UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

CARLOS EDUARDO BENEVIDES BEZERRA

Load Balancing Techniques for MMOG Multiserver Systems

Trabalho Individual II
TI-002

Prof. Dr. Cláudio Fernando Resin Geyer
Advisor

Porto Alegre, July 2010

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

BSP	Binary space partition
DVE	Distributed virtual environment
MMG	Massively multiplayer games
MMORPG	Massively multiplayer online role-playing games
NPC	Non-player character
P2P	Peer-to-peer

LIST OF FIGURES

ABSTRACT

Electronic games have become undoubtedly popular in the last few decades. This popularity gained an even higher raise when the multiplayer feature was added – multiple players could play with, or against, each other in the same match. The latest advance, however has been the massively multiplayer online games – or MMOGs, in short –, which allows the simultaneous participation of tens of thousands of players in the same game. However, to cope with such a demanding application, a powerful infrastructure is required: be it a powerful server-cluster or a geographically distributed collection of low-cost volunteer server nodes, there must be some way to distribute the load imposed by the game on the many computers involved in serving it. This work presents some interesting papers related to load balancing techniques used in massively multiplayer online games, commenting each one of them about its relevance. In the end, it is presented an idea of a possible improvement for load balancing by using a generic binary space partition (BSP) tree.

Keywords: MMOG, Massively Multiplayer Online Games, Distributed Interactive Simulation, Multiserver, Load Balancing.

1 INTRODUCTION

The main characteristic of massively multiplayer online games is the large number of players, having dozens, or even hundreds, of thousands of participants simultaneously. This large number of players interacting with one another generates a traffic on the support network which may grow quadratically compared to the number of players (CHEN; HUANG; LEI, 2006), in the worst case (Figure 1.1).

When using a client-server architecture, it is necessary that the server intermediates the communication between each pair of players – assuming that the game is intended to provide guarantees of consistency and resistance to cheating. Obviously, this server will have a large communication load, thus, it must have enough resources (available bandwidth) to meet the demand of the game. For this reason, it must be considered that the main resource to analyse is the available bandwidth, for this is the current bottleneck for MMOGs (FENG, 2007).

The problem is that, when using a distributed server, it must be delegated to each server node a load proportional to its power. Thus, no matter to which server each player is connected, their game experience will be similar, regarding the response time for their actions and the time it takes to be notified of actions from other players as well as of state changes in the virtual environment of the game.

An initial idea might be to distribute the players among servers, so that the number of players on each server would be proportional to that server's bandwidth. However, this distribution would not work, for the burden caused by players also depends on how they are interacting with one another. For example, if the avatars of two players are too distant from each other, probably there will be no interaction between them and therefore the server needs only to update each one of them with the result of their own actions. However, if these avatars are close to each other, each player should be updated not only about the results of his own actions, but also of the actions of the other player.

Normally, players can freely move their avatars throughout the game world. This makes possible the formation of *hotspots* (AHMED; SHIRMOHAMMADI, 2008), around which the players are more concentrated than in other regions of the virtual environment

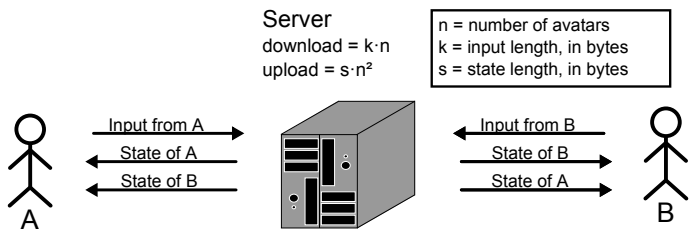


Figure 1.1: Quadratic growth of traffic when avatars are close to each other

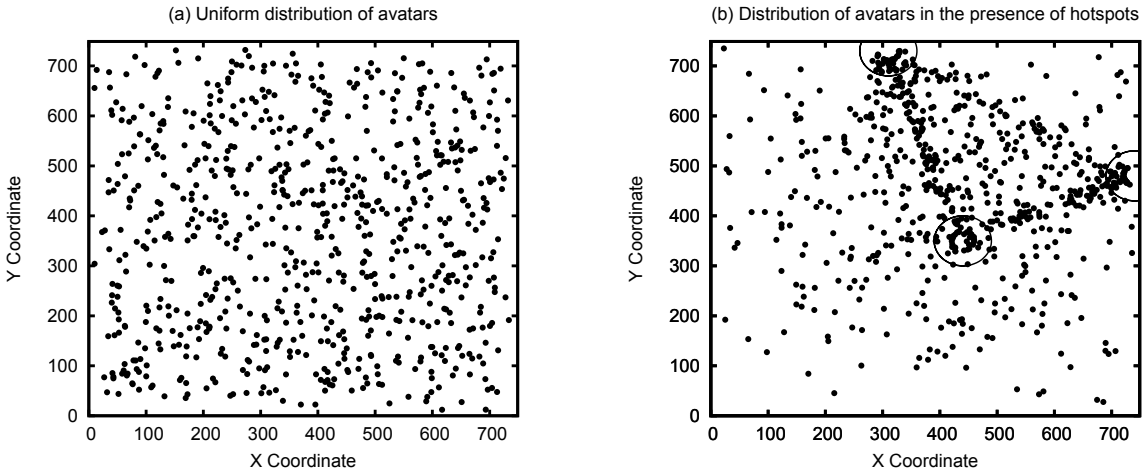


Figure 1.2: Distribution of avatars with and without hotspots

(Figure 1.2). Moreover, many massively multiplayer online RPGs not only permit but also stimulate, to some extent, the formation of these points of interest. In the worlds of these MMORPGs, there are entire cities, where the players meet to chat, exchange virtual goods or even fight, and there are also desertic areas, with few attractions for the players, and where the number of avatars is relatively small compared to other places of the environment.

For this reason, it is not enough just to divide the players between servers, even if this division is proportional to the resources of each one of them. First, in some cases the usage of the server's bandwidth may be square to the number of players, while in others it may be linear. It is shown in (CHEN; HUANG; LEI, 2006) that, in a group of avatars who are neighbors of one another, the rate of packets sent to each player is proportional to the number of avatars in that group (as in Fig. 1.1). This reason alone is enough to define a new criterion for load balancing.

Moreover, there is another important issue: the overhead of the distribution. As the servers need to communicate with one another, there must be a way to minimize this traffic, reducing the waste of resources of the server system. The load balancing scheme for MMOGs must, then, prevent the presence of hotspots from degrading the quality of the game beyond a tolerable limit.

In this work, it is presented a few recent papers which try to solve this problem, achieving a significant fair load distribution. Besides reviewing these works, some comments are added based on the advantages and weaknesses of each solution presented by the authors. In the end of this paper, a new idea is presented, suggesting the use of a generic binary space partition (BSP) tree.

2 A MICROCELL ORIENTED LOAD BALANCING TECHNIQUE

In this work, (AHMED; SHIRMOHAMMADI, 2008) present a microcell oriented load balancing mechanism in the context of a hybrid MMOG (massively multiplayer online game) architecture. The objective of the proposed approach is to define a fair load balancing model for peer-to-peer MMOGs and virtual environments that minimizes peer-to-peer distortions in the overlays. In the case of an overloaded server, the proposed approach identifies appropriate microcell(s) that reduces overlay maintenance cost and minimizes inter-server communications. The load is relieved by devolving these microcells to the other less loaded servers.

2.1 Using microcells

In this model, a set of microcells defines the virtual game world that is maintained and coordinated by a set of servers called server pool. The server pool having n servers is represented by a set $S = S_1, S_2, ..., S_n$. Each server, e.g. S_i, serves one or more microcells depending on load distribution and avatars' population in the corresponding microcells. The microcells are given unique identification numbers: left to right, top to bottom in order. In zonal MMOGs, the game world is usually divided into several manageable regions like zones. The zone concept is not entirely new and has been considered over the last few years. The hexagonal shape has a couple of advantages over others like circular, hence considered in this design to define the shape of a microcell. Each microcell is run by a server called master: a special node that coordinates the interactions in a collaborative manner with the active participation of the microcell members, i.e. avatars. So, a set of master nodes, i.e. the server pool, regulates the operation of the MMOG and provides overlay services. In that sense, the system is hybrid as it combines the benefits of both centralized and distributed systems. The layout of microcells and a random microcell distribution to servers are shown in Figure 2.1.

2.2 Authors' Load definition

In this work, the authors try to deal with hotspots. So, the implied resolution in this regard is that the server pool can handle the total system load. A hotspot refers to a crowded region in a zonal MMOG, leading to the need for load balancing. But, the key question is how to figure out a hotspot. Most of the approaches consider the total number of players in a zone/microcell as an identifying attribute. If the total number of players in a microcell exceeds the maximum affordable limit, it is a hotspot. Logically, it is correct.

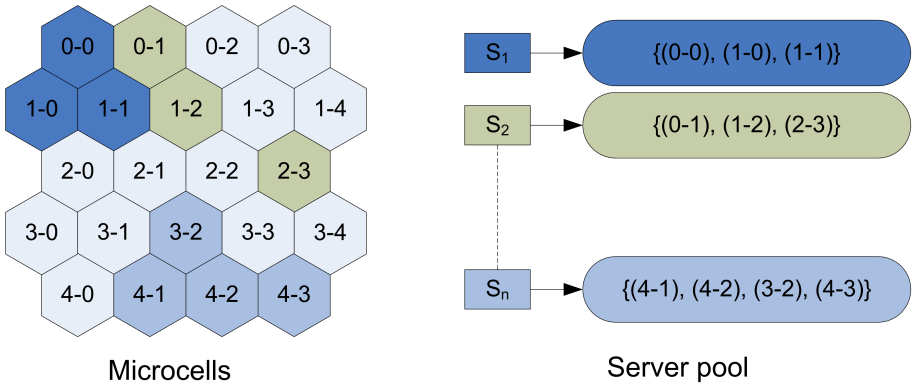


Figure 2.1: The layout of microcells and a random microcell distribution to servers

But, here the authors look at it in a different way. In MMOGs, there are many players like soldiers, robots, aircrafts, etc. with various features. In addition, the actions that actually take place in a game have an influence on message generation rate. For example, in spite of the regular updates from a player, there may be an immediate message due to a trigger in a pistol by an avatar. Every message requires reaching to every other interested player in the interested region which eventually increases the load of the master. So, the load is not completely related to the number of players, actually, it is purely linked with the message generation rate. Thus, the message generation rate, R^m, is used to estimate the load of the masters.

2.3 Marking of a loaded server

Say, the server, S_i, is in charge of a set of microcells, $M_i = m_{i_1}, m_{i_2}, ..., m_{i_k}$. Let $R_i^m = r_{i_1}, r_{i_2}, ..., r_{i_k}$ are the message generation rate of the microcells $m_{i_1}, m_{i_2}, ..., m_{i_k}$ respectively where all of them are governed by server S_i. Thus, the load of the server S_i according to message generation rate is $R_i^m = \sum_{j=1}^{k} r_{i_j}$. Considering the term T^m as a message threshold, a server is loaded or there is a hotspot if $R_i^m > T^m$.

2.4 An approach to load balance

When there is a hotspot, the load of the concerning server is relieved by moving one or more microcells to other less loaded servers. As the movement of microcell introduces complexities, some intelligence is required to identify the best possible set of microcells that need to be moved.

The reduced deployment and maintenance overhead of the hybrid MMOGs (peers or avatars perform some of the message relaying tasks) come at the cost of performance due to peer dynamics. Many techniques have been proposed in literatures such as entity typing and the smart interest driven zone crossing with buffer region to keep the desired level of performance. As it is difficult to predict avatars' movement around a cell boundary (possible frequent in and out movements), it could be effective to share the relevant state information between two adjacent microcells. Instead of sharing the complete state information, a buffer region is defined between two adjacent microcells, as shown in Figure 2.2. This greatly reduces exchanging of unimportant messages. But, if two adjacent microcells are handled by a single master, there is no buffer region in between, as shown in Figure 2.2. According to that figure, there is a buffer region between Microcell 2-4

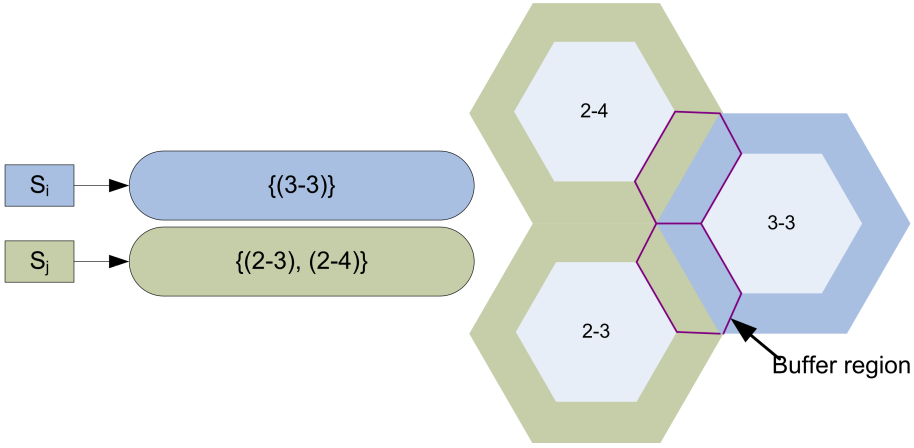


Figure 2.2: The buffer region and microcell covering

and Microcell 3-3 as they are administered by two different servers but Microcell 2-4 and Microcell 2-3 have a common master and hence no buffer region in between.

