A Dynamic Load Sharing Algorithm for Massively Multiplayer Online Games

Ta Nguyen Binh Duong   Suiping Zhou
School of Computer Engineering
Nanyang Technological University, Singapore 639798

***Abstract*-To support hundreds of thousands of players in massively multiplayer online games, a distributed client-server architecture is widely used in which multiple servers are deployed and each server handles a partition of the virtual world. Because of the unpredictable movements and interactions of avatars, the concentration of avatars in some regions of the virtual world may cause some servers be overloaded. Existing load balancing schemes for distributed virtual environments and multiplayer games try to balance the workload among servers by transferring some workload of an overloaded server to other servers. While load balancing algorithms can minimize the average response time of the system, they may also result in frequent client migrations, which may damage the interactivity of an online game. In this paper, we propose a dynamic load sharing algorithm together with an efficient client migration scheme based on the concept of subscription regions. Simulation study has also been done to verify the effectiveness of our scheme.**

I. INTRODUCTION

With recent advances in computer graphics, network technologies and CPU power, Internet-based massively multiplayer online games (MMOGs) have attracted significant attentions as a new stream of the entertainment industry. The online gaming market are growing rapidly, mainly based on a monthly subscription model: unlike traditional PC games, MMOG players have to pay a subscription fee each month in order to continue playing the game. In a research report [1] released by Zona Inc., and Executive Summary Consulting Inc., the total subscription fee of an estimated number of 6 million MMOG players around the world in 2002 is nearly $500 million, and it is expected to be $2.7 billion in 2006 with a number of 19 million players around the world.

MMOGs can be characterized as online games in which thousands, or even hundreds of thousands players interact with each other in a very large virtual world. Some of the most popular MMOG titles are Ultima Online [10], Everquest [11] and Asheron's Call [12], etc. Because of the huge number of simultaneous players and the use of Internet as a communication medium, MMOGs face some challenging problems. For example, the non-deterministic and large message transmission delay in the Internet may cause inconsistency in players' perceptions and bad interactivity. Moreover, the increasing number of players in the game may cause serious scalability problem.

Currently there are two common network architectures for Internet multiplayer games: the peer-to-peer architecture, in which clients are connected directly with each other, and the client-server architecture, in which clients are all connected to a central server. A peer-to-peer architecture has the following advantages: minimized transmission latency, no single point of failure, and the game companies do not have to invest too much money on server hardwares and network bandwidth. The disadvantages are the high potential of cheating and difficulties in game state management. In contrast, a client-server architecture facilitates the game state management and prevents cheating. However, these advantages are achieved at the cost of the expenses on servers and bandwidth, and the decreased interactivity since all actions issued by a client have to be sent to the server before they are distributed to other clients [13]. Currently most game companies adopt the client-server architecture. Clients may connect to the game server via cable modem, Digital Subscriber Line (DSL) or even dialup connections.

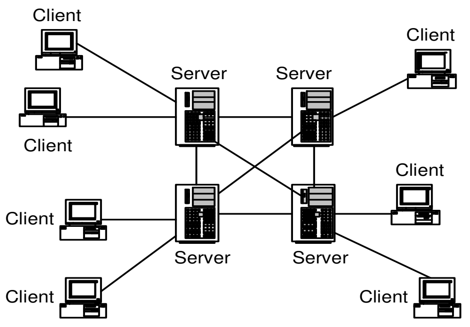


Fig. 1. The distributed client-server architecture

Typically, each client requires many kinds of resources on the server side: memory, CPU cycle and network bandwidth. As the number of clients increases, a single powerful server may not satisfy the high demands on resources. In [3], a distributed client-server architecture is proposed to support large-scale interactive games on the Internet, as shown in Fig. 1. In this architecture, multiple servers are connected to each other in a peer-to-peer mode via high speed links (e.g., LAN or optical fibre), and a client is connected to one of the servers. Clients can be distributed among servers based on their physical locations in the real world or their virtual locations in the virtual world. With the *physical location based partitioning*, a client connects to a server with the least *ping* time. In this case, the huge virtual world needs to be mirrored at every server, which is not scalable. On the other hand, with the *virtual location based partitioning,* each server manages a portion of the whole virtual world. Client connects to a server if its avatar is currently residing in the portion managed by this server.

