription

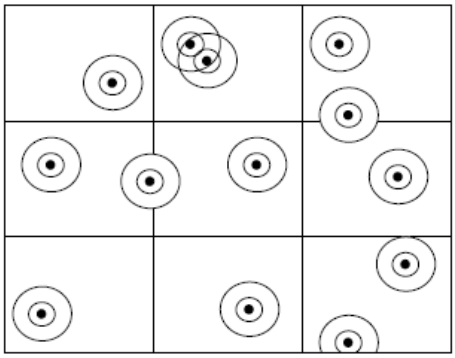


Figure 3.5: World map divided with players avatars represented

to other servers (peers) where the areas of interest of their players can overlap. In the Cloud DReAM this subscription is done by registering the server with its peers when it first connects. The second action that is required is the object subscription. Servers need to determine which objects that belong to a different server, should be subscribed for updates. In the Cloud DReAM system, every object currently present on the peers pool is migrated to the connecting server upon registering. This creates two different sets of objects. The first ones correspond to the objects owned by the server, which represent players in the server's current area of the map. The second ones are subscribed objects, which correspond to players in other servers area of responsibility, but whose state updates are important in order to enforce VFC. The updates are sent to clients according to the values defined by the VFC consistency vector. It is important to note that servers do not perform any VFC operations between themselves. Each server enforces VFC for the clients currently connected to him. It does however use the information provided by other servers to do this. Using this strategy does not optimize the communication that the servers need to perform between each other, but ensures that every server has the most recent values to work with.

Servers can join and leave the network at any given time depending on the game's current needs. Most servers join at the beginning of the game, but there can be a case when a server needs to join while the game is running. This mainly happens in order to reduce the load on another server, so that playability is always guaranteed.

When a server is added to the network, it registers itself with the cloud manager. The cloud manager sends him the area over which it is responsible. This area originated from the division of the area of an overloaded server into two areas. After this the cloud manager informs the neighbour servers of this change. Fig. 3.6 shows how the partitions can work. Each partition is managed by a different server. As we see partition 2 and 4 are bigger than the other remaining ones. In order to mitigate the problem of different sized areas, we use an algorithm to split areas that is described in section 3.3.4

3.3.4 Map Division

As we have described previously, the game map is divided between servers. This division is performed when the need for it arises. This can happen when a server is overloaded and needs to share some load. We are assuming the map base structure is always a square (even though the structures that can

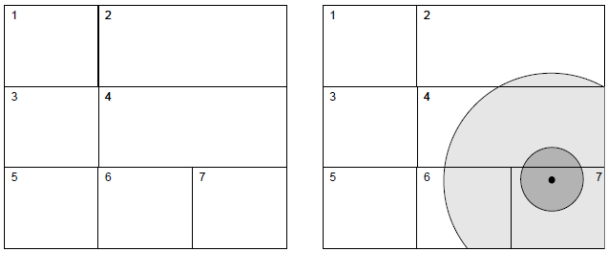


Figure 3.6: Example of different partition sizes and a player's area of interest spreading across them

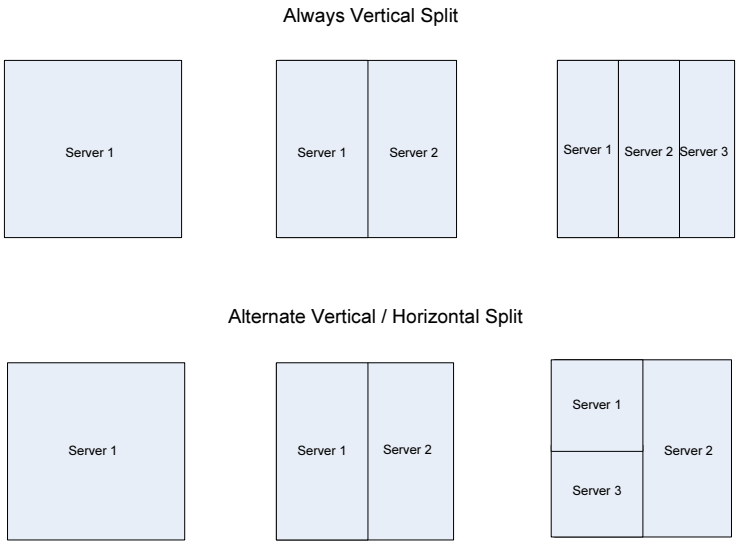


Figure 3.7: Two alternative area split methods

be created inside the map can limit it to different shapes), so we are using a linear division scheme as described earlier in section 3.3.3. When an area needs to be divided, it is split in half into two equal zones. The division of an area alternates between vertical and horizontal division. This is done so that we do not end up with a large number of rectangular areas instead of smaller square areas. To better illustrate this methodology, we present an example in Fig. 3.7. The first alternative shows a vertical only division, where in a limit situation we would have a series of rectangular areas that would cover the full length of the map while having a very small width. The second option shows an alternation between vertical and horizontal split. In this case, a record is kept of whether the current area was originated in a vertical or an horizontal split. When the area is to be split again we use that information to split it vertically if it was last split horizontally, or horizontally if was last split vertically.

3.3.5 Client Connection

When a client connects to a game session, apart from the initial connection with the cloud manager, two other connections are established. The first connection is created between the game client and the game server. The second one is between the VFC client and the VFC server. In order to know to which server

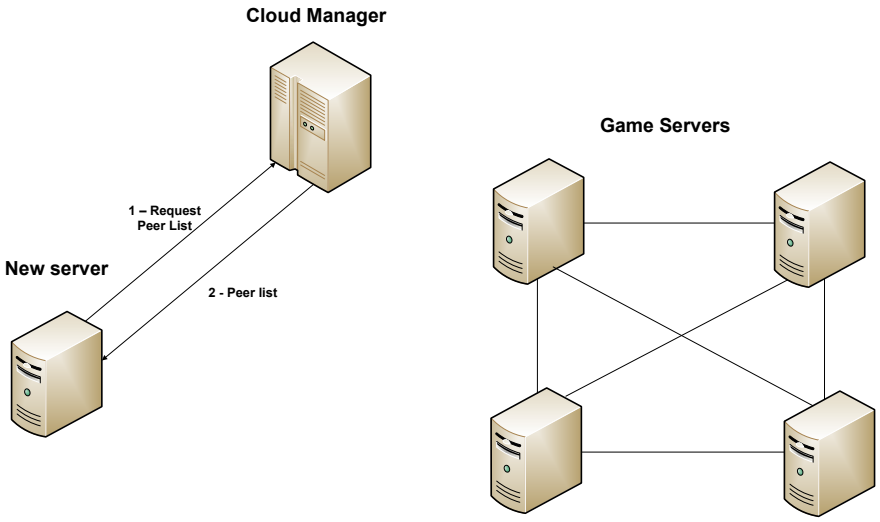


Figure 3.8: Server requesting peer list from the cloud manager

it should connect, the game client contacts the cloud manager upon connection. The cloud manager knows all the existing servers, and knows which areas they manage. The reply from the cloud manager to a connection request is the address of the server to which the client should connect. This choice of server is based on the area of the map where the client's avatar is on when it connects. Additionally a client is issued a unique client identifier, that is provided by the server when they connect. this identifier enables the distinction of the client throughout the game session.

3.3.6 Server Connection

Servers are Virtual Machine instances that are launched by the cloud manager when necessary. When a server starts running, it attempts to register itself with the cloud manager. When successfully registered, the server is issued an unique identifier by the cloud manager, that represents the server while it is active. The server also requests from the cloud manager an area of the map over which it is responsible. Finally, it requests a list of direct peers from the cloud manager as illustrated by Fig 3.8. We can see in the illustration that message one corresponds to the request for the peer server list. Message two represents the reply from the cloud manager containing the peer servers list. Every server that manages an area that is adjacent to the current connecting server's area, is considered a direct peer.

After successfully registering with the cloud manager and receiving the direct peer list, the server attempts to register with its peers as illustrated in Fig. 3.9. On the illustrated example, only two of the servers are considered peers, so the new server only contacts those two servers. On connection with a peer, a server identifies itself with its unique id. Both peers register each others addresses in order to communicate. The peer that requested the connection then requests a copy of the objects that are currently present in the pool of its peer, and add them to its own pool to be used by VFC. These objects

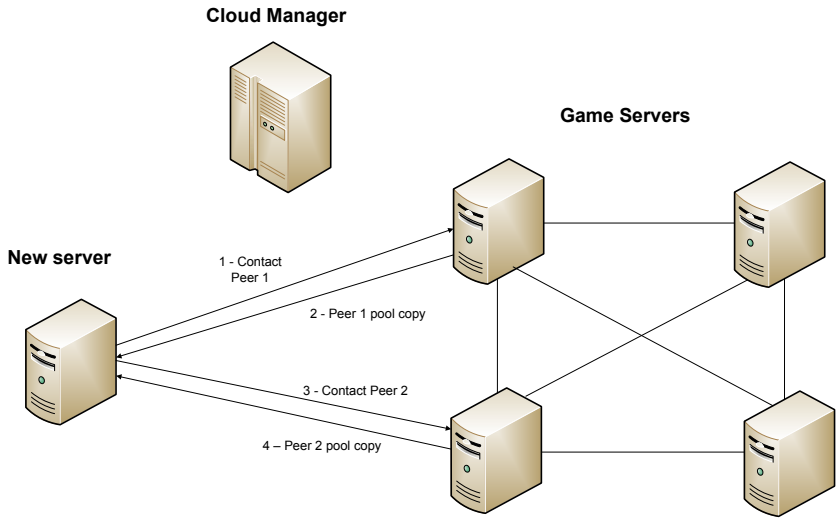


Figure 3.9: Server contacting the two peer servers

are marked with their owning server ID. This process is repeated for each of the peers in the list that was provided by the cloud manager.

3.3.7 Client Redirection

Clients may not stay connected to the same server throughout a game session. Since servers divide the game map among themselves, a client is bound to require a server change as he moves from one area of the map to another. As previously said, this area division is not evident to the player, since as far as he is concerned the game is a single large map over which he can move.

When a server performs an update round to the objects in its pool, it also checks if the objects that it owns are still currently within the limits of its own area. If it finds that an object that it owns is outside the area he manages, it notifies the cloud manager to that fact. The cloud manager acknowledges this fact and store the information. The client migration is not immediate. If a client is found to be outside of the current server area of responsibility for more than a predetermined number of update rounds, the cloud manager then instructs the client to change servers as illustrated by Fig. 3.10. This is done to avoid the problems that can rise in the border between two game zones. A client can be walking in the border between two zones and occasionally move to the neighbour area, and immediately back to the original one. This can occur a few times in a row, and would trigger a series of unjustified migrations. On our system, only after the client is found to have consistently changed to a different area, it is migrated. An example of this situation is illustrated in Fig 3.11, where a player moves several times across the border and ends in the original server. In this case the client is not migrated between servers

When a migration is triggered, the client connects to the new server. After the connection is successful,

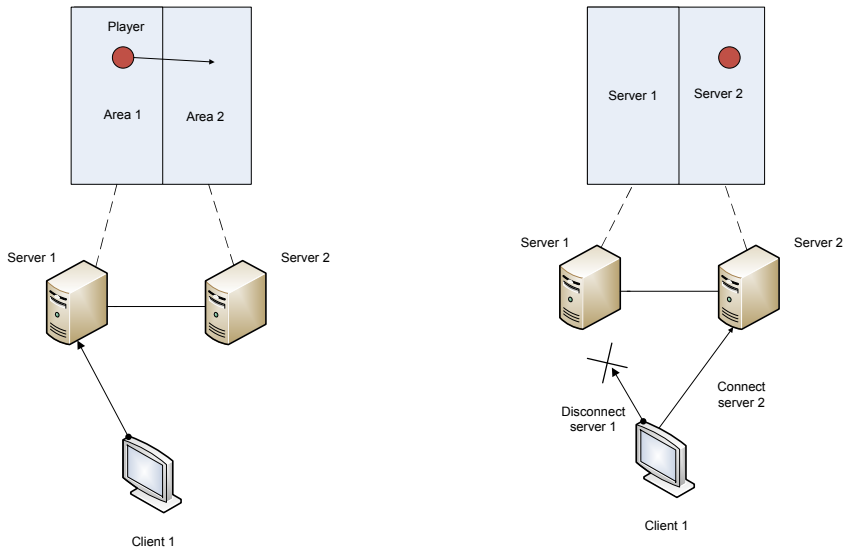


Figure 3.10: Effective migration of a client

the client disconnects from the old server, and changes the ownership of object that represents himself to the ID of the new server. This change is propagated to the other servers by the new server during the next update round.

