Adaptive Load Balancing for MMOG servers using kd-trees

(invited extended version of the 2009 SBGames paper)

CARLOS EDUARDO B. BEZERRA, JOÃO L. D. COMBA and CLÁUDIO F. R. GEYER
Universidade Federal do Rio Grande do Sul

In Massively Multiplayer Online Games (MMOGs) there is a great demand for high bandwidth connections with irregular access patterns. Such irregular demand is due to the fact that players, which can vary from few hundreds to several tens of thousands, often occupy the virtual environment of the game in different ways with varying densities. Therefore, there is a great need for decentralized architectures with multiple servers that employ load balancing algorithms to manage regions of the virtual environment. In such systems, each player only connects to the server that manages the region where his avatar is located, whereas each server is responsible for mediating the interaction between all pairs of players connected to it. Devising the proper load balancing algorithm to take into account spatial and variable occupation is a challenging problem, which requires adaptive (and possibly dynamic) partitioning of the virtual environment. In this work, we propose the use of a kd-tree for partitioning the game environment into regions, and dynamically adjust the resulting subdivision based on the distribution of avatars in the virtual environment. We compared our algorithm to competing approaches found in the literature and show that our algorithm performed better in most aspects we analyzed.

Categories and Subject Descriptors: C.2.4 [**Computers Systems Organizations**]: Computer-Communication Networks—*Distributed Systems*; I.3.5 [**Computing Methodologies**]: Computer Graphics—*Computational Geometry and Object Modeling*

General Terms: Algorithms, Management, Performance

Additional Key Words and Phrases: distributed server, kd-trees, load balancing, MMOGs

1. INTRODUCTION

Massively Multiplayer Online Games (MMOGs) is a genre of computer games that demands high network bandwidth to allow interaction among players. The main characteristic of MMOGs is the large number of players interacting simultaneously, which currently can reach up to tens of thousands [Schiele et al. 2007]. Client-server architectures are often configured to allow communication among players, with the server intermediating the communication between each pair of players. To interact, each player sends commands to the server, which calculates the new game state and propagates it back to all players affected by the state change. This mechanism can lead to many state update messages sent by the server, which may be quadratic on the number of players in the worst case – when all players interact with one another. Therefore, the cost of maintaining a centralized infrastructure like this

Authors address: Programa de Pós-Graduação em Computação, Instituto de Informática, UFRGS, Caixa Postal 15064, 91501-970, Porto Alegre - RS - Brasil.

is high when lots of players are connected, thus restricting the MMOG market to large companies with enough resources to pay the upkeep of the server.

To reduce this cost, several decentralized solutions have been proposed. Examples include peer-to-peer networks [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] or distributed servers [Ng et al. 2002; Chertov and Fahmy 2006; Lee and Lee 2003; Assiotis and Tzanov 2006], which are composed of low-cost nodes connected through the Internet. Common to both approaches is the fact that the "world", or virtual environment of the game is divided into regions which are managed by a server (or a group of peers). Each region must not violate the network capacity of its respective server, and thus must not have more content than the load imposed on the corresponding server.

Players are assigned to servers based on their location. 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. This server becomes responsible for processing the input from that player and for sending update messages in response. If a server becomes overloaded due to an excessive number of avatars in its region, one way to reduce its load is by repartitioning the virtual environment. Simply assigning players to different servers might not suffice, since they might not be able to absorb the excessive load.

Virtual environments are often divided into smaller cells, which are grouped into regions and distributed among servers. This approach has several limitations in its granularity due to cells of fixed size and position. More elaborate partitioning algorithms based in spatial data structures[Samet 2005] allow better player distribution among different servers. In this work, we use a kd-tree to dynamically partition the virtual environment. When a server is overloaded, it triggers a load balancing algorithm, which changes the limits of each region by changing the split coordinates stored in the kd-tree. We validate our proposal with several simulations and compare the results to previous works that uses the cell division technique.

2. RELATED WORK

Different authors address the spatial partitioning of virtual environments in MMOGs for better distribution among multiple servers [Ahmed and Shirmohammadi 2008; Bezerra and Geyer 2009]. A simple approach is to partition the space into a static set of cells of fixed size and position, which are grouped into regions that are assigned to one of the servers (Figure 1). When a server becomes overloaded, part of the load is transferred to another server.

In the work of [Ahmed and Shirmohammadi 2008], a cell-oriented load balancing model is proposed. Their algorithm first enumerates all clusters of cells that are managed by the overloaded server. The smallest cluster is found, and from this cluster is selected the cell with the least interaction with other cells on 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 from A and the other from B. The selected cell is then transferred to the server with the smallest load[1].

[1]considering "load" as the bandwidth used to send state updates to the players whose avatars are positioned in the cells managed by that server.