The virtual location based partitioning has a better scalability in that each server only needs to manage a portion of the whole virtual world. However, due to the movements of the avatars in the virtual world, we cannot expect a uniform distribution of avatars in the entire virtual world. Some regions of the virtual world may attract more avatars than other regions. Thus, some servers may be overloaded with many clients, which may damage the interactivity of the game.

In this paper, we will address the scalability problem in MMOGs, assuming a distributed client-server architecture with virtual location based partitioning. A load sharing algorithm is proposed to facilitate an overloaded server to transfer some of its load to other non-overloaded servers. In addition, a client migration scheme is proposed to reduce the overhead of the load sharing algorithm.

The rest of the paper is organized as follows: Section II describes the related work. Section III discusses the requirements on a load distributing algorithm for MMOG. Section IV presents our load sharing algorithm. Simulation studies are described in section V, and section VI summarizes the paper.

II. Related Work

A. *Load distributing in distributed systems*

The load distributing problem in conventional distributed systems has been studied extensively in the past decades and can be classified into static, dynamic and adaptive algorithms, where adaptive algorithms may be regarded as a special class of dynamic algorithms. Dynamic load distributing algorithms can be further classified into load sharing algorithms and load balancing algorithms [2]. A load sharing algorithm tries to avoid the situation in which some nodes are overloaded while other nodes are underloaded. A load balancing algorithm also tries to avoid this situation by attempting to equalize the workload of each node in a distributed system. In general, load balancing is more efficient in reducing the average response time of tasks compared to load sharing, but the additional overhead of balancing the workload among servers may damage the algorithm's effectiveness [2].

Generally, a load distributing algorithm consists of the following components: transfer policy, selection policy, location policy and information policy. The transfer policy determines if a node should participate in load distributing or not. Threshold, measured by units of load, is commonly used in the transfer policy. An effective load index in distributed systems is CPU queue length [2]. The selection policy selects a task for transfer. The location policy finds a suitable "transfer partner" via polling in a decentralized location policy or via a coordinator in a centralized location policy. The information policy determines when the state information of nodes should be exchanged, which may include demand-driven (sender-initiated, receiver-initiated and symmetrically-initiated), periodic and state-change-driven policies [2].

B. *Load balancing in distributed virtual environments and MMOG*

As an interesting research area, Distributed Virtual Environments (DVEs) have been studied for some time. DVE provides a 3D virtual world in which many users can interact with each other at the same time. Some examples of academic DVE systems are NPSNET, DIVE, BrickNet, etc [6]. In fact, a MMOG may be regarded as a DVE in which multiple game players interact with each other on the Internet. To support many users in a DVE application simultaneously, a distributed client-server architecture may be used, e.g., in DIVE and in BrickNet. Thus, how to avoid bottlenecks caused by overloaded servers is an important issue in these systems.

To enhance the scalability of the system and avoid overloaded servers, generally the virtual world in a DVE or a MMOG is divided into partitions and each server manages one partition of the virtual world. These partitions can be resized dynamically according to the load (measured by the number of clients) of each server. In conventional distributed systems, the tasks submitted by users are the units that will be transferred in load distributing, and they can be transferred to any eligible nodes in the system. In DVEs and MMOGs, regions of a virtual world partition, which contain the avatars of some clients, will be transferred from an overloaded server to its neighbor servers only. Here, neighbor servers of a server S_i are the servers that manage the partitions adjacent to S_i's partition.

In [4], the authors proposed a load balancing algorithm for the distributed client-server architecture in MMOGs based on a space partitioning strategy. The virtual world is divided vertically against the X-axis. Each game server manages a partition of the virtual environment. Dynamic load balancing is achieved by relocating the partition line along the X-axis. An overloaded server may transfer some of its load to its two neighboring servers only. However, if these two neighbors cannot help, cascading load migrations may occur, which may damage the interactivity of the application.

