

# 國立交通大學

資訊科學與工程研究所

## 碩士論文

大型多人線上遊戲中介軟體之容錯及負載分享

Fault Tolerance and Load Sharing Mechanism for MMOG

Middleware

研究生：范志歆

指導教授：袁賢銘 教授

中華民國九十五年六月

大型多人線上遊戲中介軟體之容錯及負載分享

Fault Tolerance and Load Sharing Mechanism for MMOG Middleware

研究生：范志歆

Student：Chih-Shin Fan

指導教授：袁賢銘

Advisor：Shyan-Ming Yuan

國立交通大學

資訊科學與工程研究所



Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年六月

# 大型多人線上遊戲中介軟體之容錯及負載分享


學生：范志歆

指導教授：袁賢銘

國立交通大學資訊科學系

## 摘要

多人大型線上遊戲的市場在近年來不斷成長，關於如何設計一個好的多人大型線上遊戲中介軟體的議題也廣泛的在許多研究中被討論。在本篇論文中，我們主要關注在線上遊戲的可取得度上。



HAMS 是我們在本篇論文中開發的一個完全分散式的系統，他是一個用於大型線上多人遊戲中介軟體的高可取得度服務，並且提供了開發錯誤回復及負載全域分享的中介軟體之環境；此外，HAMS 也被有彈性的設計並適用於大部份的大型多人線上遊戲開發中介軟體中。另外在本篇文章中，我們也測量了 HAMS 的效率並且針對其結果做討論。

# Fault Tolerance and Load Sharing Mechanism for MMOG Middleware

Student: Chih-Shin Fan

Advisor: Shyan-Ming Yuan

Department of Computer and Information Science

National Chiao Tung University

## Abstract

The market of MMOG (Massively Multiplayer Online Game) grows up greatly recent years. The design issues of a good MMOG middleware are widely discussed in many researches. In this paper, we mainly concern about the availability of online games.

HAMS is a fully distributed system that we design in this paper. It is a high availability service for MMOG middleware and provides development environment which is failure recoverable and can share server load globally. Also, HAMS is flexibly designed to be used in most of the MMOG middleware architectures. Additionally in this article, we evaluate the performance of HAMS and provide some discussions of the result.

---

# Acknowledgement

---



---

# Table of Contents

---

<b>Acknowledgement .....</b>	<b>v</b>
<b>Table of Contents.....</b>	<b>vi</b>
<b>List of Figures .....</b>	<b>ix</b>
<b>List of Tables .....</b>	<b>xi</b>
<b>Chapter 1 Introduction .....</b>	<b>1</b>
1.1. Preface .....	1
1.2. Design Issues of MMOG Middleware.....	1
1.3. Motivation and Objectives .....	2
1.4. Summary.....	4
<b>Chapter 2 Background.....</b>	<b>5</b>
2.1. Common Network Architectures of MMOG middleware.....	5
2.1.1. Client – Server .....	5
2.1.2. Client – Gateway – Server.....	6
2.2. Important Game Data of MMOG Middleware.....	7
2.3. DOIT.....	8
2.4. AIS (Application Interface Specification).....	9
2.5. Discussion.....	11
<b>Chapter 3 Related work.....</b>	<b>13</b>
3.1. Load Distributing Technology .....	13
3.2. Peer To Peer MMOG .....	14
3.3. Mirror Game Server .....	15
3.4. DOIT Coordinator .....	16
3.5. Summary.....	16

