Hybrid Load Balancing for Online Games

Rynson W.H. Lau

Department of Computer Science, City University of Hong Kong, Hong Kong

ABSTRACT

As massively multiplayer online games are becoming very popular, how to support a large number of concurrent users while maintaining the game performance has become an important research topic. There are two main research directions based on the multi-server architecture, global load balancing, which is optimal but computationally expensive, or local load balancing, which is not optimal but efficient. In this paper, we propose a hybrid load balancing approach to support massively multiplayer online gaming. Our idea is to augment a local load balancing algorithm with some global load information, which may be obtained less frequently. We propose two methods to implement the hybrid approach. Our results show that the proposed methods reduce the frequency of server overloading and improve the overall game performance significantly.

Categories and Subject Descriptors: C.4 [**Performance of System**]: Performance attributes. I.3.2 [**Graphics Systems**]: Distributed/network graphics.

General Terms: Algorithms, Performance, Design, Experimentation, Theory.

Keywords: Multiplayer online gaming, multi-server architecture, distributed virtual environments, distributed load balancing.

1. INTRODUCTION

Multiplayer online games are becoming very popular in recent years. They allow remote users (or game players) to participate and collaborate in a game mission over the Internet. One of the basic requirements of computer games is interactive response. As the user interacts with an object or another user in a game, it expects the object/user to respond immediately to the interaction. However, as the number of users accessing the game increases, the server may become overloaded and may not be able to react to the users' actions immediately. This seriously affects the interactivity of the game and the users' interest on the game.

To address this scalability problem, multi-server methods may be used. However, unlike the traditional load balancing problem in distributed systems where the processing loads may be assigned to any servers available, a user in a computer game can see other users around him inside the game scene. Hence, update messages will need to be sent among the corresponding clients via the server. However, if two clients are served by two different servers, extra messages will need to be sent between the two servers, adding extra communication costs and delay. To address this problem, we need to try and cluster nearby users to be served by the same server as much as we can.

There are a number of multi-server methods proposed to handle the increasing workload of online games, as a result of the increasing number of users. In this paper, we discuss the limitations of existing multi-server approaches and propose a hybrid approach to address the load balancing problem of multi-server online games. Our approach makes use of the global load information, which may not need to be updated very frequently, to estimate locations of high processor availability. When an overloaded server needs to perform a local load balancing process, it may identify a neighbor server that is closest to the direction where there is high processing availability for load transfer. Our experimental results show that this helps achieve much better long-term load balancing by reducing the average number of overloaded servers.

The rest of the paper is organized as follows. Section 2 briefly summarizes existing multi-server works. Section 3 presents our hybrid load balancing method. Section 4 discusses some experiments and Section 5 briefly concludes this paper.

MM'10, October 25–29, 2010, Firenze, Italy.

2. RELATED WORK

A number of multi-server methods have been proposed to address the server overloading problem of online games. A popular approach used by most online games is to simply divide the users into groups, with each group served by one server, e.g., Quake III Arena (*www.idsoftware.com*) and Diablo II (*www.blizzard.com*). When the number of users served by a server reaches a certain value, a new server is started to serve additional users. However, as the users served by different servers may not interact with each other, each server may be considered as running a separate game.

Instead of dividing the users, a different approach is to divide the game scene into static regions, with each region served by one server, e.g., EverQuest (*everquest.station.sony.com*), Ultima Online (*www.uo.com*) and Asheron's Call (*ac.turbinegames.com*). However, when a large number of users move into the same region, the server serving the region can still be overloaded. An enhanced approach to this problem is to dynamically subdivide the game scene into regions, with each region served by one server. Currently, there are two main directions of research, *global load balancing* and *local load balancing*.

In [Lui02], a global load balancing method is proposed. It models the loads of a distributed virtual environment (DVE) as a graph, with each node representing a user and each edge representing the communication costs between two adjacent nodes. The load balancing problem becomes finding the best way to partition the graph into regions (or partitions) to be handled by different servers. Although this approach may produce the best partitions, it is NP-complete, involving processing all the nodes in the DVE. As such, it is very costly and difficult to meet the real-time requirement of DVEs. [Mori05] proposes a global load balancing method that uses a monitoring server to keep check of the load generated by each of the users and hence the total amount of loads of each server. It reassigns the users to different servers in order to redistribute the loads among the servers. However, this monitoring server may potentially become a center of failure. In addition, as the load balancing process is performed remotely by this monitoring server, it may actually take longer for an overloaded server to start redistributing the excessive loads.