In [5], Lui and Chan uses a parallel incremental graph partitioning algorithm to divide the load among distributed servers. The DVE system is mapped into a graph G = (V,E), in which V represents all avatars in the system and E contains all edges. An edge represents that there is communication between two avatars. Participants in the DVE will be migrated according to the result of the graph partitioning algorithm. The algorithm may take much execution time for large-scale DVE systems. Thus, the partitioning algorithm is not really suitable for large-scale real-time applications such as MMOGs.

In [7], [8] and [9], the virtual world is divided into a grid of cells. Each server manages a group of adjacent cells. Load balancing is achieved by transferring cells from an overloaded server to its neighboring servers. However, if the workload of the system is highly skewed, these algorithms may not produce effective results since an overloaded server may not be able to transfer workload to its neighbors if all of its neighbors are also overloaded. The algorithms in [7], [8] and [9] are regarded as *local load balancing* (LLB) algorithm, in which only neighboring servers are involved, while the algorithms [4] and [5] are regarded as *global load balancing* (GLB) algorithms, in which all servers in the system are involved [14].

All of the above algorithms adopt load balancing scheme. Load balancing schemes may incur too much client migrations, which will damage the performance of the overall system.

III. Requirements On Load Distributing Algorithms For MMOGs

In this section, we will discuss some requirements on load distributing algorithms for MMOGs based on the characteristics of MMOGs. For a typical MMOG, we have the following observations:

The applications are highly interactive. MMOGs are real-time, interactive applications. Thus, a load distributing algorithm for MMOGs should not be executed very frequently and should not incur too much execution overhead on the servers, which are already busy servicing many clients.

Client migrations are heavy tasks. Load distributing in DVEs and MMOGs is implemented by transferring clients among servers. Basically, a client has to disconnect to its current server (the old server) and establish a new connection to another server (the new server), which in general may take a long time compared to the message transmission delay in the Internet. In addition, the old server has to select and forward all data and events related to the migrated client to the new server. This procedure may also take some time. Therefore, the new server may not be able to provide updates to the client immediately, i.e., the interactivity of the game may be damaged. Thus, a load distributing algorithm for MMOGs should limit the number of migrations, and a smooth client migration scheme is also necessary.

MMOG system is large. In order to support hundreds of thousands of players simultaneously, a large number of servers is required for a MMOG. For example, Everquest currently uses more than 50 servers. Thus, a load distributing algorithm should be designed carefully to avoid a significant cost on load information collection among servers.

Load cascading may cause much client migrations. Since each server manages a portion of the virtual world, the load (clients) can only be transferred directly from a server to its neighbors. When a server S_i transfers l units of load (l clients) to its neighbor S_j, the number of client migrations is l. In load cascading, a server need to transfer some of its load to its neighbor in order to receive load from other neighbors. For example, suppose server S_i wants to transfer l units of load to its neighbor S_j, and S_j has already reached its workload threshold thus cannot accept any more load. In this case, S_j will attempt to transfer l units of load to its neighbor S_k if S_k can accept the load. If successful then S_i is able to transfer l units of load to S_j. The total number of client migrations in this case is $m = l$ (from S_j to S_k) + l (from S_i to S_j) = $2l$. It is obvious that the larger is the number of servers involved in load cascading, the higher is the total number of client migrations. Thus, a load distributing algorithm for MMOG should limit the number of servers involved.

Based on the above observations, we recognize that load sharing algorithms are more suitable for MMOGs than load balancing algorithms. In general, load sharing algorithms incur less client migrations and can be executed more quickly than load balancing algorithms, since load sharing algorithms only attempt to avoid overloaded servers and do not attempt to balance the workload among servers.

IV. A Dynamic Load Sharing Algorithm

A. System model

The whole virtual world is divided into a grid of N equal square cells as shown in Fig. 2. The number of servers is n, where $n \le N$. Each server manages a group of neighboring cells, called a *partition*. Initially, we assume that all partitions have the same size, i.e., they have the same number of cells. Servers are denoted as $S_1, S_2, \dots, S_n$. Similarly, cells are denoted as $c_1, c_2, \dots, c_N$. For simplicity, we represent a partition that each server manages as a single square as shown in Fig. 3, though the size of each partition may change with time. Fig. 3 demonstrates how a group of 64 servers manage the virtual world, which is divided into 64 partitions.