The objective of changing a microcell ownership from one master to another is to approach hotspot. But, the key point that must be taken care of is to minimize inter-server communication cost while carrying out such microcell ownership alteration. Let, I_i is the number of interested links completely internal to the Microcell i among the avatars and $B_{i,j}$ is the number of interested links involved between Microcell i and Microcell j in the buffer region.

Let, the Server S_i is overloaded and currently serving k microcells: $M_i = m_{i_1}, m_{i_2}, ..., m_{i_k}$. The target is to determine an ordered list of microcells so that the microcell ownership alteration according to that order will introduce less performance penalty in the overlays of hybrid MMOGs. The algorithm initially forms group of microcells based on the microcell's neighborhood properties. A microcell can be a member of a nonempty group if and only if it has a common edge with any microcells of that group. Let, after the application of this grouping policy, Server S_i has l groups $G_1^i, G_2^i, ..., G_l^i$ in order of their cardinality, i.e. $|G_1^i| \leq |G_2^i| \leq ... \leq |G_l^i|$. The general strategy is to find a microcell within a group with a minimum number of interested links among the microcells of that group. Due to the change of ownership, these links become foreign links that will cause inter-server communication. Thus, a group with more microcells may have more common edges than a small group, at least heuristically. Thus, it can be effective to keep those groups intact. So, a group with a fewer microcells is chosen for load balancing and that is why we start with the group G_1^i, then G_2^i and so on, if necessary. Finally within a group, a microcell with a minimum number of potential foreign links is considered for ownership alteration.

2.5 Work analysis

In this work, (AHMED; SHIRMOHAMMADI, 2008) have proposed a microcell oriented load balancing mechanism for peer-to-peer MMOGs. The objective of the proposed approach was to discover one or more microcells of a loaded server in terms of low movement cost. In the case of an overloaded server, the proposed approach identifies microcell(s) that reduces overlay maintenance cost and lowers inter-server communications. The load is reduced by devolving these microcells to the least loaded server.

The problem with this approach, although it considers important aspects of load bal-

ancing for MMOGs, is that the server to which the moved microcell is destined is always the least loaded one – besides the fact that the authors chose to consider a homogeneous server system. By transferring microcells this way between servers, the load balancing scheme induces to regions to become more and more fragmented, which ultimately leads to a higher inter-server communication. This problem, thus, is dealt with by the technique presented in the next chapter.

3 A GRAPH PARTITIONING BASED TECHNIQUE

In this work, (BEZERRA; GEYER, 2009) propose a balancing scheme with two main goals: allocate load on server nodes proportionally to each one's power and reduce the inter-server communication overhead, considering the load as the occupied bandwidth of each server. Four algorithms were proposed, from which ProGReGA is the best for overhead reduction and ProGReGA-KF is the most suited for reducing player migrations between servers. The authors also made some comparisons with a state-of-the-art related work, where their approach performed better.

3.1 Related work

The servers receive the action performed by a player, calculate its outcome and send it to all interested players, who are usually those whose avatars are close to the avatar of the first player. If two players are split between different servers, each of these need not only to send the state update to the player served by it, but it has also to send the update to the server to which the other player is connected. This server, in turn, forwards that state update to the other player (Figure 3.1).

It is perceived, then, that each state will be sent twice for each pair of players who communicate through different servers. This overhead not only causes the waste of resources of the servers, but it also increases the delay to update the status of the replica of the game in the players' machines, damaging the interaction between them.

Therefore, players who are interacting with each other should, ideally, be connected to the same server. However, it is possible that all players are linked through relations of

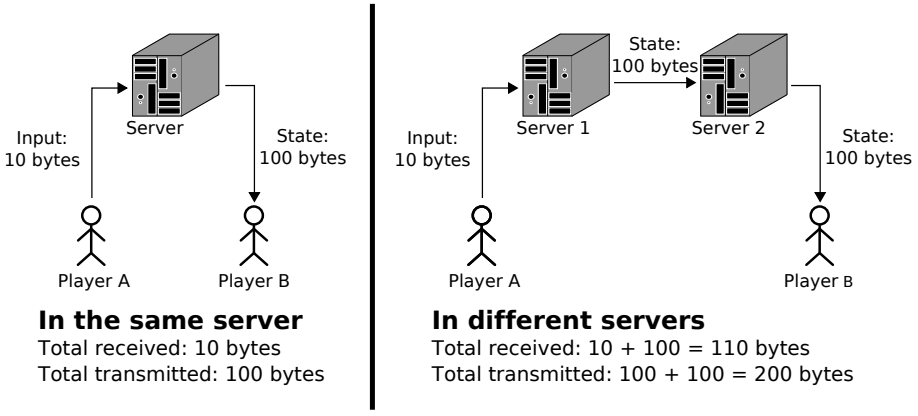


Figure 3.1: Overhead caused by the interaction of players connected to different servers

interaction. For example, two avatars of two different players, may be distant from each other, but both could be interacting with a third avatar, between them. Anyway, it is still necessary to divide the players among servers. The question is how many pairs of players and which of them will be divided into different server nodes. It is therefore necessary to decide a criterion to group players. Many works have been done in the past – such as (DE VLEESCHAUWER et al., 2005; LU; PARKIN; MORGAN, 2006; CHEN et al., 2005; DUONG; ZHOU, 2003; AHMED; SHIRMOHAMMADI, 2008) – trying to find a near optimal, yet fast, load balancing technique.

In (DE VLEESCHAUWER et al., 2005), for example, a few algorithms are proposed. Their idea is to transfer regions managed by overloaded servers to lightly loaded ones, in a way such that the maximum overload among the servers is reduced. Some very interesting principles are used, as the concepts of microcells and macrocells (which will be described in the next section). However, their model has some differences to the one presented here. For instance, they consider the number of players as the load to distribute, heterogeneous systems are not supported, and the algorithm they use to refine an existing partitioning is simulated annealing. This one may be a little too general, considering that there are algorithms specific to partitioning, such as (KERNIGHAN; LIN, 1970) or (FIDUCCIA; MATTHEYSES, 1982), which may perform better. Finally, some of the algorithms proposed in (DE VLEESCHAUWER et al., 2005) do not consider the communication overhead between servers.

It is proposed in (LU; PARKIN; MORGAN, 2006) a load balancing of cpu usage among the nodes of a cluster. The load balancing is done by an "off the shelf" NAT load balancer, which distributes the players' connections among the cluster nodes in a round-robin manner. Thus, each cluster node will end up with the same number of players, on the average. This approach, although it does balance the number of clients per cluster node, communication between the server nodes is not considered, for they are connected via a high-speed and low latency network. Furthermore, the authors were concerned only with cpu usage, and not bandwidth occupation, which is the most important aspect in this context.

Chen et al. propose, in (CHEN et al., 2005), a load shedding technique, in which an overloaded server attempts do shed its load to its neighbors. After finding a lightly loaded neighbor server – if there is one – it transfers some of its boundary microcells to that neighbor. To form the group of microcells to transfer, a microcell from the border is chosen, and others are added in a breadth-first search (BFS) order. A very similar work, which also uses BFS, is described in (DUONG; ZHOU, 2003). However, these models present a few points which could be improved. For example, a server is considered overloaded when a certain number n of players is reached. As mentioned before, this is not the best measure, since the load on a server may vary completely, depending on how these n players are distributed across the region managed by the server. Also, the use of BFS to merge two partitions or split one of them may not be ideal. Some other algorithm for graph partitioning applied to distributed systems (KERNIGHAN; LIN, 1970; FIDUCCIA; MATTHEYSES, 1982; KARYPIS; KUMAR, 1998; HENDRICKSON; LELAND, 1995) could be used, for most of these algorithms were designed exactly to balance load and minimize dependence between nodes.

The model presented by (AHMED; SHIRMOHAMMADI, 2008) – and described in the previous chapter – has been compared to the one of (BEZERRA; GEYER, 2009), for they share many design considerations.

Next, some principles, defined in other works, will be presented. Based on some of

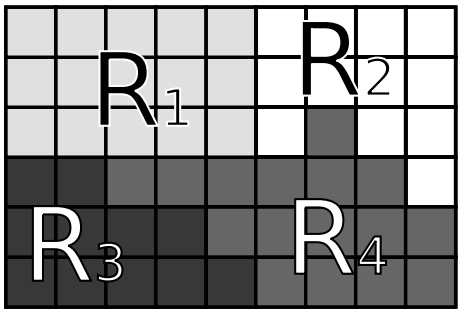


Figure 3.2: Cells grouped into four regions (R_1, R_2, R_3 and R_4)

these principles, and on research and implementation carried out, the authors defined the load balancing scheme presented in this chapter.

3.1.1 Microcells and macrocells

One way to explore the locality of the players is by grouping them according to the position occupied by their avatars in the virtual environment. One of the ideas proposed in the literature (DE VLEESCHAUWER et al., 2005) follows the principle of dividing the virtual environment in cells of fixed size and position. These cells are relatively small – or **microcells** – and they can be grouped, forming an area called **macrocell**. Each macrocell is then assigned to a different server, which will manage not a large cell of fixed size and position, but a variable set of small cells. These microcells can then be moved dynamically between different macrocells, maintaining the load on each one of the servers under a tolerable limit.

Obviously, the microcells designated to the same server node do not generate additional traffic to synchronize with each other, but the synchronization overhead of the macrocell is unpredictable, because the number of neighbors of each one of them is not foreknown, for its shape is variable. However, it was demonstrated (DE VLEESCHAUWER et al., 2005) that this overhead is compensated by a better distribution of the load between servers in the game.

3.1.2 Load balancing in local scope

In (LEE; LEE, 2003), it is also proposed a scheme for dynamic load balancing for the servers of a multi-server virtual environment. Following the scheme proposed by the authors, an overloaded server starts the process by selecting a number of other servers to be part of the load redistribution. The set of selected servers depends on the load level of the initiating server as well as on the amount of idle resources of the other servers. After the formation of this set, its elements allocate portions of the virtual environment using a graph partitioning algorithm, so that the servers involved have similar final load.

To achieve this goal, the authors also subdivided the virtual environment in rectangular **cells** – similar to the microcells –, where the number of servers is much smaller than the number of cells. The cells are grouped into **regions** – or macrocells – and each region is managed by one server (Figure 3.2). Each server handles all the interactions between avatars located in the region assigned to it. It receives the inputs of the players controlling these avatars and sends back to them the up-to-date game state.

Two cells are called adjacent (or neighbors) if they share a border. Similarly, two regions – and the servers assigned to them – are called adjacent (neighbors) if there is a pair of adjacent cells, each of which belonging to one of the two regions. The workload

of a cell was defined as the number of avatars present in that cell. The authors assumed that all players receive state updates of the same length and in the same frequency, so that the burden of processing (computing and communication) that a cell requires from a server is proportional to the number of users in that cell. The workload of a region and its designated server is defined as the sum of the individual workloads of the cells which form the region. Each server periodically evaluates its workload and exchange this information with its neighbors. They have assumed, too, that these servers are connected through a high-speed network. Thus, the overhead to exchange workload information among neighbors is limited and considered negligible compared to other costs of the distribution. For the same reason, they also assumed as negligible the overhead of communication between servers in different regions when players are interacting.

The main aspect of the solution proposed by the authors was the use of local information (the server that initiated the balancing process and its neighbors), rather than global information (involving all servers in the balance). When a server is overloaded, it searches for lightly loaded servers close to it in the overlay network. A breadth-first lookup for lightly loaded neighbor servers – as proposed in (DUONG; ZHOU, 2003) – causes a small overhead, but it may not solve the problem efficiently in a few steps, as overloaded servers tend to be adjacent. The global approach, in turn, is able to divide the workload in the most balanced way possible, but its complexity may become too high.

The solution proposed by the authors is then by involving only a subset of servers, such that its cardinality varies according to the need (if the neighbors of the server which triggered the load balancing are also overburdened, more servers are selected). However, this set is formed in a more clever way than by simply using a breadth-first search. The algorithm used to find the servers is described in the next section.