In [Ng02], a local load balancing method is proposed. It divides the DVE into regions, each served by a server. When a server is

overloaded, it only contacts its neighbor servers to determine a suitable server for load redistribution. Results from the paper show that this method is very efficient. However, since the overloaded server does not have the load status of all servers, the load balancing process may only be considered as a short-term one and the server may quickly become overloaded again. We will explain this problem in Section 3. In [Lee03, Lee07], a method is proposed to extend [Ng02] to include additional servers for load redistribution. An overloaded server will first look at its neighbor servers to see if it may transfer all its extra loads to its neighbors. If not, it will consider the neighbor servers of the available neighboring servers and so on. When there are enough available servers to take up all the excessive loads, the regions managed by all these selected servers will be repartitioned using a graph partitioning algorithm similar to [Lui02]. This method is more efficient than [Lui02] but slower than [Ng02]. Again, as the server does not have a global picture of server load information, it may only be considered as a short-term solution.

Finite element analysis research faces similar problems to ours in that they need to dynamically subdivide a mesh for processing and analysis. However, as their concern is more on the quality of the subdivision, they typically make use of global information to optimize the partitioning [Diek00, Hu01, Ou97]. Hence, their methods are typically too slow for our purpose.

3. HYBRID LOAD BALANCING

In general, the global load balancing approach considers the load information of all servers and processes all the nodes to determine an optimal solution to partition the game scene. Hence, methods of this approach are generally very slow. On the other hand, the local load balancing approach only involves the local servers in the load redistribution process. While this may improve the efficiency of the algorithm, the resulting solution is typically short-term, due to the lack of consideration of global server load status. Figure 1 shows an example where a large crowd of users is moving from left to right in the game scene. This causes server S_o, which is managing region R_o, to be overloaded. With the local load balancing approach, S_o will tend to redistribute the load to the upper servers (i.e., S_N and S_{NE}), as they have the lowest load among all the neighbor servers of S_o. However, this will cause S_N and S_{NE} to get overloaded as the crowd continues to move to the right. With S_N, S_{NE}, and possibly their surrounding servers too, being overloaded, it will be difficult for S_N and S_{NE} to start the load redistribution without first waiting for their surrounding servers to complete theirs. This will seriously affect the interactivity of those users served by S_N and S_{NE}. On the other hand, if we have a global picture of the server load status, it is not difficult to recognize that a better solution is for S_o to redistribute its load to the lower servers (in particular, S_{SW} and S_S), as there are a lot of lightly loaded servers in the lower half of the game scene.

There are generally two kinds of loading that we need to consider, server load and communication load. The server load is defined as the processing cost of updating the status and relevant data structures of each user (or each dynamic object) managed by the server, at each frame as the user moves. The communication load is defined as the communication cost of synchronizing the state of each user among relevant servers at each frame as the user moves and the need to transfer user information across servers when a server is overloaded. Our objective here is to minimize both the processing cost and the communication cost in each server, while at the same time, minimizing the two costs as a result of the load balancing algorithm itself.

The basic idea of our approach is to augment the local load balancing algorithm with some global load information. With the load information of every server, we identify server clusters with high processor availability. Our load balancing process is based on the local approach and we only search the neighbor servers for load transfer. However, as we select a neighbor server for load transfer, we do not only look at the loading of the neighbor servers as in the other local load balancing methods, we also look at globally where there is a high processor availability and we choose a neighbor server which is as close in direction as possible to the location where there is a high processor availability. Using Figure 1 as an example, we may select server S_{SW} or S_S, instead of S_N or S_{NE}, for load transfer, as they are along the direction to where there are a lot of free regions. However, if both S_{SW} and S_S are also highly loaded, then we may prefer to select S_{SE}. We discuss how we obtain the global load information and how we make use of it in a local load balancing algorithm next.

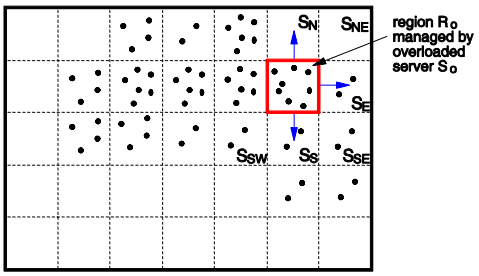

Figure 1. An example that local load balancing is less effective.