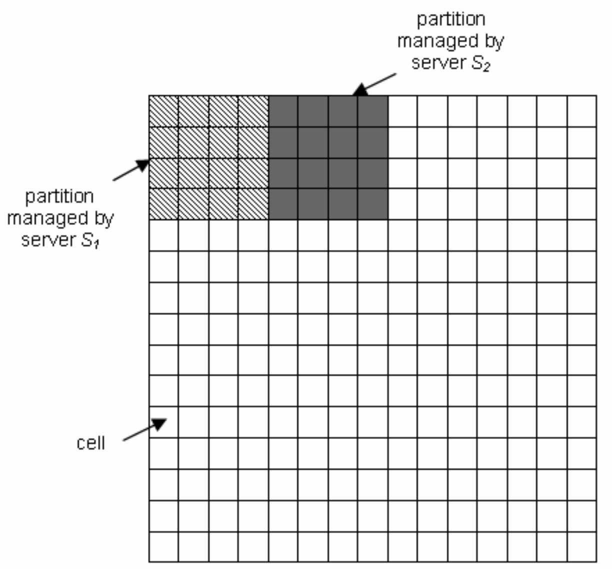

partition managed by server S_2
partition managed by server S_1
cell

Fig. 2. The virtual world represented by a grid of cells

B. Definitions

The terms and symbols that are used in the dynamic load sharing (DLS) algorithm are defined as follows:

Neighboring server. S_i is called a neighboring server of S_j and vice versa if S_i and S_j manage two partitions that share the same border line in the virtual world. For example in Fig. 3, S_1 is a neighboring server of S_2 and vice versa. However, S_1 and S_{10} are not neighboring servers of each other. The set that consists of all of S_i's neighboring servers is denoted by $NB(S_i)$.

k-level domain. The k-level domain of S_i, denoted by $D(S_i, k)$, is defined as follows:

$$D(S_i,k) = \begin{cases} NB(S_i), \text{ if } k = 1 \\ D(S_i,k-1) \cup D(S_j,1), S_j \in D(S_i,k-1), \text{ if } k > 1 \end{cases}$$

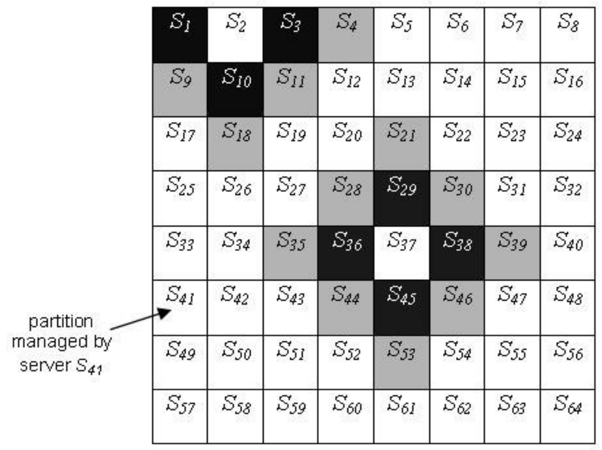
k is called the *level* of the domain.

For example, in Fig. 3, $D(S_2, 1) = \{S_1, S_3, S_{10}\}$ and $D(S_2, 2) = \{S_1, S_{10}, S_3, S_9, S_{18}, S_{11}, S_4\}$.

In the rest of this report, $D(S_i, k)$ is denoted as $D(i, k)$ for simplicity.

Load. The load of S_i, denoted by L_i, is represented by the number of clients currently connected to server S_i.

Subscription region. An avatar will receive updates from other avatars within the same partition. In addition, an avatar near the border of one partition should be able to "see" the border avatars within another partition, e.g., avatar a in Fig. 4 should receive updates of avatar b. To facilitate interactions of border avatars in adjacent partitions as shown in Fig. 4, S_i needs to subscribe to receive updates from the region belongs to S_j and adjacent to S_i. The width of this region can be set to equal to the maximum view radius of all avatars in the game [3]. In this paper, we assume that the width is the length of one cell. This region is called *subscription region* (*SR*). The region managed by S_j and subscribed by S_i is called SR_{ij}. Similarly, the region managed by S_i and subscribed by S_j is called SR_{ji} as shown in Fig. 4.