3.3.8 Communication Model

We consider two different types of communication options on the Cloud DReAM. The first one corresponds to the periodic updates messages (either between servers or between servers and clients). Their periodic nature poses the problem of excessive use of bandwidth. This problem is addressed by the interest management techniques and by the protocols used in the transport of the information. This messages are not guaranteed to arrive, but since they are periodic, it is not problematic to loose one message. An example of a message of this type are the status updates between clients and servers.

The second type of communication considered are the messages with non-periodic nature, and with a necessity for reliability of the transmission. This type of messages are not filtered by the VFC technique and need to be guaranteed to arrive. An example of an event that could use such a message in a game is the firing of a shot by a player. This message is only sent when a player performs this action, and it is important that it is not lost, or the enjoyability of the game will not be maintained. To address this issue, Cloud DReAM provides an abstraction to this messages which are known as events. Events are guaranteed to arrive to their destination. When an event is generated, it is replicated among peer servers so that every client that might be affected by it receives the information. Since events are more bandwidth consuming, the servers only replicate events to the peers that contain players that might be interested on this information.

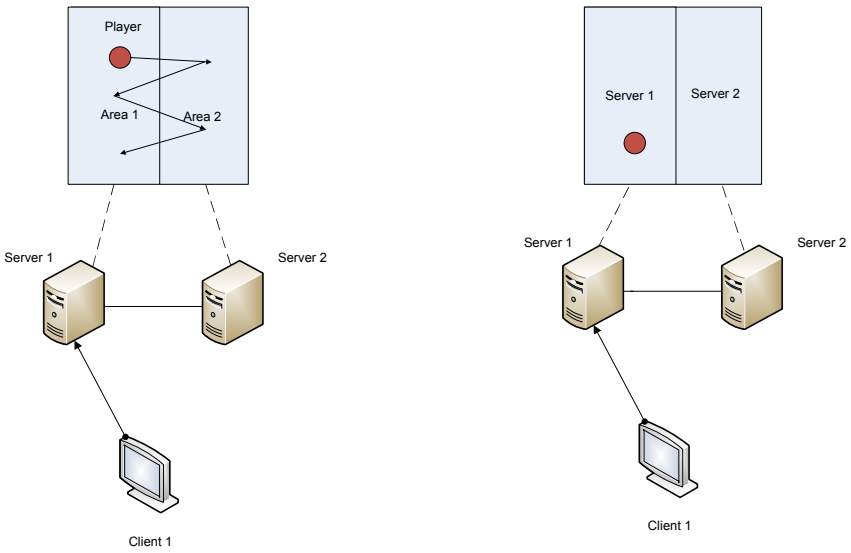


Figure 3.11: Client moving back and forth in the border and not being migrated

3.3.9 Scaling Algorithm

The scaling algorithm is split between the servers and the cloud manager. We chose to use a simple algorithm for the first version of the system. The servers monitor themselves in regular time intervals. This monitoring happens in 30 second intervals. The current version of the algorithm uses the processor current usage as a metric for the scaling to happen. When the server CPU is found to be above a threshold value for more than a given number of monitoring rounds, the scaling operation is triggered. Both the threshold value and the number of monitoring rounds that should be tolerated before scaling, can be easily configured and are not fixed.

When the decision to scale is made, the requesting server contacts the cloud manager, informing him of that fact. The cloud manager knows how many instances of the servers are running, as well as the remaining capacity of the cloud to launch more instances. If the total capacity of the cloud has not yet been reached, the cloud manager orders the launch of a new instance.

The downscaling operation is very similar. During the server monitoring rounds, if it finds itself to be under loaded for more than a given amount of rounds, the server informs the cloud manager to that fact. In this situation the cloud manager is aware of the current load on the remaining machines. It checks if the remaining machines will become overloaded as a result of this downscale operation. If the answer is positive, the machine is not terminated, as it is very likely that after its termination, a new instance would have to be immediately launched. If the cloud manager believes that there is a small risk of overloading the remaining machines, then the requesting machine can be safely terminated.

Before any machine can be terminated, the clients that are currently connected to that server, need to

be informed that they have to redirect to another server. The new server to where they should connect corresponds to the one that takes control over the area of the terminated server. The machine is not immediately terminated, since there may be some information there that is not replicated yet. Any client data that needs to be migrated to the peer servers, is migrated. When this process is complete, the machine instance can be safely terminated. In Fig. 3.12 we present a flowchart that summarizes this algorithm.

3.4 Summary

In this section we described in detail the architecture of our system. We started with a general overview on the system and its most relevant aspects. Then, we described the working details of our system components. We also described the various algorithms that we use to perform roles such as interest management and load balancing. We finalized our architectural description by detailing our scaling algorithm.

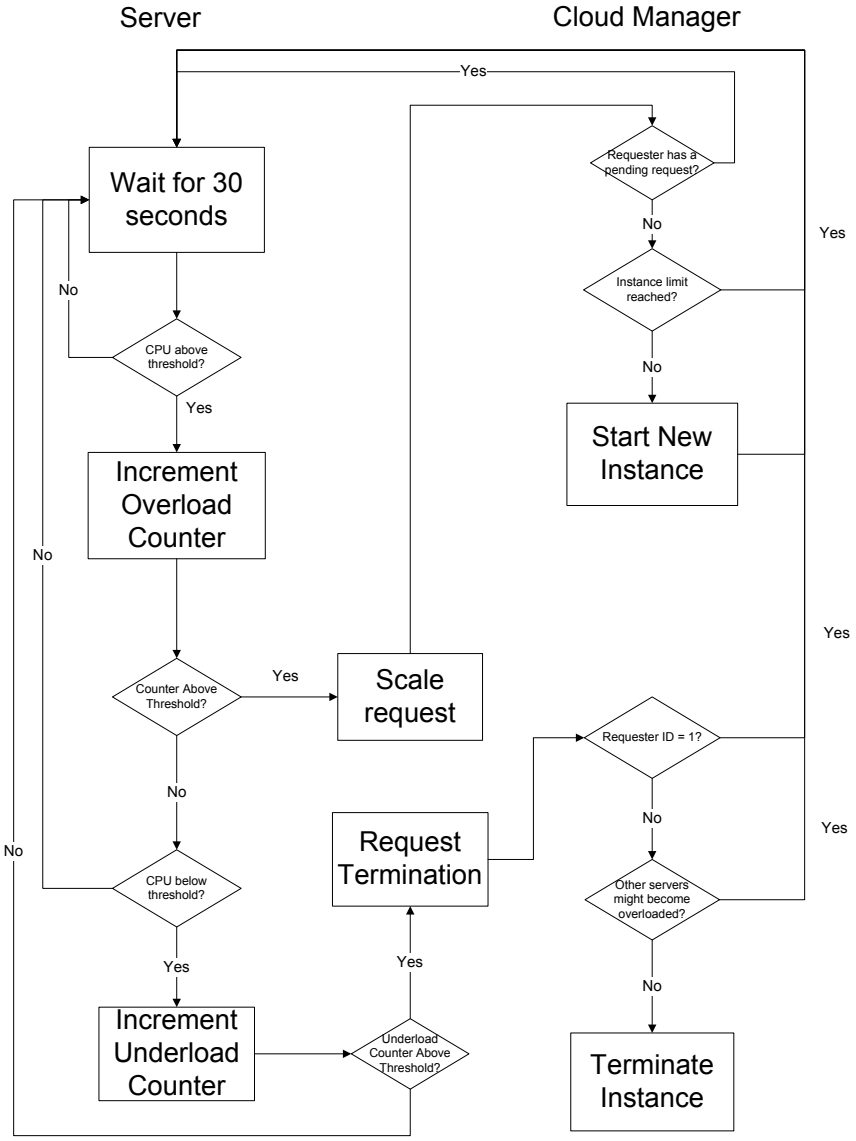


Figure 3.12: Flow chart of the scaling algorithm

4 Implementation

This chapter describes the implementation details of our system. We first present the development environment and why it was chosen. We then move on to describe the APIs that were changed and created to achieve our goals. We also present the data structures that we created to support our cloud environment.

4.1 Game Choice

The game that was chosen for this work was Cube 2: Sauerbraten. It is an open source first person shooter (FPS), which enjoys some popularity online. The game is developed in C++, and can easily interact with a C# application, in which the Cloud DReAM middleware was developed. The game maps have big dimensions, and there are a number of extra maps developed by the community. We also have a previous implementation of the VFC interest management technique for this game. This allows us to benefit from the performance improvements provided by this technique in our implementation.

4.2 Development Environment

The Cube 2 game is implemented in C++ language. The implementation we are using for VFC (VFC4FPS) was developed as a library implemented in C# using Microsoft .NET platform[1]. The game communicates with the VFC4FPS library through an API that can be invoked directly from the native game code. We have decided to use the same framework in our system. Since we are extending the VFC4FPS implementation to work in a cloud environment, we think it would not make sense to change this approach. The .NET platform provides all the resources we need to develop our system, such as the remoting features that allow an easy communication between every components.

4.3 Eucalyptus cloud

Since the purpose of this work is not the development of a dedicated cloud platform, we use an already existing and tested cloud solution. We use the Eucalyptus cloud platform to serve as a base for our work. Eucalyptus is a widely used software platform for private clouds, providing an infrastructure as a service (IaaS) type of cloud.

[1] http://www.microsoft.com/net/

We chose to use this platform for a number of reasons. First of all it is an open-source license platform, which allows us to change it if need be, and can be used without any further costs. The platform is organized in a modular fashion, with all of its components very well-defined, which also allows for further customization in case we need to switch one of the original components with a different one. These components can also be installed in different machines as the users sees best fit.

Eucalyptus is also compatible with the API of the Amazon's EC2 and S3[2] services. This is important to the persistence aspect of the system. The EC2 is a very widely used cloud computing platform, by assuring compatibility with these two API's, Eucalyptus can interact with both services to further expand the features it provides. This allows us to create a system that has a wider range of compatibility with other cloud platforms that use these two common API's. This is an important feature, since our aim is a middleware system that can support different games on different cloud platforms. Furthermore, Eucalyptus is classified as a Hybrid cloud, it can draw resources from private clouds and again thanks to the compatibility with the EC2 API, it can also draw further resources from public clouds, which increases the capacity of the system. This aspect helps the system to have a bigger scaling potential.

Finally it allows for the installation of different types of operating systems to run on top of the cloud, which makes this approach rather flexible and extensible. For our game we needed to use a Windows system in order to run the game server, so this is an important feature. This platform is also hypervisor agnostic.

By using Eucalyptus we have direct control over the resources and their management. We can control to which machine the clients connect, which is essential in managing the game status and logic. This is not the case with every cloud platform, so it makes sense to choose Eucalyptus for our implementation prototype.

