Noname manuscript No.
(will be inserted by the editor)

A load balancing scheme for massively multiplayer online games

Carlos E. B. Bezerra · Cláudio F. R. Geyer

Received: date / Accepted: date

Abstract In a distributed MMOG (massively multiplayer online game) server architecture, the server nodes may become easily overloaded by the high demand from the players for state updates. Many works propose algorithms to distribute the load on the server nodes, but this load is usually defined as the number of players on each server, what is not an ideal measure. Also, the possible heterogeneity of the system is frequently overlooked. We propose a balancing scheme with two main goals: allocate load on server nodes proportionally to the 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. We also make a review of related works and some comparisons were made, where our approach performed better.

Keywords MMOGs · load balancing · distributed server · graph partitioning

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 [1], in the worst case (Figure 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. We consider here that the main resource to analyse is the available bandwidth, for this is the current bottleneck for MMOGs [2].

Carlos Eduardo Benevides Bezerra, Cláudio Fernando Resin Geyer
Universidade Federal do Rio Grande do Sul
Av. Bento Gonçalves, 9500, Porto Alegre, Brazil
Tel.: +55-51-84065986
Fax.: +55-51-33087308
E-mail: carlos.bezerra@inf.ufrgs.br, geyer@inf.ufrgs.br

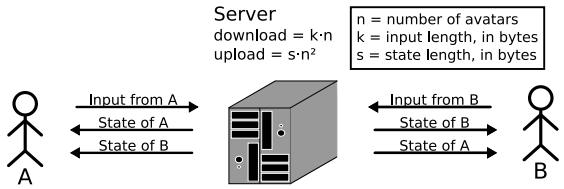


Fig. 1 Quadratic growth of traffic when avatars are close to each other

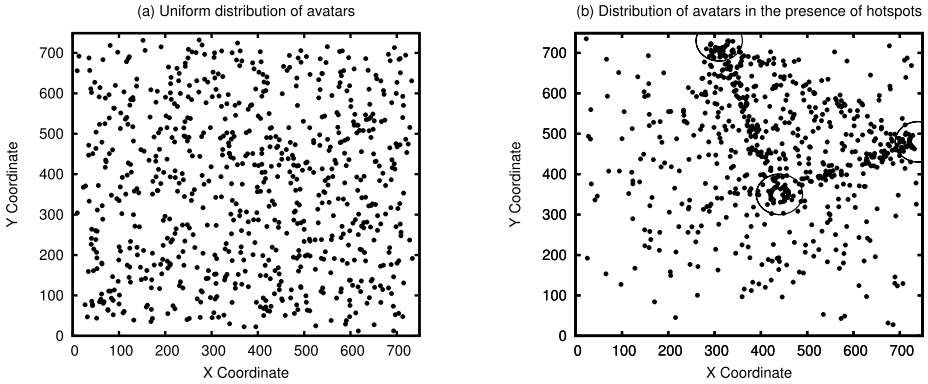


Fig. 2 Distribution of avatars with and without hotspots

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 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* [3], around which the players are more concentrated than in other regions of the virtual environment (Figure 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 [1] that, in a group of avatars who are neighbors of one another, the

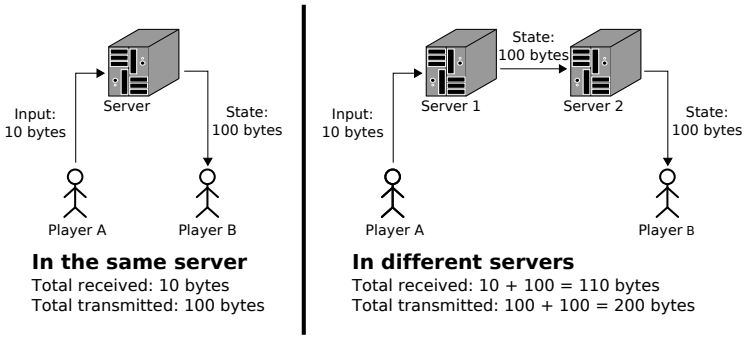


Fig. 3 Overhead caused by the interaction of players connected to different servers

rate of packets sent to each player is proportional to the number of avatars in that group (as in Fig. 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.

Differently from other works, we propose here a balancing scheme which considers the upload bandwidth occupation of the server as the load to distribute, what is done among servers with different levels of resources, also reducing the inter-server communication overhead by using a greedy graph partition growing algorithm.

2 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).

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 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 [4–7, 3] – trying to find a near optimal, yet fast, load balancing technique.

In [4], 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 ours. 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 [8] or [9], which may perform better. Finally, some of the algorithms proposed in [4] do not consider the communication overhead between servers.