S1 S2 S3 S4 S5 S6 S7 S8
S9 S10 S11 S12 S13 S14 S15 S16
S17 S18 S19 S20 S21 S22 S23 S24
S25 S26 S27 S28 S29 S30 S31 S32
S33 S34 S35 S36 S37 S38 S39 S40
S41 S42 S43 S44 S45 S46 S47 S48
S49 S50 S51 S52 S53 S54 S55 S56
S57 S58 S59 S60 S61 S62 S63 S64
partition managed by server S41
Fig. 3. Domains

Threshold. A predetermined threshold δ is used in the algorithm. The threshold is the maximum number of clients that a server can handle smoothly. For simplicity, we assume all servers have the same processing power, thus they have the same threshold. When the load of a server S_i becomes larger than δ, i.e., $L_i > \delta$, the server is considered overloaded, and load distributing for this server is necessary.

Extra load. The extra load of a server S_i, denoted by EL_i, is defined as $EL_i = \delta - L_i$. It is the load that S_i can take from other servers without being overloaded.

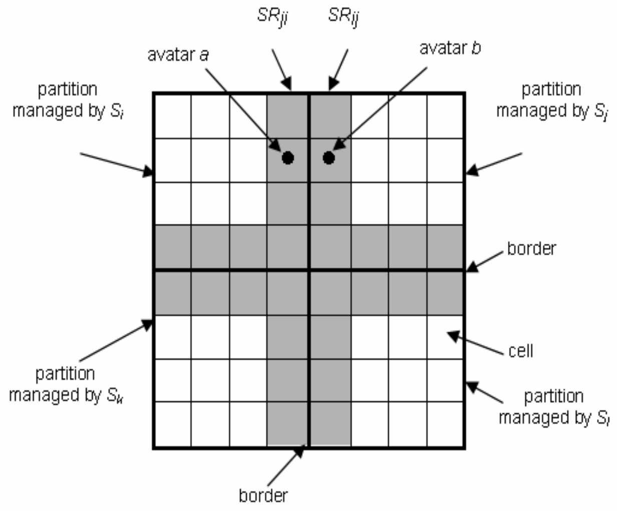


Fig. 4. Subscription regions

C. Algorithm

The DLS algorithm is a decentralized *sender-initiated* load sharing algorithm. The pseudo-codes for the algorithm are shown in Fig. 5, 6 and 7. Each server in the system will periodically check its load to perform the corresponding load sharing action, as shown in Fig. 5. At the same time, each server also waits to receive load sharing requests from other servers, as shown in Fig. 6.

In Fig. 5, if a server S_i is overloaded, i.e., $L_i > \delta$, it will attempt to share its load with servers in its k-level domain $D(i, k)$. Initially $k = 1$, which means S_i will share the load with its neighboring servers first. Function *send*(*load_sharing*,OL_i, S_j, k) sends a message to a server S_j in $D(i, k)$, indicating that S_i wants to transfer load OL_i. This function then waits to receive S_j's reply and transfer some of its load to S_j if S_j is able to take more load. This procedure is repeated until $OL_i = 0$, or $k > MAX\_LEVEL_i$ (when $k = MAX\_LEVEL_i$, all the servers in the system belong to $D(i, k)$).

```
begin
  $OL_i = L_i - \delta$;
  k = 1; /*level of domain*/
  while(($OL_i$ > 0) or (k <= $MAX\_LEVEL_i$)) do
    /*send load_sharing to neighbors only*/
    for(each $S_j \in D(i,1)$) do
        $EL_j$ = send(load_sharing,$OL_i$,$S_j$,k);
        if($EL_j$ > 0)
          transfer($EL_j$,$S_j$);
          $OL_i = OL_i - EL_j$;
          if($OL_i$ <= 0) break; /*exit for*/
        end if
    end for
    k = k + 1;
  end while
end
```