For all that was stated and due to the fact that Eucalyptus allows for a fine control of the cloud actions through its tools, we feel that this platform is the ideal one to use for our solution.

4.4 Game Conversion

As we have previously mentioned, Cube 2 is not a multiple server game. In its native implementation, one server manages the entire game map, and all the clients connected to that game on any given moment as illustrated by Fig. 4.1. In order to use this game for our cloud system, we had to proceed with its conversion into a multiple server game.

There were multiple steps on this process. We had to perform changes to the game logic on both client and server components of the original game. Fig. 4.2 illustrates the modified version of the game, with all the differences it has with regard to the original one. We now describe those changes in more detail.

[2] http://aws.amazon.com/pt/s3/

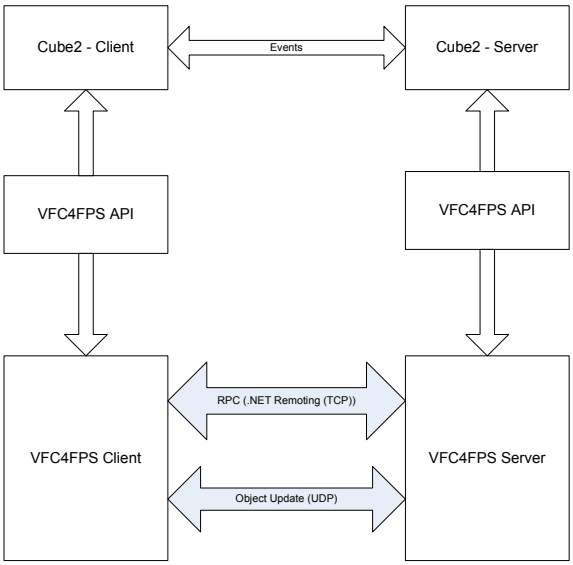


Figure 4.1: Implementation of the Cube2 game with the VFC4FPS system

4.4.1 Server Conversion

In order for the server to work on a cloud environment we converted the client ID's to a unique ID for each client. On the original version of the game, each client was issued a number by the server, that identified the client in the game session. We have kept the unique number approach for client identification, but we have made it centralized at the cloud manager. Whenever a client connects to the game, the server checks if he is a new client or an existing one. If the connection belongs to a new client, the server requests a new client number from the cloud manager using the remoting features of the .NET platform. The server then creates the local object with the given ID and reports the ID to the client so that it knows its own number. These ID's are unique across servers, so no two clients have the same ID even if they connect to different servers. The cloud manager maintains the register of this unique number within a simple variable that is incremented with every request. Since the remoting library is multi-threaded, the cloud manager also applies locking mechanisms to assure that no race conditions are created on the access to this variable.

In the original solution and on the VFC solution, the servers only sent state updates to their connected clients. On our solution that would not be possible. The servers were adapted so that when they receive an update from one of their clients, they immediately send that update to their peer servers. This extra communication between servers is performed by the Cloud DReAM server component. The approach used is similar to that used by clients to send updates to servers, using UDP. In fact the same UDP socket that is used to receive the client updates, is used for the status updates between servers. The immediate update of status between servers is done so that peer servers have the most recent state for clients they do not own, in order to perform the VFC updates to their own clients. VFC is managed by each server independently and is not centralized in any way.

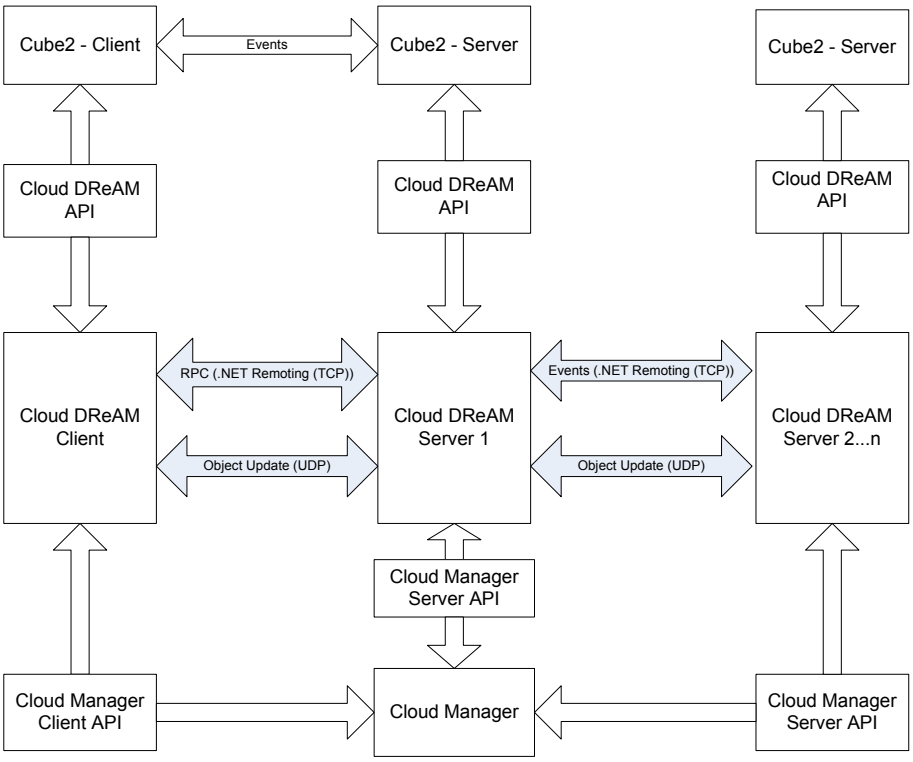


Figure 4.2: Implementation of the Cloud DReAM version of the game

Servers also need to replicate events across each other. The VFC4FPS did not address this issue, as it was not its main focus, but we need to address it in order to make the game work in the cloud environment. When an event takes place, a server processes it as the original game solution would process it. It also adds it to the Cloud DReAM server list of events to send to peer servers. If only a limited number of clients are involved in an event, the server identifies to which servers those clients belong, and sends the event to those servers. If it is an event that affects every client on neighbour servers, it is replicated to every other server on the neighbours list. This replication is performed using the .NET remoting features. This uses the TCP protocol, which has a bigger communication overhead. We have opted to accept this overhead in order to ensure that no events are lost, which is something that may happen with UDP. Furthermore, events are not as frequent as the clients state updates, so the impact of the TCP overhead is contained. Finally if the event only concerns clients connected to the current server, the event is not replicated to the peer servers. When servers during their update cycle process their local events, they request the list of pending remote events from the Cloud DReAM server and process those as well in order to update the global state of the game. After the list request by the game server, the Cloud DReAM server resets the remote events list and waits for new events.

4.4.2 Client Conversion

The main change that was performed to clients was the addition of support for other clients that are connected to different servers. As we have previously explained, clients can be connected to different servers, and still be able to see each other and interact with each other. Part of this problem was addressed by replicating events in the servers, but it is also necessary for clients in different servers to see each other. To solve this, clients process not only the other clients information that they receive from the game server, but also processes information from clients that are connected to different servers. Since the state of those clients is replicated between servers, the Cloud DReAM server is able to provide its own clients with information from other clients that are not connected to the same server. Clients add the information about this objects to their local pool, so that the game logic can access them and make them visible to the player. The difference on this process relative to the original implementation is how the addition is performed. On the original solution, when a client connects, the server sends every other client that is connected a message to inform them of the new client connection. On our solution when a client requests the other clients information from the Cloud DReAM client, it also receives the information about the server to which they are currently connected. When a client from a different server is detected, we generate a dummy message similar to the one the server would send upon connection of a new client. The client logic processes it and creates the corresponding object just like it would do if there was only one server.

4.5 Application Programming Interface

The Application programming interfaces are the bridge between the the game and the Cloud DReAM components. The VFC4FPS already has some of these interfaces implemented and working. Those interfaces were extended in order to provide the extra functionalities that are required to properly function in a cloud environment. A new component was also introduced, known as Cloud Manager, that provides an interface for both clients and servers to perform a series of cloud related operations.

We now present the relevant changes that were performed to those interfaces in order to achieve a multi-server functionality that can be deployed in a cloud infrastructure, as well as the interface for the cloud manager component.

4.5.1 Cloud DReAM Client API

This section describes the new methods that were added to the existing client interface, to support the cloud operation.

Redirect:This method is used to check if the client needs to be redirect. It contacts the cloud manager in order to verify if the current client's ID is flagged for redirection. The method returns a *boolean* value of true if the client needs to be redirected, and false otherwise.

RedirOnProcess: This method is invoked by the client in order to check if it is currently being redirected. Clients keep no information regarding a redirection process. This method is needed to filter messages that are sent by the server during a connection process. Some of these messages only make sense in the context of a first time connection. An example of such a message is the one that sends the client its client ID. When a client is being redirected this special messages are processed differently according to the result of the invocation of this method. The method receives as an argument the client's ID, and returns a *boolean* value that is true if the client is being redirected, and false otherwise.

RedirectToServer: This method is used by the client to start a redirection process. It receives as an argument the server port and the client's port. This method generates a connection message to the new server. After the connection with the new server is confirmed, it disconnects the client from the old server.

4.5.2 Cloud DReAM Server API

This section describes the new methods that were added to the existing server interface, to support the cloud operation.

NewClientId: The clients unique ID is provided by this method. When a client connects, if it is connecting for the first time, this method is invoked to obtain a new ID. This method contacts the cloud manager, which provides the unique identifier. The server uses this identifier to create the local object representing the client.

NewEvent: This method is used to add a new event to the Cloud DReAM server pending events list. The events added to this list are candidates to being replicated to the peer servers. This method receives as arguments the target of the event, which is a unique client ID, and receives the object that represents the event itself.

AddHit: This method is similar to the previous one, but in this case is used for the hits. The method receives the object that represents the hit as an argument.

GetPendingEvents: This method is invoked during the update rounds of the game server. When this method is invoked, it returns the list of all the events received from peer servers that have not been processed yet. After returning the list, the method clears the pending events list on the Cloud DReAM server.

GetPendingHits: This method returns the list of pending hits on the Cloud DReAM server. After being invoked and returning the hit list, it clears the pending hits list of the Cloud DReAM server.

4.5.3 Cloud Manager API

This section describes the methods provided by the Cloud Manager API, to the server and client in order to support cloud operation.

Server Methods

ServerJoin: This method is used by the game servers. When a game server is started in the cloud (a new instance is created and the game server starts running), it contacts the cloud manager in order to register itself by invoking the *ServerJoin* method. This method generates a new *serverId* for the connecting server and add the server, and corresponding IP address, to the cloud manager list of active servers. The newly created *serverId* is returned to the invoking server.

ServerLeave: This method is the opposite of the previous method. When a server is about to be terminated, it informs the cloud manager that he is about to be disconnected. The cloud manager marks all the clients currently connected to that server as being in need for a redirection. Finally the cloud manager attributes the area managed by that server to another server, and removes the server information from its active servers list.

LaunchInstance: This method is used to create a new server instance in the cloud. A server that finds itself overloaded, invokes this method in order to request the cloud manager for help. When this method is invoked, the cloud manager checks if it is possible to launch a new server instance (taking into account the available cloud resources), and if it is possible starts the launching process. This process consists of connecting to the machine hosting the cloud, using the SSH protocol and invoking the corresponding command to launch a new instance of the specified type to assist the overloaded server. The last task of this method is to order the split of the area managed by the overloaded server. The area is split into two different areas, and one of them is immediately sent to the requesting server. The second area is placed on a queue waiting for the server that was just launched to fully start and receive the area information.

ServerArea: Connecting servers invoke this method after they finish their startup process. This method returns to the server the area over which it is responsible.

