The design of multiplayer online video game systems

Chia-chun Hsu, Jim Ling, Qing Li, C.-C. Kuo

Chia-chun Alex Hsu, Jim Ling, Qing Li, C.-C. Jay Kuo, "The design of multiplayer online video game systems," Proc. SPIE 5241, Multimedia Systems and Applications VI, (19 November 2003); doi: 10.1117/12.512201

SPIE.

Event: ITCom 2003, 2003, Orlando, Florida, United States

On the Design of Multiplayer Online Video Game Systems

Chia-chun Hsu, Jim Ling, Qing Li and C.-C. Jay Kuo

Integrated Media Systems Center and Department of Electrical Engineering
University of Southern California, Los Angeles, CA 90089-2564

ABSTRACT

The distributed *Multiplayer Online Game* (*MOG*) system is complex since it involves technologies in computer graphics, multimedia, artificial intelligence, computer networking, embedded systems, etc. Due to the large scope of this problem, the design of MOG systems has not yet been widely addressed in the literatures. In this paper, we review and analyze the current MOG system architecture followed by evaluation. Furthermore, we propose a clustered-server architecture to provide a scalable solution together with the region oriented allocation strategy. Two key issues, *i.e.* interesting management and synchronization, are discussed in depth. Some preliminary ideas to deal with the identified problems are described.

Keywords: Online game, game server, console online game, interest management, synchronization, scalability

1. INTRODUCTION

The distributed Multiplayer Online Game (MOG) service has grown very fast in recent years. Since the great success of several early multiplayer online games around mid 90's, such as DOOM and Quake from IDSoft [1], Warcraft from Blizzard [2], MOG is getting more and more investment from the entertainment industry. There are international tournament for several popular games. Besides the computer-based MOG, manufacturers of the traditional console-based video game launched online service. PS2 online and Xbox live were announced on 2001 and 2002, respectively [3, 4]. It is forecasted that about 114 million people worldwide are expected to play online games by 2006.

Typically, MOG is a kind of distributed interactive real-time applications. Three classes are prominent among the applications with distributed interactive real-time attributes. They are *military simulations*, such as flight simulators or logistical simulations, *Distributed Virtual Environments* (DVE) or so-called *Collaborative Virtual Environments* (CVE), and the spotlighted multiplayer online games [5]. However, the MOG system brings more problems than before due to its special characteristics, *e.g.* a large number of players, heterogeneous game platforms and disparity network bandwidth of players, dazzling graphic interface, a huge amount of game content, game fairness, etc. Thus, design of MOG systems gets more attention from academia recently. Christophe and Laurent [6] built a small-scale distributed multiplayer interactive application, called *MiMaze*, and studied the architecture issue and the synchronization problem. Nathaniel *et al.* [7] presented a protocol to provide the anti-cheating guarantee in MOG. Song-Jin *et al.* [8] focused on a real-time programming methodology, where the time-triggered message-triggered object (*TMO*) with a global time-stamp was proposed for a scalable online game. Eric *et al.* [9] discussed a new synchronization scheme called trailing-tail to lower the complexity. Although MOG has a huge market potential, very little research work has been published on MOG. The current research is scattered and isolated. In this paper we attempt to give an in-deep inspection of MOG to help researchers understand the system structure, the current research stage and the challenges.

The whole MOG system consists of two basic platforms [5]. The first one is the underlying infrastructure, *i.e.* the physical platform, such as networking and hardware (including processors and graphic cards, etc.) The physical platform imposes a physical limitation on the MOG system. The second platform, so-called the logical platform, is about the system architecture, *e.g.* how hardware is organized, and how software is implemented in hosts and players. The goal of the logical platform is to provide the best performance based on the existing physical platform. In this paper, we focus on the logical platform. We first analyze three MOG system architectures: peer-to-peer, client-server and mirrored/networked server. We study their merits and shortcomings. Based on the analysis, a new architecture, i.e. server-cluster, is proposed. The server in the proposed architecture is in charge of the partial game world, which is totally different from the existing architectures. This new architecture combines advantages of peer-to-peer and client-server architectures to result in a more scalable solution. This paper is organized as follows. Section 2 describes the

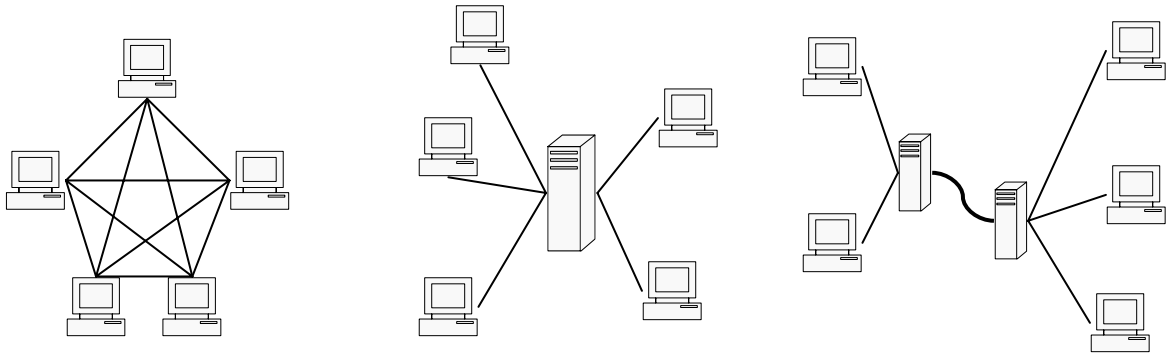
Proceedings of SPIE Vol. 5241 Multimedia Systems and Applications VI,
edited by Andrew G. Tescher, Bhaskaran Vasudev, V. Michael Bove, Jr.,
Ajay Divakaran (SPIE, Bellingham, WA, 2003) · 0277-786X/03/$15.00

current MOG system architecture and presents our new architecture. Section 3 addresses the MOG server structure, including the general conceptual model and a specific architecture. Section 4 discusses key issues on the design of MOG. Finally, concluding remarks and future work are given in Section 5.

