A fine granularity load balancing technique for MMOG servers using a kd-tree to partition the space

Carlos Eduardo B. Bezerra, João L. D. Comba, Cláudio F. R. Geyer
Instituto de Informática
Universidade Federal do Rio Grande do Sul
Av. Bento Gonçalves, 9500, Porto Alegre

Abstract

MMOGs (massively multiplayer online games) are applications that require high bandwidth connections to work properly. This demand for bandwidth is specially critical on the servers that host the game. This happens because the typical number of simultaneous participants in this kind of game varies from a few hundreds to several tens of thousands, and the server is the one responsible for mediating the interaction between every pair of players connected to it. To deal with this problem, decentralized architectures with multiple servers have been proposed, where each server manages a region of the virtual environment of the game. Each player, then, connects only to the server that manages the region where he is playing. However, to distribute the load among the servers, it is necessary to devise an algorithm for partitioning the virtual environment. In order to readjust the load distribution during the game, this algorithm must be dynamic. Some work has already been made in this direction, but with a geometric algorithm, more appropriate than those found in the literature, it should be possible to reduce the distribution granularity without compromising the rebalancing time, or even reducing it. In this work, we 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. We compared our algorithm to some approaches found in the literature and the simulation results show that our algorithm performed better in most aspects we analyzed.

Keywords: MMOGs, load balancing, distributed server, kd-trees.

Author's Contact:

{carlos.bezera, comba, geyer}@inf.ufrgs.br

1 Introduction

The main characteristic of MMOGs is the large number of players interacting simultaneously, reaching the number of tens of thousands [Schiele et al. 2007]. When using a client-server architecture for the players to communicate with one another, the server intermediates the communication between each pair of players.

To allow the interaction of players, each one of them sends his commands to the server, which calculates the resulting game state and sends it to all the players to whom the state change is relevant. We can see that the number of state update messages sent by the server may grow proportionally to the square of the number of players, if all players are interacting with one another. Obviously, depending on the number of players, the cost of maintaining a centralized infrastructure like this is too high, restricting the MMOG market to large companies with enough resources to pay the upkeep of the server.

In order to reduce this cost, several decentralized solutions have been proposed. Some of them use peer-to-peer networks, such as [Schiele et al. 2007; Rieche et al. 2007; Hampel et al. 2006; El Rhalibi and Merabti 2005; Iimura et al. 2004; Knutsson et al. 2004]. Others propose the use of a distributed server composed of low-cost nodes connected through the Internet, as in [Ng et al. 2002; Chertov and Fahmy 2006; Lee and Lee 2003; Assiotis and Tzanov 2006]. Anyway, in all these approaches, the "world", or virtual environment of the game is divided into regions and for every region is assigned a server – or a group of peers to manage it, when using peer-to-peer networks. Each of these regions must have a content such that the load imposed on the corresponding server is not greater than its capacity.

When an *avatar* (representation of the player in the virtual environment) is located in a region, the player controlling that avatar connects to the server associated to that region. That server, then, is responsible for receiving the input from that player and for sending, in response, the update messages. When a server becomes overloaded due to an excessive number of avatars in its region and, therefore, more players to be updated, the division of the virtual environment must be recalculated in order to alleviate the overloaded server.

Usually, the virtual environment is divided into relatively small cells, which are then grouped into regions and distributed among the servers. However, this approach has a severe limitation in its granularity, since the cells have fixed size and position. Using a more appropriate geometric algorithm, it should be possible to achieve a better player distribution among different servers, making use of traditional techniques that are generally used for computer graphics.

In this work, we propose the utilization of a kd-tree to perform the partitioning of the virtual environment. When a server is overloaded, it triggers the load balancing, readjusting the limits of its region by changing the split coordinates stored in the kd-tree. A prototype has been developed and used in simulations. The results found in these simulations have been compared to previous results from approaches which use the cell division technique.

The text is organized as follows: in section 2, some related works are described; in section 3, the algorithm proposed here is presented in detail; in the sections 4 and 5, we present, respectively, the simulation details and its results and, in section 6, the conclusions of this work are presented.