It is proposed in [5] 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, in this work we are not concerned with cpu usage, but bandwidth occupation.

Chen et al. propose, in [6], 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 [7]. 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 we said 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 [8–11] could be used, for most of these algorithms were designed exactly to balance load and minimize dependence between nodes.

Ahmed and Shirmohammadi propose a microcell oriented load balancing model [3]. To balance the load, their algorithm, first, finds all clusters of microcells which are managed by the overloaded server. After that, the smallest cluster (in number of microcells) is selected and then, from that cluster, it is chosen the microcell with the lowest communication with other microcells managed by the same server. The chosen microcell is then transferred to the least loaded server. This process is repeated while the server is still overloaded and there are servers which can handle the extra load of the microcells being transferred. The authors define the load on a server, as opposed to other load balancing schemes, as the summing of the message rate of every player that a server must handle, which is a much more realistic measure than the number of players on the server. Also, their model for load balancing tries to minimize the inter-server communication, improving the overall system performance. Their model will be compared to ours, for it takes into consideration many of the aspects that we consider. The comparison will be performed through some simulations, which are described in section 4.

Next, some principles, defined in previous works, will be presented. Based on some of these principles, and on research and implementation carried out, it will be defined the load balancing scheme proposed here.

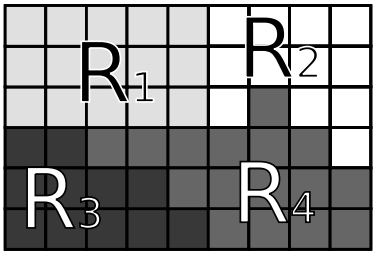


Fig. 4 Cells grouped into four regions (R_1, R_2, R_3 and R_4)

2.1 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 [4] 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 [4] that this overhead is compensated by a better distribution of the load between servers in the game.

2.2 Load balancing in local scope

In [12], 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 4). 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.

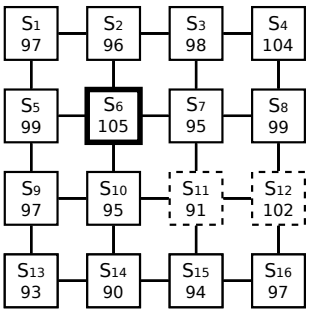
The main aspect of the solution proposed by the authors was the use of local information (the server that initiated the balancing process and its neighbors), rather than global information (involving all servers in the balance). When a server is overloaded, it searches for lightly loaded servers close to it in the overlay network. A breadth-first lookup for lightly loaded neighbor servers – as proposed in [7] – 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.

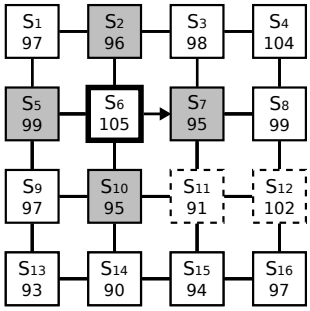
2.2.1 Selection of a local server group to balance the load

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.

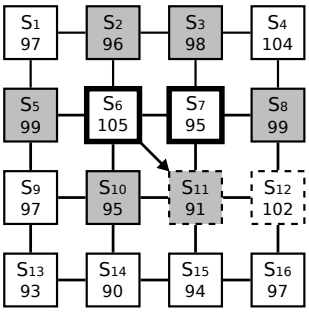
Figure 5 illustrates the operation of the algorithm. All servers have the same capacity, each one being able to handle 100 users. First, the initiator server, S_6 is inserted into SELECTED and its neighbors (S_2, S_5, S_7 and S_{10}) are added to CANDIDATES (Figure 5(a)). So S_7, which has the lowest 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 5(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 5(c)). This process continues until the average workload is under a pre-defined threshold (Figure 5(d) and Figure 5(e)).



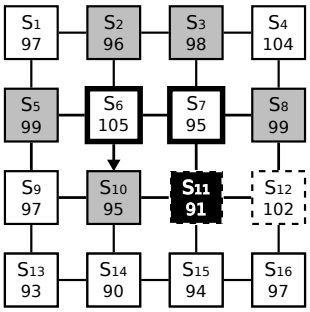
SELECTED = {}
CANDIDATES = {}



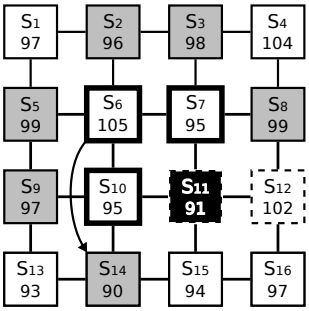
SELECTED = {S6}
CANDIDATES = {S2, S5, S7, S10}
(a)