<b>Chapter 4 System Architecture .....</b>	<b>17</b>
4.1. System Overview.....	17
4.2. HAMS Executive Part .....	19
4.2.1. Data Information .....	20
4.2.2. Membership Service .....	21
4.2.3. Game Data Service .....	21
4.2.4. Load Monitor Service.....	22
4.3. HAMS Library Part .....	22
4.4. Summary.....	24
<b>Chapter 5 Design Detail and Issues .....</b>	<b>25</b>
5.1. Chapter Overview.....	25
5.2. Game Data in HAMS .....	26
5.2.1. Game Data Information Management.....	26
5.2.2. Game Data Protection.....	27
5.2.3. Game Data Migration.....	28
5.3. Load Object in HAMS.....	29
5.3.1. Load Object Design.....	29
5.3.2. Load Object Information Management .....	30
5.3.3. Load Object Monitor .....	30
5.4. HAMS Membership Control .....	30
5.4.1. TOTEM Membership Protocol.....	31
5.4.2. HAMS Membership Protocol.....	32
5.5. Recovery Protocol .....	35
5.6. Load Sharing Protocol.....	38
5.7. Summary.....	38
<b>Chapter 6 Experiment and Discussion.....</b>	<b>40</b>

6.1.	Hardware Environment .....	40
6.2.	Game Data Store Throughput.....	40
6.2.1.	Game Data Size vs. Store Throughput .....	40
6.2.2.	Node Numbers vs. Store Throughput .....	43
6.2.3.	Store Rate vs. Store Throughput.....	44
6.3.	Game Data Read Throughput.....	46
6.4.	Failure Detection Time .....	48
6.5.	Summary.....	50
<b>Chapter 7 Conclusion and Future Work.....</b>		<b>51</b>
7.1.	Conclusion.....	51
7.2.	Future Work.....	52
<b>Bibliography .....</b>		<b>53</b>





---

## List of Figures

---

Figure 2-1 client-server network architecture .....	6
Figure 2-2 client-gateway-server network architecture.....	7
Figure 2-3 Game world partition .....	7
Figure 2-4 DOIT architecture .....	9
Figure 3-1 Global Load Sharing Algorithm in [7].....	14
Figure 3-2 Mirror Game Server Architecture .....	15
Figure 4-1 HAMS.....	17
Figure 4-2 High Availability MMOG middleware concept.....	19
Figure 4-3 HAMS executive part .....	20
Figure 4-4 HAMS library part.....	22
Figure 5-1 HAMS Game Data and OPENAIS Checkpoint.....	27
Figure 5-2 Game Data Migration.....	28
Figure 5-3 Token passing on the Totem network .....	31
Figure 5-4 State diagram in TOTEM membership protocol.....	32
Figure 5-5 HAMS membership .....	33
Figure 5-6 HAMS server unique id .....	33
Figure 5-7 HAMS initialization sequence diagram.....	34
Figure 5-8 A scenario of Recovery Protocol .....	37
Figure 6-1 Game Data Size vs. Store Throughput Software Configuration.....	41
Figure 6-2 Game Data Size vs. Store Throughput Result .....	42
Figure 6-3 Node Numbers vs. Store Throughput .....	43
Figure 6-4 Store Rate vs. Store Throughput Software Configuration.....	45
Figure 6-5 Store Rate vs. Store Throughput .....	46

Figure 6-6 Game Data Size vs. Read Throughput.....48



---

# List of Tables

---

Table 6-1 Game Data Size vs. Store Throughput Result .....42

Table 6-2 Store Rate vs. Store Throughput (KB / sec) .....45

Table 6-3 Game Data Size vs. Read Throughput .....47

Table 6-4 Failure Detection Time vs. Token Lost Timeout .....49



---

# Chapter 1 Introduction

---

## 1.1. Preface

A Massively Multiplayer Online Game (MMOG) can be defined as a computer game able to support a multitude of players which interact with each other within the same virtual world, across the Internet, and regardless of their geographical locations. However it is much harder than we can image to develop a stable, scalable and high performance MMOG. According to the report [1], online game market will reach to \$30 billion by 2009. Recently many research focus on the MMOG middleware which provide the toolkits for MMOG developers and short the time to market for game companies. The competition of this industry makes the demand of MMOG middleware raises every year and improves the research of MMOG development platform.



## 1.2. Design Issues of MMOG Middleware

There are many issues when we design a MMOG middleware.

### Scalability

It can be simply defined as how many avatars or game logics could this MMOG supports. As a popular MMOG, there are usually thousands of avatars joining this game concurrently and might be hundreds of game regions and logics operated at the same time.