2 Related Work

Different authors have tried to address the problem of partitioning the virtual environment in MMOGs for distribution among multiple servers [Ahmed and Shirmohammadi 2008; Bezerra and Geyer 2009]. Generally, there is a static division into cells of fixed size and position. The cells are then grouped into regions (Figure 1), 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 and Shirmohammadi 2008], for example, propose a cell-oriented load balancing model. To balance the load, their algorithm finds, first, all clusters of cells that are managed by the overloaded server. The smallest cluster is selected and, from this cluster, it is chosen the cell which has the least interaction with other cells of the same server – the interaction between two cells A and B is defined by the authors as the number of pairs of avatars interacting with each other, one of them in A and the other one in B. The selected cell is then transferred to the least loaded server, considering "load" as the bandwidth used to send state updates to the players whose avatars are positioned in the cells managed by that server. This process is repeated until the server is no longer overloaded or there is no more servers capable of absorbing more load – in this case, one option could be to reduce the frequency at which state update messages are sent to the players, as suggested by [Bezerra et al. 2008].

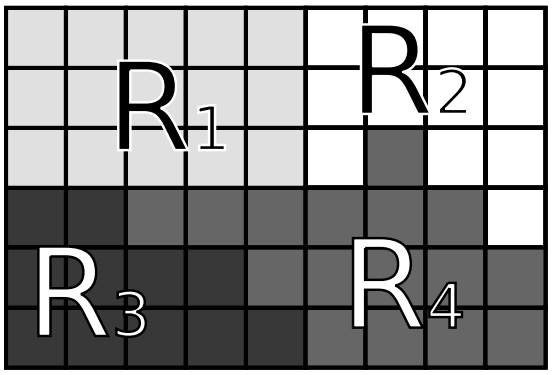

Figure 1: *Division into cells and grouping into regions*

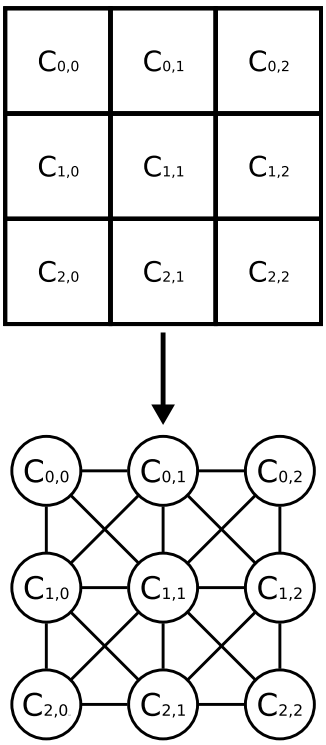

Figure 2: *Graph representation of the virtual environment*

In [Bezerra and Geyer 2009], it is also proposed the division into cells. To perform the division, the environment is represented by a graph (Figure 2), where each vertex represents a cell. Every edge in the graph connects two vertices representing neighboring cells. The weight of a vertex is the server's bandwidth occupied to send state updates to the players whose avatars are in the cell represented by that vertex. The interaction between any two cells define the weight of the edge connecting the corresponding vertices. To form the regions, the graph is partitioned using a greedy algorithm: starting from the heaviest vertex, at each step it is added the vertex connected by the heaviest edge to any of the vertices already selected, until the total weight of the partition of the graph – defined as the sum of the vertices' weights – reaches a certain threshold related to the total capacity of the server that will receive the region represented by that partition of the graph.

Although this approach works, there is a serious limitation on the distribution granularity it 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].

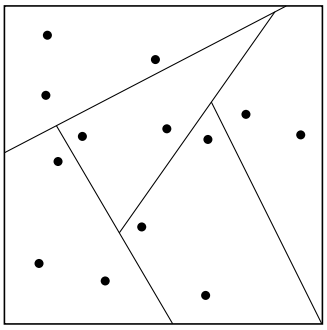

Figure 3: *Space partitioning using a BSP tree*

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 et al. 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 3). 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 readjustment 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.

3 Proposed approach

The load balancing approach proposed here is based on two criteria: first, the system should be considered 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.