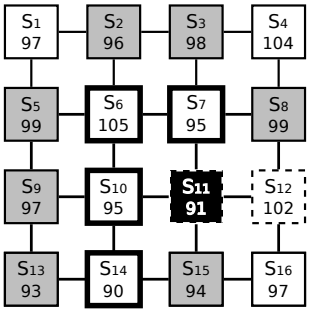
SELECTED = {S6, S7}
CANDIDATES = {S2, S3, S5, S8, S10, S11}
(b)



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)

Fig. 5 Selecting the group of servers for local rebalance

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 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 proposed in this work 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, we can greatly reduce the complexity of balancing because it will not be necessary that all servers in the game exchange messages among themselves every time that any one of them is overloaded.

To define our load balancing approach, we cannot ignore 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 we use when balancing the load of MMOG servers is the bandwidth, and not processing power. Several games, such as Age of Empires [13], 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 [2].

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 section, we make some definitions that will be necessary to specify our load balancing scheme.

3.1 Definitions

We 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 [14]. It may be represented by the function $R(A_i, A_j)$;
- **Avatar's weight**: to each avatar there are various other entities (here, we only consider avatars) 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:

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

– **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:

$$Int_c(C_i, C_j) = \sum_{i=1}^{m} \sum_{j=1}^{n} R(A_i, A_j),$$
where A_i is in C_i and A_j 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 6). 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

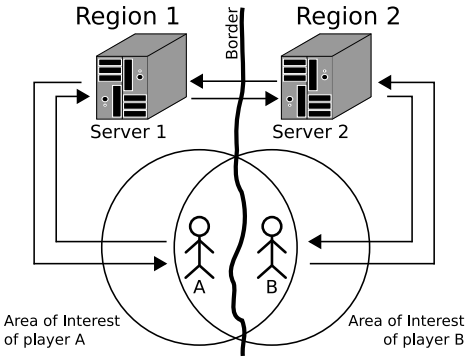


Fig. 6 Players interacting across a border between regions

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:

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

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.

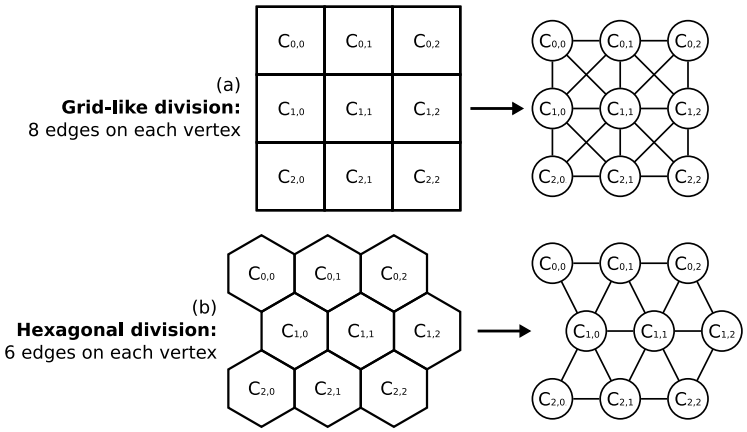


Fig. 7 Mapping of the virtual environment on a graph

Figure 7(a) illustrates how the mapping is done with square cells, while figure 7(b) shows how it would be with hexagonal cells. The objective of the balancing scheme proposed 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 [15], efficient heuristics will be used to reduce this overhead. In the following sections the algorithms proposed in this work will be presented.

3.2 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 2.2, but with some changes (section 3.2.1). 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 [8] 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 3.2.1 is for phase 1, the sections 3.2.2 to 3.2.5 are alternatives for phase 2 and the algorithm in section 3.2.6 is the refinement of phase 3.

3.2.1 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.

3.2.2 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.

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.

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**

Like we said, 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 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 10). 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 [10], 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 8 illustrates the steps of growth of a region until it reaches its load share.

In the example of Figure 8, 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 9). 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.