3.1.2.1 Selection of a local server group to balance the load

A server starts the balancing when the load assigned to it is beyond its capacity. This server selects a number of other servers to get involved with the distribution. First, it chooses the least loaded server among its neighbors and sends a request that he participates in the load balancing. The chosen server rejects the request if it is already involved in another balancing group, otherwise it responds to the server with the load information of its own neighbors. If the selected neighbor server is unable to absorb all the extra workload of the initiating server, the selection is performed again among the neighbors not only of the overloaded server, but also the neighbors of the already selected servers. The selection continues until the workload of the first server can be absorbed – that is, the workload of all selected servers becomes smaller than a certain limit.

Figure 3.3 illustrates the operation of the algorithm. All servers have the same capacity, each one being able to handle 100 users. First, the initiator server, S_6 is inserted into SELECTED and its neighbors (S_2, S_5, S_7 and S_{10}) are added to CANDIDATES (Figure 3.3(a)). So S_7, which has the lowest workload among the servers in CANDIDATES, is selected and invited to participate in the load distribution. When S_7 sends to S_6 the workload information of its neighbors (S_3, S_6, S_8 and S_{11}), it is inserted into SELECTED and its neighbors, except S_6, are added to CANDIDATES (Figure 3.3(b)). Now, S_{11}, which has the lowest workload among servers in CANDIDATES, is selected and invited to participate in the load distribution. However, S_{11} rejects the invitation, because it is involved in another distribution, initiated by S_{12}. Thus, S_{11} is removed from CANDIDATES and S_{10} is selected because it now has the lowest workload among all servers in CANDIDATES (Figure 3.3(c)). This process continues until the average workload is under a pre-defined

threshold (Figure 3.3(d) and Figure 3.3(e)).

After forming the local server set, the initiating server performs a load rebalancing in such group, using some graph partitioning algorithm. To map the virtual environment to a graph, each cell is represented by a vertex, whose weight equals to the number of avatars in that cell; and every two vertices which represent adjacent cells are connected by an edge.

3.2 Proposed load balancing scheme

In the previous section it was presented some existing works regarding load balancing in MMOGs when using multiple servers to provide the support network. The load balancing scheme presented here is based on some of the principles in the literature. One of them is the division of the virtual environment in microcells, for later grouping in macrocells. This is a relatively simple way of addressing the issue of the avatars' movement dynamics through the game world, by transferring the microcells dynamically according to need.

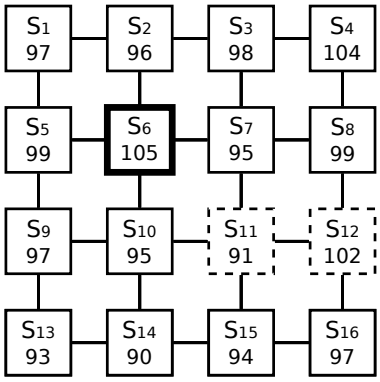
It will also be used the idea of balancing based only on local information – each server, when needing to reduce its workload, selects only a few other servers to join a local load rebalancing. Thus, the complexity of balancing can be greatly reduced because it will not be necessary for all servers in the game to exchange messages among themselves every time that any one of them is overloaded.

To define their load balancing approach, the authors consider that, usually, the traffic generated by the players is not simply linear, but square for each cluster of players. Such misunderstanding can generate considerable differences between the actual load of each server and the load estimated by the balancing algorithm. Another point to consider is the overhead, both in the delay of sending messages, as in the use of bandwidth of the servers, when players connected to different servers are interacting with each other. Some works assume that the servers are all in the same high-speed and low latency local-area network, and therefore overhead is negligible. However, when considering a geographically distributed server system, this assumption cannot be made. This overhead must be taken into account, no matter which load balancing algorithm is being used.

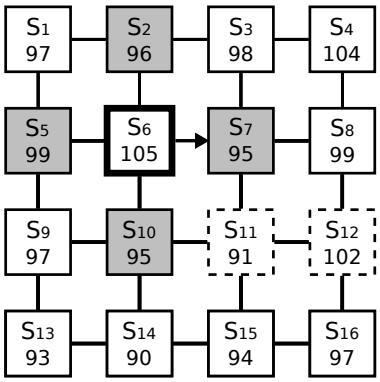
Another important point is that the main criterion the authors use when balancing the load of MMOG servers is the bandwidth, and not processing power. Several games, such as Age of Empires (MICROSOFT, 1997), include simulations of virtual environments with hundreds or thousands of entities, which are performed smoothly on current personal computers. However, if this game was multiplayer and each of these entities was controlled by a different player on the Internet, it would most likely generate a traffic amount which would hardly be supported by a domestic connection (FENG, 2007).

Moreover, the upload bandwidth must be taken into account, much more than download. This happens for two reasons: first, the usage of the download bandwidth of each server node grows linearly with the number of players connected to it, while the usage of the upload bandwidth may have a quadratic growth, getting quickly overwhelmed; second, domestic connections – which are majority in volunteer peer-to-peer systems – usually have an upload bandwidth much lower than the download one.

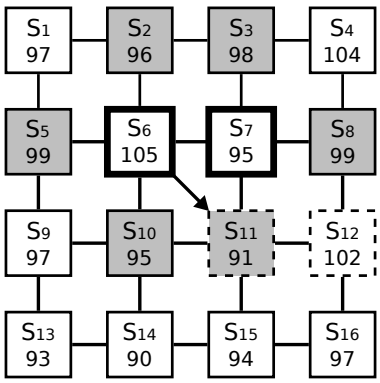
There are also other issues which must not be overlooked. One of them is that the server system is probably not homogenous – considering that it is geographically distributed with, likely, different connections to the Internet. Therefore, one cannot assume that the servers have the same amount of cpu power or network bandwidth. In the next



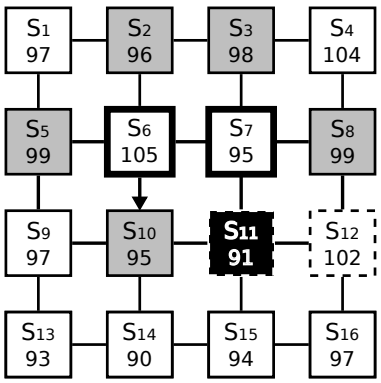
SELECTED = {}
CANDIDATES = {}



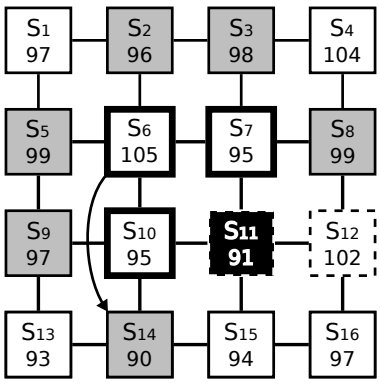
SELECTED = {S6}
CANDIDATES = {S2, S5, S7, S10}
(a)



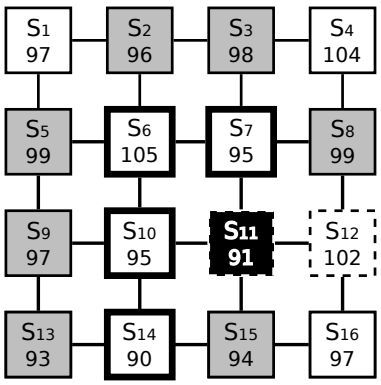
SELECTED = {S6, S7}
CANDIDATES = {S2, S3, S5, S8, S10, S11}
(b)



SELECTED = {S6, S7}
CANDIDATES = {S2, S3, S5, S8, S10}
(c)



SELECTED = {S6, S7, S10}
CANDIDATES = {S2, S3, S5, S8, S9, S14}
(d)



SELECTED = {S6, S7, S10, S14}
CANDIDATES ={S2, S3, S5, S8, S9, S13, S15}
(e)

Figure 3.3: Selecting the group of servers for local rebalance

section, it is made some definitions that will be necessary to specify the load balancing scheme proposed by the (BEZERRA; GEYER, 2009).

3.2.1 Definitions

The authors also use the idea of mapping the virtual environment on a graph. The graph will then be partitioned to distribute the workload of the game between the different servers. It is necessary first to define some terms which will be used on the proposed algorithms.

- **Server**: here, server is defined as a node belonging to the distributed system to serve the game. Each server can be assigned a single region;
- **Server power**: the server power, $p(S)$ is a numerical value proportional to the server's upload bandwidth;
- **Server power fraction**: given a set of servers $Servers = \{S_1, S_2, ..., S_n\}$, the power fraction of a server S, $frac_p(S)$, is equal to its power divided by the summed power of all servers in *Servers*:

$$frac_p(S) = \frac{p(S)}{\sum_{i=1}^{n} p(S_i)}$$

- **System power**: the total power of the system, P_{total}, equals the sum of the powers of the n servers which form it:

$$P_{total} = \sum_{i=1}^{n} p(Si)$$

- **Cell**: similar to the microcells, it is considered here the environment being divided into small cells, each one with fixed size and position. If two cells share a border, they are said to be **adjacent** or **neighbors**;
- **Region**: the cells are grouped, forming what is called regions. Usually these areas are contiguous, although in some cases the subgraph that represents them may be disconnected, resulting in the presence of cells isolated from each other. Each region is assigned to a server, and only one, and $s(R)$ is the server associated to the region R. It may be referred throughout the text to region's "power", which in fact refers to the power of the server associated with that region, i.e. $p(s(R))$;
- **Relevance**: the relevance of an avatar A_j to another one, A_i, determines the frequency of updates of the state of A_j the server should send to the player controlling A_i (BEZERRA; CECIN; GEYER, 2008). It may be represented by the function $R(A_i, A_j)$;
- **Avatar's weight**: to each avatar there are various other entities (here, only avatars are considered) of the game, each with a frequency of state updates that need to be sent to the player who controls that avatar. Thus, for each avatar A , its individual weight – or of upload bandwidth that the server uses to send state updates to its player – $w_a(A)$ depends on which other entities are relevant to it, and how much. Let $\{A_1, A_2, ..., A_t\}$ be the set of all avatars in the virtual environment, we have:

$$w_a(A) = \sum_{i=1}^{t} R(A, Ai)$$

- **Cell's weight**: here, the total weight of a cell (or the use of upload bandwidth of its server) will be equal to the sum of the individual weights of the avatars in it. Consider cell C, where the avatars $\{A_1, A_2, ..., A_n\}$ are present. Also, consider that the avatars $\{A_1, A_2, ..., A_t\}$ are all the avatars in the virtual environment. The weight of this cell, $w_c(C)$, is:

$$w_c(C) = \sum_{i=1}^{n} w(Ai) = \sum_{i=1}^{n} \sum_{j=1}^{t} R(A_i, A_j)$$

- **Region's weight**: the weight of a region is the sum of the weights of the cells that compose it. Let R be the region formed by $\{C_1, C_2, ..., C_p\}$. The weight of R is:

$$w_r(R) = \sum_{i=1}^{p} w_c(Ci)$$

- **Region's weight fraction**: given a set of regions $Regions = \{R_1, R_2, ..., R_n\}$, the weight fraction of R, relative to $Regions$, is:

$$frac_r(R) = \frac{w_r(R)}{\sum_{i=1}^{n} w_r(R_i)}$$

- **Region's resource usage**: fraction that indicates how much of the power of the server of that region is being used. It is defined by:

$$u(s(R)) = \frac{w_r(R)}{p(s(R))}$$

- **World weight**: the total weight of the game, W_{total}, will be used as a parameter for the partitioning of the virtual environment. It is defined as the sum of the weights of all cells. Let $\{C_1, C_2, ..., C_w\}$ be the set of all cells in which the game world is divided, we have:

$$W_{total} = \sum_{i=1}^{w} w_c(Ci)$$

- **System usage**: fraction that indicates how much resources of the system as a whole is being used. It is defined by:

$$U_{total} = \frac{W_{total}}{P_{total}}$$

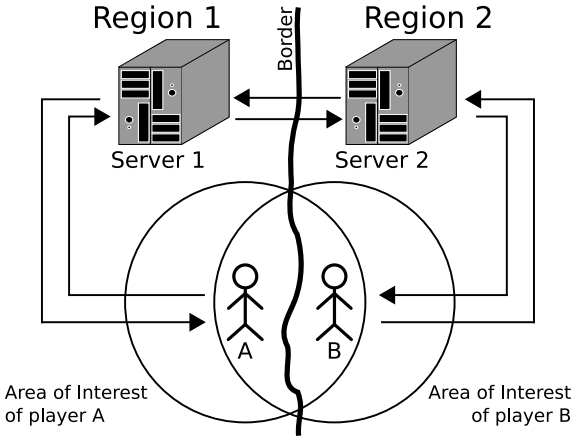


Figure 3.4: Players interacting across a border between regions

- **Cell interaction**: the interaction between two cells is equal to the sum of all interactions between pairs of avatars where each one of them is located in one of these cells. The interaction between cells C_i and C_j is given by:

$$Int_c(C_i, C_j) = \sum_{i=1}^{m} \sum_{j=1}^{n} R(A_i, A_j),$$
$$\text{where } A_i \text{ is in } C_i \text{ and } A_j \text{ is in } C_j.$$