As mentioned in the introduction, to divide the environment of the game into regions, we propose 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, we propose 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 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.

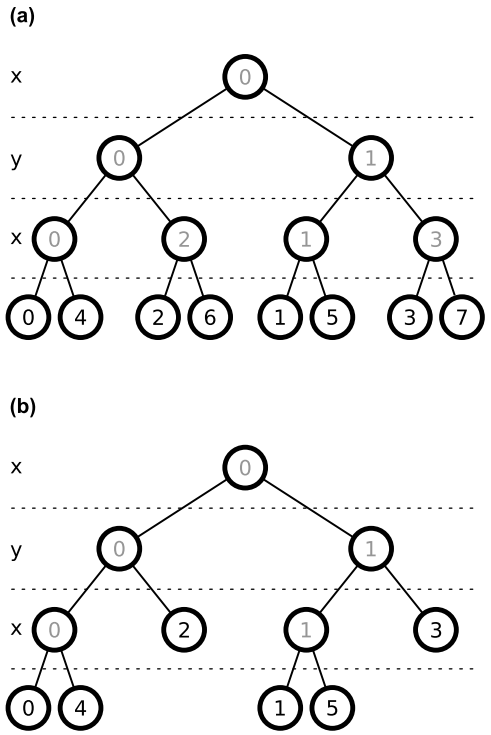

Figure 4: *Balanced kd-trees built with the described algorithm*

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.

3.1 Building the kd-tree

To make an initial space division, it is constructed a balanced kd-tree. For this, we use the recursive function shown in Algorithm 1 to create the tree.

Algorithm 1 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 1, 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 (a), we have a full kd-tree formed with this simple algorithm and, in Figure 4 (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.

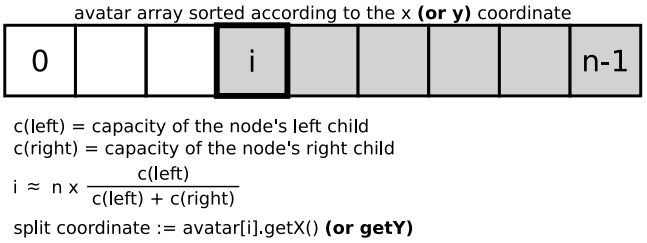

Figure 5: *A load splitting considering only the number os avatars*

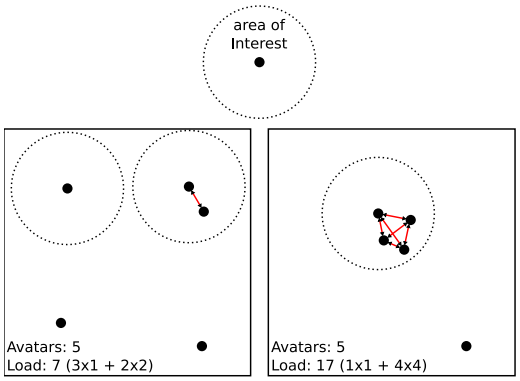

Figure 6: *Relation between avatars and load*

3.2 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 right child (Figure 5). 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 6). For this reason, it is not sufficient only to consider the number of players to divide them among the servers.

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

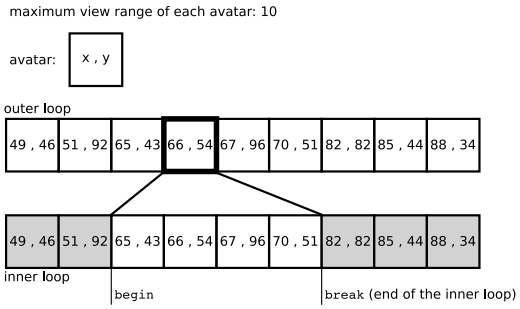

Figure 7: *Sweep of the sorted array of avatars*

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 7).

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.

3.3 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.