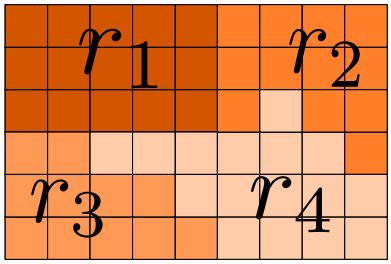


Fig. 1. Division into cells and grouping into regions

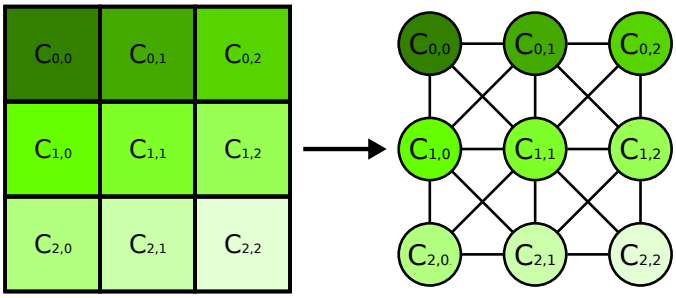


Fig. 2. Graph representation of the virtual environment

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].

In [Bezerra and Geyer 2009], it is also proposed the division into cells. To perform the division, the environment is represented by a graph, where each vertex represents a cell (Figure 2). Every edge in the graph connects two vertices representing neighboring cells. The weight of a vertex is the server's bandwidth used to send state updates to the players whose avatars are in the cell represented by that vertex. The interaction between any two cells defines 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.

This approach has a limited granularity distribution. If a finer granularity is desired, it is necessary to use smaller cells. This increases the number of vertices in the graph that represents the virtual environment and, consequently, the time required to perform load balancing. Also, the control message with the list of cells designated to each server becomes longer. A possibility to circumvent this problem relies on partitioning the virtual environment using spatial data structures [Bentley 1975], for example kd-trees or Binary Space Partitioning (BSP)-trees (Figure 3).

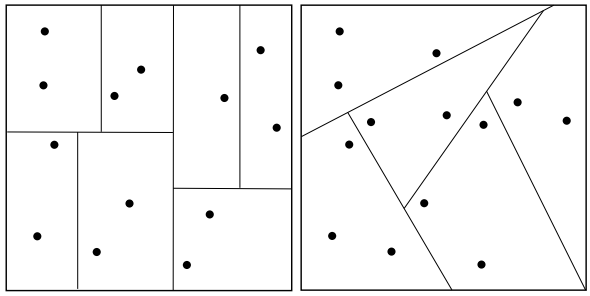


Fig. 3. Space partitioning using a kd-tree (a) and a BSP tree (b).

Spatial data structures are widely used in Computer Graphics, and can be used to partition the virtual environments of MMOGs. There is a vast literature on techniques that aim at creating spatial partitions with balanced "loads". In [Luque et al. 2005], the goal is to reduce the time needed to calculate collision detection between pairs of objects in a dynamic environment. A BSP-tree is used to distribute the objects in the scene. Objects that are in different partitions do not collide and there is no need to perform a complex collision test. It is proposed 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 collision detection. Such ideas are used for load balancing among servers in MMOGs.

3. PROPOSED APPROACH

In this section we describe our load balancing algorithm. Our proposal is based on two criteria: first, we define the *load* of a server as the amount of bandwidth it uses to send state updates to its clients; and, second, the system should be considered heterogeneous (i.e. every server may have a different amount of available bandwidth). The reason why we believe that the load on each server should be proportional to the amount of bandwidth required to send state update messages to the players connected to it and *not* necessarily proportional to the number of players is because every player may be interacting to many others. This way, each player may have to receive state updates from many other players, leading to a quadratic total number of state update messages to be sent by the servers. Also, upload rates are usually lower than download rates and, on top of that, the number of commands received by a server grows linearly with the number of players. Hence, we consider only the upload bandwidth used to send state updates, ignoring the download rate used by the servers to receive commands from the clients.

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 is stored a split coordinate. Each 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.

Every node of the tree also stores two other values: capacity and load of the subtree. The load of a leaf node is equal to the load of the region associated to it. Similarly, the capacity of a leaf node corresponds to the network capacity of its associated server. For each non-leaf node, the load and capacity values correspond to the sum of those stored in its child nodes. The tree root stores, therefore, the total weight of the game and the total capacity – upload bandwidth – of the server system.

In the following sections, we will describe 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 – by instantiating a single root node which, then, makes the call $node.build\_tree(0, 0, n)$, where n is the number of leaf nodes (corresponding to the number of servers, in our case).

Algorithm 1 node::build_tree(id, level, n)

```
this.node_id ← id;
if id + 2^level ≥ n then
  left_child ← NIL;
  right_child ← NIL;
else
  left_child ← new_node();
  left_child.parent ← this;
  right_child ← new_node();
  right_child.parent ← this;
  left_child.build_tree(id, level + 1, n);
  right_child.build_tree(id + 2^level, level + 1, n);
end if
```

In Algorithm 1, the id value is used to calculate whether each node has children or not. The purpose of this is to create a balanced tree where the leaf-count (number of leaf nodes) difference between the two sub-trees of any given node is one, at most.