### Availability

Distributed technologies are usually adopted by modern MMOG middleware. In order to handle plenty of avatars and game content simultaneously, MMOG is always

hosted on lots of different hardware such as servers, gateways and network devices. However these sophisticated devices are not as reliable as we suppose. Each crash will cause serious damages to avatars' benefit and moneyed lost to MMOG companies.

Another point of availability is load distributing. Because of the unpredictable movements of avatars in the virtual world, the load varies with time between different game servers. While the server is overloaded, it will decrease the performance of the system (e.g. network latency, CPU throughput) and even cause game server crash.

### **Interactivity**

Avatars update it own status by sending events or messages to game server. The results will be calculated by server and reply to game players. The interactivity of MMOG middleware defines the response time of these messages exchange. Different kinds of MMOG have their own interactivity constraints.

### **Consistency**

Consistency is a traditional problem of distributed technologies. In MMOG, this usually means the “view” synchronization of avatars. Consistency is easy to maintain in single server, but complicate in server cluster.

## **1.3. Motivation and Objectives**

As we mentioned in the previous section, the availability of a MMOG takes an important role when we develop a MMOG middleware. However, even the most popular MMOGs today do not confront this problem. The most popular solution is duplicating the game world, and managing them separately in different servers. Players create their characters in different servers can not interact each other. In this way, each server can balance the load naturally and will not affect the other servers while one of them crashes. Obviously this is not a mature solution, because it still

suffers hardware failure and overload. Also, separating the avatars affects the game fun.

In this paper, we design a high availability service named HAMS (High Availability MMOG Service) that is fault resistant and is able to share load dynamically in MMOG middleware. Also, we hope this service could be applied to most of the MMOG middleware.

In order to reach our goal, there are many features we should provide.

### **Distributed technology**

Because a fault resistant system must overcome the single point of failure, a distributed control mechanism is necessary for HAMS. There must be no central controller in the MMOG middleware because it is not reliable. Since the controller maybe fail, it will suffer the single point of failure. Also the data should be duplicated to guarantee the availability. After one replication of data being destroyed because of hardware crash, there should be always a backup.

### **High performance**

The performance of fail-recovery mechanism indicates that the repairing time while hardware failure. We can take the mechanism into two phases. First, when failure occurs, it takes time to detect which node is down. The faster we find the fail node the earlier we can start to recover data. Second, after detecting which node failed, it still needs time to recover the game data belongs to the failure node. The objective is to shorten the time of these two phases. The best result we want is that players can hardly feel anything different between the fail-recovery, at least a tolerable affection (Eg: a short time rollback).

The performance of load distributing mechanism indicates the time we cost from the moment we find overload problem to the time the load be shared.

### **Flexibility**

Different kinds of MMOG middleware should be able to implement their own recovery policy or load distribution mechanism.

Load distributing and recovery policy should be adjustable because different game design may have different consideration. For example, sometimes we would like to share our clients to server which is responsible for the clients “near” us in order to reduce the server communications since avatars may interact to avatars belong to other servers. But sometimes we hope to the clients to a “far away” server just because the remote server’s CPU utilization is much more less than the neighbor server and the communication between local server and neighbor is not frequent.

## 1.4. Summary

Because the MMOG industry is getting hotter, a good development environment of MMOG becomes necessary. A MMOG middleware is used to help the developer to design their MMOG products short the time to market.

There are many design issues in MMOG middleware. In this paper, we focus on the availability of MMOG middleware. We hope to provide a general solution for most MMOG middleware which want to have a high availability environment.

The availability issue can be separated into two major problems, fail-recovery and load sharing. In order to provide a good solution, our system should be designed with distributed architecture and must be flexible and high performance.