Fig. 5. Algorithm for S_i to share its load

At S_j , function *accept*(S_i) is used to read the value OL_i of the load that S_i wants to transfer, as shown in Fig. 6. If $k = 1$, function *reply*(EL_j , S_i) is called to answer immediately the *load_sharing* request from S_i as shown in Fig. 6, where EL_j indicates the extra load that S_j can take without being overloaded. If $k > 1$, S_j will seek help from servers in $D(j, 1)$ by sending a *load_sharing* request to each server S_m in $D(j, 1)$, except server S_i. If S_j finds out that S_m can take an extra load EL_m, it will then transfer EL_m units of load to S_m and accumulate EL_m in the variable EL as shown in Fig. 6. This procedure is repeated until $OL_i = 0$ or all servers in $D(j, 1)$ are involved. Then S_j replies S_i that it can take an extra load of $EL = \sum EL_m$.

The load sharing procedure at S_i continues until $OL_i = 0$ or all servers in the system are involved, i.e., $k > MAX\_LEVEL_i$.

To transfer load L from S_i to S_j , function *transfer*(L, S_j) in as shown Fig. 7 is executed at S_i. To facilitate smooth client migrations, the function selects to transfer the cells that are in SR_{ji}. Cells that are near to the border of S_i and S_j are transferred first. Since S_j subscribed to receive all updates from the SR_{ji}, S_i does not need to forward to S_j the data and events related to the avatars in SR_{ji} when these avatars are migrated to S_j. This results in less client migration overhead. After transfer a cell c_m to S_j, all adjacent cells of c_m are added to SR_{ji}. This procedure is repeated until all the required load L is transferred.

```
begin
  for(each load_sharing request) do
    $OL_i$ = accept($S_i$);
    if(k == 1)
      /* $S_i$ needs to transfer a load $OL_i$*/
      $EL_j$ = $\delta$ - $L_j$;
      if($EL_j$ <= $OL_i$)
        reply($EL_j$,$S_i$);
        $L_j$ = $\delta$;
      else
        reply($OL_i$,$S_i$);
        $L_j$ = $L_j$ - $OL_i$;
      end if
    elseif(k > 1)
      /*$S_i$ needs to transfer a load $OL_i$*/
      for(each $S_m \in D(j,1)$ and $S_m$ # $S_i$) do
        $EL_m$ = send(load_sharing,$OL_i$,$S_m$,k-1);
         /*variable to accumulate load*/
        EL = 0;
        if($EL_m$ >= $OL_i$)
          transfer($OL_i$,$S_m$);
          EL = EL + $OL_i$;
          break;/*exit for*/
        elseif($EL_m$ > 0)/*0 < $EL_m$ < $OL_i$*/
          transfer($EL_m$,$S_m$);
          EL = EL + $EL_m$;
          $OL_i$ = $OL_i$ - $EL_m$;
          if($OL_i$ <= 0) break /*exit for*/
        end if
      end for
      reply(EL,$S_i$);
    end if
  end for
end
```

Fig. 6. Algorithm for S_j to receive load from S_i

```
begin
  /*transfer load L from $S_i$ to $S_j$*/
  while(L > 0)do
    /*$c_m$ is closest to the border*/
    select $c_m \in SR_{ji}$;
    transfer $c_m$ to $S_j$;
    add all cells adjacent to $c_m$ into $SR_{ji}$;
    L = L - number_of_clients_in_$c_m$;
  end while
  return;
end
```

Fig. 7. Function *transfer*(*L*, *S_j*)

V. Experiments And Results

A. Performance metrics

The following metrics are used to evaluate the performance of the proposed load distributing algorithm:

Overload reduction ratio. This metric is defined as follows:

$$\alpha = \frac{\sum_i (LB_i - \delta) - \sum_j (LA_j - \delta)}{\sum_i (LB_i - \delta)}$$

where LB_i is the load of an overloaded server S_i (i.e., $LB_i > \delta$) right before the load distributing algorithm is executed, and LA_j is the load of an overloaded server S_j right after the load distributing algorithm is finished. It is obvious that $0 \le \alpha \le 1$, and a larger α means that the load distributing algorithm is more efficient in reducing skewed workloads that cause servers to be overloaded, which results in better average response time of the system.