- **Overhead between two regions**: if there is only one server and one region, comprising the entire virtual environment of the game, the use of the upload bandwidth of the server will be proportional to W_{total}. However, due to its distribution among various servers, there is the problem of having players from different regions interacting with each other very close to the border between the regions (Figure 3.4). Because of that, each state update of these players' avatars will be sent twice. For example, let A_i be the avatar of the player P_i, connected to the server S_i, and A_j the avatar of the player P_j, connected to the server S_j. In order for P_i to interact with P_j, it is necessary that S_i sends the state of A_i to S_j, which then forwards to P_j. The same happens on the other way around. The overhead between regions R_i and R_j is equal, therefore, to the sum of interactions between pairs of cells where each one of them is in one of these regions. If R_i and R_j have respectively m and n cells, we have that the interaction – or overhead – between them is given by:

$$Int_r(R_i, R_j) = \sum_{i=1}^{m} \sum_{j=1}^{n} Int_c(C_i, C_j),$$
$$\text{where } C_i \in R_i \text{ e } C_j \in R_j.$$

- **Total Overhead**: the total overhead on the server system is calculated as the sum of overheads between each pair of regions. So we have:

$$OverHead = \sum_{i} \sum_{j, j \neq i} Int_r(Ri, Rj)$$

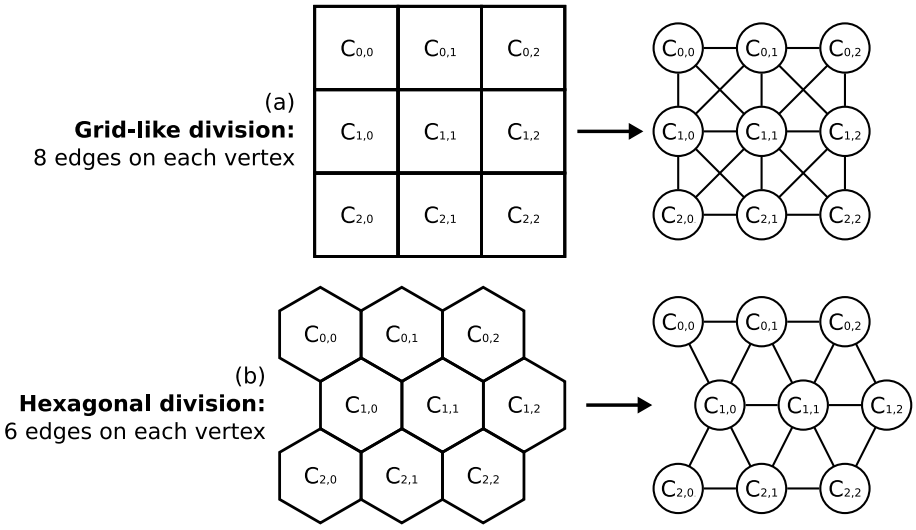


Figure 3.5: Mapping of the virtual environment on a graph

As the concepts needed to understand the proposed load balancing scheme have been defined, now it will be described how the virtual environment is mapped on a weighted graph, which will then be partitioned. Let $GW = (V,E)$ be a graph that represents the game world, where V is the set of vertices and E is the set of edges connecting vertices. Each component of this graph, and what it represents, is described below:

- **Vertex**: each vertex in the graph represents a cell in the virtual environment;
- **Edge**: each edge in the graph connects two vertices that represent adjacent cells;
- **Partition**: each partition of the graph GW – a subset of the vertices of GW, plus the edges that connect them – represents a region;
- **Vertex weight**: the weight of each vertex is equal to the weight of the cell that it represents;
- **Edge weight**: the weight of the edge connecting two vertices is equal to the interaction between the cells represented by them;
- **Partition weight**: the weight of a partition is equal to the sum of the weights of its vertices, i.e. the weight of the region that it represents;
- **Edge-cut**: the edge-cut in a partitioning is equal to the sum of the weights of all the edges which connect vertices from different partitions. This value is equal to the sum of the overheads between each pair of regions. Thus, the edge-cut of the graph GW is equal to the total overhead on the server system.

Figure 3.5(a) illustrates how the mapping is done with square cells, while figure 3.5(b) shows how it would be with hexagonal cells. The objective of the balancing scheme presented here is to assign to each server a weight proportional to its capacity, reducing as much as possible the edge-cut of the graph that represents the virtual environment and, consequently, reduce the overhead inherent to the distribution of the game on multiple servers. Although this is an NP-complete problem (FEDER et al., 1999), efficient heuristics will be used to reduce this overhead. In the following sections the algorithms proposed by the authors will be presented.

3.2.2 Proposed algorithms

It is considered that an initial division of the virtual environment has already been made. Each server should then check regularly if there is an imbalance and trigger the algorithm. Although the overhead resulting from the distribution of the virtual environment is part of the workload on servers, there is no way to know it beforehand without executing the repartitioning first. For this reason, the "weight" to be distributed does not include this extra overhead.

When there is an unbalanced region, it is selected a local group of regions, similar to what was shown in section 3.1.2, but with some changes (section 3.2.2.1). After this selection, it will be used an algorithm whose parameters are only the loads of the cells and their interactions, for such data is available prior to distribute. Finally, with the regions already balanced, the algorithm of Kernighan and Lin (KERNIGHAN; LIN, 1970) will be used to refine the partitioning, reducing the edge-cut and, thus, the overhead, while keeping the balance.

The proposed scheme is then divided into three phases:

1. Select the group of local regions;
2. Balance these regions, assigning to each one a weight which is proportional to the power of its server;
3. Refine the partitioning, reducing the overhead.

A decision to this balancing scheme is that a region R is considered overloaded only when the use of its resources is greater than the total use of the system, also considering a certain tolerance, tol, to avoid constant rebalancing. Thus, the server starts the balancing of R when, and only when, $u(s(R)) > U_{total} \times tol$. Thus, even if the system as a whole is overburdened, a similar quality of game will be observed among the different regions, dividing the excessive weight between all servers fairly. What can be done when $U_{total} > 1$ is gradually reduce the amount of information sent in each state update, leaving for the application the task of extrapolating the missing information based on previous updates.

Another important aspect is that each server always has an associated region. What might happen would be a region being empty – without any cells – and its server not participating in the game. This is useful when the total capacity of the server system is much greater than the total load of the game, or $P_{total} \gg W_{total}$. In this case, the introduction of more servers would only increase the communication overhead of the system, without improving its quality – except when introduced to provide fault tolerance.

The algorithms were developed oriented to regions, instead of servers, in order to be more legible, because of the constant transfers of cells. Moreover, it becomes easier to extend, in the future, the balancing model used here to allow more than one server managing the same region. The proposed algorithms are described as follows. The one in section 3.2.2.1 is for phase 1, the sections 3.2.2.2 to 3.2.2.5 are alternatives for phase 2 and the algorithm in section 3.2.2.6 is the refinement of phase 3.

3.2.2.1 Local regions selection

The algorithm for selection of regions (Algorithm 1) aims to form a set of regions such that the average usage of resources of the servers of these regions is below a certain limit. Starting from the server that has started the balancing, its neighboring regions with the least usage of resources are added. When the average usage is less than 1, or less than

U_{total} (line 5), the selection ends and phase 2 begins, with the region set as input. These two conditions are justified because there are two possibilities: $U_{total} \leq 1$ and $U_{total} > 1$.

In the case when $U_{total} \leq 1$, there is sufficient power in the system so that all servers have a usage smaller than 100%. Thus, regions are added to the group until all the servers involved are using fewer resources than they have. However, when $U_{total} > 1$, there is no way to all servers be using less than 100% of its resources at the same time. Thus, it is sufficient that all servers are similarly overburdened, and that some kind of adjustment is made, which will probably be a reduction in the information sent to players in each state update. Although this approach seems to lead the system to an inconsistent state, due to the lack of available bandwidth, the U_{total} is calculated based on the theoretical value of a region's weight. In practice, the system as a whole will take two measures: first, the frequency at which state updates are sent to players is diminished and, second, it will deny access to new players who try to join the game. Denying access when the game is overpopulated with players is the usual policy adopted by some MMOGs.

If even after all the neighbors, and the neighbors of the neighbors and so on, are selected, the criterion is not met, empty regions – belonging to idle servers – will be inserted in the group (line 8) because the overhead of interaction between regions introduced by them is justified by the need for more resources.

Algorithm 1 Local regions selection

1: $local\_group \leftarrow \{R\}$
2: $local\_weight \leftarrow w_r(R)$
3: $local\_capacity \leftarrow p(s(R))$
4: $average\_usage \leftarrow \frac{local\_weight}{local\_capacity}$
5: **while** $average\_usage > max(1, U_{total})$ **do**
6: **if** there is any not selected region neighbor to one of $local\_group$ **then**
7: $R \leftarrow$ not selected region neighbor to one of $local\_group$, with smallest $u(s(R))$
8: **else if** there is any empty region **then**
9: $R \leftarrow$ empty region with highest $p(s(R))$
10: **else**
11: stop. no more regions to select.
12: **end if**
13: $local\_weight \leftarrow local\_weight + w_r(R)$
14: $local\_capacity \leftarrow local\_capacity + p(s(R))$
15: $average\_usage \leftarrow \frac{local\_weight}{local\_capacity}$
16: $local\_group \leftarrow local\_group \cup \{R\}$
17: **end while**
18: run phase 2 passing $local\_group$ as input.

3.2.2.2 ProGReGA

ProGReGA, or proportional greedy region growing algorithm, seeks to allocate the heaviest cells to the regions managed by the most powerful servers. As input, the algorithm receives a list of the regions to balance. Details are shown in Algorithm 2.

Like said before, it is passed as input a list of the regions whose load will be rebalanced. This makes possible the use of this algorithm both in local and global scope, just by choosing between passing some regions or all regions of the environment. The distribution is based on information from that set of regions, whose total weight and total

Algorithm 2 ProGReGA

1: $weight\_to\_divide \leftarrow 0$
2: $free\_capacity \leftarrow 0$
3: **for** each region R in *region_list* **do**
4: $weight\_to\_divide \leftarrow weight\_to\_divide + w_r(R)$
5: $free\_capacity \leftarrow free\_capacity + p(s(R))$
6: temporarily free all cells from R
7: **end for**
8: sort *region_list* in decreasing $p(s(R))$ order
9: **for** each region R in *region_list* **do**
10: $weight\_share \leftarrow weight\_to\_divide \times \frac{p(s(R))}{free\_capacity}$
11: **while** $w_r(R) < weight\_share$ **do**
12: **if** there is any cell from R neighboring a free cell **then**
13: $R \leftarrow R \cup \{$neighbor free cell with the highest $Int_c(C)\}$
14: **else if** there is any free cell **then**
15: $R \leftarrow R \cup \{$heaviest free cell$\}$
16: **else**
17: stop. no more free cells.
18: **end if**
19: **end while**
20: **end for**

power are calculated in lines 1 to 7. To be redistributed later, all cells associated with these regions are released (line 6).

To provide a partitioning which is balanced, proportional and with low edge-cut since the second phase of the balancing, the regions are sorted in decreasing order of server power (line 8). The ProGReGA then runs through this list, seeking to assign heavier cells to more powerful servers.

In line 10, it is calculated the weight share for each region, considering the total weight that is being divided and the total power of the servers of those regions. The server power fraction of a region, $\frac{p(s(R))}{free\_capacity}$ should be the same fraction of weight that must be attributed to it. Even if this weight is greater than the power of that server, resulting in an overload, all the servers are similarly overloaded, satisfying the criterion of balance that has been defined. The condition for the end of the allocation of new cells in a region is that its weight is greater than or equal to its weight share.

It is important to notice that the while condition (line 11) may lead to the assignment of a cell whose weight is higher than the weight share left on the region. However, it is very unlikely to assign to a region its exact weight share, so it is preferable to surpass this value, guaranteeing that every cell will be assigned to some region. Also, assigning to a region a weight which is higher than its calculated share does not imply that it will become overloaded. First, the weight share of a region is not equal to the power of its server. When there are enough resources, the region's weight share will be smaller than its capacity. Furthermore, as ProGReGA is a greedy algorithm, the heaviest cells will be assigned first, to the most powerful servers.

The criterion to choose a cell to include in the region is to make the heaviest edges on the graph *GW* connect vertices in the same partition, reducing the edge-cut and, thus, the overhead. In each step, it is selected the cell which not only is adjacent to a cell already

present in the region, but whose edge connecting them is the heaviest possible. This algorithm is based on a principle similar to the one used in GGGP (KARYPIS; KUMAR, 1998), a greedy algorithm for graph partitioning. The objective of GGGP is to split a graph in two partitions of the same weight, while reducing the edge-cut with the heaviest edge heuristics. Figure 3.6 illustrates the steps of growth of a region until it reaches its load share.