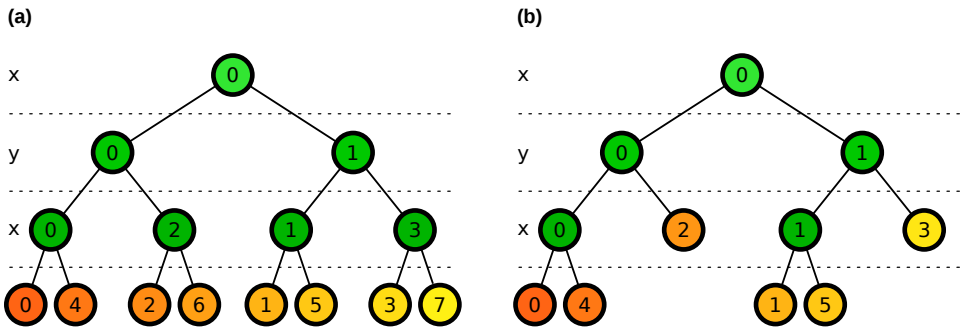


Fig. 4. Balanced kd-trees built with the described algorithm

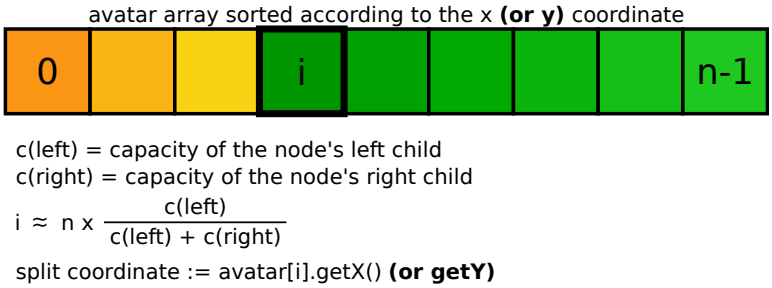


Fig. 5. A load splitting considering only the number os avatars

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.

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 in a way such that the number of players on each server is proportional to the its upload bandwidth. To calculate the split coordinate, it would be enough to simply sort the n avatars based on their coordinates on the split axis (x or y) and, then, assign the first n' avatars to the left child and the last $n - n'$ ones to the right child, where n' and $n - n'$ are proportional to the capacity of the left and right child, respectively (Figure 5). At each node, this operation would have a cost of $O(nlogn)$, due to the sorting. Assigning all the set of avatars to the root node and recursively executing this operation would have a complexity of $O(nlog^2n)$, as it must me repeated for each of the $logn$ levels of the kd-tree.

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 to update each one of them only about the outcome of his own actions – for these, the growth in the number of

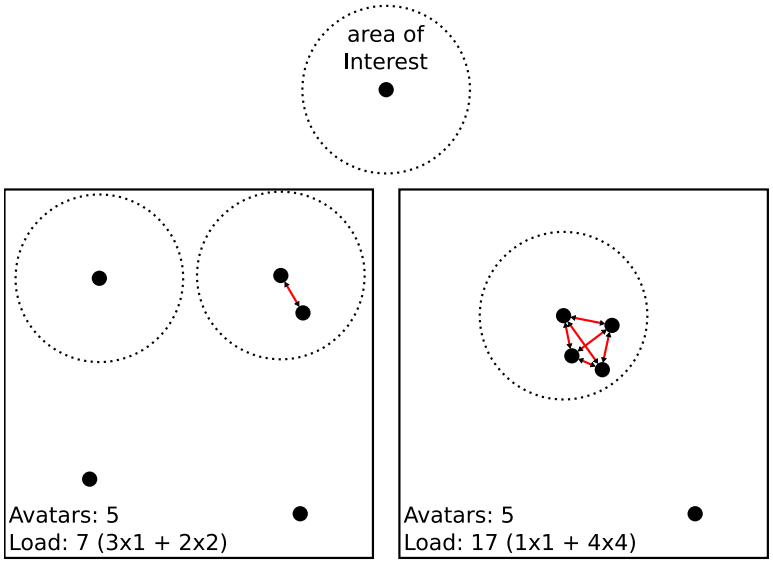


Fig. 6. Relation between avatars and load

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 enough to consider only the number of players when dividing 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 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[2] 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,

[2]The relevance of an object to an avatar may be used to define how many state update messages its player must receive about that object per time unit [Bezerra et al. 2008] – for that matter, avatars are also considered objects. One possible approach is to define the relevance of an object to an avatar as their proximity in the virtual environment.

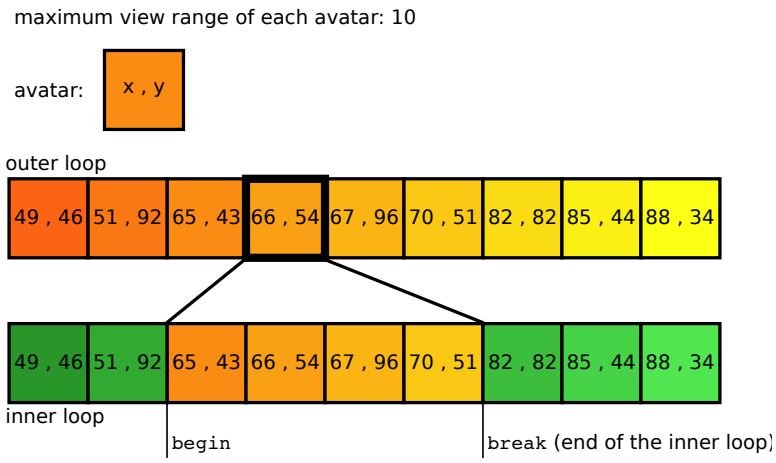


Fig. 7. Sweep of the sorted array of avatars

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 – this can be done because each avatar is located in a region, which is represented by a leaf node. Recursively, the load is calculated for the other nodes of the kd-tree. As mentioned before, to each leaf node a server and a region of the virtual environment are assigned. As 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, the load of each leaf node is equal to the sum of the weights of the avatars located in the region represented by it.