ComputeNeighbours: When a new server connects, it needs to know which servers are managing areas that are neighbours to its own area. The method receives the unique identification for the server, and checks its database for areas that are neighbours to the area currently registered for the given identification. When it finishes, the method returns a *Dictionary* structure with the neighbour server ID's and respective IP addresses.

ServerReady: After launching a new server, it is not immediately ready to receive incoming connections. The server first needs to receive the information that is relevant to him, taking into account the area that it is managing. When this warm-up process is completed, the server invokes the *ServerReady* method, which receives a *serverId* as an argument. This invocation, tells the cloud manager that the server is ready to perform its task, and can now start to receive connections from players.

GetClientId: If a connecting client is new to the game, it needs to have and ID. When a server receives a request from a new client to connect to the game, the server invokes *GetClientId* to request the cloud manager for a new unique id for the client. The cloud manager generates the new ID and returns it to the server, which is responsible for forwarding it to the client.

ToBeRedirected: Throughout the game, clients move from one server to the other. Servers monitor the clients currently connected to them to see if they are still within their area. When they find a client

outside of their area of responsibility, they notify the cloud manager using the *ToBeRedirected* method. This method receives a list of *clientId* corresponding to the clients that were outside of the responsibility area. The cloud manager keeps track of this notifications, and after he receives a predetermined number of notifications for the same client, it is marked for redirection.

Client Methods

ClientConnection: This method is used by the game client to connect to a server. Before the client connects to the server, it contacts the cloud manager using the *ClientConnection* method. The method receives the client ID as an argument. The return of this method is the address of the server to which the client should connect.

ClientNeedRedir: Clients often check if they are marked for redirection. The method *ClientNeedRedir*, receives a *clientId* as an argument, and checks if that ID is currently marked for redirection. The return value of this method is a *boolean* value that represents the need for the client to start a redirection process.

ClientRedirection: This method represents the start of the redirection process. If the result of the *ClientNeedRedir* method is true, the client invokes *ClientRedirection* to start the redirection. This method receives the *clientId* and the position of the player in the game world. Based on the player's position on the game world, the cloud manager determines the address of the server to which the client should be redirected. The return of the method corresponds to the IP address of the new server.

4.6 Data Structures

In order to support the multi-server functionality required for the cloud environment, we had to create some new data structures, and extend existing ones. In Fig. 4.3 we illustrate where this structures are used in our system. In this section we describe what was changed in the existing data structures, as well as the new ones that were created.

4.6.1 *CubeOriObj*

This class implements the *IOrientableObject* interface, and represents the player's avatar status. The original VFC4FPS implementation only had information relating to the player's position. On our implementation, since we have to consider events as well (which VFC4FPS did not consider), we need to have some extra information regarding the player. This extra information is needed since not all players are connected to the same server. The information that was previously obtainable directly through the game server, now has to be replicated in a different way.

We extended the *CubeOriObj* to have some additional fields. The fields are the player's health information (*lifesequence, health, armour, armourtype*). This information is important for the client program to

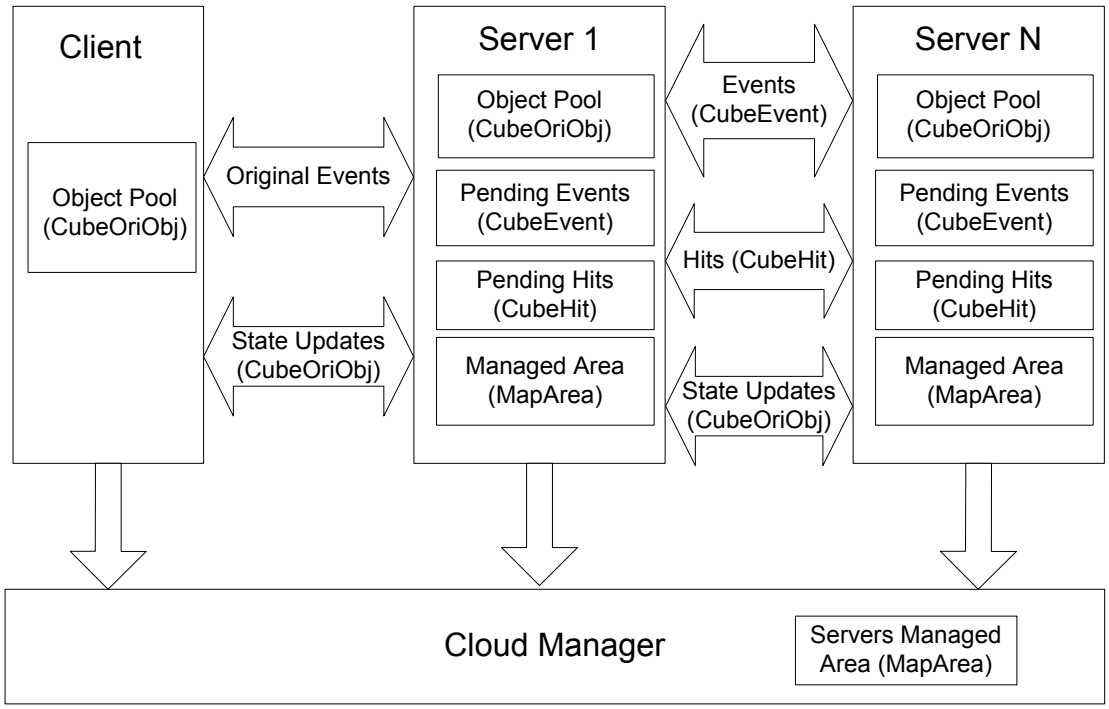


Figure 4.3: Data structures distribution across the Cloud DReAM system

know which of the objects are supposed to be displayed on screen. This type of object is used by both servers and clients as illustrated in Fig. 4.3.

4.6.2 *CubeEvent*

The *CubeEvent* class represents the various events that can occur on the original game. Event such as a shot fired by an avatar, an avatar respawn, can be represented by this type of object. This class was created due to the previously mentioned need for an event replication system. When an event is created in a server, it creates the corresponding *CubeEvent* object to represent that event. The created object is replicated to the peer servers to which it is relevant as illustrated in Fig. 4.3.

This object can represent various types of events, that shared most of the information that needed to be replicated. This object contains the player responsible for the creation of the event (*cliId*), the identification of the event type (*type*) and the event id generated by the game (*id*). It also has a temporal sequence number that is maintained by the servers which is marked in the event (*millis*). In the specific case of a shot event, the object keeps the coordinates of origin of the shot (*fromX, fromY, fromZ*), and destination of the shot (*toX, toY, toz*). This information about the shot is needed by the VFC system to decide which shots are relevant to each client across different servers.

4.6.3 *CubeHit*

The *CubeHit* object represents a hit in the game. A hit can be from a shot or an explosion that occurs in the game world, and affects the players health. An hit is also treated as an event by the game. We have decided to make it a different type of object, because the information it needs to share is very different from the information shared by the remaining events.
This object contains information about the player that generated the hit, and the player affected by the hit (*clientId, targetId*). It contains information about the type of gun used to generate the hit (*gun*), information about any type of power up to the weapon damage (*quadmillis*), the distance of the hit (*dist*) and information concerning graphic effects, such as the number of rays generated by a shot (*rays*).

4.6.4 *MapArea*

The division of the game map among servers is represented by the *MapArea* class. Each server has one instance of the class that represents the zones he is currently responsible for as illustrated in Fig. 4.3. The cloud manager also knows which are belongs to each server, as it keeps a list of *MapArea* associated with the server ID. Since the map division is only made considering two dimensions, this class contains two *Point* objects, which are two dimensional. The first *Point* represents the starting point of the area (*start*) and the second one represents the end of the area (*end*). Since we are only considering square or rectangular areas on this implementation, this representation is enough to describe the areas.
This object also provides the methods to determine if two areas are next to each other, from now on known as neighbour areas. The method used for this calculation is *NeighbourAreas*. The implementation of this method, takes two areas, and verifies if they are next to each other in any direction.
Another important aspect of the system is the division of one area into two distinct areas. The *MapArea* class also provides a method that splits the current area into two distinct areas, known as *SplitArea*. This method can perform a vertical area split or an horizontal area split. As previously said this is done to prevent the generation of large rectangular areas with a big length that can span across the entire map area. To help perform this task, the class contains information about the orientation of the last split performed on the current area *vertical*. Depending on the value contained on this variable, the area is either divided vertically or horizontally.
Finally this class provides a method to determine if a given *Point* is contained within the area that is represented. This functionality is implemented in the method *PointBelongsToArea*.

4.7 Load Balancing Mechanisms

The load balancing is performed by player position within the game world. Every time a server receives and processes an update from a player currently connected to him, the server checks if that player is still within its area of responsibility. To do this, the server compares the position contained in the *CubeOriObj* that represents the player, with the area represented by *MapArea*. If the player is found to be outside

of the server area, the cloud manager is notified. The cloud manager registers the fact that the player is outside of the server area, but does not immediately issue the redirection order. The notification causes the cloud manager to increment a counter that represents the player. If enough consecutive requests are received relative to the same player, the cloud manager marks the player ID as in need for a redirection, on the redirection list. If the notifications received from the server are not consecutive, the cloud manager does not trigger the redirection. This enforces what was explained in section 3.3.7.

4.8 Scaling Mechanisms

Our scaling algorithm is working on two of our components, the cloud manager and the server. Each server has a monitoring thread that is sleeping for the most part of the game. This thread wakes up every thirty seconds, and inspects the current CPU usage percentage. The server checks if the value obtained is above the defined threshold value. In the positive case, the server increments a counter (*cpuAboveTresh*). When this counter reaches the defined number of rounds to tolerate before scaling, the scaling request is performed by notifying the cloud manager. The same thread also checks if a server is underloaded and uses a similar process to detect it and notify the cloud manager.
The cloud manager keeps information about the scaling operations. It keeps a list that contains the ID's of the servers that have a pending scale request. If a server has requested a scale which is still in progress, and then tries to make a second request before the first one completes, the cloud manager blocks that second request. When a scale request is received by the cloud manager, it contacts the Cloud Controller (CLC) of the eucalyptus cloud to launch a new instance. To do this, the cloud manager opens an SSH connection to the CLC machine and invokes the *euca-launch-instance* command. The execution of this command returns a unique instance ID, that identifies the instance within the eucalyptus cloud. This ID is different from the server ID that we use in our system, this ID is only used by the eucalyptus platform. The cloud manager stores this ID in a Dictionary, associated with the ID used by our system to identify the server. When an instance needs to be terminated, the cloud manager uses the stored Eucalyptus instance ID to do it. It connects to the CLC through SSH and invokes the command *euca-terminate-instance instanceId*.

4.9 Server Image

Our server application runs on a Windows environment. For the deployment of our servers in the cloud we created an image of a Windows machine where we installed our application. We used Windows 7 as our target system. This image had all the necessary components for our server to run installed, such as the .NET 4.0 platform. Eucalyptus requires some components to be installed on this image in order for it to run on the cloud. This components were installed according to the Eucalyptus administration guide, and using the tools provided with Eucalyptus.

We also performed some simple configurations to better suit our needs. We have setup a user in the machine image named eucalyptus. This user has administration privileges on the machine. For this user we have a password less login in place, so that when the machine is booted, it immediately logs in to this user's area. Finally we have configured the machine to run our server application when a user logs in, so that it is automatically launched when Eucalyptus creates a new virtual machine instance.

4.10 Summary

On this section we described the most important implementation details of our system. We detailed the process of conversion that was necessary for the Cube2 game to work with our system. We also detailed the communication interfaces between our system components, as well as the data structures that support our system's operation.

5 Evaluation

In this chapter we describe the testing process to the Cloud DReAM system. We begin with a description of how the evaluation was performed and the infrastructures used on the tests. We conclude with a qualitative evaluation of the game in order to understand if overall playability and enjoyment of the game was maintained.