In the example of Figure 3.6, there are two servers, S_1 and S_2, where $p(S_1) = 30$ and $p(S_2) = 18$. The total weight of the environment being repartitioned is $W_{total} = 32$. For the division to be proportional to the capacity of each server, the weights assigned to S_1 and S_2 are 20 and 12, respectively. The selection starts with the vertex of weight 6 (free cell with the highest weight) and, after that, at each step the vertex connected by the heaviest edge is added to the partition. The selected edges and the vertices belonging to the new partition are highlighted.

In the first step of the cycle starting in line 11, if the region does not have any cell yet, ProGReGA gets the heaviest free cell (line 15). The same occurs when a region is compressed between the borders of other regions and has no free neighbor, getting cells from somewhere else (Figure 3.7). This may generate fragmented regions and possibly increase the overhead of the game. However, this happens more often in the last steps of the distribution, when most of the cells would be already allocated to some region. Because the algorithm is greedy, when it reaches that stage of its execution, the free cells would probably the lightest cells of the environment, causing little overhead.

3.2.2.3 ProGReGA-KH

A possible undesirable effect of the ProGReGA algorithm is that by releasing all the cells and redistributing them, it might happen that one or more regions completely change their place, causing several players to disconnect from their servers and reconnect to a new one. Trying to reduce the likelihood of such event, the authors propose a variation of the algorithm, called **ProGReGA-KH** (proportional greedy region growing algorithm keeping heaviest cell). This new algorithm (shown in Algorithm 3) is similar to the original version, except that each region maintains its heaviest cell (lines 6 and 8), from which a region similar to the previous one can be formed, so that several players will not need to migrate to other server. However, to keep one of the cells of each region, it might be preventing a better balancing to occur. Also, the fixation of that cell might cause fragmented regions, increasing the overhead.

3.2.2.4 ProGReGA-KF

Another way of trying to minimize the migration of players between servers because of the rebalancing is the **ProGReGA-KF**, or proportional greedy region growing algorithm keeping usage fraction (Algorithm 4). In this algorithm, each region will gradatively release its cells in increasing order of weight, until its weight fraction is less than or equal to the power fraction of its server. Thus, the heaviest cells remain on the same server and, therefore, most players do not need to migrate. After that, the cells that were released are redistributed among the regions with lowest server resource usage (line 13). The disadvantage of this algorithm is, as in ProGReGA-KH, the possibility of fragmenting the regions, with many isolated cells, increasing the overhead.

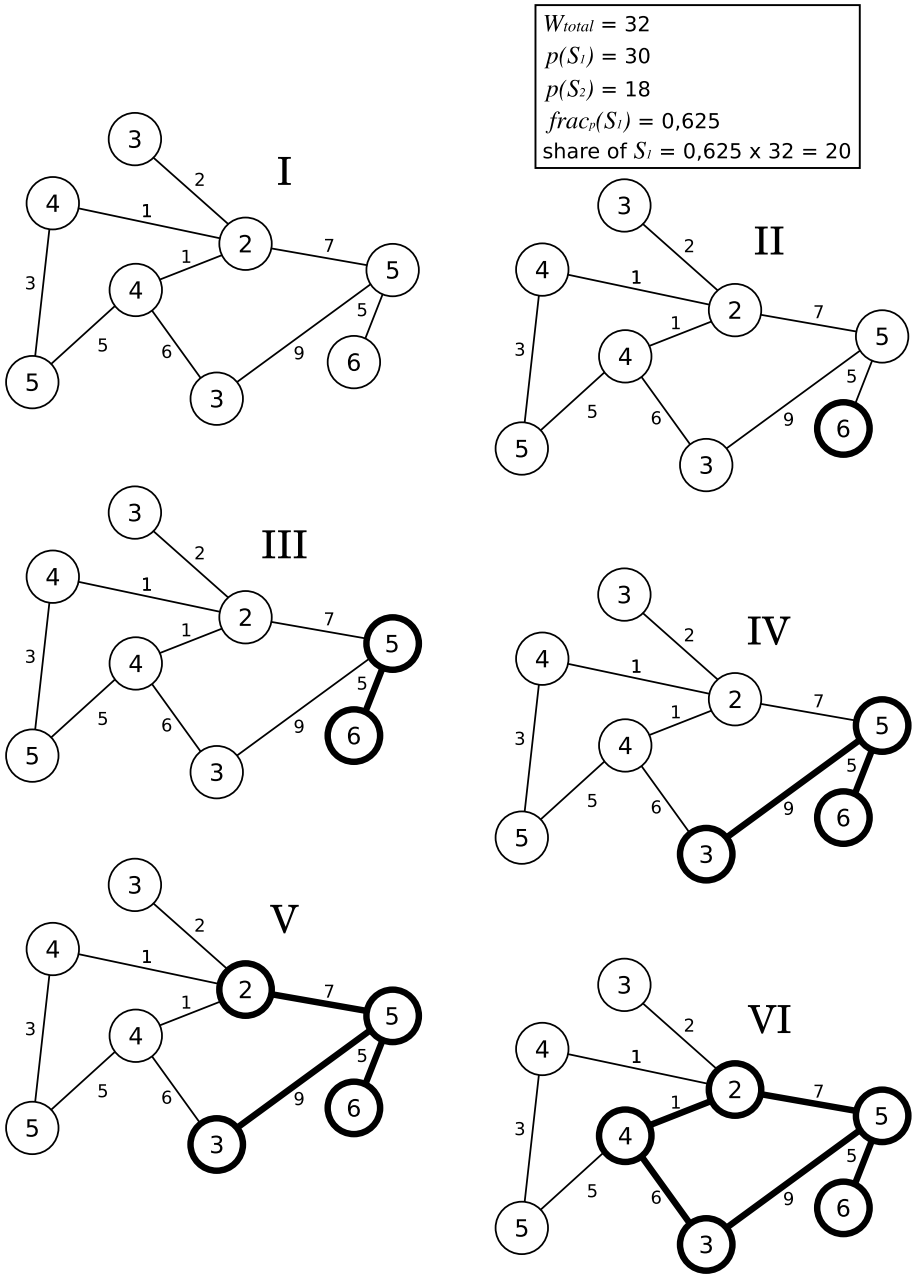


Figure 3.6: Growth of a partition (region) with ProGReGA

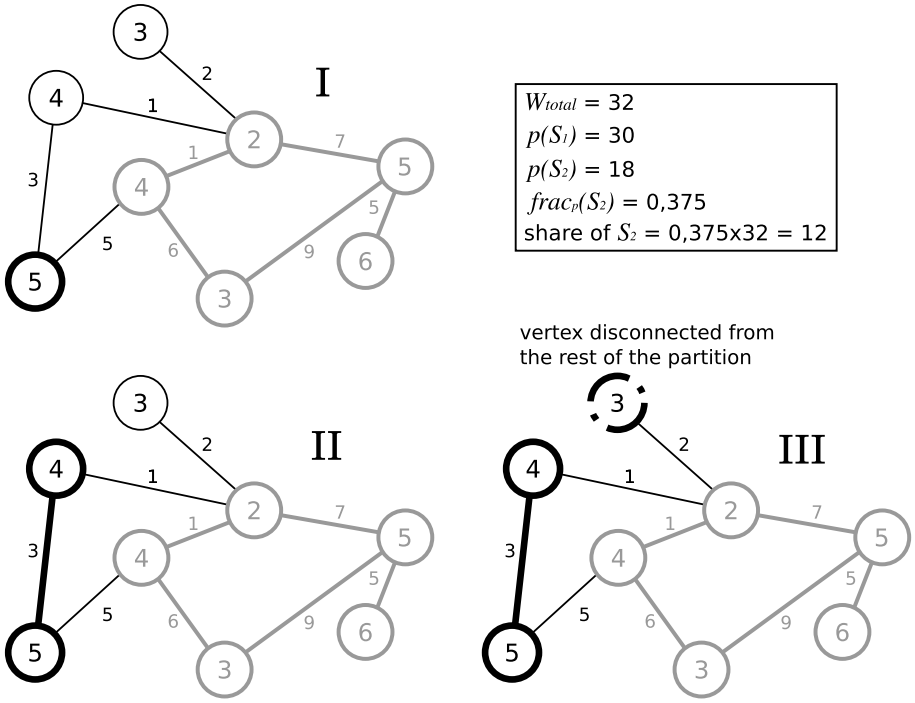


Figure 3.7: Formation of disconnected partition (fragmented region)

Algorithm 3 ProGReGA-KH

```
weight_to_divide ← 0
free_capacity ← 0
for each R in region_list do
    weight_to_divide ← weight_to_divide + w_r(R)
    free_capacity ← free_capacity + p(s(R))
    c ← heaviest cell from R
    temporarily free all cells from R
    R ← R ∪ {c}
end for
sort region_list in decreasing p(s(R)) order
for each region R in region_list do
    weight_share ← weight_to_divide × p(s(R)) / free_capacity
    while w_r(R) < weight_share do
        if there is any cell from R neighboring a free cell then
            R ← R ∪ {neighbor free cell with the highest Int_c(C)}
        else if there is any free cell then
            R ← R ∪ {heaviest free cell}
        else
            stop. no more free cells.
        end if
    end while
end for
```

Algorithm 4 ProGReGA-KF

1: $weight\_to\_divide \leftarrow 0$
2: $free\_capacity \leftarrow 0$
3: **for** each region R in *region_list* **do**
4: $weight\_to\_divide \leftarrow weight\_to\_divide + w_r(R)$
5: $free\_capacity \leftarrow free\_capacity + p(s(R))$
6: *cell_list* $\leftarrow$ list of cells from R in increasing order of weight
7: **while** $frac_r(R) > frac_p(s(R))$ **do**
8: $C \leftarrow$ first element from *cell_list*
9: remove C from R
10: remove C from *cell_list*
11: **end while**
12: **end for**
13: sort *region_list* in increasing order of $u(s(R))$
14: **for** each region R in *region_list* **do**
15: $weight\_share \leftarrow weight\_to\_divide \times \frac{p(s(R))}{free\_capacity}$
16: **while** $w_r(R) < weight\_share$ **do**
17: **if** there is any cell from R neighboring a free cell **then**
18: $R \leftarrow R \cup \{$neighbor free cell with the highest $Int_c(C)\}$
19: **else if** there is any free cell **then**
20: $R \leftarrow R \cup \{$heaviest free cell$\}$
21: **else**
22: stop. no more free cells.
23: **end if**
24: **end while**
25: **end for**

3.2.2.5 *BFBCT*

The **BFBCT** (best-fit based cell transference) was proposed by the authors as an alternative to ProGReGA and its variants. The objective of the algorithm is to check what is the weight excess in each region and transfer it to free regions whose capacity is the closest to that value. This is done by transferring cells whose weight is the closest to free capacity of the receiving region, observed two restrictions: first, the total weight transferred can not be larger than the free capacity of the destination region and, second, it should not be transferred a load greater than the necessary to eliminate the overload. The second restriction is justified because a weight transfer larger than necessary would probably result in a larger amount of migrating players. The exception to this rule is when a cell is heavier than the other, not for having more avatars, but because they are closer, with quadratic traffic growth between them. Algorithm 5 describes in detail the operation of BFBCT.

Algorithm 5 BFBCT

1: **for** each region R_i in *region_list* **do**
2: $weight\_to\_lose \leftarrow w_r(R_i) - W_{total} \times frac_p(s(R_i))$
3: $destination\_regions \leftarrow region\_list - \{R_i\}$
4: sort $destination\_regions$ in decreasing order of $u(s(R))$
5: **for** each region R_j in $destination\_regions$ **do**
6: $free\_capacity \leftarrow frac_p(s(R_j)) \times W_{total} - w_r(R_j)$
7: $weight\_to\_this\_region \leftarrow min(weight\_to\_lose, free\_capacity)$
8: **while** $weight\_to\_this\_region > 0$ **do**
9: **if** R_i has a cell C such that $w_c(C) \leq weight\_to\_this\_region$ **then**
10: $C \leftarrow$ cell from R_i with weight closest to, but not larger than, $weight\_to\_this\_region$
11: $R_i \leftarrow R_i - \{C\}$
12: $R_j \leftarrow R_j \cup \{C\}$
13: $weight\_to\_this\_region \leftarrow weight\_to\_this\_region - w_c(C)$
14: $weight\_to\_lose \leftarrow weight\_to\_lose - w_c(C)$
15: **else**
16: continue with next R_j.
17: **end if**
18: **end while**
19: **end for**
20: **end for**

3.2.2.6 *Refining with the Kernighan-Lin algorithm*

After balancing the load between servers in phase 2, all the servers will have a similar resource usage ratio. However, the algorithm in phase 2 cannot measure the final interaction between regions before repartitioning. This may lead to a partitioning that, although every server uses a similar percentage of its upload bandwidth to its clients, there may be a high inter-server communication overhead, causing a waste of resources of the server system.