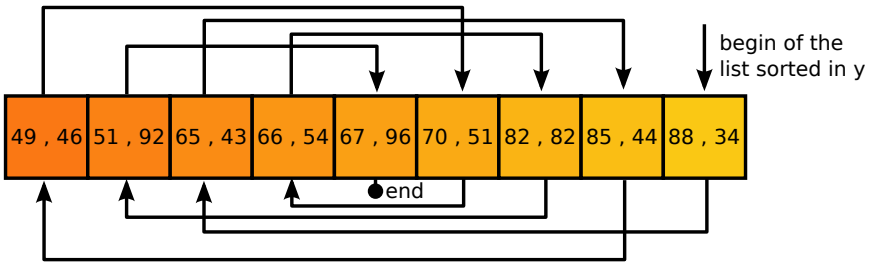


Fig. 8. Avatar array sorted by x, containing a list sort by y

3.3 Dynamic load balancing

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 – list of avatars, their positions and loads – 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 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 associated to it and going one level up at each step until it finds an ancestor node with a capacity greater than or equal to the load. While this node is not found, the algorithm continues recursively up the tree until it reaches the root. At each step, the algorithm visits a node which has a sub-tree containing other servers (leaf nodes), whose avatar list, load value and capacity value are probably unknow or outdated to the initiating server. A request is then sent to these servers, requiring the current avatar list and the current values of load and capacity (Figure 9). With these data, and the initiating server's own list of avatars and values of load and capacity, the algorithm can calculate the load and capacity of the ancestral nodes 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 initiating server 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

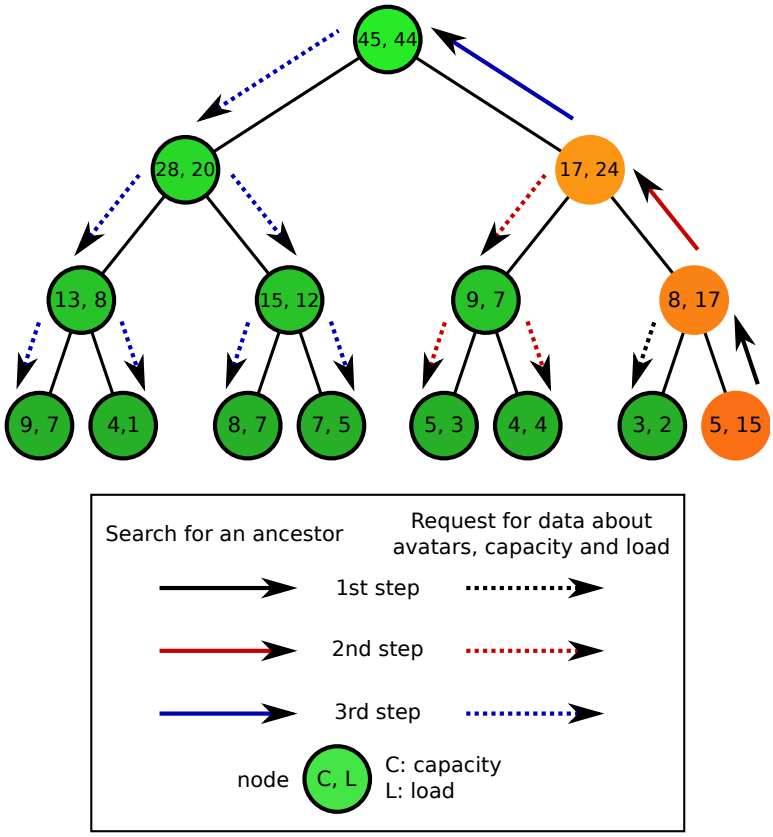
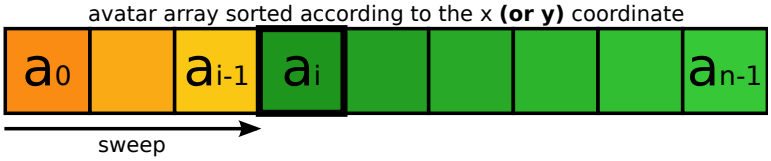


Fig. 9. Search for an ancestor node with enough resources

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 managed by it, the rebalance time is $O(nlogS)$, where n is the number of avatars in the game and S is the number of servers. The communication cost is $O(n)$, caused by the sending of data related to te n avatars. The merging of all avatar lists has $O(n)$ complexity, for the avatars were already sorted by the servers. At each level of the kd-tree, $O(n)$ avatars are swept in the worst case, in order to find the i index whose avatar's coordinate will be used to split the regions defined by each node of the tree (Figure 10). As this is a balanced tree with S leaf nodes, it has a height of $\lceil logS \rceil$, explaining the $O(nlogS)$ overall complexity of the rebalancing algorithm described.