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 8). 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

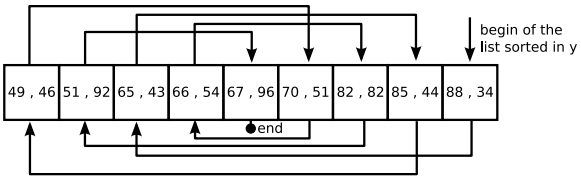

Figure 8: *Avatar array sorted by x, containing a list sort by y*

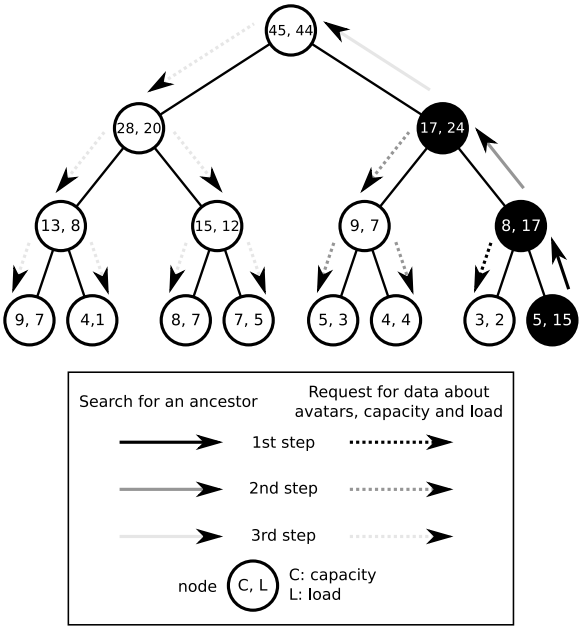

Figure 9: *Search for an ancestor node with enough resources*

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 9). 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.

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 10). 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 man-

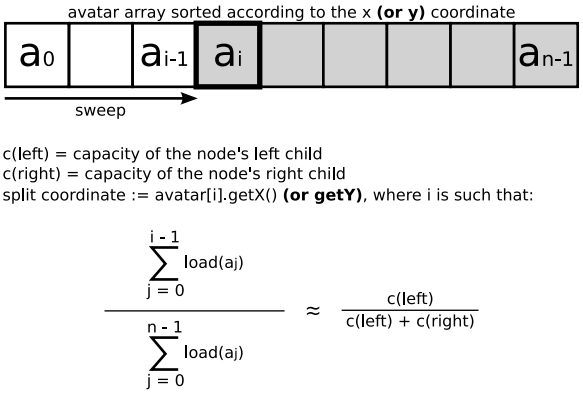

Figure 10: *Division of an avatar list between two brother nodes*

aged 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 10). As this is a balanced tree with S leaf nodes, it has a height of $\lceil logS \rceil$.

4 Simulations

To evaluate the proposed dynamic load balancing algorithm, a virtual environment across which many avatars moved was simulated. Starting from a random point in the environment, each avatar moved according to the random waypoint model [Bettstetter et al. 2002]. To force a load imbalance and stress the algorithms tested, we defined some *hotspots* – points of interest to which the avatars moved with a higher probability than to other parts of the map. This way, a higher concentration of avatars was formed in some areas. Although the movement model used is not very realistic in terms of the way the players move their avatars in real games, it was only used to verify the load balance algorithms simulated. For each algorithm tested, we simulated two situations: one with the presence of hotspots and one without hotspots.

The proposed approach was compared to the ones presented in section 2, from other authors. However, it is important to observe that the model employed by [Ahmed and Shirmohammadi 2008] considers hexagonal cells, while in our simulations we used rectangular cells. Furthermore, the authors considered that there is a transmission rate threshold, which is the same for all servers in the system. As we assume a heterogenous system, their algorithm was simulated considering that each server has its own transmission rate threshold, depending on the upload bandwidth available in each one of them. However, we kept what we consider the core idea of the authors' approach, which is the selection of the smallest cell cluster managed by the overload server, then choosing that cell with the lowest interaction with other cells of the same server, and finally the transferring of this cell to the least loaded server. Besides Ahmed's algorithm, we also simulated some of the ones proposed in [Bezerra and Geyer 2009] – Progrega and BFBCT.