To attenuate this problem, it was also proposed by (BEZERRA; GEYER, 2009) to use the algorithm of Kernighan and Lin (KERNIGHAN; LIN, 1970). This algorithm receives as input the graph that represents the virtual environment. Given a pair of regions

(partitions), Kernighan-Lin searches for pairs of cells (vertices) that, when exchanged between their regions, the overhead (edge-cut) is reduced.

In the case of a virtual environment distributed among various regions, generally more than two, Kernighan-Lin is run for each pair of regions. Moreover, after each swap performed by the algorithm, the balance must be kept – otherwise, a completely unbalanced partitioning, with the lowest possible overhead, could be reached. The Kernighan-Lin algorithm is widely known in the area of distributed systems, so details will not be provided here. Consider only that it returns a value of $true$ if any change was made and $false$ otherwise. It runs for all pairs of regions. until it returns a value of false, indicating that no exchange will provide additional gain. Algorithm 6 shows how the Kernighan-Lin algorithm is called.

Algorithm 6 Kernighan-Lin

```
1:  swapped ← true
2:  while swapped = true do
3:      swapped ← false
4:      for each region R_i in region_list do
5:          for each region R_j in region_list do
6:              if Kernighan-Lin(R_i, R_j) = true then
7:                  swapped ← true
8:              end if
9:          end for
10:     end for
11: end while
```

3.3 Work Analysis

The technique presented by (BEZERRA; GEYER, 2009) is good because it reduces significantly – proved via simulations presented by the authors – inter-server communication overhead, saving up resources of the system. However, there are a few issues with the approach. The first problem is that the granularity of the approach is fixed and limited by the cell size. If a finer granularity is desired, smaller cells should be used, increasing proportionally the complexity of the algorithm, and the potential fragmentation of the regions as well. Actually, another problem not solved with this approach is the very likely region fragmentation, as a region might contain disconnected cells. These two problems are well solved with the technique presented in the next chapter.

4 A KD-TREE BASED APPROACH

In order to readjust the load distribution during the game, the load balancing algorithm must be dynamic. Some work has already been made in this direction, but with a geometric algorithm, more appropriate than those presented in the past chapters, it should be possible to reduce the distribution granularity without compromising the rebalancing time, or even reducing it. In this work, (BEZERRA; COMBA; GEYER, 2009) propose the use of a kd-tree for dividing the virtual environment of the game into regions, each of which being designated to one of the servers. The split coordinates of the regions are adjusted dynamically according to the distribution of avatars in the virtual environment.

4.1 Related Work

As presented in the past two chapters of this review, different authors have tried to address the problem of partitioning the virtual environment in MMOGs for distribution among multiple servers. Generally, there is a static division into cells of fixed size and position. The cells are then grouped into regions, and each region is delegated to one of the servers. When one of them is overwhelmed, it seeks other servers, which can absorb part of the load. This is done by distributing one or more cells of the overloaded server to other servers. (AHMED; SHIRMOHAMMADI, 2008) and (BEZERRA; GEYER, 2009) were presented as examples of this kind of approach.

Although these approaches works with fair results, there is a serious limitation on the distribution granularity they can achieve. If a finer granularity is desired, it is necessary to use very small cells, increasing the number of vertices in the graph that represents the virtual environment and, consequently, the time required to perform the balancing. Besides, the control message containing the list of cells designated to each server also becomes longer. Thus, it may be better to use another approach to perform the partitioning of the virtual environment, possibly using a more suitable data structure, such as the kd-tree (BENTLEY, 1975).

This kind of data structure is generally used in computer graphics. However, as in MMOGs there is geometric information – such as the position of the avatars in the environment –, space partitioning trees can be used. Moreover, we cand find in the literature techniques for keeping the partitions defined by the tree with a similar “load”. In (LUQUE; COMBA; FREITAS, 2005), for example, it is sought to reduce the time needed to calculate the collisions between pairs of objects moving through space. The authors propose the use of a BSP (binary space partitioning) tree to distribute the objects in the scene (Figure 4.1). Obviously, if each object of a pair is completely inserted in a different partition, they do not collide and there is no need to perform a more complex test for this pair. Assuming an initial division, it is proposed by the authors a dynamic readjust-

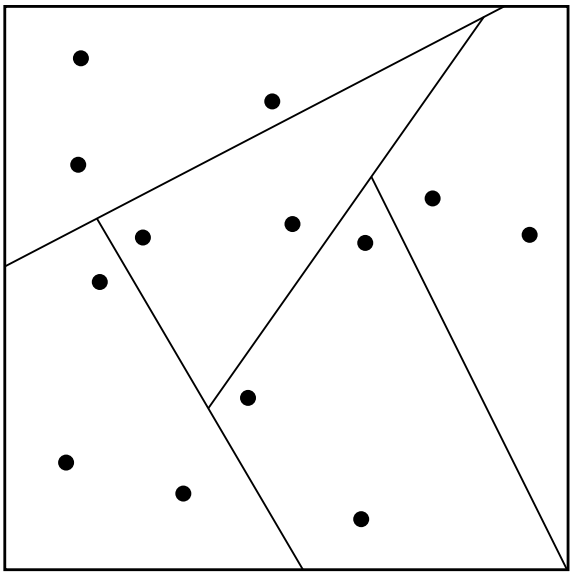


Figure 4.1: Space partitioning using a BSP tree

ment of the tree as objects move, balancing their distribution on the leaf-nodes of the tree and, therefore, minimizing the time required to perform the collision detection. Some of the ideas proposed by the authors may be used in the context of load balancing between servers in MMOGs.

4.2 Proposed approach

The load balancing approach proposed by (BEZERRA; COMBA; GEYER, 2009) is based on two criteria: first, the system should be considered as potentially heterogeneous (i.e. every server may have a different amount of resources) and, second, the load on each server is *not* proportional to the number of players connected to it, but to the amount of bandwidth required to send state update messages to them.

This choice is due to the fact that every player sends commands to the server at a constant rate, so the number of messages received by the server per unit time grows linearly with the number of players, whereas the number of state update messages sent by the server may be quadratic, in the worst case.

To divide the environment of the game into regions, it is proposed the utilization of a data structure known as kd-tree. The vast majority of MMOGs, such as World of Warcraft (BLIZZARD, 2004), Ragnarok (GRAVITY, 2001) and Lineage II (NCSOFT, 2003), despite having three-dimensional graphics, the simulated world – cities, forests, swamps and points of interest in general – in these games is mapped in two dimensions. Therefore, the authors proposed to use a kd-tree with k = 2.

Each node of the tree represents a region of the space and, moreover, in this node it is stored a split coordinate. Each one of the two children of that node represents a subdivision of the region represented by the parent node, and one of them represents the sub-region before the split coordinate and the other one, the sub-region containing points whose coordinates are greater than or equal to the split coordinate. The split axis (in the case of two dimensions, the axes x and y) of the coordinate stored alternates for every level of the tree – if the first level nodes store x-coordinates, the second level nodes store y-coordinates, the third level goes back to x-coordinates and so on. Every leaf node also

represents a region of the space, but it does not store any split coordinate. Instead, it stores a list of the avatars present in that region. Finally, each leaf node is associated to a server of the game. When a server is overloaded, it triggers the load balancing, which uses the kd-tree to readjust the split coordinates that define its region, reducing the amount of content managed by it.

Every node of the tree also stores two other values: capacity and load of the subtree. The load of a non-leaf node is equal to the sum of the load of its children. Similarly, the capacity of a non-leaf node is equal to the sum of the capacity of its children nodes. For the leaf nodes, these values are the same of the server associated to each one of them. The tree root stores, therefore, the total weight of the game and the total capacity of the server system.

In the following sections, it will be described the construction of the tree, the calculation of the load associated with each server and the proposed balancing algorithm.

4.2.1 Building the kd-tree

To make an initial space division, it is constructed a balanced kd-tree. For this, it is used the recursive function shown in Algorithm 7 to create the tree.

Algorithm 7 node::build_tree(id, level, num_servers)

if id + $2^{level} \geq num\_servers$ **then**
  $left\_child \leftarrow right\_child \leftarrow NIL$;
  return;
else
  $left\_child \leftarrow$ new_node();
  $left\_child.parent \leftarrow$ **this**;
  $right\_child \leftarrow$ new_node();
  $right\_child.parent \leftarrow$ **this**;
  $left\_child$.build_tree$(id, level+1, num_servers)$;
  $right\_child$.build_tree$(id+2^{level}, level+1, num_servers)$;
end if

In Algorithm 7, the *id* value is used to calculate whether each node has children or not and, in the leaf nodes, it determines the server associated to the region represented by each leaf of the tree. The purpose of this is to create a balanced tree, where the number of leaf nodes on each of the two sub-trees of any node differs, in the maximum, by one. In Figure 4.2 (a), we have a full kd-tree formed with this simple algorithm and, in Figure 4.2 (b), an incomplete kd-tree with six-leaf nodes. As we can see, every node of the tree in (b) has two sub-trees whose number of leaf nodes differs by one in the worst case.

4.2.2 Calculating the load of avatars and tree nodes

The definition of the split coordinate for every non-leaf node of the tree depends on how the avatars will be distributed among the regions. An initial idea might be to distribute the players among servers, so that the number of players on each server is proportional to the bandwidth of that server. To calculate the split coordinate, it would be enough to simply sort the avatars in an array along the axis used (x or y) by the tree node to split the space and, then, calculate the index in the vector, such that the number of elements before this index is proportional to the capacity of the left child and the number of elements from that index to the end of the array is proportional to the capacity of the

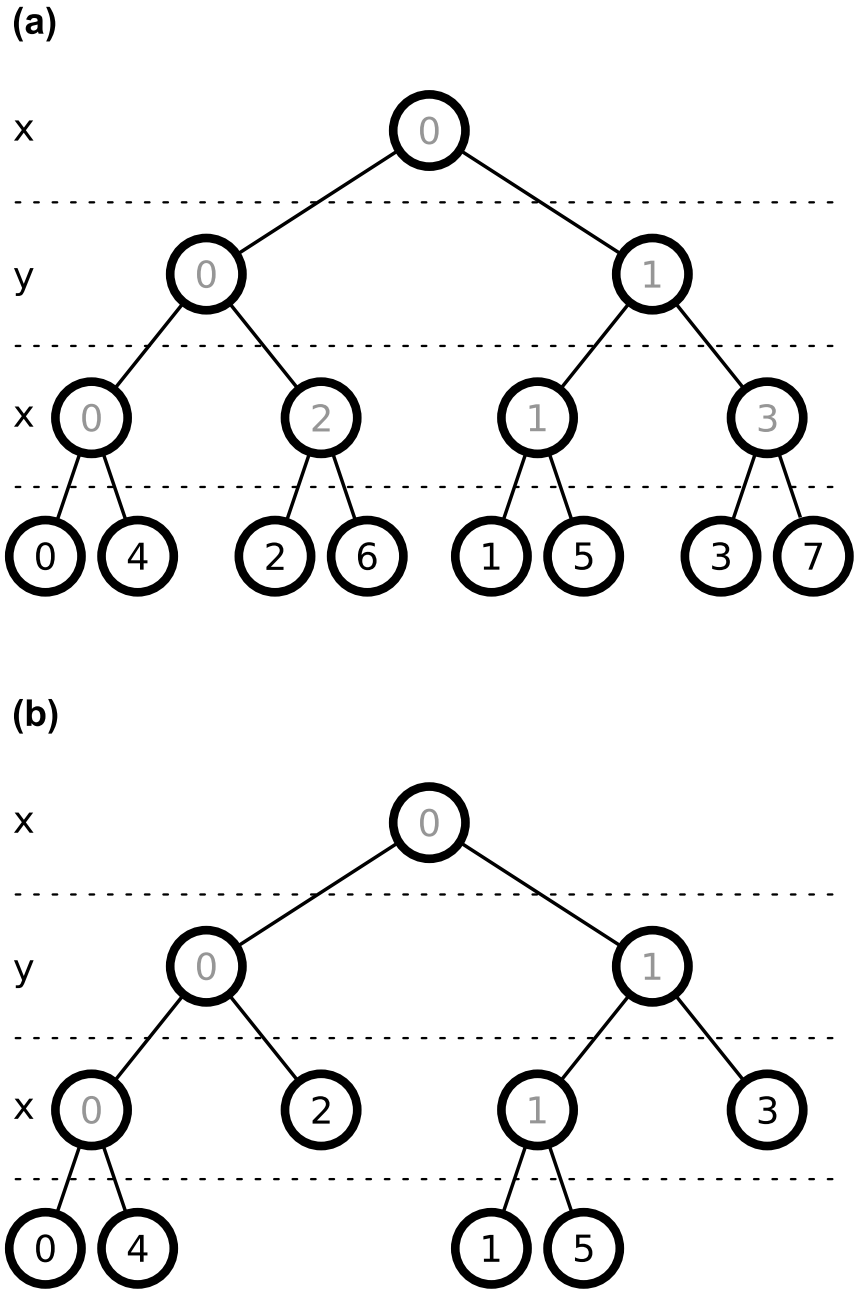


Figure 4.2: Balanced kd-trees built with the described algorithm

right child (Figure 4.3). The complexity of this operation is $O(nlogn)$, due to the sorting of avatars.