2. SYSTEM ARCHITECTURE

We can classify logical platforms into three types: the peer-to-peer (distributed) architecture, the server-client (centralized) architecture, and the mirrored-server architecture. Since the mirrored-server is the hybrid architecture of the former two architectures, we compare the two former two cases first.

2.1 Peer-to-Peer vs. Client-Server Architecture



(a) Peer-to-Peer architecture (b) Client-Server architecture (c) Mirrored-Server architecture

Figure 1 Topology for Multiplayer Online Game system

Let us compare the characteristics of the peer-to-peer and the client-server architectures below.

- **Scalability:** Since the main server has to collect all participants' data and server all participants of a game, the server may become saturated when the number of participants goes up. The total amount of data transferred over the network can be reduced in a distributed architecture.
- **Delay:** In a centralized architecture, data reach their destination through the server, which can increase the network delay up to two times the delay in a distributed architecture, where data only crosses the network once to reach their destinations.
- **Robustness:** It is desirable that the failure of any participant has no effect on other participants in the centralized system. This means all participants are equivalent and independent, and has all necessary information to compute the state of game at any time. The server-client architecture is less robust since the system may fail due to a single point failure (*i.e.* the failure of the server).
- **Consistency:** Since all participant received information from the server only in a centralized architecture, global consistency is guaranteed. Moreover, the server introduces a natural synchronization among players.
- **Cheat-proof:** The presence of a centralized server makes cheating difficult. In a distributed architecture, each entity makes its own decisions, and there may be no authority to identify potential cheaters. Consequently, the deployment of distributed architectures will require a specific distributed mechanism to deal with cheating of participants.
- **Easy to charge:** This feature allows game companies to charge players based on the amount of time they played or the duration of their participation. It favors the centralized solution.

The first three factors favor the distributed architecture while the last three favor the centralized architecture. We can say that a distributed architecture improves scalable and real-time properties of MOG applications at the expense of ease of management. Despite the latency, single point of failure and scalability problems in the client-server architecture, it is by far the most commonly used architecture in MOG. The reasons include easier implementation (quick adaptation from the single player version) and more administrative control. Some comparison results are summarized in Table 1.

Table 1 Characteristics with three system architectures

	Peer-to-Peer architecture	Mirrored-Server Architecture	Client-Server architecture
Robustness	Good	Medium	Poor
Scalability	Good	Medium	Poor
Delay	Good	Medium	Poor
Consistency	Poor	Medium	Good
Cheat-proof	Poor	Medium	Good
Easy to charge	Poor	Medium	Good
Best fit	Good for term-base MOG		Good for small scale, latency-sensitive MOG

2.2 Mirrored-Server Architecture

The mirrored server architecture, also referred to as the server-network (or the server pool) [10], is a compromise between the client-server and the peer-to-peer architectures. As in the client-server architecture, the mirrored server can be centralized managed. However, as similar to the peer-to-peer architecture, messages do not pay the latency cost due to the traveling to a centralized server. The price is the requirement of a complicated synchronizing algorithm to maintain consistency among mirrors. If the total amount of bandwidth consumption of all three architectures is comparable, the mirrored server architecture consumes more bandwidth on the private network, whose bandwidth is plenty as compared to the busy public network. The mirrored server architecture has another advantage, *i.e.* the mirrored server architecture can use the private network between mirrored servers to allow IP multicast [9].

2.3 Clustered-Server Architecture

The game server in the mirrored server or the server network architecture contains the entire game world. The only shared resource is the bandwidth. However, there will be more intensive interactions among users in the future MOG systems. Then, the server will have heavier load for the whole game world, which may become a main issue. To our best knowledge, there is no existing MOG system they could let players interact with other players on a different server even they are playing the same game.

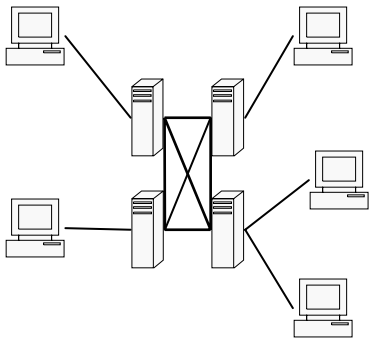


Figure 2 Clustered-server Architecture

We propose a new clustered-server architecture to improve the client-server architecture. After replacing the single central by a group of connected servers, called clustered servers or the server cluster, we can make the system more scalable. This strategy, called the *multi-server content* or *multi-homing*, is widely used in non-gaming modern websites. However, simple algorithms such as *DNS Round Robin* in *multi-homing* are not applicable to MOG. We need more advanced load-balancing and synchronization schemes among clustered servers to make MOG scalable and practical. In fact, the mirrored-server is the evolution of the peer-to-peer architecture. The major difference between them is the game world. In the mirrored-server architecture, each server has its own local copy of the game world and synchronizes with others. In contrast, each server has a partial game world and is responsible to provide a unified world-view for that part in the cluster-server architecture.