The simulated virtual environment consisted of a two-dimensional space, with 750 moving avatars, whose players were divided among eight servers ($S_1, S_2, ..., S_8$), each of which related to one of the regions determined by the balancing algorithm. For the cell-oriented approaches simulated, the space was divided into a 15×15 cell grid, or 225 cells. The capacity of each server S_i was equal to $i \times 20000$, forming a heterogeneous system. This heterogeneity allowed us to evaluate the load balancing algorithms simulated according to the criterion of proportionality of the load distribution on the servers.

In addition to evaluate the algorithms according to the proportionality of the load distribution, it was also considered the number of player migrations between servers. Each migration involves a

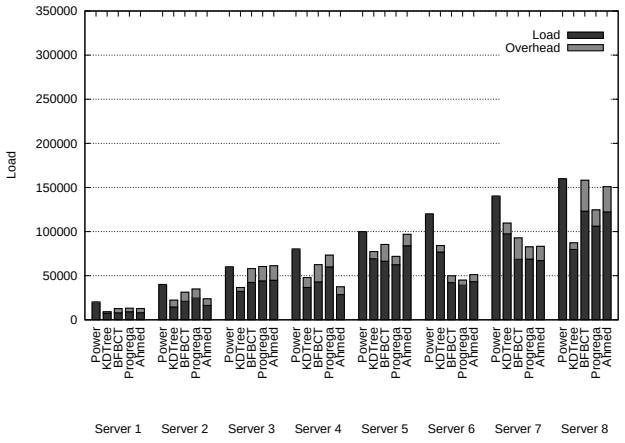

Figure 11: *Average load on each server (by algorithm, without hotspots)*

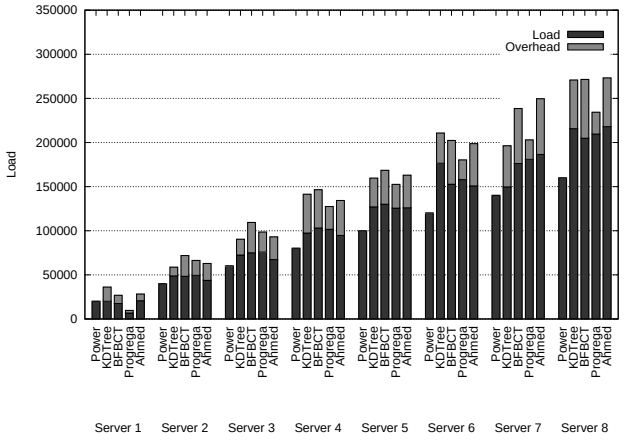

Figure 12: *Average load on each server (by algorithm, with hotspots)*

player connecting to the new server and disconnecting from the old one. This kind of situation may occur in two cases: the avatar moved, changing the region in which it is located and, consequently, changing the server to which its player is connected; or the avatar was not moving and still its player had to migrate to a new server. In the latter case, obviously the player's transfer was due to a rebalancing. An ideal balancing algorithm performs the load redistribution requiring the minimum possible number of player transfers between servers, while keeping the load on each server proportional to its capacity.

Finally, the inter-server communication overhead will also be evaluated. It occurs when two players are interacting, but each one of them is connected to a different server. Although the algorithm proposed in this work does not address this problem directly, it would be interesting to evaluate how the load distribution performed by it influences the communication between the servers.

5 Results

Figures 11 and 12 present the average load (plus the inter-server communication overhead) on each server, for each algorithm tested. The first figure shows the values in a situation without hotspots and, therefore, a smaller total load. The second, in turn, presents the load distribution when the server system is overloaded. We can see, in Figure 11, that all algorithms have met the objective of keeping the load on each server less than or equal to its capacity, when the system has sufficient resources to do so. In Figure 12, it is demonstrated that all the algorithms managed to dilute – in a more or less proportional manner – the load excess on the servers. It is important to observe, however, that the load shown in Figure 12 is only theo-