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

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

Fig. 10. Division of an avatar list between two brother nodes

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 scenarios with 750 and 1500 players, with and without hotspots. In the cases with hotspots, we simulated different probabilities (70% and 90%) of each player selecting one of the hotspots as the next waypoint to move. Combining all these variables, we simulated six different scenarios for each algorithm – Figure 11 illustrates each situation.

The reason why we chose three kinds of avatar distribution – no hotspots, mildly concentrated around hotpots (70%) and densely concentrated (90%) – was because we wanted to make realistic simulations. In most commercial MMOGs, there are points of interest, like towns and dungeons. Usually, the players wander throughout the game world, looking for adventure and treasures, going to some dungeon and coming back to town once in a while – therefore, creating a mildly concentrated avatar population around these places. Frequently, however, the game company creates special game events that attract most players to some locations in the game world, like a very powerful monster invading a town or a very rare and valuable treasure in a dungeon, creating a population density in these areas much higher than usual – case to which the heavily concentrated simulation scenario refers. However, it is very uncommon to have absolutely no hotspots in a real MMOG,

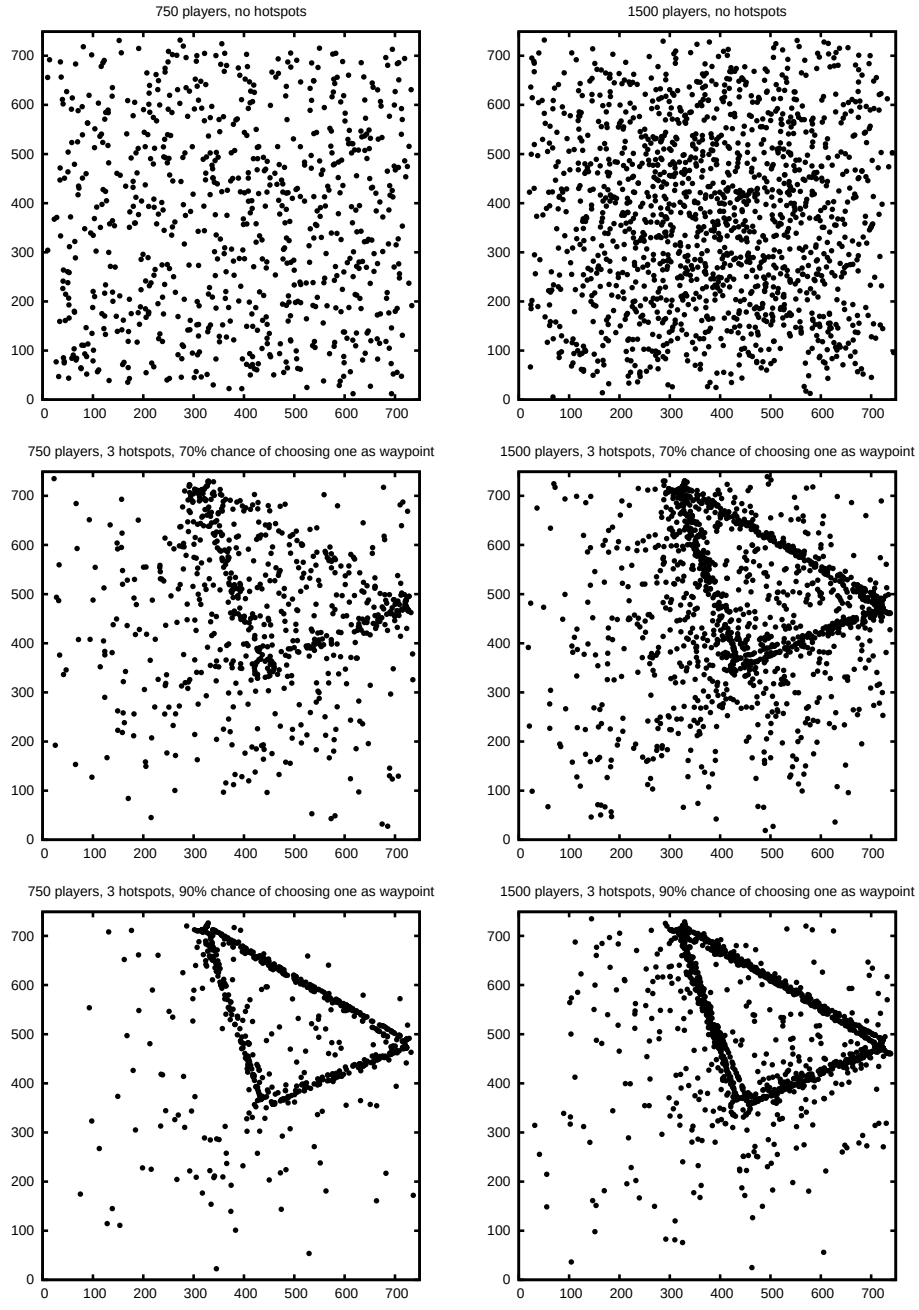


Fig. 11. Scenarios in which the algorithms were simulated

but this case was also simulated in order not only to cover this kind of no-hotspot situation, but also for its results to serve as a reference to the results of the scenarios with 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 for 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 overloaded server, then choosing from this cluster the 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**[3] 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 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 under (or proportional to) its capacity.

Finally, the inter-server communication **overhead**[4] was also 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.