3.1 Global Loading Information

Let the game scene, ***GS***, be divided into n regions, such that ***GS*** = $\{R_1, R_2, \ldots, R_n\}$, and served by n servers, ***S*** = $\{S_1, S_2, \ldots, S_N\}$. Whenever we receive the load information from all server, ***L*** = $\{L_1, L_2, \ldots, L_N\}$, we first compute a processor availability map, which is essentially a negation of ***L***, ***M*** = $\{M_1, M_2, \ldots, M_n\}$, where $M_i = L_{max,i} - L_i$ and $L_{max,i}$ is the maximum load that server S_i can handle. Based on this availability map, we then identify some potential target server clusters ***G*** = $\{G_1, G_2, \ldots, G_m\}$ that have high processor availability, where m is the number of server clusters. A server cluster is a group of servers serving connected regions of the game scene and a target server cluster is a server cluster with low overall load such that it may accept excessive loads from overloaded servers. We then compute the total processor availability A_i of each target server cluster G_i by summing up the processor availability of each server within G_i as:

$$A_i = \sum_{k|R_k \in G_i} M_k \quad (1)$$

To define a direction from an overloaded server S_o to a target server cluster G_T for load transfer, we compute the center of mass as the reference point. We first need to obtain the center of mass from each server, ***P*** = $\{P_1, P_2, \ldots, P_n\}$, where P_i represents the center of mass of R_i computed by finding the mean location of all users U_k within R_i as: $P_i = (\sum_{k|U_k \in R_i} p_k) / |R_i|$, where p_k is the location of U_k in R_i and $|R_i|$ returns the total number of users within R_i. We then compute the center of mass C_i of each target server cluster G_i by finding the mean center of mass of all regions within G_i as:

$$C_i = \frac{\sum_{k|R_k \in G_i} P_k M_k}{\sum_{k|R_k \in G_i} M_k} \quad (2)$$

Given an overloaded server S_o, we first select the most suitable target server cluster G_T from the set of potential target server clusters $\boldsymbol{G}$ by considering the total processor availability A_i and the Euclidean distance between the center of mass P_o of S_o and the center of mass C_i of G_i. We prefer a target server cluster with higher processor availability and nearer to the overloaded server.

There is an important note here. For a large game with servers physically distributed at different sites, it is often not possible, or otherwise too expensive, to obtain synchronized load information from all servers within each frame time. The advantage of forming server clusters here is that the processor availability information tends to be more reliable over a longer period of time for a larger cluster of regions than for a single region, as the load balancing process needs time to diffuse the load across regions.

3.2 Server Clustering

A main issue of implementing our hybrid load balancing approach is on how to construct target server clusters. We have attempted two techniques here. The first one is to go through the load map. If a server has a load below a threshold (we set this threshold to the average server load, i.e., total load / number of servers), we use it as a seed. We check its first-ring neighbor servers. All servers with a load below the threshold will join with the seed to form a potential target server cluster. We then expand this server cluster by progressing to the second-ring and third ring neighbor servers, until either the sum of all the loads exceeds a certain threshold or the area coverage of the cluster is too high. During runtime, an overloaded server may select a suitable cluster from one of the target server clusters, based on the total processor availability (Eq. (1)) and the distance of its center of mass (Eq. (2)). We refer to this technique as *general clustering*.

The second technique that we propose is an approximation method of the first. Given an overloaded server S_o, we divide the game scene into four quarters, the upper left, upper right, lower left and lower right quarters using S_o's center of mass as a reference, as shown in Figure 2. A region is said to belong to a quarter if its center of mass falls inside the quarter. We may then compute the total processor availability and the center of mass for each of these four quarters to guide the load redistribution process. We refer to this technique as *quarter clustering*.

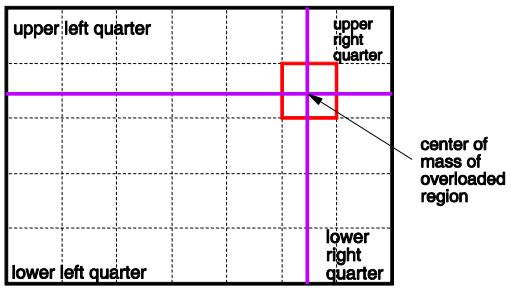

Figure 2. The game scene is divided into four quarters.