5.1 Tests Performed

In order to properly evaluate the difference between the original game approach and our system, we need to perform some tests for comparison, using the original system, and two versions of our system, one using a static infrastructure and another one using a dynamic infrastructure. We designed these tests to evaluate if our system achieves our requirements of scalability, performance and usability when compared to the original game.

5.1.1 Original Game

On this test, we use the original game with the VFC4FPS system, and play a game that lasts approximately 10 minutes (which is a normal duration of a game round on this type of game). The game is played using artificial intelligence controlled players (*bots*). For this test we use 50 *bots*, and we had a warm-up time of 10 minutes where we were connecting all the *bots* to the server. After the 10 initial minutes warm-up, when we consider the game to be on stable testing conditions, we started measuring the server usage for another 10 minutes.
This test is important in order to compare the difference between server usage on the original solution, with the one we implemented on our system.

5.1.2 Cloud DReAM with static infrastructure

This test used our modified version of the game. On this version we have multiple servers running the game, sharing the game map among them. For this test, 4 servers were used. The game map was divided equally among the 4 servers, so each one of them managed a same shape and size zone. The servers were created statically, which means we have started the game with all the 4 servers running and expecting client connections. Clients connect and move freely across the map, migrating between servers as they move from one area to the other.
We used a similar testing condition to the one performed for the original game test. We used the same 50

bots to play a 10 minutes game. We also gave the system a 10 minutes warm-up period before we start registering our values. This test is important to understand how the servers perform in a static multi server environment, where every server remains connected from the start of the game up until the end. This scenario may be compared to the most common scenario of current commercial games on the market.

5.1.3 Cloud DReAM

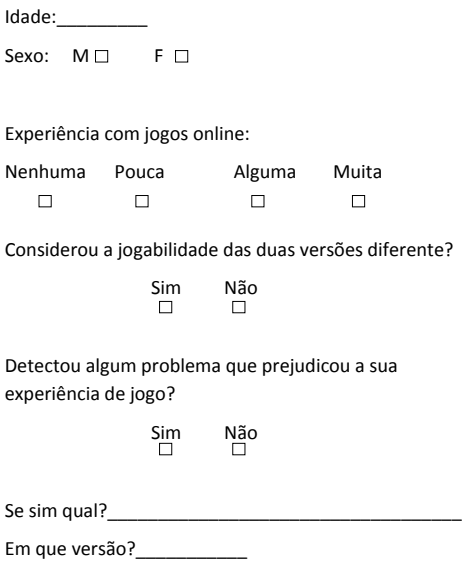
The last scenario uses our system with dynamic servers. This means that the resources (in this case servers) will be adjusted according to the needs of the system. We start the test with one single server and start connecting *bots* to the game. We have used a server number limit on our system, so it will not scale beyond that number. For this test the maximum number of servers working at any given moment is 4. This is done so that we can compare it with the static version of this test.
The scalability triggers we used on this test were CPU above the value of 50% to launch a new instance, and CPU load below 5% in order to terminate an instance. The rest of the conditions were similar to the previous tests apart from the duration. Since it takes longer for the system to adapt, these tests lasted as long as 30 minutes. The number of used *bots* was also 50.

5.1.4 Migration Tests

Since clients can migrate between servers, we want to understand what is the cost of this migration and how it can impact the game. We performed some tests with the purpose of measuring the migration times of the clients between servers. As we wanted to evaluate the impact of migrating a different number of clients, we performed a few different types of tests. The first test we performed was the migration of a single client between two servers. For the second test we migrated 10 clients simultaneously from one server to a different one. Finally, on the last test, we migrated 20 clients from the same server, to a different server. The purpose of these tests is to understand the impact of the migration for both the client and the server.

5.1.5 Usability Tests

In order to test the usability of the system we asked real users to participate in a game that was using the Cloud DReAM system. To start the test we asked the users to play a game on the normal version of the game with 45 other bot clients. After that test, we asked the same user to play the game using the graphical game client, against 45 bot clients on the VFC2 map, using the Cloud DReAM system. The users are not aware of the underlying differences between the two versions. After performing both tests, the users were asked to answer a simple enquire about their experiences with both versions. The enquire model can be seen in Fig. 5.1. The purpose of this test is to understand the impact of our intervention occurring in the background, to the enjoyment of the game. We want to understand if the existence of multiple servers hinders the game enjoyability in any way. The two main aspects we are aiming to test


Idade:_____

Sexo: M ☐ F ☐

Experiência com jogos online:

Nenhuma ☐ Pouca ☐ Alguma ☐ Muita ☐

Considerou a jogabilidade das duas versões diferente?

Sim ☐ Não ☐

Detectou algum problema que prejudicou a sua experiência de jogo?

Sim ☐ Não ☐

Se sim qual?_____

Em que versão?_____

Figure 5.1: Enquire answered by volunteers

are the events that involve players located in different servers, and the player migrations between servers. We consider these two situations to be the ones where there can potentially be noticeable differences between versions.

5.2 Used Infrastructure

To perform the previous tests we used the following systems:
For our cloud infrastructure we used a Desktop computer with an Intel Core i7 2.94GHz and 16GB of RAM. This system is running on Ubuntu 12.04 LTS and the version of Eucalyptus that is being used is 3.1. Each game server is running on a virtual machine created on the cloud. Each virtual machine has 1 CPU with 2.93GHz and has 1 GB of RAM available.
The Cloud Manager component is running on an Intel Pentium 4 3.20GHz and 1GB of RAM, with Windows XP Professional.
The client *bots* used on the tests are running on machines equipped with Intel Core2 Quad Q6600 2.40GHz and 8GB of RAM available. These machines are running Windows 7 Professional 64 bits version. All the tests were performed on a local network with $100 Mbps$ bandwidth.

5.3 Used Map

We have used for our tests the VFC2 map. This is a custom map that was designed to test the VFC4FPS system. This map is a perfect square arena, with a dimension of 2048x2048. This measurement is done in cubes, which are the native unit of measure of Cube 2. The map is a medium sized area with a fully open space. We opted to use this medium sized map due to the number of players we could simulate with our resources. We found that this map size was the ideal to exercise our load balancing and scaling algorithms.
Since this system is aimed at MMO styled games with large maps, we also performed tests with a larger map named VFC3, which is 4096x4096. This map is also a perfect square arena styled map. On this work however we only present the results from the tests performed on the VFC2 map. The tests performed on VFC3 showed that with our test setting, the players would scatter around the map, and the load generated would not be enough to stress the system. This caused the system to remain stable with only one server, which is an acceptable result. However, we wanted to test our load balancing and scaling algorithms further and thus decided to present the results of VFC2 which are more interesting to discuss.

5.4 Ideal Scenario

It is important to define the ideal scenario that describes the results of a perfect system. We use this to compare with our own results and try to understand how close we get to the ideal results. For this system, an ideal scenario can be described as the following: the game starts off with one server; clients connect to the server and start generating load on the machine; The machine considers itself to be overloaded and asks for help; A new server is launched to help deal with the increasing load; when the second server is up, the system is now capable of supporting an extra number of players; this number can, in theory, be twice the number of players that was supported by a single server; this could go on by adding more servers; in all this scaling operation, usability of the game is maintained; the server machines are kept within certain load values that are considered ideal for maintaining game usability. It is important to note that the scaling operation would only go as far as the game requires it to. This means that new servers are only created when they are needed, so that resource usage is optimized. It is also to be expected that for the same number of players on each server, the load values remain close to each other.
)

5.5 Launch Instances

The eucalyptus cloud platform needs between 4 minutes and 30 seconds, to 5 minutes and 10 seconds to launch a new instance of our server image. During this time period, eucalyptus launches a new instance in the cloud infrastructure, the instance performs the normal boot routines of the windows system, and our server application is launched. This time is measured from the moment that the launch instance

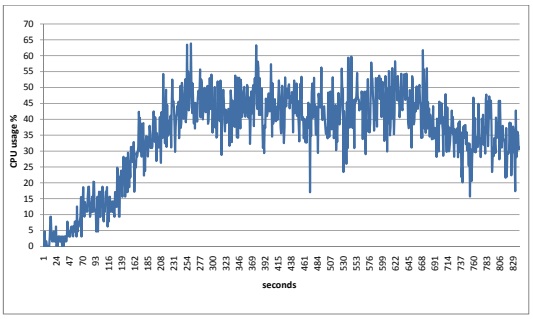


Figure 5.2: CPU Usage for the server

command is issued, up to the moment that the server instance contacts the cloud manager to register itself. This time period is something that is out of our control. It depends on the cloud platform, and on the operating system being used in our images.

5.6 Scenarios Evaluation

5.6.1 Original Game

This test reflects the original version of the system, working only with one server. This server is performing the VFC operations on the game. The results from this test can be seen in Fig. 5.2. This chart represents the CPU usage in percentage during the entire test. In the beginning we can see that the usage is very low and starts growing as we connect more players up to the maximum number that was used for this test. After this initial setup time, we can see that the CPU usage remains stable on values around 45% to 60%, throughout the entire duration of the test.The decrease of usage in the end occurs due to the disconnection of clients that was performed to finalize the test.
This setup shows what is the performance of a server, in a single server setup. We use the results obtained in this test as a baseline for comparison with the multiple server approaches. We can see that the number of players used is enough to put some stress on the server in terms of processor usage. Other factors such as networking were not considered for this case, as they are the same as described in [34]. The usage of RAM was also found to be very consistent, with values between 40 MB and 50 MB, and not very worthy of further detailing.

5.6.2 Cloud DReAM with static infrastructure

This test uses 4 servers that are active throughout the entire game. Fig. 5.3 through Fig. 5.6 show the CPU usage for the 4 servers used in this test and Fig. 5.11 shows the player distribution across servers. The charts show that the load on all the servers stays below the 50% value (for the great majority of the

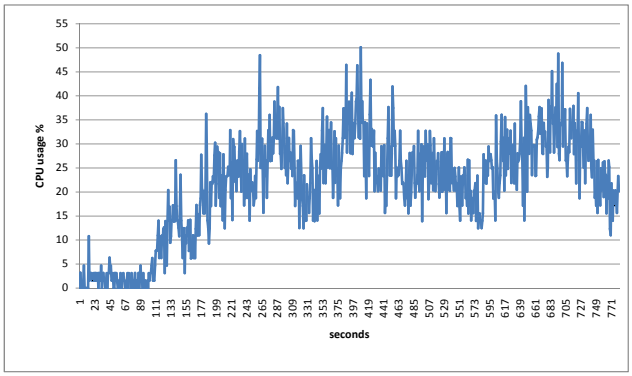


Figure 5.3: CPU Usage for server 1 (Static)

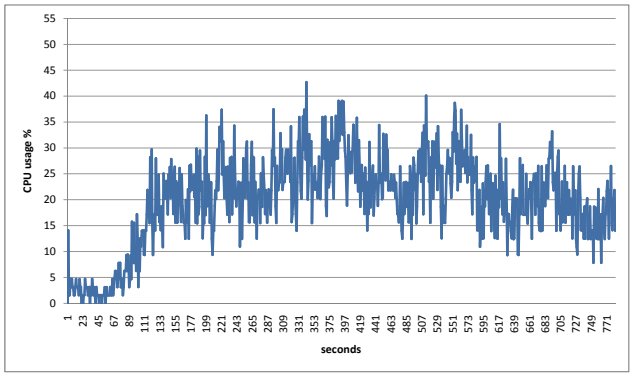


Figure 5.4: CPU Usage for server 2 (Static)