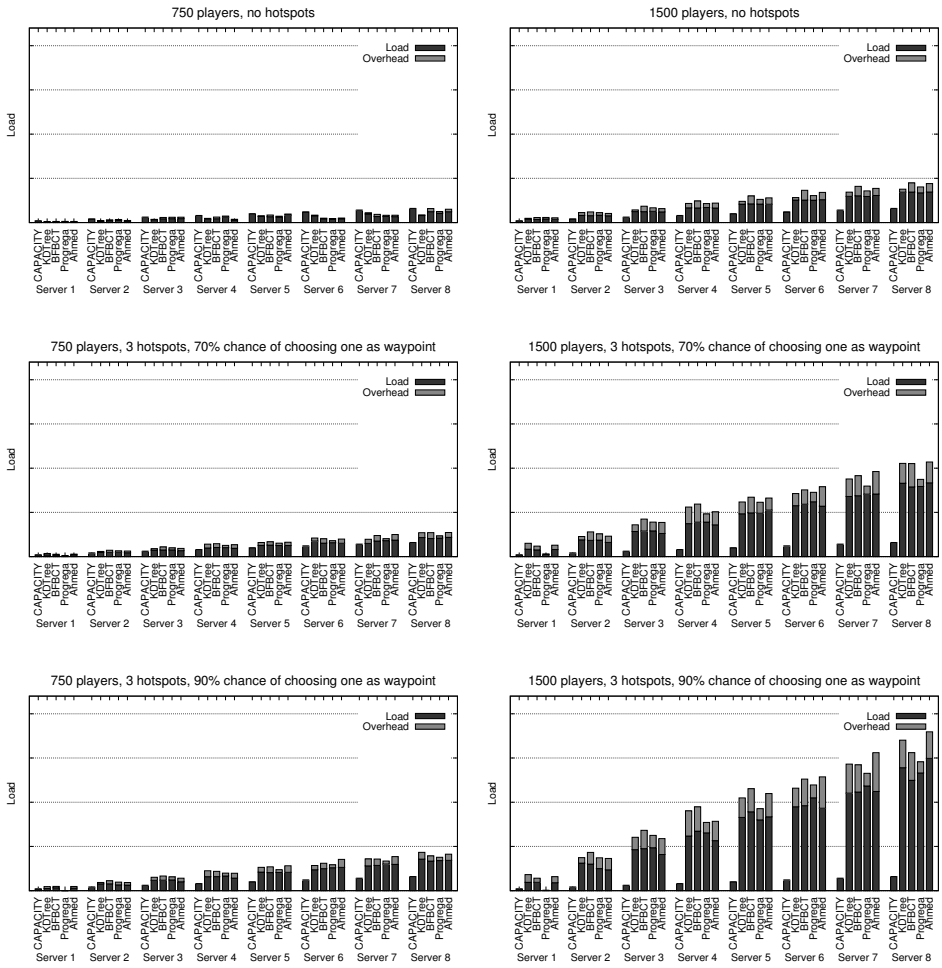


Fig. 12. Average load on each server in each scenario

5. RESULTS

Figure 12 presents the average load (plus the inter-server communication overhead) on each server, for each algorithm tested. The first two graphics show the values in a situation without hotspots and so they present a lower total load. The next two graphics, in turn, present the load distribution when the players tend to move towards one of the hotspots, increasing the number of interactions and, thus, the load on the servers. Finally, the two bottom graphics show the load on the servers

[3]The network capacity is a value proportional to the upload bandwidth of each simulated server.
[4]The inter-server communication overhead here represents the amount of bandwidth wasted by the servers when intermediating interactions of players who are in different regions. This happens because, in each of these interactions, there is more than one intermediate server.

when not only there are hotspots, but also the probability of a player moving to one of them is considerably high (90%). Although this is not the worst case – all players interacting with one another –, the load increase on the servers is significant.

We can see that all algorithms have met the objective of keeping the load on each server lower than or proportional to its capacity, in all situations considered. When the server system is overloaded (as in every graphic of Figure 12, apart from the first), 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 the figure is only theoretical. 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 Figure 13, it is shown how much the balance generated by each algorithm deviates from an ideal balance – i.e., 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, at least in three of the situations of greatest overload, 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 situations of heavy overload, the algorithm that uses the kd-tree is particularly effective, because rebalance is needed. In a situation where the system has enough resources, the proportionality of the distribution is not an important issue: it is enough that each server manages a load smaller than its capacity.

Regarding player migrations between servers, the proposed approach not only performed considerably better than the others (Figure 14), as it also demonstrated a greater scalability, increasing its advantage over the other algorithms as the simulated scenario became more heavily loaded. 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 an environment divided into very fragmented regions constantly crosses the borders 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 fewer player migrations.