However, this distribution is not optimal, for the load imposed by the players depends on how they are interacting with one another. For example, if the avatars of two players are distant from each other, there will be probably no interaction between them and, therefore, the server will need only to update every one of them about the outcome of his own actions – for these, the growth in the number of messages is linear with the number of players. On the other hand, if the avatars are close to each other, each player should be updated not only about the outcome of his own actions but also about the actions of every other player – in this case, the number of messages may grow quadratically with the number of players (Figure 4.4). For this reason, it is not sufficient only to consider the number of players to divide them among the servers.

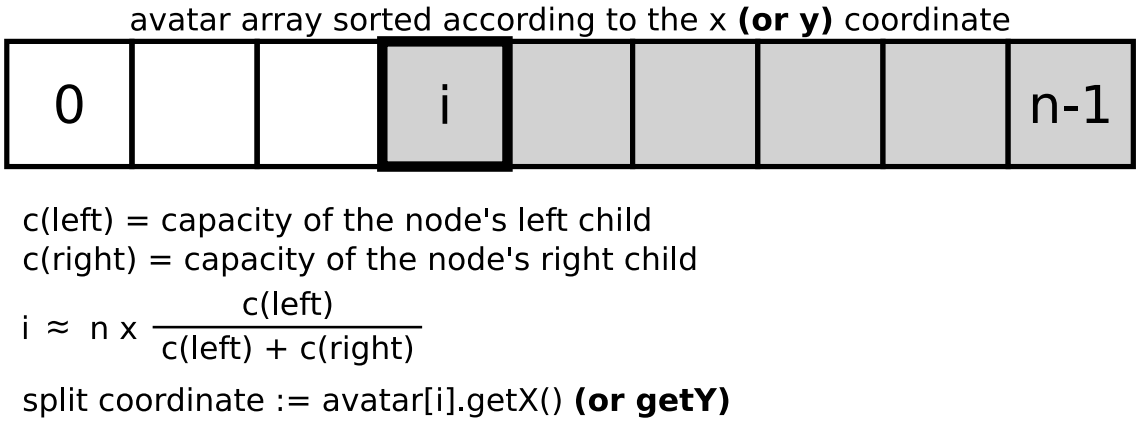


Figure 4.3: A load splitting considering only the number os avatars

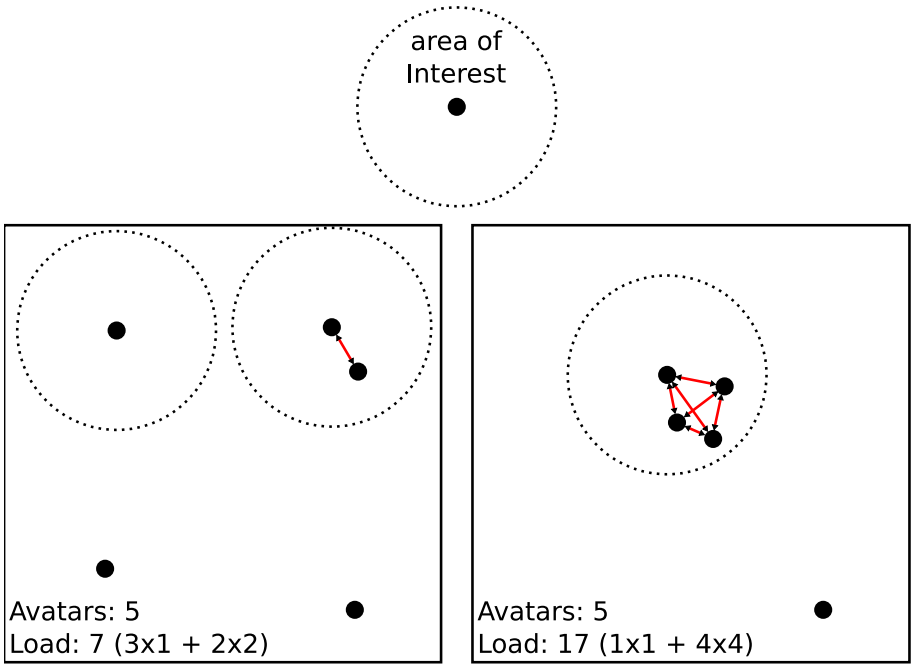


Figure 4.4: Relation between avatars and load

A more appropriate way to divide the avatars is by considering the load imposed by each one of them on the server. A brute-force method for calculating the loads would be to get the distance separating each pair of avatars and, based on their interaction, calculate the number of messages that each player should receive by unit of time. This approach has complexity $O(n^2)$. However, if the avatars are sorted according to their coordinates on the axis used to divide the space in the kd-tree, this calculation may be performed in less time.

For this, two nested loops are used to sweep the avatars array, where each of the avatars contains a *load* variable initialized with zero. As the vector is sorted, the inner loop may start from an index before which it is known that no avatar a_j has relevance to that being referenced in the outer loop, a_i. It is used a variable *begin*, with initial value of zero: if the coordinate of a_j is smaller than that of a_i, with a difference greater than the maximum view range of the avatars, the variable *begin* is incremented. For every a_j which is at a distance smaller than the maximum view range, the *load* of a_i is increased according to the relevance of a_j to a_i. When the inner loop reaches an avatar a_j, such that its coordinate is greater than that of a_i, with a difference greater than the view range, the outer loop moves immediately to the next step, incrementing a_i and setting the value of a_j to that stored in *begin* (Figure 4.5).

Let $width$ be the length of the virtual environment along the axis used for the splitting; let also $radius$ be the maximum view range of the avatars, and n, the number of avatars. The number of relevance calculations, assuming that the avatars are uniformly distributed in the virtual environment is $O(m \times n)$, where m is the number of avatars compared in the internal loop, i.e. $m = \frac{2 \times radius \times n}{width}$. The complexity of sorting the avatars along one of the axes is $O(nlogn)$. Although it is still quadratic, the execution time is reduced significantly, depending on the size of the virtual environment and on the view range of the avatars. The algorithm could go further and sort each set of avatars a_j which are close (in one of the axes) to a_i according to the other axis and, again, perform a sweep eliminating those which are too far away, in both dimensions. The number of relevance calculations would be $O(p \times n)$, where p is the number of avatars close to a_i, considering the two axes of coordinates, i.e. $p = \frac{(2 \times radius)^2 \times n}{width \times height}$. In this case, *height* is the extension of the environment in the second axis taken as reference. Although there is a considerable reduction of the number of relevance calculations, it does not pay the time spent in sorting the sub-array of the avatars selected for each a_i. Adding up all the time spent on sort operations, it would be obtained a complexity of: $O(nlogn + n \times mlogm)$.

After calculating the load generated by each avatar, this value is used to define the load on each leaf node and, recursively, on the other nodes of the kd-tree. To each leaf node a server and a region of the virtual environment are assigned. The load of the leaf node is equal to the server's bandwidth used to send state updates to the players controlling the avatars located in its associated region. This way, the load of each leaf node is equal to the sum of the weights of the avatars located in the region represented by it.

4.2.3 Dynamic load balancing

Once the tree is built, each server is associated to a leaf node – which determines a region. All the state update messages to be sent to players whose avatars are located in a region must be sent by the corresponding server. When a server is overloaded, it may transfer part of the load assigned to it to some other server. To do this, the overloaded server collects some data from other servers and, using the kd-tree, it adjusts the split coordinates of the regions.

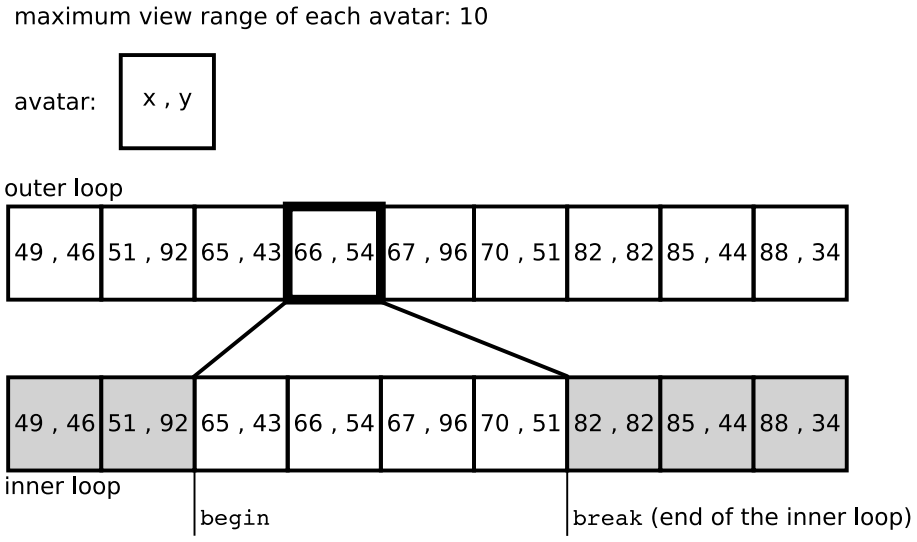


Figure 4.5: Sweep of the sorted array of avatars

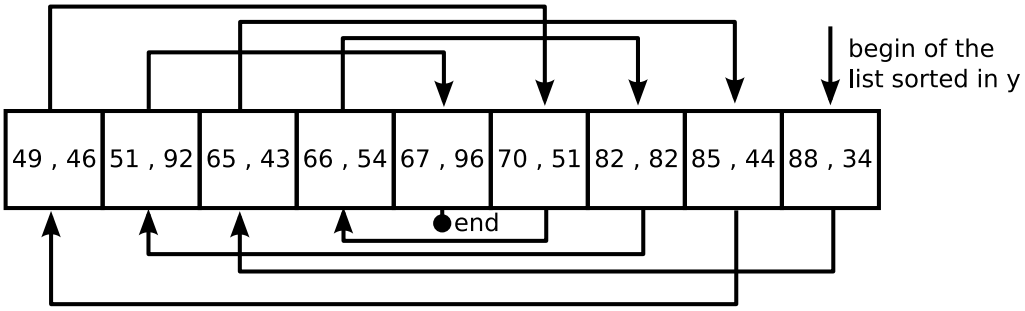


Figure 4.6: Avatar array sorted by x, containing a list sort by y

Every server maintains an array of the avatars located in the region managed by it, sorted according to the *x* coordinate. Also, each element of the array stores a pointer to another element, forming a chained list that is ordered according to the *y* coordinated of the avatars (Figure 4.6). By maintaining a local sorted avatar list on each server, the time required for balancing the load is somewhat reduced, for there will be no need for the server performing the rebalance to sort again the avatar lists sent by other servers. It will need only to merge all the avatars lists received from the other servers in an unique list, used to define the limits of the regions, what is done by changing the split coordinates which define the space partitions.

When the overloaded server initiates the rebalance, it runs an algorithm that traverses the kd-tree, beginning from the leaf node that defines its region and going one level up at each step until it finds an ancestor node with a capacity greater then or equal to the load. While this node is not found, the algorithm continues recursively up the tree until it reaches the root. For each node visited, a request for the information about all the avatars and the values of load and capacity is sent to the servers represented by the leaf nodes of the sub-tree to the left of that node (Figure 4.7). With these data, and its own list of avatars and values of load and capacity, the overloaded server can calculate the load and capacity of its ancestral node visited in the kd-tree, which are not known beforehand – these values are sent on-demand to save up some bandwidth of the servers and to keep the system scalable.