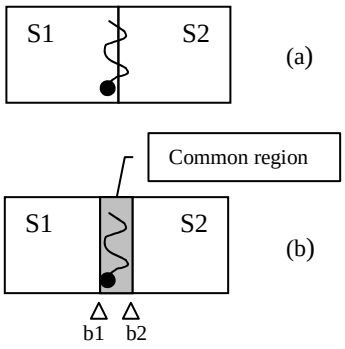


Figure 3 Common Region Method

The next question is about load distribution among servers. The *region oriented allocation* strategy is adopted by the grid-based MOG. With this strategy, the game world is divided into regions that have their own multicast group. All objects within the region send update to the corresponding server. When an object crosses the boundary of a region, a handoff operation is needed. One solution to the handoff problem is the use of the common region. A disjoint division is shown in Figure 3(a), where servers, S1 and S2, maintain their own regions. Once a player across the boundary, handoff takes place. When a player wanders around the boundary back and forth, the handoff operation must be executed repetitively, which demands a lot of resource for handoff. By

introducing a *common region* between two regions as shown in Figure 3(b), it is possible to provide a better solution. When a player or an object enters the common region from S1 as labeled by b1, S1 is responsible to maintain the player's status and keep S2 informed, only after the player leaves the common region labeled by b2, S1 would pass all the relevant information of the player to S2, and vice versa. Since in the common region, the player's status is maintained by all neighboring servers simultaneously, graceful handoff can be achieved even under the wandering case. A more complicated idea, known as the *hierarchically structured virtual object* [10], is implemented in the latest virtual environment model. Objects in the virtual world are organized into different layers based on the space location relationship. An object is revealed only when viewers reach the layer of the object. Apparently, different levels can be handled by different servers in the clustered-server architecture.

The advantages of the clustered-server architecture are summarized below.

- The cluster-server architecture is more scalable. The bottleneck arising from the large computational power requirement due to interaction among a large number of players and objects in the game world is eliminated by distributing the computational load to multiple servers.
- More efficient synchronization can be achieved. Generally speaking, the server cluster only maintains one copy of the game status jointly. Additional optimization techniques, such as the *common region*, could be used in order to furthermore boost the performance.

A powerful server that can resolve the synchronization issue quickly is essential to MOG. Future MOG would keep testing the limit on scalability with its extensive interaction nature. If a single server cannot solve the problem, the multi-server platform, such as the clustered-server architecture, provides a viable and cost effective solution.

2.4 Case Study

Besides the adoption of personal computers as MOG terminals, it becomes common to use traditional console systems to access online games. Each traditional console system is a stand-alone box capable of running games designed specifically for that game system. Recent developments have enabled such console systems to access the Internet so that it is possible to play video games online with other players. This section will give a brief overview of three popular console systems, *i.e.* Xbox Live, PS2, GameCube, and explain the latest development and deployment of online gaming components for each system.

Xbox Live is an online system for Xbox and is managed entirely by Microsoft. Unlike Sony or Nintendo, Xbox can handle all online support for its games. The system allows gamers to find their friends easily, talk to other players during the game play, download player statistics and new features for games, and play the game online with others. Xbox Live servers are located in four different cities to serve people in each region. To play MOG, users first subscribe to the service and then log on a central server with the game they wish to play loaded. From there, users either choose to host a game or join someone's game. After the initial setup phase, players are connected directly to the hosting player in a client-server fashion and the burden of running the game is shifted from the Live server to the host Xbox machine. There are already a few MOG out and many more will be released each month. Currently, Xbox Live servers are not dedicated servers, and there is some debate on how the system is going to work out.

PlayStation2 only has a handful of titles with online play capability at this moment. Since game providers are responsible for their own online systems, the software on servers varies per title. The most popular games are EverQuest and SOCOM, a 3D MUD and FPS, respectively. The EverQuest online service is handled by the Sony Online Entertainment (SoE). Multiple servers handle all games and each server is isolated. Thus, there is no communication among servers. SOCOM runs on multiple servers that host multiple games in a client-server fashion, and each server supports roughly a dozen of players. Future developments include the much anticipated Final Fantasy XI which, along with other new titles, will use *grid* computing. A *grid*, which is a collection of computers working together as a single system providing the supercomputer power at much more cost effective prices, could potentially accommodate a million users at once.

GameCube is much like PS2. The only MOG title is Phantasy Star Online, a 3D MUD, which runs in a method similar to SoE for EverQuest. The game runs in a server-client fashion and the server only provides hosting for games and account information storage. Individual character data for players are stored locally on the GameCube so that players may build their characters offline. Further online gaming for GameCube is still under development but promises to be much like Sony's version and depends on the game developers' decision rather than Nintendo's.

3. MOG SERVER STRUCTURE

In this section, we will introduce the implementation of the game server from two aspects. First, we address the game server structure. Then, the pseudo-codes for game server are abstracted for building the basic concept on the game server. Finally, special changes made for the clustered-server architecture will be discussed.

3.1 Game Server Structure

The block diagram of a MOG server is showed in Figure 4, which consists of four blocks with four basic functions. Here, we use the word "structure" to represent the logical platform, and leave the "architecture" for the physical platform.