3.2.3 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. To try to reduce the likelihood of such event, we 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

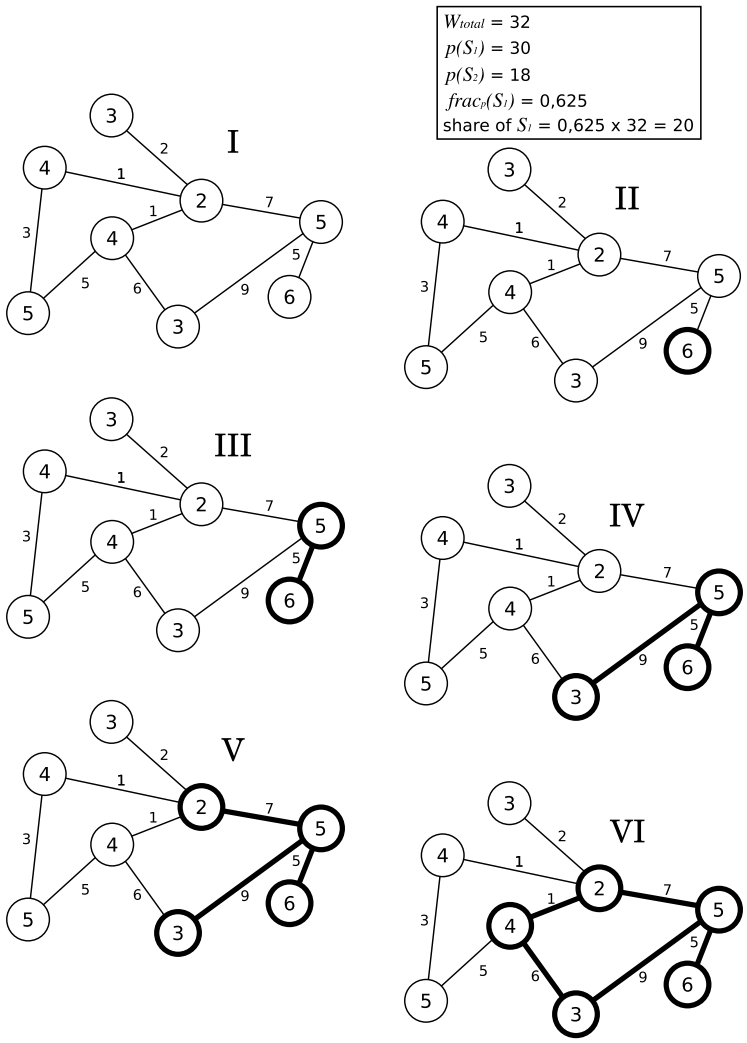


Fig. 8 Growth of a partition (region) with ProGReGA

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.

3.2.4 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,

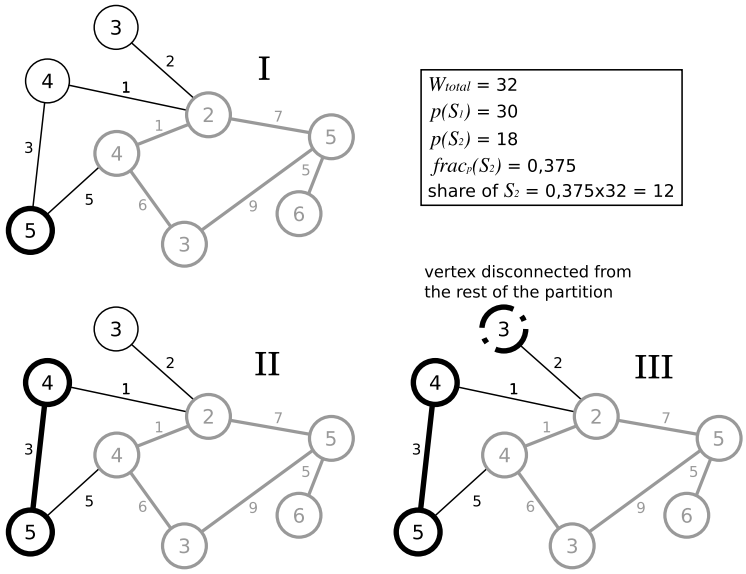


Fig. 9 Formation of disconnected partition (fragmented region)

Algorithm 3 ProGReGA-KH

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

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.

Algorithm 4 ProGReGA-KF

```
1: weight_to_divide ← 0
2: free_capacity ← 0
3: for each region R in region_list do
4:     weight_to_divide ← weight_to_divide + w_r(R)
5:     free_capacity ← free_capacity + p(s(R))
6:     cell_list ← list of cells from R in increasing order of weight
7:     while frac_r(R) > frac_p(s(R)) do
8:         C ← 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 ← weight_to_divide × 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 ← R ∪ {neighbor free cell with the highest Int_c(C)}
19:        else if there is any free cell then
20:            R ← R ∪ {heaviest free cell}
21:        else
22:            stop. no more free cells.
23:        end if
24:    end while
25: end for
```

3.2.5 BFBCT

The **BFBCT** (best-fit based cell transference) is proposed here 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, we should not transfer 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.

3.2.6 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 is proposed here to use the algorithm of Kernighan and Lin [8]. 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.