---

## Chapter 2 Background

---

In this chapter, we are going to introduce some background knowledge you need. It will help you to understand our system design clearly. Section 2.1 is an introduction of some distributed architectures we are using in MMOG middleware today. Also it will give you a basic concept of what mechanism might be needed for fail-over and load distributing. In Section 2.2, we introduce some important game data that appears commonly in MMOG middleware. The next section we are going to talk about a MMOG middleware named DOIT [2] which was developed by Distributed System Lab. of National Chiao Tung University. DOIT is a flexible, scalable, easy-to-use MMOG middleware. Section 2.4 includes a standard specification about availability which is adopted by HAMS. Finally, we will give a discussion.

### 2.1. Common Network Architectures of MMOG middleware

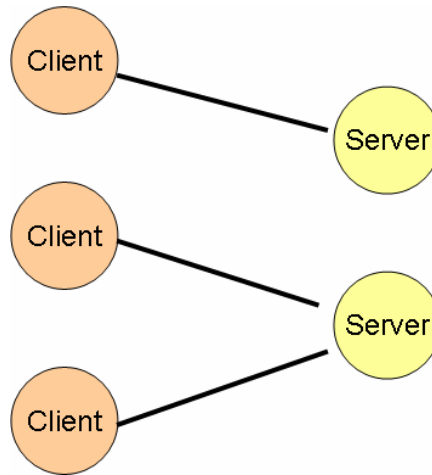
There are many kinds of network architecture of MMOG middleware [3]. Following are some commonly network topologies in MMOG. Through this overview you will see the most common MMOG middleware architectures. One of the design goals of HASM is able to be applied in all these architectures.

#### 2.1.1. Client – Server

This is the most popular network topology adopted by MMOG middleware today. Clients connect to the game server and start to play. The number of game servers depends on the scalability of the MMOG. Each game server is responsible for different game logics, and usually the servers are separated by the geographic of the game world. Avatars move to different part of game world also migrate their connections to another server at the same time. Client-server architecture is used in



many games (Quake, EverQuest) because it is cheating avoidance and easy to maintain. However it may have the single point of failure and server overloaded if there is no other mechanism with it.



*Figure 2-1 client-server network architecture*

### **2.1.2. Client – Gateway – Server**

Client-gateway-server architecture is actually an improvement of client-server. In client-server, players connect to the game server directly, because each server's I/O connections is limited, there can not be too many avatars play in the same area of game world at the same time even though the server still underload. It is not a reasonable constraint because avatars people are gregariousness. In client-gateway-server architecture, clients connect to gateway instead of connecting to game server directly. It breaks the I/O connection limitation of each game server and improves the scalability of client-server architecture. However, introducing gateways does not solve single point of failure neither overloaded problem.