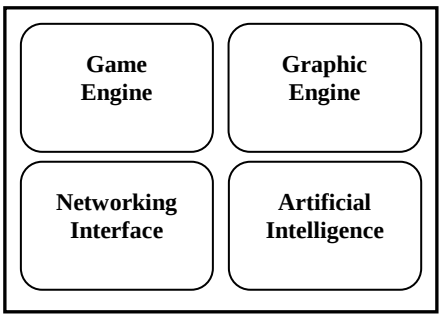


Figure 4 Basic MOG Server Structure

- The **Game Engine** is responsible for maintaining the game world view through iterative synchronization. It applies physical rules to the virtual world by checking and updating the status of every game entity or object constantly. The synchronization mechanism decides the updates and then passes the information to the networking interface. At the same time, it is also responsible for receiving the updates from the networking interface and resolves consistent conflicts. The cheat-proofing mechanism and some part of Interest Management, such as the information filter, also takes place here.
- The **Networking Interface** handles the input/output updates from/to the game engine. The other part of Interest Management, *e.g.* packet coalition, occurs here before they actually go to the network.
- **Artificial Intelligence** controls the behavior of non-player characters (NPC) and their interaction with players, *e.g.* the Multiplayer Dungeon (MUD) MOG. In Real-Time Strategy (RTS) MOG, AI not only controls NPC, but also any units that a player authorizes to gain the autonomous behavior.
- The **Graphic Engine** is responsible for graphic rendering computation. In modern MOG, the 3D world view is quite common. The graphic algorithm plays an important role in the success of certain MOG.

Combined with architecture features, MOG are coordinating through either a centralized or a distributed structure. The structures differ in the location where the game state is maintained and the mechanism in achieving synchronization. Under the client-server architecture, all entity states are maintained by the server, which computes the game state based on inputs from clients and informs clients of the current game state.

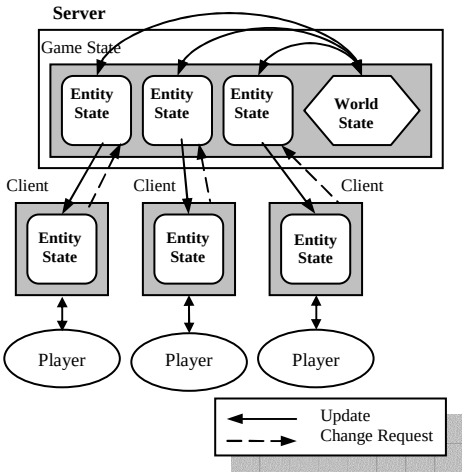


Figure 5 Server Structure for Client-Server MOG system

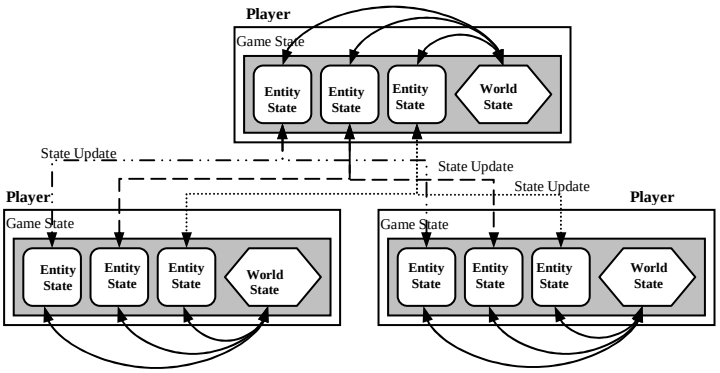


Figure 6 Server Structure for Peer-to-Peer MOG system

In a centralized structure under the client-server architecture as illustrated in Figure 5, a client sends requests for the server to change the client's entity state or a client informs the server of update decisions that have affected the client's entity state, and the server resolves synchronization between game objects or the global game state. Under a distributed

structure, each player maintains its own entity state, informs other players of its decisions, and resolves any consistent problem without the use of any centralized authority (*i.e.* the serverless approach) as shown in Figure 6. Another difference is that each client sees a partial view of the world (and only the server has the whole picture) in a centralized structure while every player maintains its own world in a distributed structure.

3.2 Pseudo codes for game server and clients

The pseudo codes for the client and the server are shown in Figure 7. The client first subscribes to a server. Then, the client sends to the server the command constructed from player's input, and updates the world according to information from the server. If no commands or updates, the client only renders the audio or video effect and do prediction, *e.g.* to use dead reckoning to predict the position. The server runs a loop waiting for clients' commands that carry players' requests. Once the command is received, the server determines how it interacts with the rest of the virtual world. Needless to say, this is a computational intensive task. After that, the server sends necessary updates to related clients.

Under the proposed clustered-server architecture, the pseudo codes need some modifications. The client side remains the same. The only difference is that the client should subscribe itself to a proper server among the sever cluster. This information is processed by a master server initially, which should be able to change the subscription list dynamically when players move to the neighbor region. Each sever only maintains the game status of its responsible region and the neighboring common region. Since interaction only happens in the region locally, no synchronization among servers is needed except when a player changes his/her located region. Once a player enters a common region, both the corresponding servers will have the information, but only the original server has to compute its interaction and then informs the other. Once the player leaves the common region, the original server removes the player from its list, and hands the responsibility to the new region server of the player.

```
connect (server_ip, server_port);
while(connected)
{
   /* if previous time slot ends
      start a new time slot */
   if (current_time_slot)
      continue;
   /* when receive input from
      player, send command to server */
   if (player_input( ))
      send_command( );
   /*when receive update from server
     update game status */
   if (recv_update( ))
      world_update( );
   /*predict for all entities*/
   prediction( );
   /*render graphic, play audio,…etc*/
   render_effect( );
}
```

```
while(!game_end)
{
   /*if any player time_out, end game*/
   if (!time_out)    continue;

   while (recv_command( ))
   /*if players send a command, execute*/
   { execute_command( ); }
   /*AI determines NPC or item reaction*/
   perform_ai( );
   /*if player needs to be synchronized,
      sends update */
   for (n=0; n<player_number; n++)
   {
      if (update(n))
         send_update(n);
   }
}
```

```
##each server performs own session on its region
while(!game_end)
{
   /*if any player time_out, end game*/
   if (!time_out)    continue;

   while (recv_command( ))
   /*if players send a command, execute*/
   { execute_command( ); }
   /*AI determines NPC or item reaction*/
   perform_ai( );
   /*if player is in common region
     inform other servers, if player needs to be
     synchronized, send update */
   for (p=0; p<region_player_number; p++)
   {
      if (update(p))
         if (common_region( ))
            inform_other_server( );
         send_update(p);
   }
}
```

Figure 7 Pseudo codes for the client (left), the server (middle) and the cluster-server (right).