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**

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: $swapped \leftarrow$ **true**
2: **while** $swapped =$ **true do**
3: $swapped \leftarrow$ **false**
4: **for** each region R_i in $region\_list$ **do**
5: **for** each region R_j in $region\_list$ **do**
6: **if** $Kernighan\text{-}Lin(R_i, R_j) =$ **true then**
7: $swapped \leftarrow$ **true**
8: **end if**
9: **end for**
10: **end for**
11: **end while**

4 Simulations and results

To perform the simulation of the load balancing, a virtual environment was simulated with various avatars. At first, the avatars are distributed uniformly throughout the virtual environment. As the simulation begins, they start to move according to the random waypoint model [16]. However, the choice of which waypoint to go was not completely random.

Three hotspots were defined, and there was a probability for the avatar to choose one of these hotspots as its next waypoint. Although this may be not very realistic, its purpose was to force an uneven distribution of avatars in the virtual environment, putting to test the load balancing algorithms. Those who take account of the existence of hotspots form the regions on this basis, reducing the distribution overhead.

As we said in section 2, we also compare our algorithms to the one proposed by Ahmed and Shirmohammadi [3]. However, it is very important to observe that some changes have been made before simulating. First, the authors divide the virtual environment in hexagonal cells, and we simulate their algorithm on a grid-like division, with square cells. Moreover, as we consider an heterogeneous system, each server S_i presents its own message rate threshold, T_i^m. Nonetheless, we keep what we consider the core of their approach, which is the selection of the smallest cell cluster managed by the overloaded server, followed by the selection of the cell of that cluster which has the lowest interaction with other cells of the same cluster and, finally, the transference of the selected cell to the least loaded server.

The environment consisted of a two-dimensional space, divided into 225 cells, forming a matrix of order 15. There were 750 avatars. Each cell always belonged to some region, although it could be transferred to another region. There were eight servers ($S_1, S_2, ..., S_8$), each one of them associated to a region, each of which could have 0 to all 225 cells. The capacity of all servers was different, with $P(S_i) = i \times 20000$. Thus, it was possible to test whether the distribution obeyed the criterion of proportional load balancing. Each session of simulated game was 20 minutes long. The total weight of the game was purposely set in a way that it was greater than the total capacity of the system, forcing the triggering of the load balancing.

The evaluation of the load balancing algorithms was done in a more abstract level, taking into consideration the formulae defined in section 3.1. The results presented here are calculated using those definitions – the inter-server communication, for example, is the $OverHead$ defined in that section –, while the simulator software puts the avatars in movement, executing the load balancing algorithms as necessary, i.e. when the weight of a region violates the balance criteria defined in section 3.2.

In Figure 10, we can see that all the algorithms simulated met the proportionality criterion in load balancing. However, some of them introduced more overhead than the others. The reason for this is the fragmented regions. The lower the number of fragments the regions have, the lower is this overhead because there are fewer boundaries between regions.

Observe that the algorithm ProGReGA is the one with the lowest overhead of all, as it was designed precisely to create the most possible contiguous regions, searching cells connected by the highest interactions, to minimize the algorithm overhead. The BFBCT algorithm, however, was developed in order to distribute the load based on a best-fit allocation, seeking to balance it as much as possible, ignoring, in phase 2, the existence of interactions between cells and regions. For this reason, the BFBCT was the one which created most fragmentation in the regions and thus most overhead between the servers.

Figure 11 shows the change in the total overhead on the server system, depending on playing time. It was observed that the overhead generated by each balancing algorithm varies relatively little with time, and the ProGReGA is the one which gives the lowest overhead in any moment of the game, and BFBCT has the largest overhead of all, for almost the entire simulation. The algorithms ProGReGA-KH and ProGReGA-KF alternate with intermediate values of overhead.

Ahmed's algorithm, in turn, presented the second highest average overhead. The most probable reason for this is the choice of the least loaded server to receive a microcell being transferred from an overloaded server. Though this is an obvious choice in terms of load