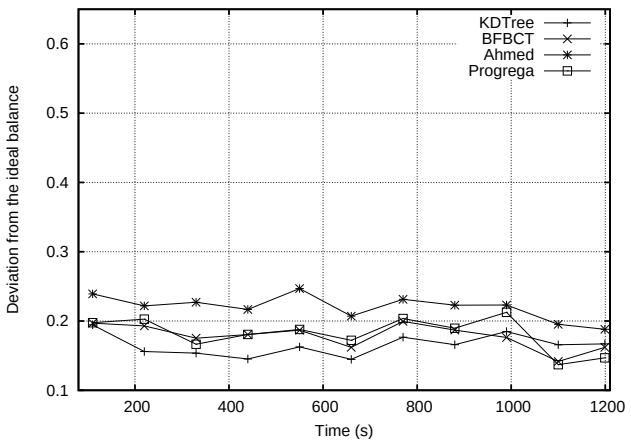

Figure 13: *Average deviation of the ideal balance of the servers (without hotspots)*

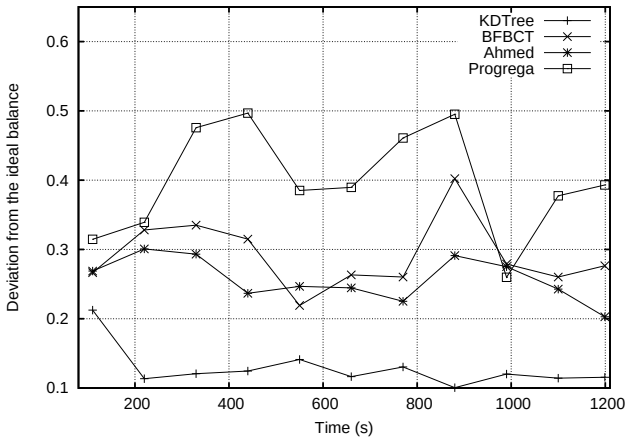

Figure 14: *Average deviation of the ideal balance of the servers (with hotspots)*

retical. Each server will perform some kind of "graceful degradation" in order to keep the load under its capacity. For example, the update frequency might be reduced and access to the game could be denied for new players attempting to join, which is a common practice in most MMOGs.

In figures 13 and 14, it is shown how much the balance generated by each algorithm deviates from an ideal balance – that is, how much, on the average, the load on the servers deviate from a value exactly proportional to the capacity of each one of them – over time. It is possible to observe that, in both situations – with and without hotspots – the algorithm that uses the kd-tree has the least deviation. This is the due to the fine granularity of its distribution, which, unlike the other approaches tested, is not limited by the size of a cell. In the situation with hotspots, the algorithm that uses the kd-tree is particularly effective, because rebalance is needed. In a situation where the system has more resources than necessary, the proportionality of the distribution is not as important: it is enough that each server manages a load smaller than its capacity.

Regarding player migrations between servers, all the algorithms – except BFBCT – had a similar number of user migrations in the absence of hotspots (Figure 15). This happens because the load of the game is less than the total capacity of the server system, which required less rebalancing and, thus, caused less migrations of players between servers. Figure 16, however, demonstrates that the algorithm that uses the kd-tree had a significantly lower number of user migrations than the other approaches. This is due, in the first place, to the fact that the regions defined by the leaf nodes of the kd-tree are necessarily contiguous, and each server was linked to only one leaf node. An avatar moving across the environment divided into very fragmented regions constantly crosses the borders