4. KEY ISSUES IN GAME SERVER

Based on the architecture and the structure of the MOG system, there are several key issues to be considered in the design of a good MOG system. They include: interest management, synchronization, cheat-proofing and scalability. Due to the page limitation, we will focus on first two issues in this section. We will give a brief overview and present the *adaptive focus control* method and *priority synchronization* scheme in interest management and synchronization, respectively.

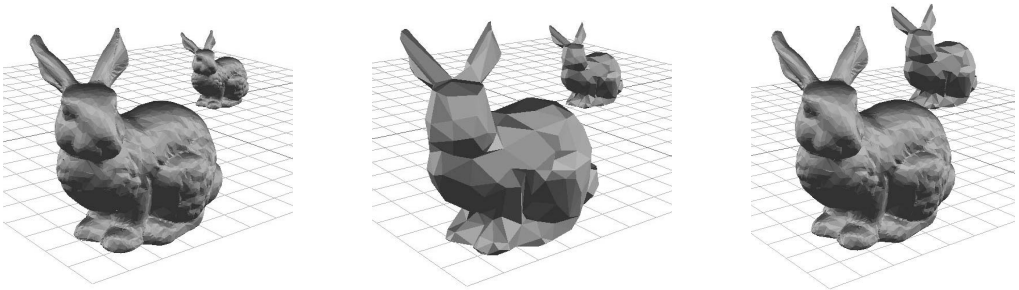
4.1 Interest Management

In a typical MOG, players do not need to know the activities of other users, who are out of their interest. The communication is expected to contain the critical information to save the bandwidth. *Interest management* (*IM*) is designed to reach this goal. It serves as a filter to remove irrelevant messages. It can minimize the network traffic and reduce the burden on the client and the server. Thus, through IM, each participant only maintains the information of a partial world. From the synchronization viewpoint, IM is used to determine which data should be synchronized. From the bandwidth saving viewpoint, IM is to minimize the amount of messages that the protocol should send out to make the game world consistent. Furthermore, IM keeps the only the necessary data on user end, thus eliminating the possibility of *reveal-hidden-information* cheating.

Aura (or *area of interest*) is an expression of interested data, and it usually correlates with the sensing capability (or the granularity) of the system being modeled. It is also strongly related to the world structure of the MOG. Aura-based IM is always symmetric. That is, the associates receive massages from each other when their auras interact. Moreover, aura can be divided into the *focus* and the *nimbus*, representing observer's perception and observed object's perceptivity [5]. This filter is crucial and practical in certain kind of game, e.g. *first person shooting* (*FPS*).

4.1.1 Adaptive Focus Control

To further reduce the load on the server and the client in terms of bandwidth consumption and computation power, a new solution called *adaptive focus control* is proposed in this section. Before describing the scheme, we introduce the concept of progressive representation of 3D objects in video games first.



(a) Fixed mesh, high resolution (b) Fixed mesh, low resolution (c) Adaptive mesh used in adaptive focus control
Figure 8 Examples of object models in the game world

Nowadays, 3D MOG uses the following method to render 3D model. They have certain models (*e.g.* triangle polygons in Quake) and image (texture) information. For rendering, they first combine the model and the image information to create the *avatar* (or object) and then shrink/enlarge it with respect to the distance parameter. However, the model keeps the fixed mesh resolution during the whole game. As a result, a dilemma faced is that a finer mesh resolution demands a higher computational power in processing polygons as shown in Figure 8(a). On the other hand, with a coarse resolution, an ugly appearance with zigzag edges of an object shows up when the object is close to the player as shown in Figure 8(b). Scalability is the basic requirement for future MOG. Thousands of players will compete within a game. Thus, this dilemma will become critical. Furthermore, the client for MOG might not have much computational power with limited resource. This is particularly true for mobile devices such as a PDA or a cellular phone. They may not have enough power/memory to process/store all finest models. Consequently, some model data may be transferred on demand through a wireless channel of limited bandwidth.

To overcome this difficulty, we propose a method called *adaptive focus control*, which is able to create a better world view at the same cost. Let us define the *max focus depth* to be the longest distance a player's focus can reach in the game world. Then, the focus depth of an object is defined as the proportion of the distance between the object and the player over the max focus depth. Furthermore, the *detail level* means that the mesh resolution level for the model of the object. The bigger the detail level is, the higher resolution the model is needed. These two parameters can be written as:

$$focus\ \ depth = \frac{distance}{focus\ \ depth_{\max}}, \quad detail\ \ level = \frac{1}{focus\ \ depth} \tag{1}$$

Every player maintains a table storing the focus depth for all objects. This table is dynamically updated according to the objects' coordinates in the game world. In particular, the player only handles an object in the detail level depending on the focus depth of this object. For example, during the rendering procedure, an object's model with a specified detail level will be requested from the server or loaded from player's local data according to the object's focus depth as shown in Figure 8(c). The shorter the focus depth is, the higher resolution the model will be rendered. Therefore, this scheme saves the resource if the object is far away and can provide a vivid portrait of high-level details if the object is near. Progressive mesh models [12] can be conveniently utilized in this scheme.