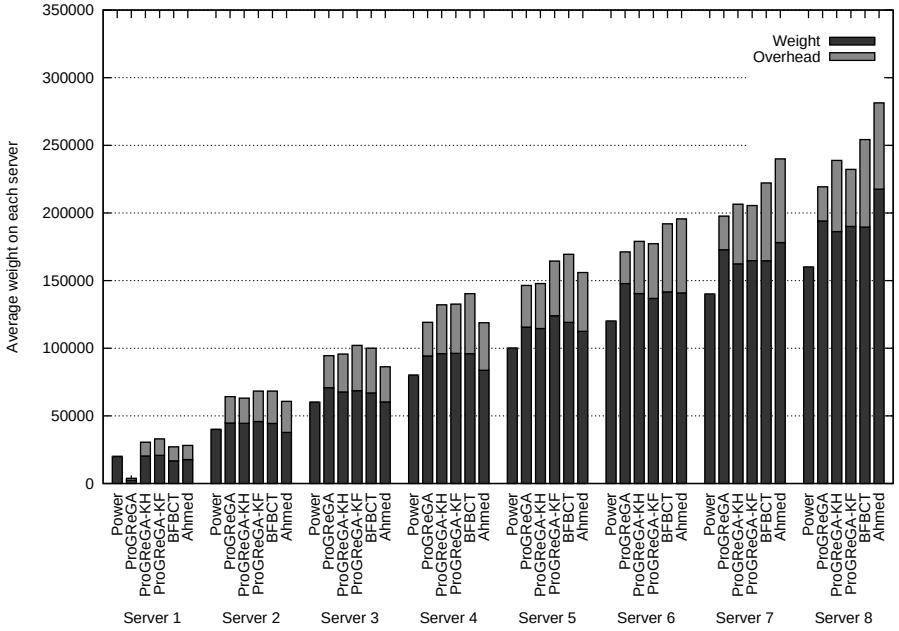


Fig. 10 Overall comparison of proposed load balancing algorithms

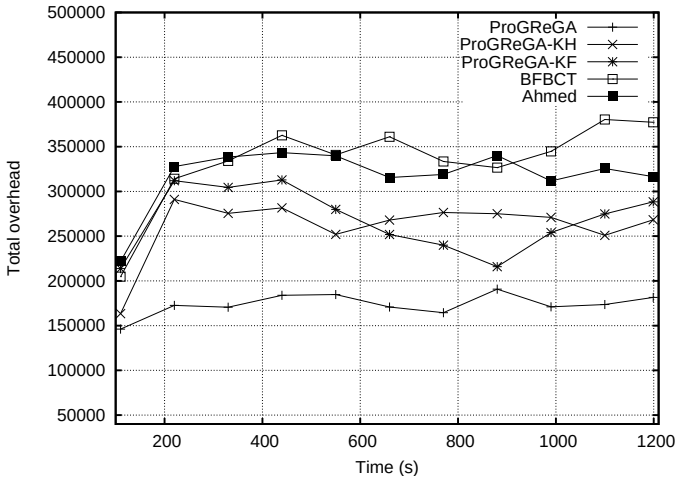


Fig. 11 Overhead introduced by each balancing algorithm during the game session

balance, it does not consider that the least loaded server may have no microcell adjacent to the microcell being transferred, creating a fragmented region and, thus, increasing the overhead on the system.

However, the introduction of overhead is not the only criterion considered. Fig. 12 shows how was the migration of users between servers throughout the simulated game session, for each algorithm. The value of migration has been divided in two: *walking migration*,

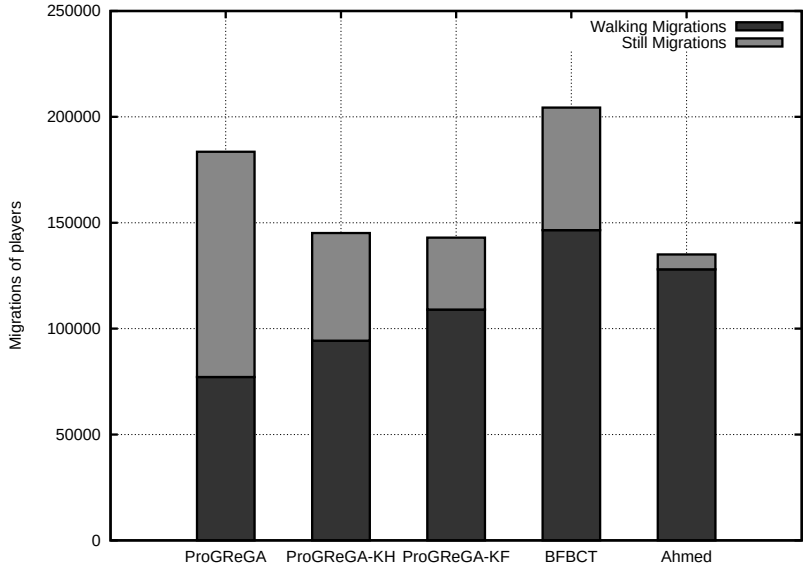


Fig. 12 Players migrating while still or walking depending on algorithm

which occurs when a player migrates to a new server because he moved his avatar from one region to another, and *still migration*, which occur when a player exchanges servers without having moved. This kind of migration occurs because the cell where his avatar was has been transferred to another region as a result of a rebalancing of the load of the game. Walking migrations are more likely to happen when the regions are not contiguous.