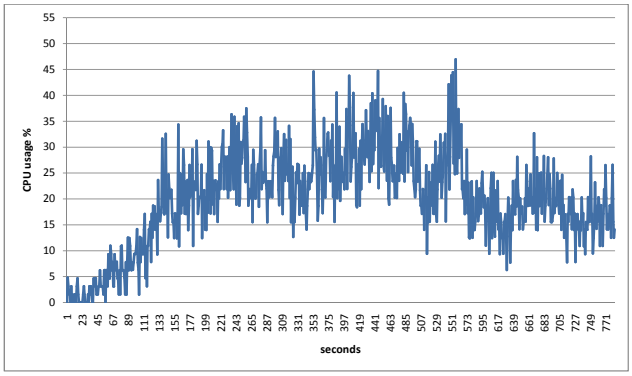


Figure 5.5: CPU Usage for server 3 (Static)

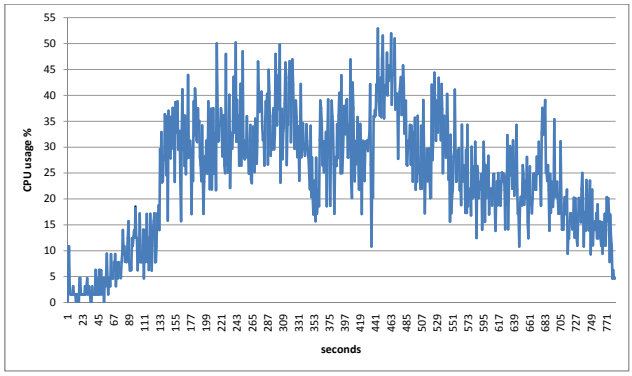


Figure 5.6: CPU Usage for server 4 (Static)

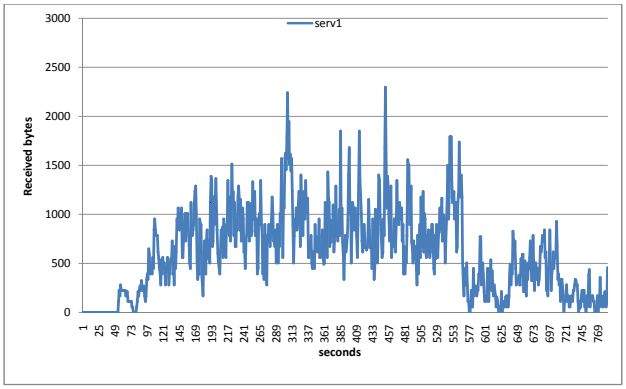


Figure 5.7: Events received for server 1 (Static)

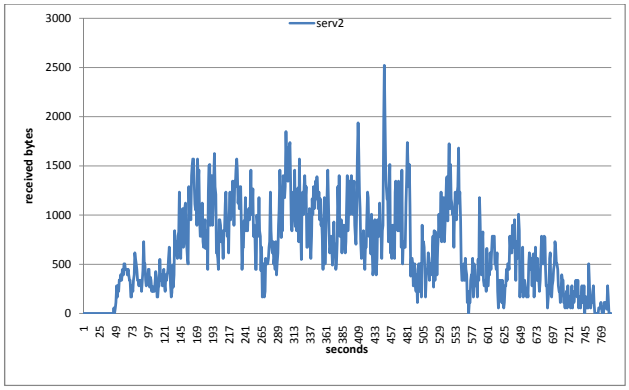


Figure 5.8: Events received for server 2 (Static)

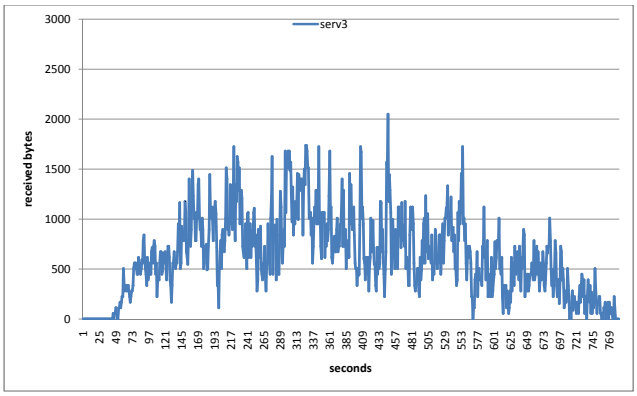


Figure 5.9: Events received for server 3 (Static)

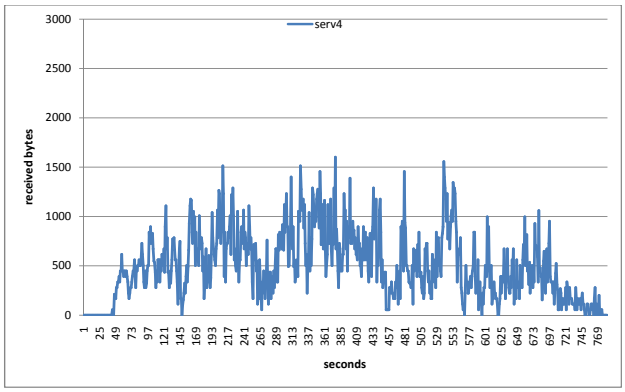


Figure 5.10: Events received for server 4 (Static)

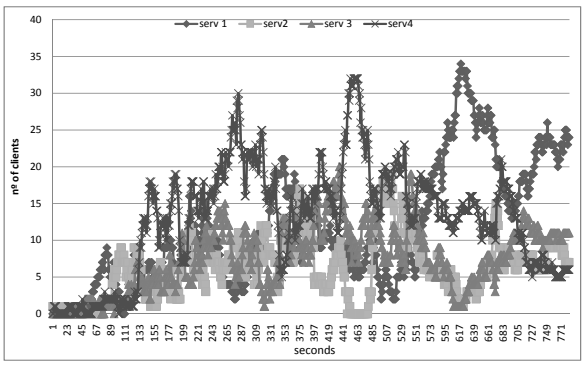


Figure 5.11: Clients connected to each server per second

duration of the experience), and remains between 15% and 30% most of the time, with some occasional peaks to around 40%. Some periods of greater load are visible on some of the servers, where the usage can go beyond 50%. Those periods can be justified with the presence of a greater number of players on the area that is being managed by that server. Recall that we are distributing players between servers based on their location inside the game world. This makes it possible for a sudden migration of every player to the same server. This type of situation will bring the CPU usage closer to what was observed in the previous test case with a single server, where the usage was always between 45% and 60%.

Comparing this test setting with the original one described in the previous section, we can observe a reduction of the CPU usage for all the server machines. This was an expected result, as the servers are splitting the load generated by the players, between themselves. We can also observe that the load is not perfectly split. In an ideal scenario, if we maintain the same number of players, every server would have approximately the same load, and it should be a fraction of the load observed in the single server case. To justify this we present in figures 5.7 through 5.10 the information related to the network communication

between servers. This communication did not exist in the original solution (since it only had one server). This extra network usage is due to the player state updates and event replication between servers. What this means is that servers have extra events and states to process, independently of the number of players that are connected to them at the moment. This extra data that is required to be processed is what causes the results to differ from the ideal case. This communication is, however, necessary and has to be performed in order to maintain the game consistency. It can still be optimized further. This test setting also allows us to conclude that we are using more resources than the ones we need. Every server machine in this game is showing CPU usages that are relatively low. This means that for the number of clients connected and consequent traffic and processing generated, the number of servers being used is higher than what would be required. This is a perfect example of the waste of resources that we are trying to mitigate with the Cloud DReAM system. We use this scenario as a comparison for the Cloud DReAM solution, since the scenario shown here is a representation of what happens with many game solutions nowadays.

5.6.3 Cloud DReAM

The results description and analysis of this test setting is focused on two distinct cases. During our experiments, we found out that these two outcomes were very common and interesting to be presented and analysed. The first case that we find interesting starts off with one server, and as the game progresses the system decides to scale to two different servers and remain this way through the rest of the game. On the second case, we also start off with one server, but in the end the system has decided to scale up to three different servers, and terminate the third one sometime before the end of the test.

First case:

The first case that we describe starts off with one server. We start to connect our players to the server. When the server considers itself to be overloaded, it ask for help from the cloud manager. The cloud manager orders the launch of a new server instance. The players distribute themselves across these two servers in an even way. The system remains stable with the two servers until the end of the test. In Fig. 5.12 and Fig. 5.13 we have the CPU values for both servers, and in Fig. 5.16 we have the players distribution across servers. The charts illustrate the scenario that we have described. On the first chart we can see a CPU usage increase around 171 seconds, that triggered the launch of the second server. After the second server is operational around 524 seconds (with relation to chart 5.12) , the players are redistributed between the two servers according to their positions. On this situation we see that after that moment, the CPU usage on both servers remains stable. Server 1 registers CPU usages of 20% to 40%, while server 2 registers values of 10% to 30%. Even though the load is not perfectly spread across both servers, it is very close. This means that the players on both servers, and the actions they perform do not generate enough load to trigger any other scaling operation.

In Fig. 5.14 and Fig. 5.15, we present the communication between servers that took place during this test. As we can see, during the first 445 seconds of the objects chart, and 520 seconds of the events chart, the communication is zero. This happens because we only have one server during these first moments.

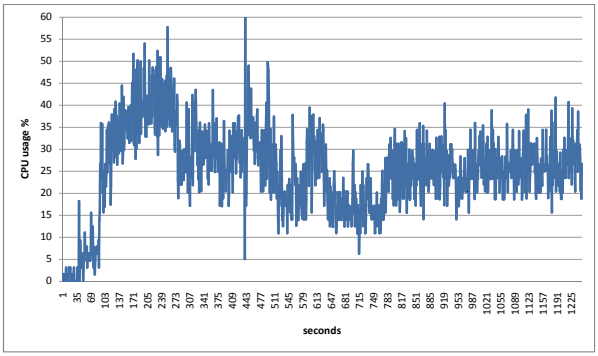


Figure 5.12: CPU Usage for server 1 (Cloud DReAM first case)

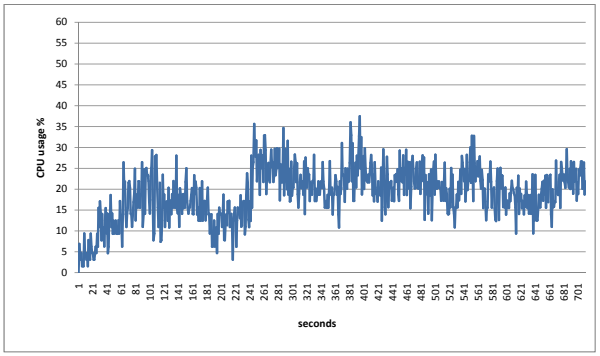


Figure 5.13: CPU Usage for server 2 (Cloud DReAM first case)

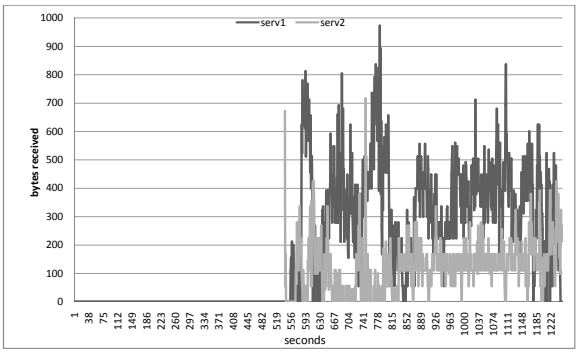


Figure 5.14: Events received for both servers (Cloud DReAM first case)