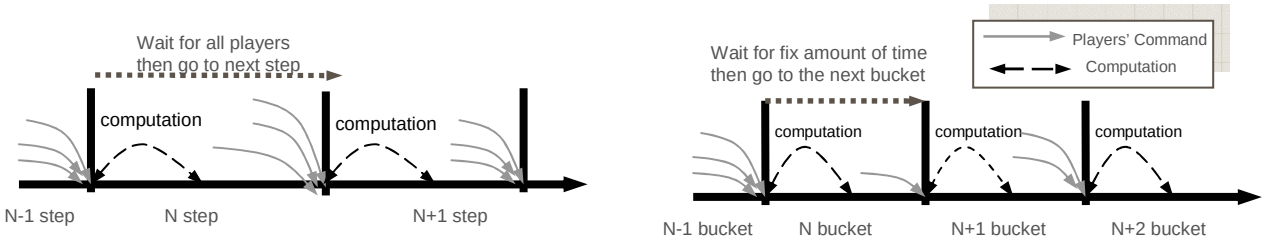
To implement the algorithm efficiently, a caching mechanism is desirable at the player end. All recently used models are stored in the cache. Only the models that are not in the cache will be transferred from the server. The model is granted the progressive feature so that it has several detail levels. We have a choice to store a larger number of basic models of lower resolution or a smaller number of models of higher resolution. Thus, the replacement policy is critical to the management of player's cache. Besides caching strategy, we also need a special data structure to maintain and arrange models associated with several levels of focus depth. In addition, each player can determine the overall detail level by adjusting the max focus depth based on the resource availability. The player with less resource can set the max focus length to a smaller value to remove all the higher detail levels.

4.2 Synchronization

Synchronization is the foundation of MOG. The most fantastic feature of MOG is to bring human competitors into the game and synchronization is used to make sure that everybody is actually playing together on a consistent base. Without a satisfying synchronization algorithm, no consistency can be guaranteed. Furthermore, decisions on synchronization also affect other issues such as cheat-proofing, scalability, etc. We will review the synchronization algorithm first, and then discuss ways to implement an efficient synchronization scheme.

4.2.1 Conservative Algorithms

Conservative algorithms solve the synchronization problem by preventing misordering directly. A typical conservative algorithm is *lockstep synchronization*, in which the server/host will take the next action only after receiving all players' commands [13], which is shown in Figure 9(a). Gautier *et al.* [6, 14] presented the *bucket synchronization* scheme in MiMaze, which is a small 3D multiplayer game over the Internet. As shown in Figure 9(b), with the bucket synchronization scheme, the server/host will wait for a short fixed-length delay before updating the game world to prevent wrong ordering of events. Many FPS online games still use bucket synchronization, which is also called the *frame-based synchronization*. For *real-time strategy games* (*RTS*), the game has to wait for the slowest users due to fairness. Turn-based synchronization is often utilized in RTS. It can be viewed as a variation of lockstep synchronization. The improvement is that the host always detects the network delay of all other hosts and adjusts the turn period accordingly.



(a) Lockstep Synchronization (b) Bucket Synchronization

Figure 9 Conservative Algorithms

Although conservative algorithms are suitable for simple games (and even for RTS games), it is unlikely that it can perform well in fast-paced games since it does not have the ability to detect and recover event misordering.

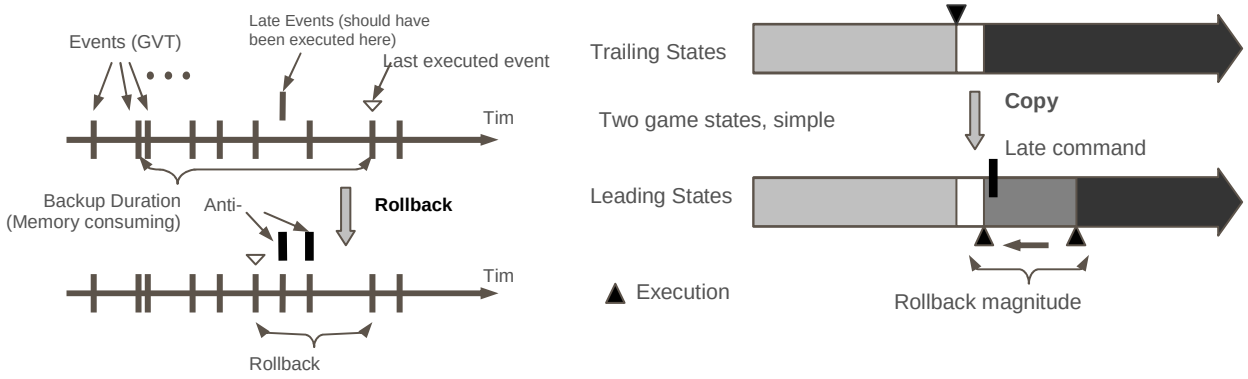
4.2.2 Optimistic Algorithms

Optimistic algorithms have a mechanism to detect and correct inconsistency and keep a constant rate of simulation, which is suitable for more complicated multiplayer interactive games. These algorithms execute events optimistically before they know surely that no earlier events could arrive, and then repair inconsistencies when their predictions were wrong.

Let us explain how out-of-order events impact the game first. In order to maintain real-time property and a smooth output, a server should have about 30 updates per second. However, since users' commands are generally less than this rate, we should compensate the lack of information between updates, especially, the position information. *Dead reckoning* methods [10] are designed to compensate the lacked information, which predict events based on some rules. Sometimes, the late coming events might be different from the predicted information so that the predicted game world state has to be rolled back. The backup method for the game world state, the rollback strategy and the misordering detection approach are all issues to be addressed by the optimistic algorithms. To evaluate the optimistic algorithms, the following three factors should be considered.