Finally, it is shown the amount of communication between servers for each simulated algorithm, over time. As we can see in Figure 15, when there are no hotspots, the algorithm 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 boundaries between them and, consequently, reducing the probability of occurring interactions between pairs of avatars, each one in a different region. However, it is also possible to see that the inter-server communication with Progrega was considerably lower than with the other algorithms in all the situations of system overload, except when there were 750 players with 70% chance of an avatar moving to a hotspot, where Progrega and the kd-tree based algorithm alternated in the

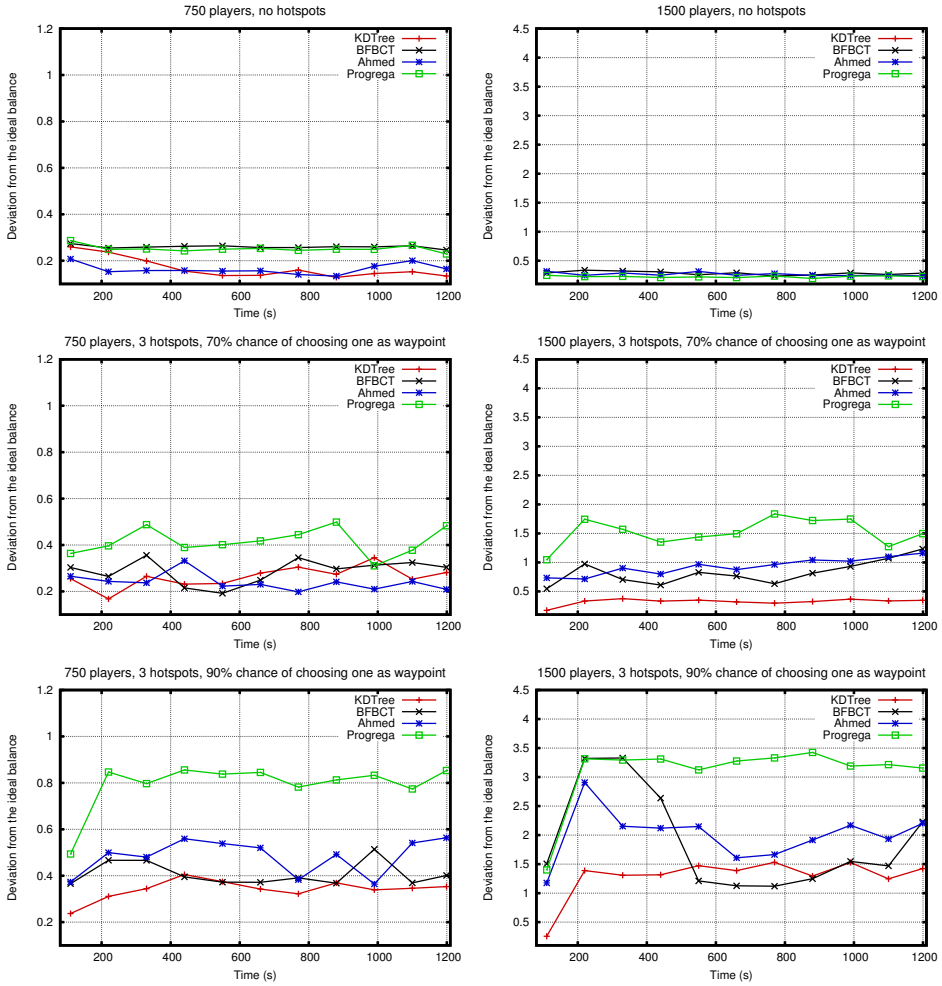


Fig. 13. Average deviation of the ideal balance of the servers in each scenario

first place. The reason for this is that the main goal of Progrega – besides balancing the load – is precisely to reduce the communication between servers. However, even though it was not built specifically for reducing the overhead due to inter-server communication, the algorithm proposed here got second place overall 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 the nodes of the kd-tree. One of the conclusions reached was that the use of such data structures to make this partitioning allows a fine granularity of the load distribution, while the readjustment of the regions becomes