Migration ratio. This metric is defined as follows:

$$\beta = \frac{M}{T}$$

where M is the total number of clients migrated due to the load distributing algorithm execution, and T is the total number of clients of the system. It is obvious that $0 \le \beta \le 1$, and a smaller β means that the load distributing algorithm causes less client migration overhead.

B. Simulation

In the simulation, the virtual world is divided into a grid of cells. Each server is responsible for only one partition which comprises 100 adjacent cells. The number of servers in the simulation is set to 16, 36, 64, 100 respectively. The threshold of each server is the same and is set to δ = 800. Three load distributing schemes are studied in the simulation: the DLS scheme, the GLB scheme and the LLB scheme.

Since the movements of human-controlled avatars in the game are unpredictable, the actual workload distribution in the system is unknown. Thus, similar to [5], three different workload distributions are used, namely uniform, clustered and skewed distribution. Due to space limitation, only the results for the skewed distribution is presented here as the worst case. The skewed distribution is simulated by uniformly distributing 25% of the total workload to 25% of partitions at one "corner" of the virtual world, and uniformly distributing the rest of the total workload to every partitions in the system. Thus, the workload is skewed at one "corner" of the virtual world.

C. Results

The simulation results are shown in Fig. 8, 9, 10 and 11. In these figures, the system load ratio is defined as the total number of clients of the system divided by $n \times \delta$, where n is the number of servers.

The skewed workload in the system cannot be well-reduced in the LLB, as shown in Fig. 8 and Fig. 10. In contrast, the overload reduction ratio of the DLS and GLB is equal to 1, which means there's no overloaded server in the system right after the DLS or the GLB is finished.

The migration ratio of the LLB is the lowest in the three schemes, as shown in Fig. 9 and Fig. 11, since only neighboring servers of an overloaded server are used in the LLB, which incurs no cascading migration. However, overloaded servers in the LLB cannot share their workloads to servers other than their neighbors, which results in highly skewed workload as expected.

The DLS and the GLB are all able to reduce all the skewed workload, but the DLS results in a less migration ratio than the GLB, as shown in Fig. 9 and Fig. 11. In addition, as shown in Fig. 9, the migration ratio of the DLS increases with the system load, while the GLB has a high migration ratio even when the system load is low.