3.3 Our Hybrid Load Balancing Algorithm

Our local load balancing algorithm is based on the one proposed by [Ng02]. When a server is overloaded, it checks its neighbor servers and identifies the lightest loaded server(s) to redistribute the load. Our idea here is that as we identify the neighbor servers for load redistribution, we consider not only the loading of the neighbor servers, but also the global load information, i.e., the locations where there are high processor availability. Our hybrid load balancing algorithm is summarized as follows:

1. A process, called the load collector (LoC), collects the load value L_i and the center of mass P_i from each region R_i of the game scene $\boldsymbol{GS}$ regularly. (In our implementation, we set it to once a second.) The LoC then constructs the processor availability map $\boldsymbol{M}$ and the set of centers of mass $\boldsymbol{P}$. It broadcasts $\boldsymbol{M}$ and $\boldsymbol{P}$ to each server regularly.
2. When server S_o is overloaded, it checks the latest load information of its neighbor servers. By neighbor servers, we refer to those servers managing regions that are immediate neighbors to R_o.
3. Based on the load information of the neighbor servers, S_o identifies all potential target neighbor servers, i.e., neighbor servers that may help take up some excessive load from S_o.
4. Given the latest global load information, i.e., $\boldsymbol{M}$ and $\boldsymbol{P}$, S_o obtains the set of potential target server clusters, $\boldsymbol{G}$. (When quarter clustering is used, this set will contain the four quarters.) Each server cluster will have two values, the total processor availability and the center of mass.
5. S_o selects the target server cluster, G_T, by considering the processor availability and the distance of the center of mass of each cluster from S_o. The actual weights of the two factors are application dependant.
6. Finally, from among the potential target neighbor servers, S_o selects the server which is closest in direction to G_T as the target server, S_t, for load redistribution. In case if S_t cannot take up all the excessive load, S_o then selects the next target server which is next closest in direction to G_T for further load redistribution. This process repeats until S_o has redistributed all its excessive loads.

4. RESULTS AND DISCUSSIONS

To evaluate the performance of the proposed method, we have implemented it in C++ and performed a number of simulated experiments on it. Our testing platform is a PC with an Intel Core Duo 2.26GHz CPU and 2GB RAM. We divided the game scene into 64 regions managed by 64 servers. Figure 3 shows a snapshot of part of the game scene, with each color region representing one of the 64 regions. To model a large number of users, we generate 300 virtual users, each moving around randomly within a small local area, and another 700 virtual users which form a crowd moving together across the center of the game scene. These virtual users are shown as black dots in Figure 3. In our experiments, we assume that the loads generated by all users are the same and that each server can handle a maximum of 25 users.

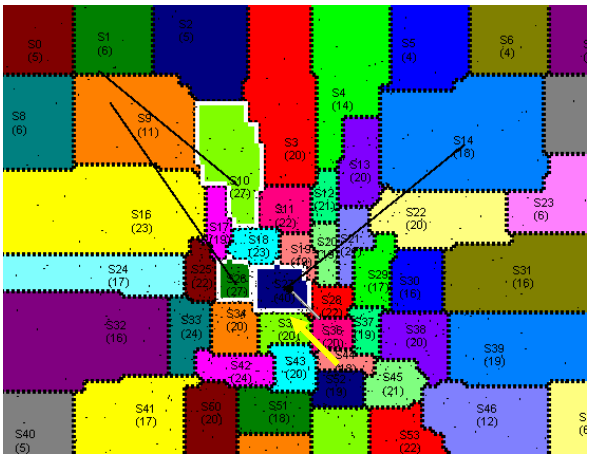

Figure 3. A snapshot of part of a game scene.

In Figure 3, each region shows the server ID and the number of virtual users within it. When a server is overloaded, it is marked with a white boundary. Here, there are three overloaded servers and three black lines connected to them. Each black line joins the center of mass of an overloaded region to the center of mass of the target cluster, indicating the direction of load redistribution. The yellow arrow indicates the direction of crowd movement. Based on these settings, we briefly describe two of the experiments that we have performed.

4.1 Experiment 1

Our first experiment is to study the computational cost of the proposed methods. Table 1 shows the average computation times of three methods: the original local load balancing method in [Ng02], our general clustering method and our quarter clustering method proposed in this paper. All three methods execute the same local repartitioning step, while the two methods that we propose require an additional server clustering step. From Table 1, we can see that the total computational cost of the quarter clustering method is about 2 times that of the original method and the total computational cost of the general clustering method is about 2.7 times that of the original method. The server clustering step of the general clustering method is more expensive than that of the quarter clustering method due to the additional cost in identifying neighboring servers to form clusters.