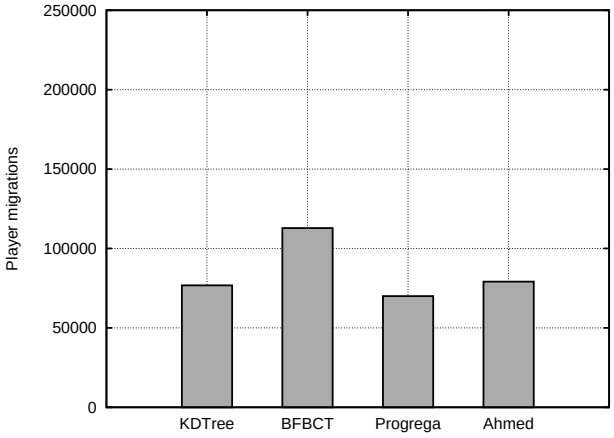

Figure 15: *Player migrations between servers (without hotspots)*

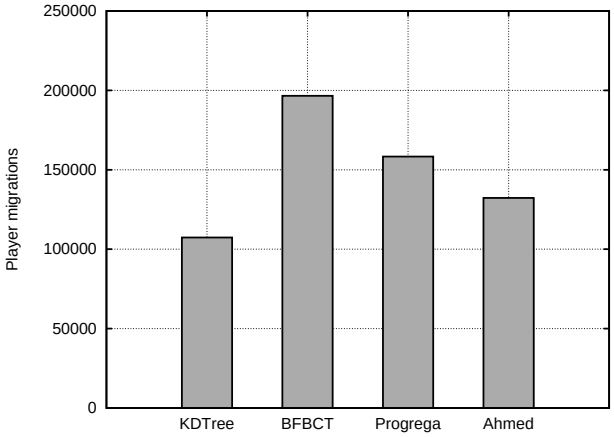

Figure 16: *Player migrations between servers (with hotspots)*

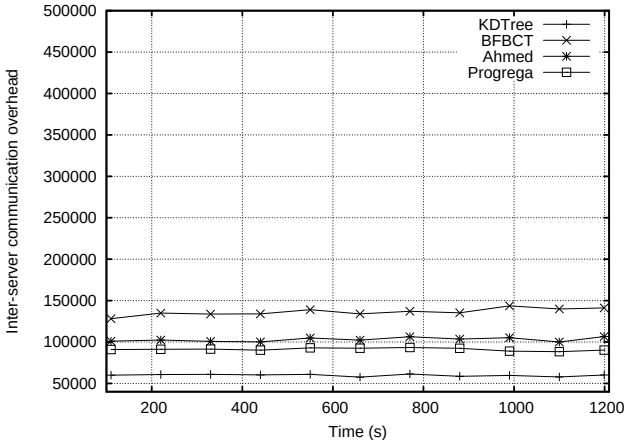

Figure 17: *Inter-server communication for each algorithm over time (without hotspots)*

between these regions and causes, therefore, its player to migrate from server to server repeatedly. Another reason for this result is that each rebalancing executed with the kd-tree gets much closer to an ideal distribution than the cell-based algorithms – again, thanks to the finer granularity of the kd-tree based distribution –, requiring less future rebalancing and, thus, causing less player migrations.

Finally, it is shown the amount of communication between servers for each simulated algorithm, over time. In Figure 17, all algorithms have similar results, and the one which uses the kd-tree is slightly better than the others. This is also explained by the fact that the regions are contiguous, minimizing the number of bound-

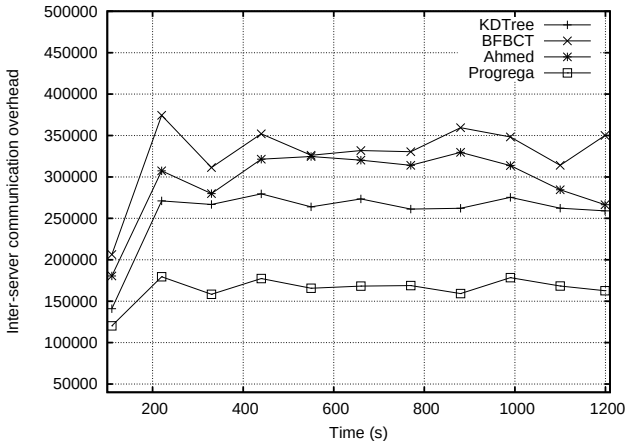

Figure 18: *Inter-server communication for each algorithm over time (with hotspots)*

aries between them and, consequently, reducing the probability of occurring interactions between pairs of avatars, each one in a different region. In Figure 18, it is possible to see that the inter-server communication caused by Progrega was considerably lower than all the others in a situation of system overload. The reason for this is that its main goal – besides balancing the load – is precisely to reduce the communication between servers. However, even not considering the additional cost, the algorithm that uses the kd-tree got second place in this criterion.

6 Conclusions

In this work, we proposed 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 fine 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 helped to reduce the number of migrations, by performing less rebalancing operations. The fact that the regions defined by the kd-tree are necessarily contiguous was one of the factors that contributed to the results of the proposed algorithm, which was better than the other algorithms simulated in most of the criteria considered.