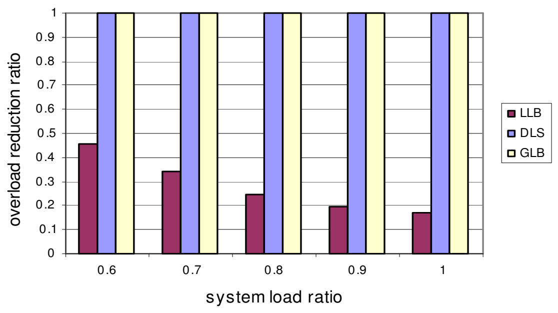


Fig. 8. Overload reduction ratio, 64 servers

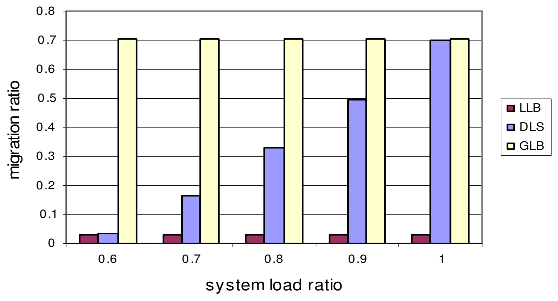


Fig. 9. Migration ratio, 64 servers

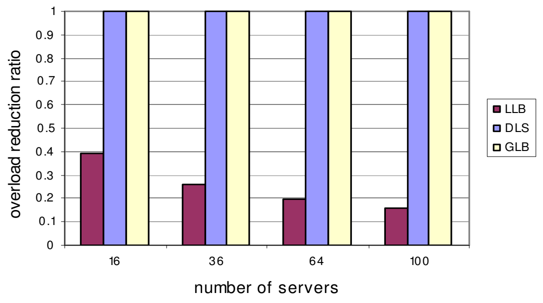


Fig. 10. Overload reduction ratio, system load ratio = 0.9

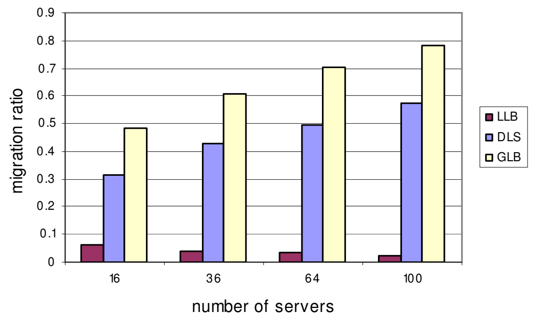


Fig. 11. Migration ratio, system load ratio = 0.9

Under the same system load ratio, the migration ratios of the DLS and the GLB both increase, as the number of servers increases, as shown in Fig. 11. This is the result of the increasing number of cascading migrations since the number of overloaded servers skewed at one "corner" of the virtual world increases with the system size.

To summarize, the proposed DLS algorithm is efficient in reducing the skewed workload, and it does not incur much client migration overheads as in the GLB algorithm.

VI. Conclusion And Future work

We have developed a dynamic load sharing algorithm for MMOGs with a distributed client-server architecture. The algorithm is based on a decentralized, sender-initiated load sharing scheme. Furthermore, a client migration scheme is also proposed to facilitate load transfers among multiple servers. Simulation study shows that the algorithm is able to reduce the skewed workload efficiently, and it does not incur much client migrations.

Our future work will be the implementation of this load sharing algorithm in a large-scale Internet game with a distributed client-server architecture.

References

[1] Zona Inc., and Executive Summary Consulting Inc, "State of Massive Multiplayer Online Games 2002: A New World in Electronic Gaming", October 2002, available at www.zona.net/news/2002mmogreport.html.
[2] G. Shivaratri, P. Krueger and M. Singhal, "Load Distributing for Locally Distributed Systems", *IEEE Computer*, 25(12), pp. 33-44, December 1992.
[3] W. Cai, P. Xavier, S. Turner and B. S. Lee, "A Scalable Architecture to Support Interactive Games on the Internet", *Proc. of the 16th Workshop on Parallel and Distributed Simulation*, pp. 60-67, May 2002.
[4] D. Min, E. Choi, Donghoon Lee and B. Park, "A Load Balancing Algorithm for a Distributed Multimedia Game Server Architecture", *Proc. of the IEEE International Conference on Multimedia Computing and Systems*, pp. 882-886, June 1999.
[5] J. Lui and M. Chan, "An Efficient Partitioning Algorithm for Distributed Virtual Environment Systems", *IEEE Transaction on Parallel and Distributed Systems*, 13(3), pp. 193-211, March 2002.
[6] S. Singhal and M. Zyda, *Networked Virtual Environments: Design and Implementation*, p37-p49, Addison-Wesley, 1999.
[7] G. Carneiro and C. Arabe, "Load Balancing for Distributed Virtual Reality Systems", *Proc. of the International Symposium on Computer Graphics, Image Processing, and Vision*, pp. 158-165, October 1998.
[8] M. Hori, T. Iseri, K. Fujikawa, S. Shimojo and H. Miyahara, "Scalability Issues of Dynamic Space Management for Multiple-Server Networked Virtual Environments", *Proc. of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pp. 200-203, 2001.
[9] Beatrice Ng, A. Si, R. Lau and F. Li, "A Multi-Server Architecture for Distributed Virtual Walkthrough", *Proc. of the ACM VRST'02*, pp. 163-170, November 2002.
[10] Ultima Online, available at www.uo.com.
[11] Everquest, available at www.everquest.station.sony.com.
[12] Asheron's Call, available at www.microsoft.com/games/zone/asheronscall.
[13] M. Mauve, S. Fishcher and J. Widmer, "A Generic Proxy System for Networked Computer Games", *Proc. of the NetGames2002*, pp. 25-28, April 2002.
[14] K. Lee and Dongman Lee, "A Scalable Load Balancing Scheme for Large-Scale Multi-Server Distributed Virtual Environment Systems", unpublished.