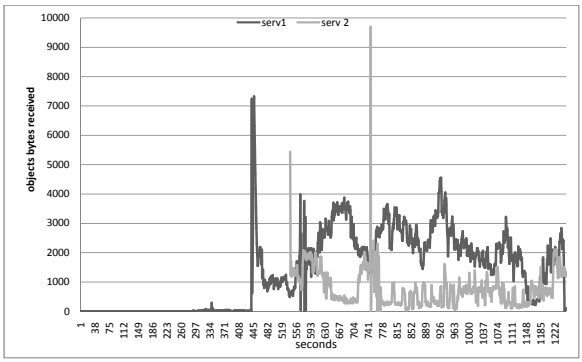


Figure 5.15: Objects received for both server (Cloud DReAM first case)

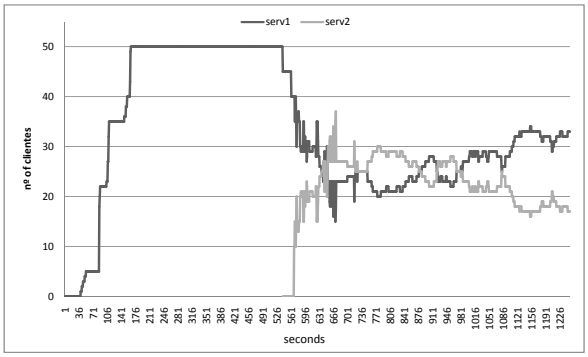


Figure 5.16: Clients connected to each server per second

The objects chart starts to have communication earlier than the events one due to the periodic nature of the objects exchange messages, which is not the case for events.
From the moment the second server connects, the two servers start exchanging updates between themselves. As we can see by comparing Fig. 5.14 with Fig. 5.15, the quantity of information exchanged between servers is mainly due to player state updates. This is due to the fact that state updates are periodic in its nature and occur every 33 ms. The events are not periodic, they occur based on player actions inside the game. Furthermore, not every event is replicated between servers. An event is only replicated when it affects players on both servers. This explains why the amount of information exchanged by the status updates is higher than that of the events.
All this extra network traffic has an impact on the server. Every one of these updates that are received, either state or event, needs to be processed. This is the main reason why the load on the CPU for the same total number of players, is not perfectly split between servers.

Second case:

The second case starts similarly to the previous one. One server is connected and players start to join the game. After a while, the server considers itself overloaded and asks the cloud manager to scale. When the second server is ready and connects, a large portion of clients immediately migrates to that server. This sudden migration happens because of the game world division. A large portion of the players is now located in the area of the second server, and thus is migrated to that server. This increase in the server load causes the second server to consider itself overloaded and requests the cloud manager for help. This generates the launch of a third server. Fig. 5.17 through Fig. 5.19 show the loads for the different servers and Fig. 5.22 we have the players distribution across servers. These charts differ from the static ones since the servers were not all running for the same period of time. They also illustrate what we have just described. We can see an increase in the CPU load on server one around 186 seconds, that triggers the scaling operation. The chart also shows that another load spike occurs on server one right after the scaling operation. This does not trigger a new scale because the cloud manager knows there is a pending scale request from server 1. After server 2 finishes the connection process, we observe the increase in its CPU load around 81 seconds (with relation to server 2 time scale) and the consequent scale request that generates the launch of server 3. Finally, nearing the end of the test, we can see that server 3 has a CPU load below 5% starting from 175 seconds (with relation to server 3 time scale). This causes this instance to be terminated. The machine was terminated because its load was low, and the cloud manager decided that it would not cause additional stress on the remaining two machines.
Similarly to what we have seen on the static test with 4 servers, the CPU load is effectively split between all three servers. What we discussed for that case also applies here. The load is not perfectly split. This is caused in part by our load balancing algorithm which may cause uneven distribution of players between the servers. An example of this is what happened in this scenario with server 2, that triggers a scale operation right after it was launched. The other factor that contributes to the extra load on the servers is the network and processing overhead that we also discussed for the static case.
In Fig. 5.20 and Fig. 5.21 we show the network values for this scenario. Similar to what happened on the

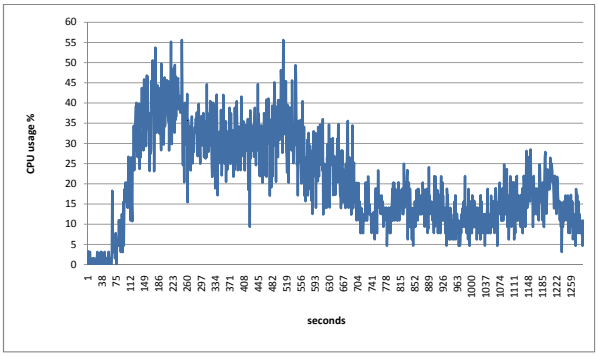


Figure 5.17: CPU Usage for server 1(Cloud DReAM second case)

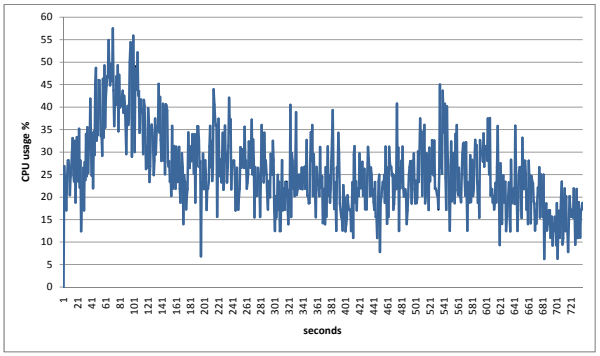


Figure 5.18: CPU Usage for server 2 (Cloud DReAM second case)

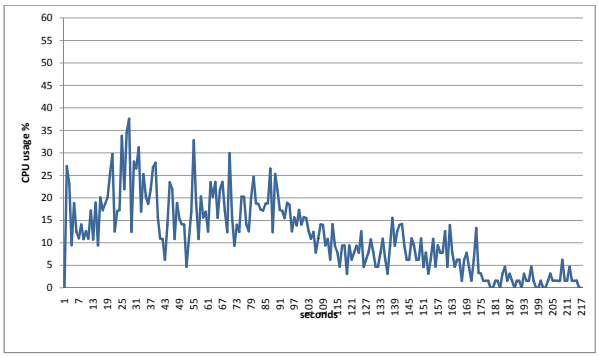


Figure 5.19: CPU Usage for server 3 (Cloud DReAM second case)

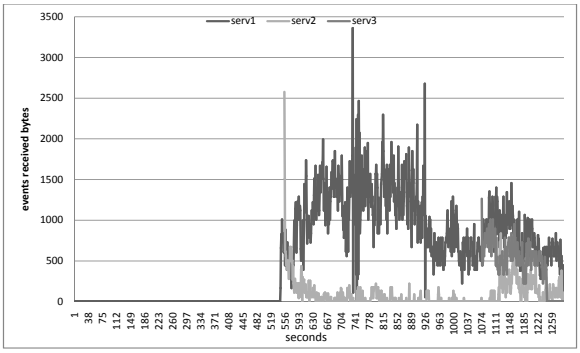


Figure 5.20: Events received for the 3 servers (Cloud DReAM second case)

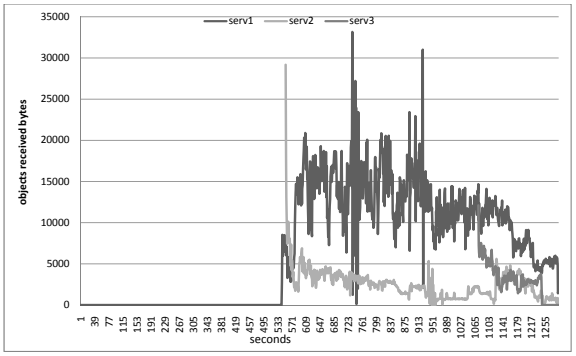


Figure 5.21: Objects received in the 3 servers (Cloud DReAM second case)

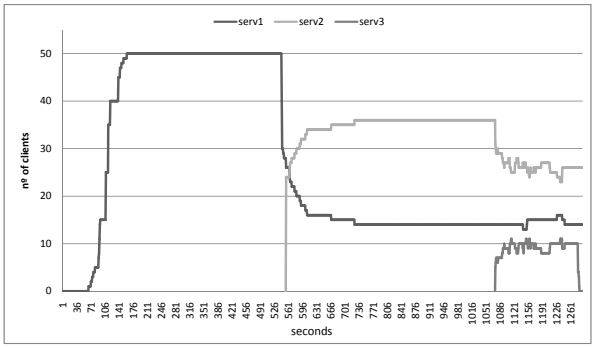


Figure 5.22: Clients connected to each server per second

previous case, the charts start with no communication until the second server connects. It is clear that in this experience, the event data is much higher than what was observed on the previous case. This means that there were more events occurring on the game, which tells us that players were interacting more with each other. This causes the status updates to have more information as well which can be seen in Fig. 5.21. We have claimed that the network communication is one of the causes for the extra load we see after a split, and this experience confirms it. We can compare this case with the previous one, where we had the same conditions, but the network values were lower. On that case the system only required two servers to maintain a stable game. On the current experience, the system required 3 server to deal with the extra load it was experiencing, but eventually stabilized and decided to terminated the third instance.

When we compare both of the scenarios described in this section with the static one described in the previous section, we can draw some conclusions. The first thing that we notice is that the scenarios that are using the Cloud DReAM system used less servers to support the same number of clients. This supports our claim that there is a waste of resources in the static approach. Furthermore this shows that our system can in fact improve the usage of resources, by dynamically deciding when to scale or reduce the number of resources that are required. Another observation that we make is that we can further optimize the communication between servers as well as the load balancing algorithms being used. We see by our tests that the load balancing algorithm based on areas of the map is not ideal and can generate weird situations. The improvement of the load balancing algorithm will allow for an even better resource usage, that will be closer to the ideal case of a perfect split of the load.

5.7 Migration test result

After performing our migration tests we obtained the following results: For the single client migration, we observed values of migration time that ranged between 60 to 80 ms. For the 10 clients migration test the range of values did not change much from the first experience. We observed values of migration time

Player	Time
1	75 ms
2	103 ms
3	90 ms
4	63 ms
5	87 ms
6	95 ms
7	92 ms
8	85 ms
9	67 ms
10	91 ms

Table 5.1: Redirection times for a simultaneous 10 player migration

Player	Time
1	227 ms
2	107 ms
3	224 ms
4	156 ms
5	156 ms
6	146 ms
7	212 ms
8	159 ms
9	98 ms
10	109 ms
11	68 ms
12	194 ms
13	136 ms
14	230 ms
15	116 ms
16	147 ms
17	120 ms
18	106 ms
19	116 ms
20	89 ms

Table 5.2: Redirection times for a simultaneous 20 player migration

ranging from 60 to 115 ms as illustrated by table 5.1 . On the final test, when we migrated 20 clients, we did notice an increase in migration time for some of the clients. The maximum value observed for a client migration on this test was 248 ms. Table 5.2 illustrates the times obtained for one of the tests. Even though the values on the last test have doubled from the previous ones, they are still acceptable and did not pose any playability issues. Furthermore, the event of 20 players migrating simultaneously is a very rare occurrence and did not seem to have a great impact neither on the server side nor on the client side. It is important to notice that these tests were performed on a local network and the clients migrated were mostly bots running on the same machine. We were unable to determine the impact of the bots running on the same machine for the migration times. From our empirical observations we believe that some of the extra delay might be caused by this factor, but it remains to be systematically proven.

5.8 Usability test result

We have asked 13 volunteers to perform this test. The average age of the volunteers was 24. All the volunteers were male. When asked if they noticed any difference between the playability of both game versions, 3 of them said they noticed differences. This means that 77% of the user enquired did not notice any difference in playability. In Fig. 5.23 we present a chart showing the relation between player experience with online games, and the differences noticed in playability.