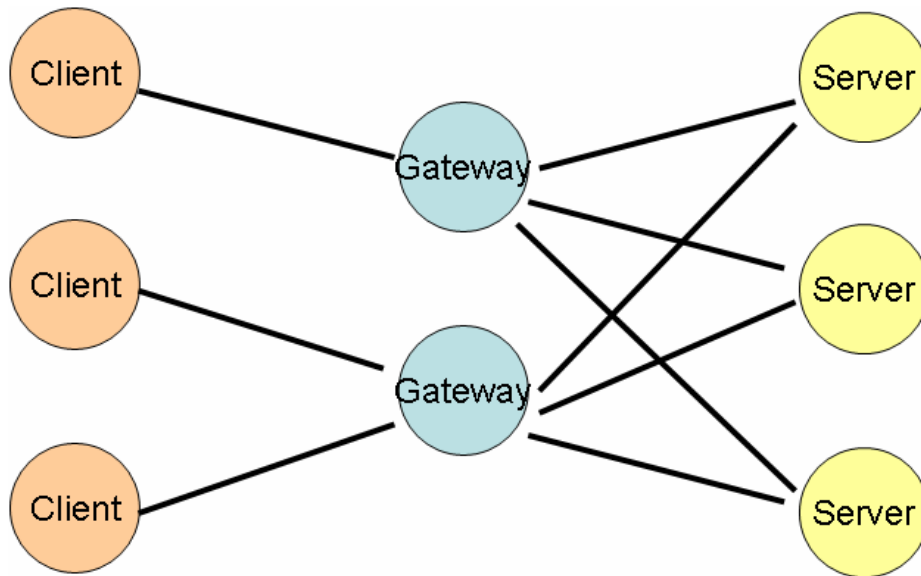


Figure 2-2 client-gateway-server network architecture

## 2.2. Important Game Data of MMOG Middleware

In section 2.1, we know that modern MMOG are always hosted in a server cluster. There are some important data includes in a game server. When we talk about availability, the most important part is game data protecting. The following will help you to understand the various game data types in MMOG.

The words region, cell or unit and are addressed in many papers related to MMOG or DVE (Distribute Virtual World). All of them represent pieces of game world. There are many approaches to divide the world such as in quadrangle or polygon. Even, there are many researches proposed dynamic partition algorithm in order to reach the most load balancing system.



Figure 2-3 Game world partition

Game logic is the largest part in a game. It indicates what will happen after each

action is created by an avatar. For example, an avatar may lose its health after he/she is hit by a NPC (non-player character) or learned new skill after the avatar use a skill book. These complicated game logics are maintained by many game servers and might be distributed into different servers just like game regions.

Avatars' status information is very important data in the game servers. It represents the current state of an avatar such as healthy, skill, level and etc. Avatar's status is updated after the result is calculated by server according to the game logic and avatar's behavior.

## **2.3. DOIT**

DOIT is a flexible, scalable, easy-to-use MMOG middleware which was developed by Distributed System Lab. of National Chiao Tung University. DOIT is based on client-gateway-server architecture and it divides the game world into "Regions". The architecture overview of DOIT is in Figure 2-4. The architecture and the design issues of DOIT did a great inspiration of this paper.

### **Region**

In DOIT, the game world is divides into many regions. The Avatar connects to one gateway and be dispatched to the server which handles the region that this avatar logged out last time. As the avatar move to region that does not belongs to current server, it will be migrated to another game server and gateway will start to dispatch avatar's message to new server.

### **Coordinator**

Coordinator in DOIT is a command line interface that provides game maintainers some utilities such as region migration. Through coordinator, we can move regions form one server to another.

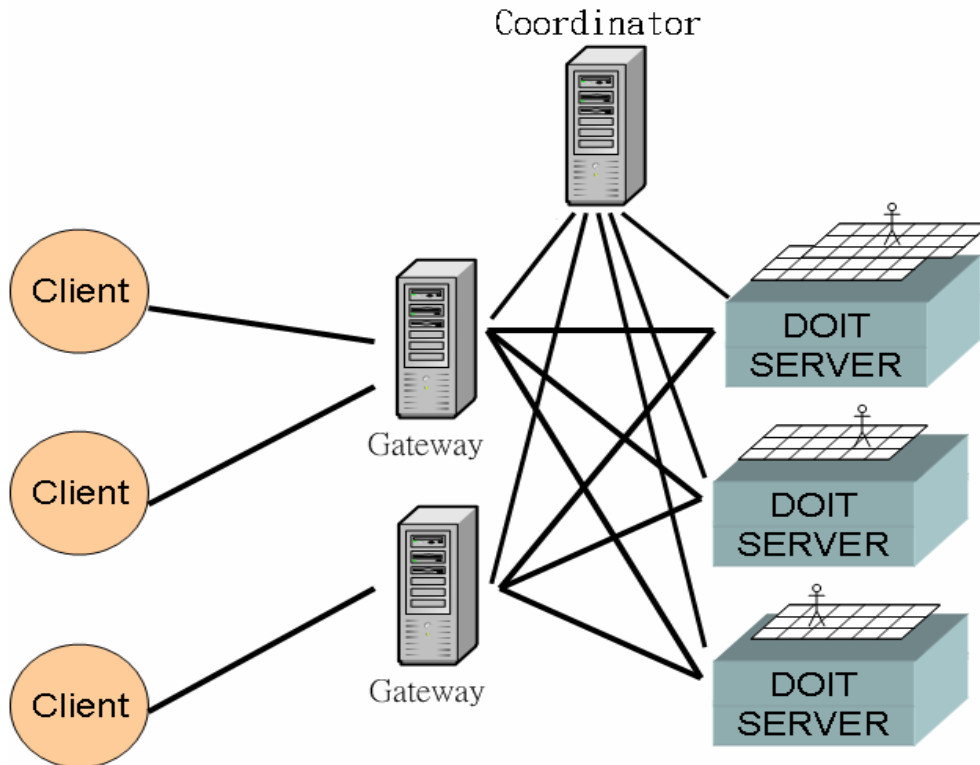
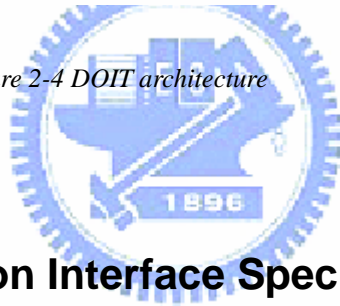