In conclusion, it was possible to use methods that can reduce the complexity of 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.

Acknowledgements

The development of this work has been supported by the National Research Council (CNPq) and by the Coordination for Improvement of the Higher Education Personnel (CAPES).

References

AHMED, D., AND SHIRMOHAMMADI, S. 2008. A Microcell Oriented Load Balancing Model for Collaborative Virtual Environments. In *VECIMS*, 86–91.

ASSIOTIS, M., AND TZANOV, V. 2006. A distributed architecture for MMORPG. In *Proceedings of the ACM SIGCOMM workshop on Network and system support for games, NetGames, 5.*, New York: ACM, Singapore, 4.

BENTLEY, J. 1975. Multidimensional binary search trees used for associative searching.

BETTSTETTER, C., HARTENSTEIN, H., AND PÉREZ-COSTA, X. 2002. 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.*, New York: ACM, Atlanta, GA, 7–14.

BEZERRA, C. E. B., AND GEYER, C. F. R. 2009. A load balancing scheme for massively multiplayer online games. *Massively Multiuser Online Gaming Systems and Applications, Special Issue of Springer's Journal of Multimedia Tools and Applications*.

BEZERRA, C. E. B., CECIN, F. R., AND GEYER, C. F. R. 2008. 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.*, Washington, DC: IEEE, Vancouver, Canada, 35–42.

BLIZZARD, 2004. World of warcraft. 2004. Available at: <http://www.worldofwarcraft.com/>. Last time accessed: 24 jul. 2009.

CHERTOV, R., AND FAHMY, S. 2006. Optimistic Load Balancing in a Distributed Virtual Environment. In *Proceedings of the ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video, NOSSDAV, 16.*, New York: ACM, Newport, USA, 1–6.

EL RHALIBI, A., AND MERABTI, M. 2005. Agents-based modeling for a peer-to-peer MMOG architecture. *Computers in Entertainment (CIE) 3*, 2, 3–3.

GRAVITY, 2001. Raganarök online. 2001. Available at: <http://www.ragnarokonline.com/>. Last time accessed: 24 jul. 2009.

HAMPEL, T., BOPP, T., AND HINN, R. 2006. A peer-to-peer architecture for massive multiplayer online games. In *Proceedings of the ACM SIGCOMM workshop on Network and system support for games, NetGames, 5.*, New York: ACM, Singapore, 48.

IIMURA, T., HAZEYAMA, H., AND KADOBAYASHI, Y. 2004. Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games. In *Proceedings of the ACM SIGCOMM workshop on Network and system support for games, NetGames, 3.*, New York: ACM, Portland, USA, 116–120.

KNUTSSON, B., ET AL. 2004. Peer-to-peer support for massively multiplayer games. In *Proceedings of the IEEE Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM, 23.*, [S.l.]: IEEE, Hong Kong, 96–107.

LEE, K., AND LEE, D. 2003. 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*, New York: ACM, Osaka, Japan, 160–168.

LUQUE, R., COMBA, J., AND FREITAS, C. 2005. Broad-phase collision detection using semi-adjusting BSP-trees. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, ACM New York, NY, USA, 179–186.

NCSOFT, 2003. Lineage ii. 2003. Available at: <http://www.lineage2.com/>. Last time accessed: 24 jul. 2009.

NG, B., ET AL. 2002. A multi-server architecture for distributed virtual walkthrough. In *Proceedings of the ACM symposium on Virtual reality software and technology, VRST*, New York: ACM, Hong Kong, 163–170.

RIECHE, S., ET AL. 2007. Peer-to-Peer-based Infrastructure Support for Massively Multiplayer Online Games. In *Proceedings of the 4th IEEE Consumer Communications and Networking Conference, CCNC, 4.*, [S.l.]: IEEE, Las Vegas, NV, 763–767.

SCHIELE, G., ET AL. 2007. Requirements of Peer-to-Peer-based Massively Multiplayer Online Gaming. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid, CCGRID, 7.*, Washington, DC: IEEE, Rio de Janeiro, 773–782.