We can see that ProGReGA is the one which has fewer walk migrations, precisely because their regions are contiguous. However, the fact of not trying to minimize the transfer of cells – the ProGReGA main objective is to minimize the overhead – the number of migrations is still the largest of all its variations, lower only than BFBCT, which was also the worst algorithm on the user migration criterion. The strategy of ProGReGA-KH, and more strongly in the ProGReGA-KF, to maintain the maximum possible number of cells when rebalancing, has as a result the lowest numbers of migration of players of all the proposed algorithms.

The algorithm proposed by Ahmed had even less migrations than ProGReGA-KF. Though the number of walk migrations is the second highest – as expected from a more fragmented partitioning –, the number of still migrations is almost negligible. We believe that a clever choice of cells to transfer led to this result. Cells which interact less with other cells will, probably, contain less avatars and, then, transferring them will not cause many player migrations.

Another detail to be considered is the uniformity of distribution: all servers must have an usage rate as similar as possible so that the distribution is considered fair. To measure this uniformity, the standard deviation, σ, was calculated for $u(S)$ of all servers for each simulated algorithm. Figure 13 shows the variation of $\sigma(u(S))$ over time.

To summarize the evaluated algorithms, we have: ProGReGA, which introduced an overhead that increased the system load by 13% on the average, but it presents the second highest number of migrations of players, and the least fair load distribution; ProGReGA-KH, whose overhead increases the system load by 28%, but it presents the third best number of

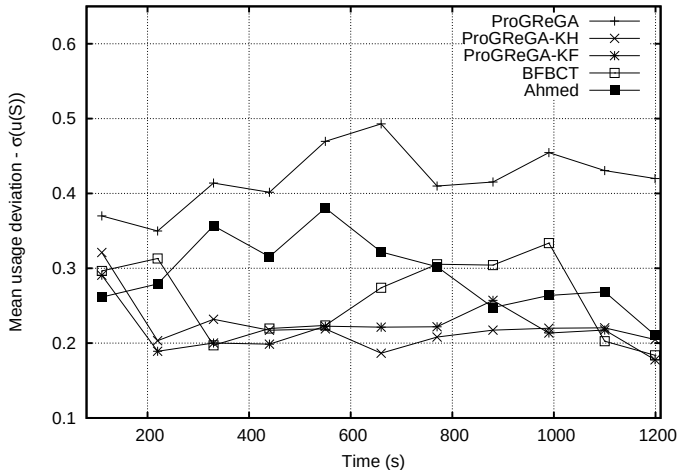


Fig. 13 Deviation of the ideal usage value

migrations and its distribution is one the most fair; ProGReGA-KF, whose load increase is of 22%, with the second lowest number of migrations and is also one of the most fair algorithms; and we also have BFBCT, which increased the load by 34%, with the highest number of migrations and not as fair as ProGReGA-KH or ProGReGA-KF. Finally, Ahmed's algorithm increased the system load by 29%, on the average, with the least number of migrations and a fairness level varying a lot over time.

Although the servers presented the largest variation in the resources usage rate, the ProGReGA was the one with the least overall real weight (which includes overhead: $W_{total} + OverHead$) of all simulated algorithms. Its variant ProGReGA-KF had the second lowest number of user migrations, along with the second lowest overhead and a fair load distribution, on the average, when compared to the other algorithms. Which of these two is better depends on the specific game. In a real-time game, users constantly migrating between servers can introduce delay and hinder the interaction between players. ProGReGA-KF, or Ahmed's algorithm, would be more suited for this situation. However, when it is not possible to use some kind of "graceful degradation" to reduce the quality of the game and save up the servers' resources, ProGReGA would be the best option.

5 Conclusions and future work

It was proposed here a load balancing scheme for distributed MMOG servers, taking into account the use of upload bandwidth of the server nodes. We considered important aspects such as the quadratic growth of the traffic when the avatars are close to one another, and the distribution overhead when players connected to different servers are interacting. The scheme, which is divided into three phases, proposed different algorithms for phase 2 (the balancing phase), and ProGReGA presented the lowest overhead of all, while ProGReGA-KF presented the second fewest migrations of players between servers, along with the second lower overhead introduced and a fair load distribution. For this reason, we recommend the use of ProGReGA-KF for most cases. However, in some situations where the system load must be reduced as much as possible, ProGReGA would perform better.

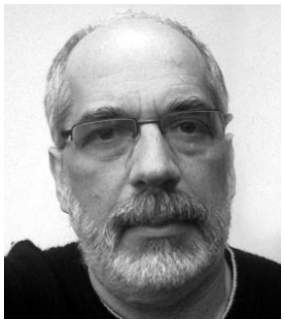
As a future work, one could create a load balancing scheme where there will be not only one server node, but a group of server nodes managing each region. In this case, ways to balance the load on the system, considering this two-level distribution, may be investigated.