Figure 2-4 DOIT architecture



## 2.4. AIS (Application Interface Specification)

The Application Interface Specification (AIS) [4][5] is a standard for high availability middleware which is developed by Service Availability Forum (SA Forum). It provides many high availability standard APIs which will be widely used in our system.

This is originally defined for carrier-grade applications which are developed by vendors themselves, and not portable across different vendors' implementations. The term high availability (HA) represents the services availability should be up to a range near 99.999%, or simply imply the application down time should not be larger than 6 minutes in one year. There are some other requirements for carrier-grade applications in [6].

There are six services defined in AIS: Availability Management Framework, Cluster Membership Service, Checkpoint Service, Event Service, Message Service and Lock Service. We introduce three kinds of services that are useful in our solution.

### **Cluster Membership Service**

The Cluster Membership Service provides membership information about the nodes in a cluster to the applications. A cluster is simply a set of nodes with unique identifier that are connected together by network. As nodes join and leave the cluster, cluster membership changes.

The Cluster Membership Service provides functions that allow applications to retrieve the membership information and monitor the changes in membership.

### **Checkpoint Service**

The Checkpoint Service provides a facility for an application to preserve its state across hardware and software faults. If a node fails or restart, it can recover its state quickly by retrieving the checkpoint data.

Checkpoints are cluster-wide entities that are designated by unique names. A copy of the data stored in a checkpoint is called a checkpoint replica, which is typically stored in main memory rather than on disk for performance reasons. A given checkpoint may have several checkpoint replicas stored on different nodes in the cluster to protect it against node faults.

### **Event Service**

The Event Service is a publish/subscribe multipoint-to-multipoint communication mechanism that is based on the concept of event channels, where a publisher communicates asynchronously with one or more subscribers over an event channel.

Multiple publishers and multiple subscribers can communicate over the same event channel, and publishers can also be subscribers on the same event channel.

Individual publishers and individual subscribers can communicate over multiple event channels.

The OPENAIS is an open source project to implement Application Interface Specification. OPENAIS began development by MontaVista Software in January 2002. It covers all services defined in AIS and another extends services named Evs.

OPENAIS uses TOTEM multicast network [7] for its communication protocol. TOTEM first been proposed in [8] and be improved in many researches [9][10]. Also Efstration Thomopoulos measured the message latency of TOTEM in [11]. The recovery performance has been proven in our previous work [12], and the message latency is short even in large scale. TOTEM provides total ordering and reliable multicast and membership protocol which maintain all cluster nodes' join and leave.

## 2.5. Discussion



In this chapter, we describe the most common architectures of MMOG today and the basic concept of what a MMOG looks like. We know that the MMOG middleware are usually operated in many game servers which are responsible for pieces of the game world. Avatars (clients) connect to the servers directly or through gateways and start to interact with other characters in the game world. There are many important data in the game server such as game regions, game logics and avatars' status. Protecting these game data helps us to relieve the damage from server failure. Game data migration could decrease the load in servers.