- **Overhead:** the penalty paid for *rollback*. It is better to have a lower overhead in saving and restoring states.
- **Memory Usage:** the memory required for storing the game world state for backup. Obviously, a algorithm with smaller memory usage is better.
- **Complexity:** the complexity to fix the inconsistencies. Higher complexity means higher delay.

Time Warp Synchronization (*TWS*) takes a snapshot of the state at each execution, and issues a *rollback* to an earlier state if an event earlier than the last executed event is ever received as shown in Figure 10(a). On a rollback, the state is first restored to that snapshot, and then all events between the snapshot and the execution time are re-executed. Time Warp assumes that events directly generate new events. As part of the rollback, *anti-messages* are sent out to cancel previously generated events that have become invalid, which in turn trigger other rollbacks if these messages have already been executed, which in turn trigger more *anti-message*, and so on. This explosion of *anti-message* may congest the network and tie up servers with anti-message processing instead of executing the game. The algorithms developed later try to resolve the anti-message explosion problem.



(a) Time Warp Synchronization

(b) Trialing State Synchronization

Figure 10 Optimistic Algorithms

Trailing state synchronization (*TSS*) [15], which is shown in Figure 10(b), was developed for Quake under the mirrored game architecture. Similar to TWS, TSS also executes *rollback* when inconsistency is detected. However, it implements rollback intelligently to avoid high memory and processor overheads demanded by TWS. Instead of keeping snapshots of every command, TSS keeps two copies of the same game world, each at a different simulation time separated by synchronization delay. The latest one in the time domain is called the leading state. The other is called the trailing state. When inconsistency is detected in the leading state and rollback is required, instead of copying the state from a snapshot taken as TW does, TSS just simply copies the game status from the trailing state to the leading state, and then performs all commands between the inconsistency point and the present point again. TSS does not actually solve the rollback problem originated from TW. It will have better performance when the following two situations are present. First, the

game state is large and it is expensive to store the snapshot. Second, the gap between states' delay is small. Time warp is the guiding algorithm in the synchronization area today. In order to rollback, we need to have copies of the past checkpoint. It is still a challenge to do it with less memory and overhead. The amount of anti-message is another problem. This problem may be less serious when most commands are independent of each other (*e.g.* in the Quake game).

4.2.3 Insights on Synchronization

Let us provide two insights into the synchronization problem below.

Priority-based Synchronization

To further enhance the performance of current synchronization schemes, we have to discern the nature of events in the game world. Our observation is that different events have different consistency requirements. *Real-time events* have strict timeliness and lax consistency. In contrast, *consistent events* have lax timeliness and strict consistency requirements. The most challenging events are *consistent real-time events*, which have both strict timeliness and consistency requirements. All events that take more than 100ms (time order of human reflex) to react to are classified as *consistent events*. Otherwise, they are considered as either *real-time events* or *consistent real-time events*, depending on their consistency degree. For example, *Quake* commands (events) can be divided into five types as shown in Table 2.

The requirements for different events lead to different priorities on processing and rollback. We call the synchronization scheme that takes event priority into account the *priority-based synchronization*. The priority-based synchronization algorithm may use different approaches to handles different kinds of events, *e.g.* taking more snapshots to minimize the cost of rollback on real-time consistent events, and putting less effort and resource on real-time events, since the rollback cost is low. By treating different events with different methods, we can achieve better overall performance under the same resource and delay constraints.

Table 2 Command classifications for the Quake

Event Type	Event Description	Event Property
Move event	An avatar moves to a new position	Real-time
Fire event	An avatar fires a rocket	Consistent real-time
Impact event	The rocket impacts and detonates	Consistent real-time
Damage/die event	An avatar takes damage and possibly die	Consistent
Spawn event	An avatar is reborn in a random part of the map	Consistent

Synchronization Scheme for Clustered-server Architecture

As mentioned above, the multi-server architecture has more advantages as compared with the peer-to-peer and the client-server architecture. However, few synchronization algorithms today are designed to exploit the merit of the multi-server architecture. Here, we discuss the synchronization scheme applicable to the proposed clustered-server architecture. Recall the common region concept in the clustered-server architecture; the two servers have to synchronize the common region when there are players in the region. The server having the player's physical network connection is called the *ingress node*, the other server as the *egress node* as shown in Figure 11. Several questions arise. How is the synchronization conducted? Where should be the game status calculated? More specifically, should it take place in the ingress or the egress node?

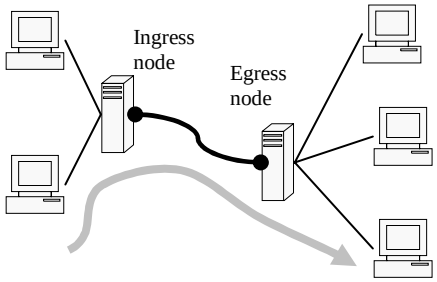


Figure 11 Synchronization for clustered-server architecture

Typically, the ingress node can calculate the new state and send it to the egress node. Another possibility is that the ingress node just forwards the incoming updates to the egress node and does the calculation there. An even more complicated scenario is that the calculation is partly done in the ingress node and completely finished in the egress with an arbitrary partition.