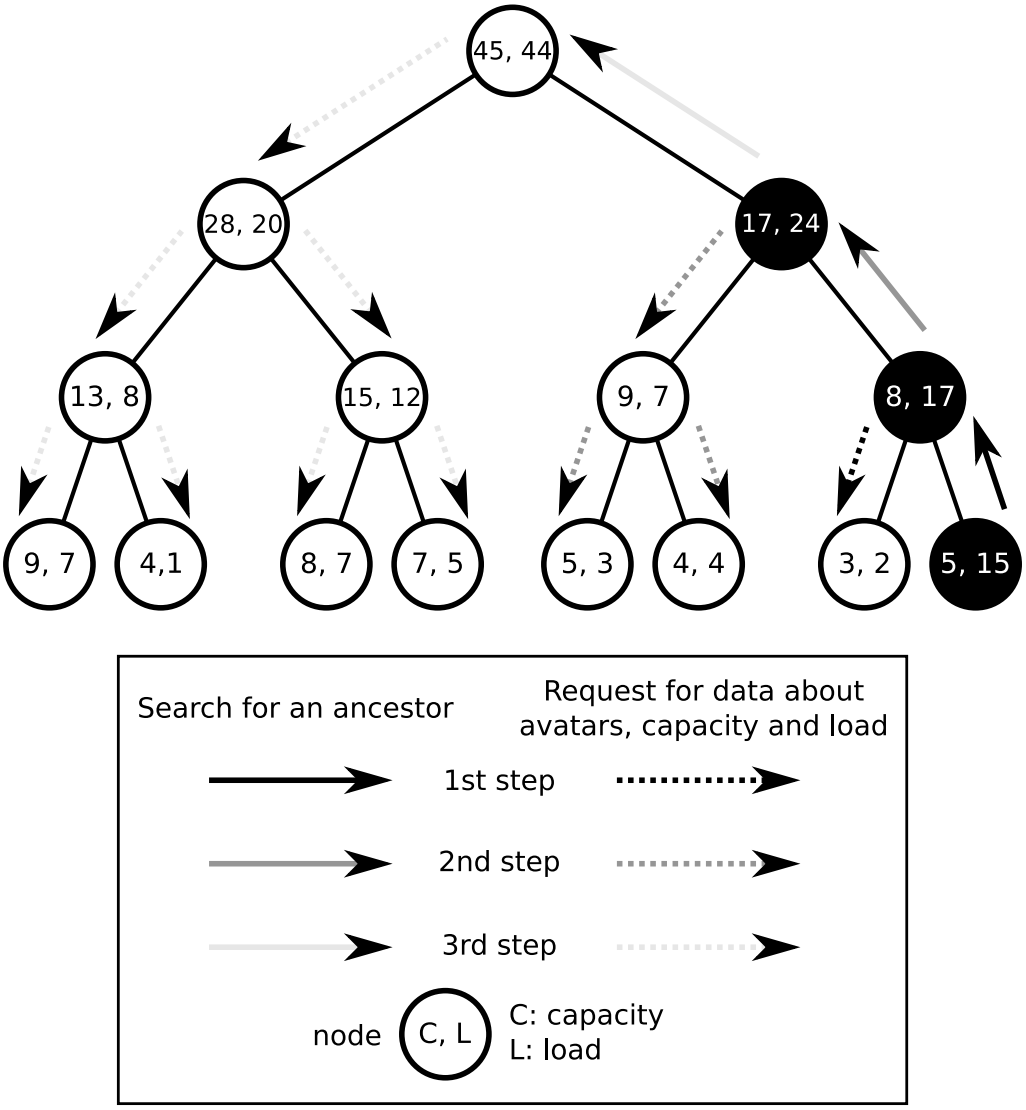
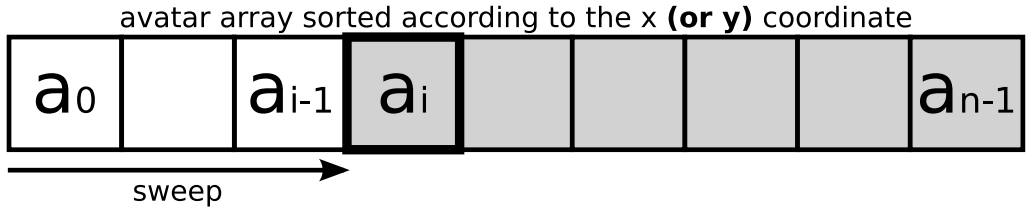


Figure 4.7: Search for an ancestor node with enough resources



c(left) = capacity of the node's left child
c(right) = capacity of the node's right child
split coordinate := avatar[i].getX() **(or getY)**, where i is such that:

$$\frac{\sum_{j=0}^{i-1} load(a_j)}{\sum_{j=0}^{n-1} load(a_j)} \approx \frac{c(left)}{c(left) + c(right)}$$

Figure 4.8: Division of an avatar list between two brother nodes

Reaching an ancestral node with capacity greater than or equal to the load – or the root of the tree, if no such node is found – the server that initiated the balance adjusts the split coordinates of the kd-tree nodes. For each node, it sets the split coordinate in a way such that the avatars are distributed according to the capacity of the node's children. For this, it is calculated the load fraction that should be assigned to each child node. The avatar list is then swept, stopping at the index i such that the total load of the avatars before i is approximately equal to the value defined as the load to be designated to the left child of the node whose split coordinate is being calculated (Figure 4.8). The children nodes have also, in turn, their split coordinates readjusted recursively, so that they are checked for validity – the split coordinate stored in a node must belong to the region defined by its ancestors in the kd-tree – and readjusted to follow the balance criteria defined.

As the avatar lists received from the other servers are already sorted along both axes, it is enough to merge these structures with the avatar list of the server which initiated the rebalance. Assuming that each server already calculated the weight of each avatar managed by it, the rebalance time is $O(nlogS)$, where n is the number of avatars in the game and S is the number of servers. The communication cost is $O(n)$, caused by the sending of data related to te n avatars. The merging of all avatar lists has $O(n)$ complexity, for the avatars were already sorted by the servers. At each level of the kd-tree, $O(n)$ avatars are swept in the worst case, in order to find the i index whose avatar's coordinate will be used to split the regions defined by each node of the tree (Figure 4.8). As this is a balanced tree with S leaf nodes, it has a height of $\lceil logS \rceil$.

4.3 Work analysis

In this chapter, we presented a work where (BEZERRA; COMBA; GEYER, 2009) propose the use of a kd-tree to partition the virtual environment of MMOGs and perform the load balancing of servers by recursively adjusting the split coordinates stored

in its nodes. One of the conclusions reached was that the use of kd-trees to make this partitioning allows a finer granularity of the load distribution, while the readjustment of the regions becomes simpler – by recursively traversing the tree – than the common approaches, based on cells and/or graph partitioning.

The finer granularity allows for a better balancing, so that the load assigned to each server is close to the ideal value that should be assigned to it. This better balance also helps to reduce the number of migrations, for it can perform less rebalancing operations to achieve the same distribution fairness. The fact that the regions defined by the kd-tree are necessarily contiguous also contribute to a lower number of migrations of players between servers.

It was also possible to use methods with a lower complexity for each rebalancing operation. This is due, first, to the reduction of the number of operations for calculating the relevance between pairs of avatars by sweeping a sorted avatar list and, secondly, to keeping at each server an avatar list already sorted in both dimensions, saving the time that would be spent on sorting the avatars when they were received by the server executing the rebalance.

This approach, however, has a serious limitation: the split axes are always parallel to the x and y axes. That may lead, sometimes, to a worse division of the players among servers, as each region is necessarily a rectangle with its sided aligned to the coordinates axes. Also, each server has an indivisible and contiguous region assigned to it. Although this may reduce the number of player migrations, a fragmented region could be of use sometimes, adding some flexibility to the world partitioning.

5 FINAL REMARKS

The works presented here represent some of what is the newest techniques in terms of load distribution in multiserver systems for massively multiplayer online games. They all take in account the most important resource in a massively multiplayer game server: network bandwith. Although processing power used is also a resource to be used by the game servers, certainly the greatest concern is with the bandwidth occupation. To reach a saturation level where a game server CPU is overloaded, it is required a number of players much higher than that which would saturate the network connection of an average desktop personal computer.

Besides, comments were made – that were disposed throughout the text and in the sections of "work analysis" – for each of the presented works, showing what looks good and what appears to be bad in each one of them. While there are some weaknesses in the papers presented, one can make use of several ideas proposed by them, in order to create a new, more refined approach to balance the load in a massively multiplayer games multiserver system.

5.1 Future work

Considering what has been shown in the works presented, a possible increment would be the use of a generic BSP tree. In that data structure – which is a more general case of the KD-tree – each node of the tree contains a linear equation that represents a straight line in the plane. This line, then, divides the region represented by that tree node in two sub-regions, each one represented by other tree nodes. These tree nodes, therefore, are children of the first one.

By using such a more generic space partitioning data structure, such as the described BSP tree, the load balancing algorithm may achieve an ever better distribution than that made with the KD-tree and the other techniques presented in this work. This structure, however, would cause a significantly higher rebalancing complexity, as each tree node defines an arbitrary line, represented as a linear equation, instead of a simple x or y coordinate.

Another possible improvement would be to create a tree whose number of leaf nodes is higher than the number of servers in the multiserver system. This way, it could be designated multiple leaf nodes – that is, multiple regions – to each server. With such an approach, an even finer and more fairer load balacing. However, to allow for the multiple node assignment for each server, a different criterion for the region distribution must be devised.

REFERENCES

AHMED, D.; SHIRMOHAMMADI, S. A Microcell Oriented Load Balancing Model for Collaborative Virtual Environments. In: IEEE CONFERENCE ON VIRTUAL ENVIRONMENTS, HUMAN-COMPUTER INTERFACES AND MEASUREMENT SYSTEMS, VECIMS, 2008, Istanbul, Turkey. **Proceedings...** Piscataway: NJ: IEEE, 2008. p.86–91.

BENTLEY, J. Multidimensional binary search trees used for associative searching. **Communications of the ACM**, [S.l.], v.18, n.9, p.517, 1975.

BEZERRA, C.; COMBA, J.; GEYER, C. **A fine granularity load balancing technique for MMOG servers using a kd-tree to partition the space**. 2009. 17–26p.

BEZERRA, C. E. B.; CECIN, F. R.; GEYER, C. F. R. A3: a novel interest management algorithm for distributed simulations of mmogs. In: IEEE/ACM INTERNATIONAL SYMPOSIUM ON DISTRIBUTED SIMULATION AND REAL-TIME APPLICATIONS, DS-RT, 12., 2008, Vancouver, Canada. **Proceedings...** Washington: DC: IEEE, 2008. p.35–42.

BEZERRA, C.; GEYER, C. A load balancing scheme for massively multiplayer online games. **Multimedia Tools and Applications**, [S.l.], v.45, n.1, p.263–289, 2009.

BLIZZARD. **World of Warcraft**. Available at: http://www.worldofwarcraft.com/.

CHEN, J.; WU, B.; DELAP, M.; KNUTSSON, B.; LU, H.; AMZA, C. **Locality aware dynamic load management for massively multiplayer games**. 2005. 289–300p.

CHEN, K.; HUANG, P.; LEI, C. Game traffic analysis: an mmorpg perspective. **Computer Networks**, [S.l.], v.50, n.16, p.3002–3023, 2006.

DE VLEESCHAUWER, B. et al. Dynamic microcell assignment for massively multiplayer online gaming. In: ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, NETGAMES, 4., 2005, Hawthorne, NY. **Proceedings...** New York: ACM, 2005. p.1–7.

DUONG, T.; ZHOU, S. **A dynamic load sharing algorithm for massively multiplayer online games**. 2003. 131–136p.

FEDER, T. et al. Complexity of graph partition problems. In: ACM SYMPOSIUM ON THEORY OF COMPUTING, STOC, 31., 1999, Atlanta, GA. **Proceedings...** New York: ACM, 1999. p.464–472.

FENG, W. **What's Next for Networked Games?** Available at: http://www.thefengs.com/wuchang/work/. (NetGames 2007 keynote talk, nov. 2007).

FIDUCCIA, C.; MATTHEYSES, R. A Linear-Time Heuristic for Improving Network Partitions. In: CONFERENCE ON DESIGN AUTOMATION, 19., 1982, Las Vegas, NV. **Proceedings...** New York: ACM, 1982. p.175–181.

GRAVITY. **Raganarök Online**. Available at: http://www.ragnarokonline.com/.

HENDRICKSON, B.; LELAND, R. An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations. **SIAM Journal on Scientific Computing**, Philadelphia, PA, USA, v.16, n.2, p.452–452, Mar. 1995.

KARYPIS, G.; KUMAR, V. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. **SIAM Journal on Scientific Computing**, Philadelphia, PA, USA, v.20, n.1, p.359–392, Aug. 1998.

KERNIGHAN, B.; LIN, S. An efficient heuristic procedure for partitioning graphs. **Bell System Technical Journal**, [S.l.], v.49, n.2, p.291–307, 1970.

LEE, K.; LEE, D. A scalable dynamic load distribution scheme for multi-server distributed virtual environment systems with highly-skewed user distribution. In: ACM SYMPOSIUM ON VIRTUAL REALITY SOFTWARE AND TECHNOLOGY, 2003, Osaka, Japan. **Proceedings...** New York: ACM, 2003. p.160–168.

LU, F.; PARKIN, S.; MORGAN, G. **Load balancing for massively multiplayer online games**. 2006.

LUQUE, R.; COMBA, J.; FREITAS, C. **Broad-phase collision detection using semi-adjusting BSP-trees**. 2005. 179–186p.

MICROSOFT. **Age of Empires**. Available at: http://www.microsoft.com/games/empires/.

NCSOFT. **Lineage II**. Available at: http://www.lineage2.com/.