The Application Interface Specification defines many useful services to manage the application data and hardware resources. Its high efficient membership service and checkpoint service is useful in distributed system architecture. The open source implementation OPENAIS is a high performance and fully distributed middleware. The membership service in OPENAIS has been proofed by our previous work. Many

developers are now trying to deploy their application on OPENAIS, so the OPENAIS develop group still keep working on this open source. The latest distribution 0.74 has become very stable and we think it should be useful in HAMS.



---

## Chapter 3 Related work

---

There are many researches and articles related to the issues of availability of MMOG. In this chapter, we are going to introduce you some representative approaches. These approaches can briefly be divided into two kinds. Section 3.1 is some algorithm base solutions of availability. They proposed algorithms to solve the load distributing problem. The other kind of approach is architecture based. We introduce you three kinds of architectures which have the ability to solve failure or overloaded problems. They mainly get these advantages from the architecture they design but will not propose any algorithm in the availability issues.

### 3.1. Load Distributing Technology

In [13], they proposed a global load sharing algorithm called DLS which have the capability to distribute the load to all servers through recursively dispatching their load to neighbor server. These solutions may invoke many of network transfer in one time and might cause network burst. There is another similar algorithm proposed by Kyngmin Lee and Dongman Lee in [14].

Figure 3-1 is an example of the algorithm. While a server S00 found that it is overloaded, it first sent to its neighbor server S10, S11, S12 and S13. Each neighbor will recursively execute the sharing algorithm until the S00 is underload or there is no other server to share. This time complexity of this algorithm depends on how many servers are there in the cluster.

Another kind of load distributing strategy is local load distributing. In [15], Dugki Min provided a local load sharing algorithm which just distributes the load to neighbor server. It reduces the network traffic through the execution but can not distribute the load to all servers once.



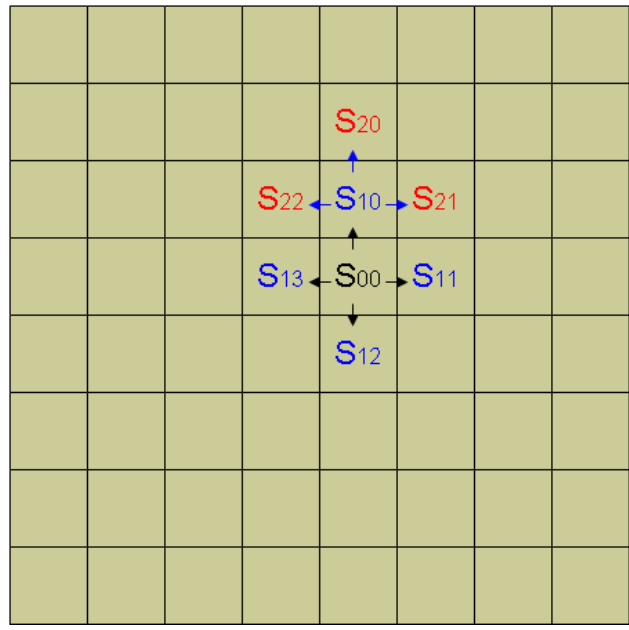


Figure 3-1 Global Load Sharing Algorithm in [13]

### 3.2. Peer To Peer MMOG



Peer-to-peer is a newly technology comparing with client-server architecture. The main idea of P2P is let the clients maintain their own game logic calculation and state copy. Often, P2P games are based on fully connected overlay network which has no central server and. Generally, in P2P, the players' status change should be sent to other peers so they can update there "scene" synchronously. The most well known advantage of P2P architecture is robustness. There is no single point of failure and overloaded in P2P games because clients are responsible for their own calculation and even one of them crashes, it does not affect others. Moreover, we can easily recover the failed data from other peers when the failed client reboots again. However, since interactions and accounting information are not verified by any server, cheating is possible. And also, each client's status change should be sent to all other clients, it cause heavy network overhead and complicated consistency issue in P2P. There are

few P2P MMOG game, the representative of them is MiMaze [16] [17]. Also, some researches about p2p game cheating problem [18].