When asked if they had seen any differences that affected their experience with the game, 4 of the volunteers answered positively. The differences they have reported were inconsistencies on the score table, such as scores being reset to zero; Some glitches on enemy players, that caused them to miss shots; some

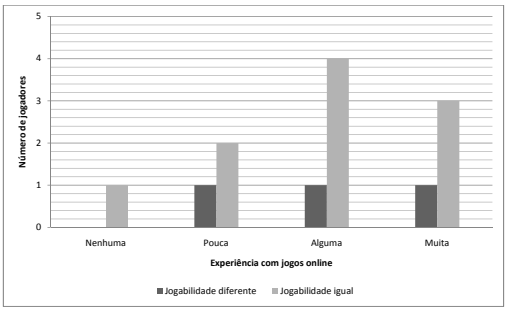


Figure 5.23: Online game experience vs. differences noticed

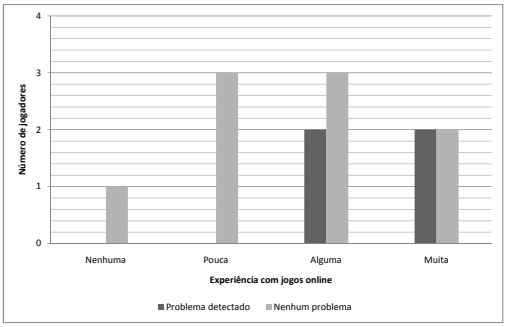


Figure 5.24: Online game experience vs. enjoyability compromising problems

animation bugs, such as a player shooting his weapon and no animation is performed on his avatar. All of these differences were reported on the Cloud DReAM version of the game. In Fig. 5.24 we can see an illustrative chart of the answers.

While the differences that were presented are important in terms of gameplay, they are not game breaking. Most importantly, none of the volunteers noticed any type of differences caused by migration of players between servers, or interactions between players of different servers. This is a positive aspect, that shows that the impact of our system on playability is very reduced and was not very noticeable to players. We can conclude that apart from some minor implementation bugs, we managed to achieve our goal of maintaining playability and enjoyment of the game even with Cloud DReAM performing its operations.

5.9 Summary

On this section we have presented the results from the evaluation to the Cloud DReAM system. We have performed tests to evaluate the performance of our system. We have also evaluated the adaptability capabilities of our system when load on the servers changes. We have compared the results obtained for the Cloud DReAM system, with common static implementations. Finally we have presented the results

from the usability tests performed.

The evaluation conducted in this section represents the first step in the evaluation of this system. The test settings need to be further expanded with more players and more servers. This is important to understand the real benefits that may be gained from using a cloud platform to support an MMOG.

6 Conclusion

On this document we have analysed the current state of the art in terms of multiplayer games. We have analysed the most common infrastructure setups, the load balancing algorithms and interest management techniques. We have also analysed some existing systems that have very similar goals to our own. With resource usage optimization in mind, we opted to develop an approach to be deployed on a cloud environment. The result was the Cloud DReAM system presented in this work. From our evaluation, we can conclude that there is in fact enough potential for a solution based on the cloud computing paradigm to be used. Even though the game used for this prototype is not an MMO type of game, and has important differences that are relevant to be discussed, the solution showed encouraging results. We have shown that it is possible to use the cloud infrastructure to manage the available resources and still provide a usable game environment, where users remain unaware of our actions in the background. We managed to create a first approach to resource usage optimization that has managed to maintain the game usability and performance as we required it to. In terms of scalability, our test setting is not enough to claim with certainty that the system is scalable. The game used poses some limitations on this matter, as well as the resources we had available for testing. However we believe that with a game that is capable of supporting more players and is developed with a cloud infrastructure in mind, a good scalable system would be achievable.

It is a fact that our results are encouraging, but they also show that there is still a large margin for improvement. We have some aspects that are all ready identified and will be subject to improvement. The more important ones that we have identified are: The optimization of the communication between servers; The reduction of the information sent relative to the game events; Optimization of the load balancing and scaling algorithm.

6.1 Future Work

This work is a first step into deploying the game servers on a cloud environment. As such there are improvements that can be performed:

- The VFC consistency system is being enforced by the servers with relation to the clients that are directly connected to them. It would be interesting to have the servers performing some form of interest management between themselves. This would reduce the bandwidth usage even more than the current implementation does.
- The usage of different scaling algorithms is potentially interesting to be explored. An algorithm

that can consider for instance statistical values relative to previous usage history on the servers could help optimize the solution.

- There are different approaches to splitting the map and to divide the players across servers. It would be interesting to evaluate the impact of different algorithms on the performance of the system.
- The solution is deeply connected to some of the specific details of the Cube2 game. It would be interesting to make it a more generic platform that could be extended to different games with ease.

Bibliography

[1] Dewan Tanvir Ahmed and Shervin Shirmohammadi. A microcell oriented load balancing model for collaborative virtual environments. *School of Information Technology and Engineering, University of Ottawa*.

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. *UC Berkeley Reliable Adaptive Distributed Systems Laboratory*, February 2009.

[3] Marios Assiotis and Velin Tzanov. A distributed architecture for mmorpg. *Netgames*, October 2006.

[4] Daniel Bauer, Seam Rooney, and Paolo Scotton. Network infrastructure for massively distributed games. *Netgames*, April 2002.

[5] Carlos Eduardo B. Bezerra, João L.D. Comba, and Claudio F.R. Geyer. A fine granularity load balancing technique for mmog servers using a kd-tree to partition the space. *Insituto de Informática, Universidade Federal do Rio Grande do Sul*.

[6] Carlos Eduardo B. Bezerra and Claudio F.R. Geyer. A load balancing scheme for massively multiplayer onoine games. *Special issues of Springer's Journal of Multimedia Tools and Applications*, 2009.

[7] Jean Botev, Alexander Hohfeld, Hermann Schloss, Ingo Scholtes, and Markus Esch. The hyperverse - concepts for a federated and torrent-based 3d web. *Proceedings First International Workshop on Massively Multiplayer Virtual Environments*, March 2008.

[8] Jean-Sébastien Boulanger, Jörg Kienzle, and Clark Verbrugge. Comparing interest management algorithms for massively multiplayer games. *School of Computer Science, McGill University, Canada*.

[9] Eliya Buyukkaya and Maha Abdallah. Data management in voronoi-based p2p gaming. *Proceedings of IEEE CCNC*, 2008.

[10] Jin Chen, Baohua Wu, Margaret Delap, Björn Knutsson, Honghui Lu, and Cristiana Amza. Locality aware dybamic load management for massively multiplayer games. *PPoPP*, June 2005.

[11] Ta Nguyen Binh Duong and Suiping Zhou. A dynamic load sharing algorithm for massively multiplayer online games. *IEEE*, 2003.

[12] Lu Fan, Phil Trinder, and Hamish Taylor. Design issues for peer-to-peer massively multiplayer online games. *School of Mathematical and Computer Sciences , Heriot-Watt University.*

[13] D. Frey, J. Royan, R. Piegay, A.M. Kermarrec, E. Aceaume, and F. Le Fessant. Solipsis: A decentralized architecture for virtual environments. *Proceedings First International Workshop on Massively Multiplayer Virtual Environments*, March 2008.

[14] Thorsten Hampel, Thomas Bopp, and Ribert Hinn. A peer-to-peer architecture for massive multiplayer online games. *Netgames*, October 2006.

[15] R. Housely, W. Fordand W. Polk, and D. Sodo. Internet x.509 public key infrastructure. *Internet Engineering Task Force Draft, PKIX Working Group.*

[16] Shun-Yun Hu, Shao-Chen Chang, and Jehn-Ruey Jiang. Voronoi state management for peer-to-peer massively multiplayer online games. *Proceedings of IEEE CCNC*, 2008.

[17] Guan-Yu Huang, Shun-Yun Hu, and Jehn-Ruey Jiang. Scalable reputation management for p2p mmogs. *Proceedings First International Workshop on Massively Multiplayer Virtual Environments*, March 2008.

[18] Xinbo Jiang and Farzad Safaei. Supporting a seamless map in peer-to-peer system for massively multiplayer online role playing games. *IEEE Conference on Local Computer Networks*, October 2008.

[19] Wang Junzhong and Yue Zhigang. A finding less-loaded server algorithm based on mmog and analysis. *International Conference on Intelligent Computation Technology and Automation*, 2010.

[20] Rynson W.H. Lau. Hybrid load balancing for online games. *ACM Multimedia International Conference*, October 2010.

[21] Bruno Loureiro, Luis Veiga, and Paulo Ferreira. Vfc-game. *INESC-ID/IST*, 2010.

[22] Dugki Min, Eunmi Choi, Donghoon Lee, and Byungseok Park. A load balancing algorithm for a distributed multimedia game server architecture. *Department of Computer Science and Engineering, Konkuk University.*

[23] K. L. Morse. Interest management in large-scale distributed simulations. *Department of Information & Computer Science, University of California, Irvine*, 1996.

[24] Vlad Nae, Radu Prodan, and Thomas Fahringer. Cost-efficient hosting and load balancing of massively multiplayer online games. *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference*, October 2010.

[25] Vlad Nae, Radu Prodan, Thomas Fahringer, and Alexandru Iosup. The impact of virtualization on the performance of massively multiplayer online games. *Network and Systems Support for Games (NetGames), 2009 8th Annual Workshop*, November 2009.

[26] André Filipe Pessoa Negrao. Vfc large-scale: consistency of replicated data in large scale networks. *Instituto Superior Técnico*, September 2009.

[27] Sylvia Ratnasamy, Paul Francis, Mark Handley, and Richard Karp. A scalable content-addressable network. *SIGCOMM*, August 2001.

[28] Simon Rieche, Klaus Wehrle, Marc Fouquet, Heiko Niedermayer, Leo Petrak, and Georg Carle. Peer-to-peer based infrastructure support for massively multiplayer online games. *RWTH Aachen University / University of Tübigen.*

[29] Simon Rieche, Klaus Wehrle, Marc Fouquet, Heiko Niedermayer, Timo Teifel, and Georg Carle. Clustering players for load balancing in virtual worlds. *Proceedings First International Workshop on Massively Multiplayer Virtual Environments*, March 2008.

[30] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. *Proc. of The 18th IFIP/ACM International Conference on Distributed Systems Platforms*, November 2001.

[31] Gregor Schiele, Richard Suselbeck, Arno Wacker, Tonio Triebel, and Christian Becker. Consistency management for peer-to-peer based massively multiuser virtual environments. *Proceedings First International Workshop on Massively Multiplayer Virtual Environments*, March 2008.

[32] Rudiger Schollmeier. A definition of peer-to-peer networking for the classification of peer-to- peer architectures and applications. *IEEE*, 2002.

[33] Richard Suselbeck, Gregor Schiele, and Christian Becker. Peer-to-peer support for low-latency massively multiplayer online games in the cloud. *Network and Systems Support for Games (NetGames)*, November 2009.

[34] Luís Veiga, André Negrao, Nuno Santos, and Paulo Ferreira. Unifying divergence bounding and locality awareness in replicated systems with vector-field consistency. *INESC-ID, Lisboa Portugal.*

[35] Bart De Vleeschauwer, Bruno Van Den Bossche, Tom Verdickt, Filip De Turck, Bart Dhoedt, and Piet Demeester. Dynamic microcell assignement for massively multiplayer online gaming. *Netgames*, October 2006.

[36] Arno Wacker, Gregor Schiele, Sebastian Schuster, and Torben Weis. Towards an authentication service for peer-to-peer based massively multiplayer virtual environments. *Proceedings First International Workshop on Massively Multiplayer Virtual Environments*, March 2008.

[37] Jim Waldo. Scaling in games and virtual worlds. *Communications of the ACM*, 51, August 2008.