Acknowledgements This work was supported by the National Research Council (CNPq) and by the Coordination of Improvement of Higher Education (CAPES), both Brazilian research funding agencies.

References

1. Chen, K., Huang, P., Lei, C.: Game traffic analysis: An MMORPG perspective. Computer Networks **50**(16), 3002–3023 (2006)
2. Feng, W.: What's Next for Networked Games? (2007). Available at: http://www.thefengs.com/wuchang/work/. (NetGames 2007 keynote talk, nov. 2007)
3. Ahmed, D., Shirmohammadi, S.: A Microcell Oriented Load Balancing Model for Collaborative Virtual Environments. In: Proceedings of the IEEE Conference on Virtual Environments, Human-Computer Interfaces and Measurement Systems, VECIMS, pp. 86–91. Piscataway, NJ: IEEE, Istanbul, Turkey (2008)
4. De Vleeschauwer, B., et al.: Dynamic microcell assignment for massively multiplayer online gaming. In: Proceedings of the ACM SIGCOMM workshop on Network and system support for games, NetGames, 4., pp. 1–7. New York: ACM, Hawthorne, NY (2005)
5. Lu, F., Parkin, S., Morgan, G.: Load balancing for massively multiplayer online games. In: Proceedings of the 5th ACM SIGCOMM workshop on Network and system support for games. ACM New York, NY, USA (2006)
6. Chen, J., Wu, B., Delap, M., Knutsson, B., Lu, H., Amza, C.: Locality aware dynamic load management for massively multiplayer games. In: Proceedings of the 10th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 289–300. ACM New York, NY, USA (2005)
7. Duong, T., Zhou, S.: A dynamic load sharing algorithm for massively multiplayer online games. In: Proceedings of the 11th IEEE International Conference on Networks, pp. 131–136
8. Kernighan, B., Lin, S.: An efficient heuristic procedure for partitioning graphs. Bell System Technical Journal **49**(2), 291–307 (1970)
9. Fiduccia, C., Mattheyses, R.: A Linear-Time Heuristic for Improving Network Partitions. In: Proceedings of the Conference on Design Automation, 19., pp. 175–181. New York: ACM, Las Vegas, NV (1982)
10. Karypis, G., Kumar, V.: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM Journal on Scientific Computing **20**(1), 359–392 (1998)
11. Hendrickson, B., Leland, R.: An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations. SIAM Journal on Scientific Computing **16**(2), 452–452 (1995)
12. Lee, K., Lee, D.: A scalable dynamic load distribution scheme for multi-server distributed virtual environment systems with highly-skewed user distribution. In: Proceedings of the ACM symposium on Virtual reality software and technology, pp. 160–168. New York: ACM, Osaka, Japan (2003)
13. Microsoft: Age of empires (1997). Available at: http://www.microsoft.com/games/empires/
14. Bezerra, C.E.B., Cecin, F.R., Geyer, C.F.R.: A3: a novel interest management algorithm for distributed simulations of mmogs. In: Proceedings of the IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications, DS-RT, 12., pp. 35–42. Washington, DC: IEEE, Vancouver, Canada (2008)
15. Feder, T., et al.: Complexity of graph partition problems. In: Proceedings of the ACM Symposium on Theory of Computing, STOC, 31., pp. 464–472. New York: ACM, Atlanta, GA (1999)
16. Bettstetter, C., Hartenstein, H., Pérez-Costa, X.: Stochastic Properties of the Random Waypoint Mobility Model: epoch length, direction distribution, and cell change rate. In: Proceedings of the ACM international workshop on Modeling analysis and simulation of wireless and mobile systems, 5., pp. 7–14. New York: ACM, Atlanta, GA (2002)

Carlos Eduardo Benevides Bezerra is a full-time Ph.D. student at UFRGS (Universidade Federal do Rio Grande do Sul). He obtained his M.Sc. from the same university in 2009 and his B.Sc. degree from UFBA (Universidade Federal da Bahia) in 2007. His research interests include massively multiplayer online games, interest management algorithms, parallel and distributed algorithms, peer-to-peer networks and computational geometry.



Cláudio Fernando Resin Geyer is an associate professor at the Informatics Institute of the Federal University of Rio Grande do Sul. His research interests include massively multiplayer games, grid computing and ubiquitous computing. He received his Ph.D. in informatics from Joseph Fourier University. He is a member of ACM and of the Brazilian Computer Society (SBC).