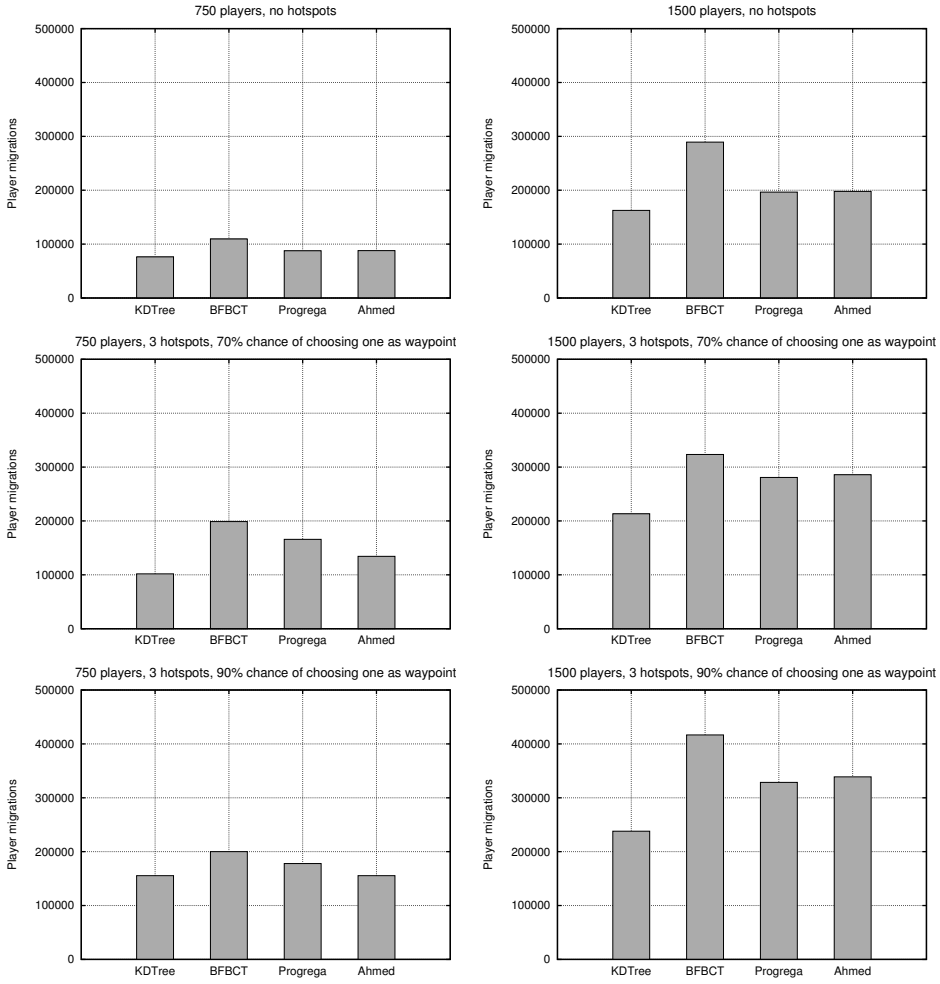


Fig. 14. Player migrations between servers in each scenario

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.

We have made an extensive set of simulations, comparing all the algorithms in six different scenarios, varying the number and distribution of avatars throughout the virtual environment of the game. By performing these tests, we have concluded

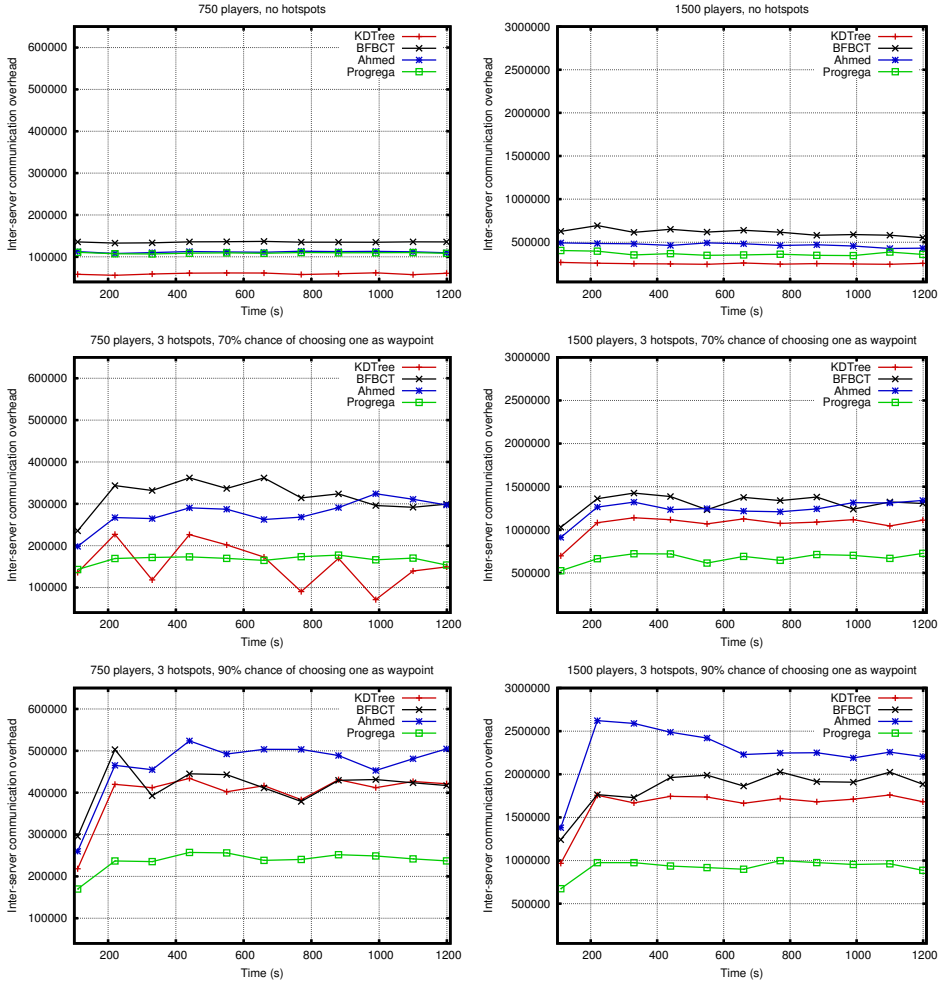


Fig. 15. Inter-server communication over time, for every algorithm in each scenario

that the proposed algorithm not only performs better than the others, but also that it scales better, at least in the kind of situations we considered, which we tried to make as close as possible to real MMOG scenarios – although we defined very high load values in order to push the simulated algorithms and verify their behavior in worst case scenarios.

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, second, 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.

6.1 Acknowledgments

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*. Istanbul, Turkey, 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., 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.

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*.

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. Raganar 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.

Samet, H. 2005. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

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.

Received December 2009