There is one important point to note here. In both our general and quarter clustering methods, there is no need to perform the server clustering step every time when the server is overloaded, since the computed clusters will be valid until the server has received an update of the global load information. Since we only update the global load information once a second in our experiments, we only need to perform the server clustering step once per second. As such, the computational cost of the server clustering step may be neglected. In terms of communication cost, the two proposed methods only need to obtain the load value and the center of mass from each server once a second. The additional communication cost of the proposed methods can be neglected.

Table 1. Computational cost of the proposed method.

	Original Method	General Clustering	Quarter Clustering
Server Clustering Step	0 ms	10.63 ms	6.21 ms
Repartitioning Step	6.16 ms	6.16 ms	6.16 ms
Total Time	6.16 ms	16.79 ms	12.37 ms

4.2 Experiment 2

Our second experiment is to study the effectiveness of the proposed methods in reducing the chance of server overloading. Figure 4 shows the numbers of overloaded servers of the same three methods during a frame sequence. We can see that both the general and quarter clustering methods have much lower numbers of overloaded servers than that of the original method, resulting in a much better server performance. The advantage of augmenting the local load balancing method with global load information is very significant. However, we are surprised to see that the general clustering method produces nearly the same number of overloaded servers as the quarter clustering method does. We were in fact expecting a lower number as the general clustering method was supposed to produce more accurate clustering. Our explanation of this phenomenon is that as it may take many frames for the load from the overloaded server to reach the target server cluster, the computed target server cluster at each frame only serves as a reference and hence its accuracy is not critical. As a result, we may conclude that the quarter clustering method is better due to its simplicity. In terms of communication cost, since our methods have a much lower number of overloaded servers, the amount of users needed to be transferred, i.e., communication cost, is expected to be lower than the original methods

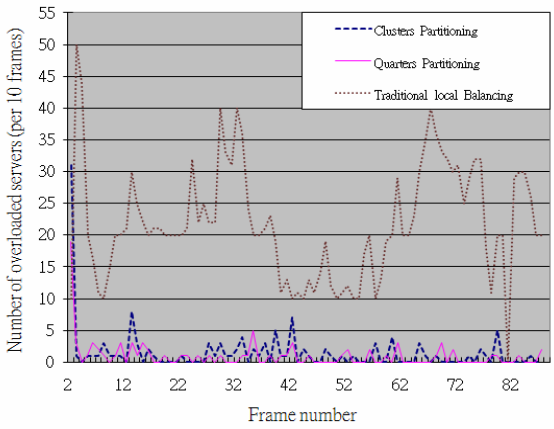

Figure 4. Server overloading with the proposed methods.

5. CONCLUSION

We have discussed two main multi-server approaches to handle the increasing number of users in online games, the global load balancing approach and the local load balancing approach. In general, the main difficulty of these two approaches is to be able to perform effective load balancing with minimal computational costs. To address this issue, we have proposed in this paper a hybrid load balancing approach by augmenting a local load balancing algorithm with global load information to guide the direction of load redistribution. We have proposed two methods of this approach, the general clustering method and the quarter clustering method. Our results show that the proposed approach is effective in reducing the number of server overloading and the quarter clustering method is preferred due to its simplicity.

REFERENCES

[Diek00] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw, "Shape-optimized Mesh Partitioning and Load Balancing for Parallel Adaptive FEM," *Parallel Computing*, **26**:1555-1581, 2000.

[Hu98] Y. Hu, R. Blake, and D. Emerson, "An Optimal Migration Algorithm for Dynamic Load Balancing," *Concurrency: Practice and Experience*, **10**(6):467-483, Dec. 1998.

[Lee03] K. Lee and D. Lee, "A Scalable Dynamic Load Distribution Scheme for Multi-Server Distributed Virtual Environment Systems with Highly-Skewed User Distribution," *Proc. ACM VRST*, 2003.

[Lee07] D. Lee, M. Lim, S. Han, and K. Lee, "ATLAS: A Scalable Network Framework for Distributed Virtual Environments," *Presence*, **16**(2):125-156, April 2007.

[Lui02] J. Lui and M. Chan, "An Efficient Partitioning Algorithm for Distributed Virtual Environment Systems," *IEEE Trans. on Parallel and Distributed Systems*, **13**(2):193-211, Mar. 2002.

[Mori05] P. Morillo et al., "Improving the Performance of Distributed Virtual Environment Systems," *IEEE Trans. on Parallel and Distributed Systems*, **16**(7):637-649, 2005.

[Ng02] B. Ng, A. Si, R. Lau, and F. Li, "A Multi-server Architecture for Distributed Virtual Walkthrough," *Proc. ACM VRST*, p. 163-170, Nov. 2002.