Computing the game status in the ingress node has the fast response advantage and the calculation only occurs once. However, there is no way to detect or avoid inconsistency. Computing the game in the egress node is more natural. In this scenario, every ingress node forwards the command to the egress node and then each ingress and each egress node calculates its own copy of states. Obviously, inconsistency can be detected and avoided by redundant calculation done by each egress node. Unfortunately, the latter method cannot solve the handoff problem since it requires that both the ingress and the egress nodes have the copy of the player in the common region. Inconsistency detecting at the ingress node is just same as the single server case. Consequently, the previous choice might be suitable for the clustered-server architecture. Combining the concept of different event priority, a more complicated synchronization scheme emerges. That is, we can compute real-time events at the ingress node and verify the consistent event at the egress node except for real-time events. The rest usually belongs to consistent events that have no time-restriction.

5. CONCLUSIONS AND FUTURE WORK

We reviewed and analyzed the current MOG system architectures followed by evaluation. Furthermore, we proposed a clustered-server architecture to provide a scalable solution together with the region oriented allocation strategy. Interest management, synchronization, cheat-proofing and scalability are four key issues in the design of MOG. Due to the page limit, we focused on interest management and synchronization only in this paper. To deal with problems arising in interest management and synchronization, we presented some solutions. Their detailed implementation and evaluation will be our future work.

ACKNOWLEDGEMENTS

The research has been funded in part by the Integrated Media Systems Center, a National Science Foundation Engineering Research Center, Cooperative Agreement No. EEC-9529152. Any Opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the National Science Foundation.

REFERENCES

1. Doom, Quake, ID Software, Inc. http://www.idsoftware.com
2. WarCraft, StarCraft, Blizzard Entertainment, http://www.blizzard.com
3. Sony Online Entertainment, http://www.sonyonline.com
4. Xbox live, http://www.xbox.com/live
5. Jouni Smed, Timo Kaukoranta, Harri Hakonen, *A Review on Networking and Multiplayer Computer Games*, Technical Report 454, 2002
6. Christophe Diot, Laurent Gautier, *A Distributed Architecture for Multiplayer Interactive Applications on the Internet*, *IEEE Networks magazine. Vol. 13. No. 4. Pp.6-1, July-August 1999*
7. Nathaniel E. Baughman, Brian Neil Levine, *Cheat-Proof Playout for Centralized and Distributed Online Games*, *Proc. IEEE INFOCOM 2001.* April 2001. Anchorage, Alaska
8. Sung Jin et al., *A global timestamp-based scalable framework for multiplayer online game*, Proceedings of the IEEE Fourth International Symposium on Multimedia Software Engineering (MSE'02), 2002
9. Eric Conin, Burton Filstrup, Anthony R. Kurc, Sugih Jamin, *An Efficient Synchronization Mechanism for Mirrored Game Architecture*, Proc. NetGames2002, Apr 2002
10. Jouni Smed, Timo Kaukoranta, Harri Hakonen, *Aspects of Networking in Multiplayer Computer Games*, The Electronic Library, 20(2):87-97, 2002
11. Chris Greenhalgh, Jim Purbrick, Dave Snowdon, I*nside MASSIVE-3: Flexible Support for Data Consistency and World Structuring*, *Proceedings of the Third ACM Conference on Collaborative Virtual Environments (CVE 2000)*, San Francisco, CA, USA, September 2000, pp. 119-127, ACM Press
12. Jiankun Li, Jin Li, C.C. Kuo, "*Progressive Compression on 3D Graphic Models*", International Conference for Multimedia Computing and Systems, P.135-142, 1997.

13. Eric Cronin, Burton Filstrup, Sugih Jamin, *Cheat-Proofing Dead Reckoned Muliplayer Games*, In Proc. ADCOG 2003, January 2003
14. Laurent Gautier and Christophe Diot, *Design and Evaluation of MiMaze,a Multi-Player Game on the Internet*, IEEE Multimedia Systems Conference. Austin. June 28 - July 1, 1998
15. E. Cronin, B. Filstrup, A. Kurc, *A Distributed Multiplayer Game Server System*, UM EECS589 Course Project Report, May 2001
16. Andrew Rosenbloom, *A Game Experience ~ In Every Application*, Communications of the ACM, July 2003, vol. 47, number 7
17. Alois Ferscha, Johannes Luthi, *Estimating Rollback Overhead for Optimism Control in Time Warp*, Proceedings of the 28th Annual Simulation Symposium (Phoenix, Arizona, April 9-13, 1995), pp 2 - 12, IEEE Computer Society Press, 1995.
18. Ahmed Abdelkhalekm, Angelos Bilas, Andreas Moshovos, *Behavior and Performance of Interactive Multiplayer Game Servers*, *Proc. of The 2001 International IEEE Symposium on Performance Analysis of Systems and Software (ISPASS01)*, November 2001
19. Michael Buro, *ORTS: A Hack-free RTS Game Environment*, Proceedings of the Third International Conference on Computers and Games, Edmonton (Canada) 156-161, 2002
20. Michael Buro, Igor Durdanovic, *An Overview of NECI's Generic Game Server*, Proceedings of the 6th Computer Games Olympiad Workshop, Maastricht (The Netherlands), 32-37, 2001
21. Andrew Kirmse and Chris Kirmse, *Security in online games*, Game Developer, vol. 4, no. 4, pp. 20-8, July 1997
22. Warcrafiii.net, *Cheat Prevention:Steps Blizzard Will Take to Protect the WarCraft World*, 2001, http://www.warcraftiii.net/articles/cheat-prevention.shtml
23. Harvey Smith, *The Future of Game Design: Moving Beyond Deus Ex and Other Dated Paradigms*, ION Storm Austin, 2001, http://www.igda.org/articles/hsmith_future.php