### 3.3. Mirror Game Server

In [19] is architecture of MMOG middleware called Mirror Game Server. Mirror Game Server is actually an improvement of client-server architecture. In client-server architecture, each game server is responsible for one part of game world. But in mirror game server, there is a group of game servers that serve clients connected to the same part of game world. Duplicating each server to a group has some advantages, such as load balancing and fault resisting. However, since clients distributed in the server group are related to the same game world, each status update should be sent to all servers in the group. This cause complicated consistency issue between the servers ([20], [21]) and also increase the network traffic load. Moreover, because the availability relies on the number of the “mirrors” (game servers in the same group), we may add too many servers to one group that is not necessary and waste the hardware resource.

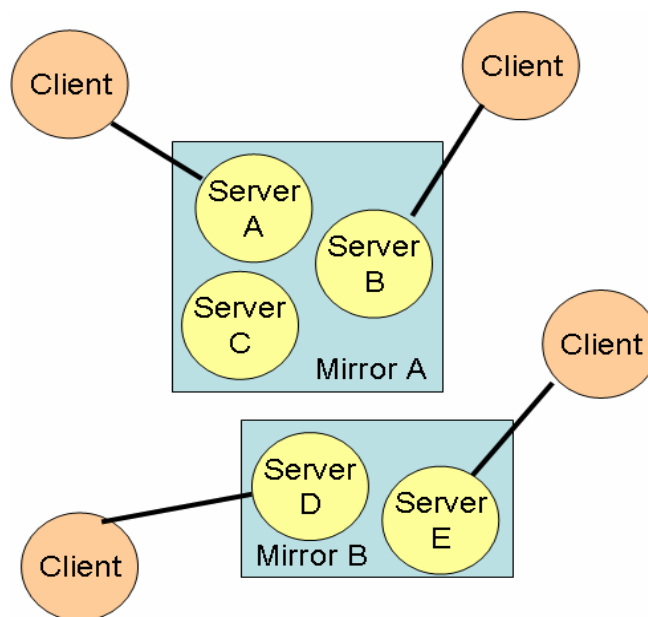


Figure 3-2 Mirror Game Server Architecture

Figure 3-2 shows the architecture of mirror game server. The servers belong to the same mirror ( $\{A,B,C\}$   $\{D,E\}$ ) are replications of each other. Clients connected the nearest game server and begin to play the MMOG. The game data information is totally synchronized in each server belongs in the same group. While server A crashed, both server B and C can take over the clients which were connecting with A.

### 3.4. DOIT Coordinator

Another approach to solve the overloaded problem was proposed in [2] DOIT by adding a central controller named Coordinator as Figure 2-4. Because coordinator is a central control technology, while the coordinator fails, there is no way to balance to load. Moreover, it only provides region migrate, game developers can not move other game object by coordinator.

### 3.5. Summary



Above are some researches in availability issues of MMOG middleware. All of them had big contributions. Unfortunately, many of them have their weakness. Our solution takes their good concept and fixes the drawback.

In HAMS, we hope that there is no central controller so we can overcome the single point of failure. Second, HAMS is aimed to the widely used server architecture of MMOG middleware which means it must be easily to management and difficult to cheat. Under this consideration, the p2p and mirror server architecture are both not proper for HAMS. Eventually, HAMS hopes to provide more efficient solution in both fail-recovery and load sharing and also a flexible interface for MMOG middleware developers.

---

## Chapter 4 System Architecture

---

HAMS is the abbreviation of High Availability MMOG Service. It provides developers many services to design their own high available MMOG middleware. In this chapter, you will see the system architecture of HAMS. And we will depict the components in HAMS step by step in the following sections.

### 4.1. System Overview

HAMS is a high availability service for MMOG middleware. In Chapter 2, we talked about the most common architecture of modern MMOG. We hope that HAMS can solve the availability problem in all these architecture.

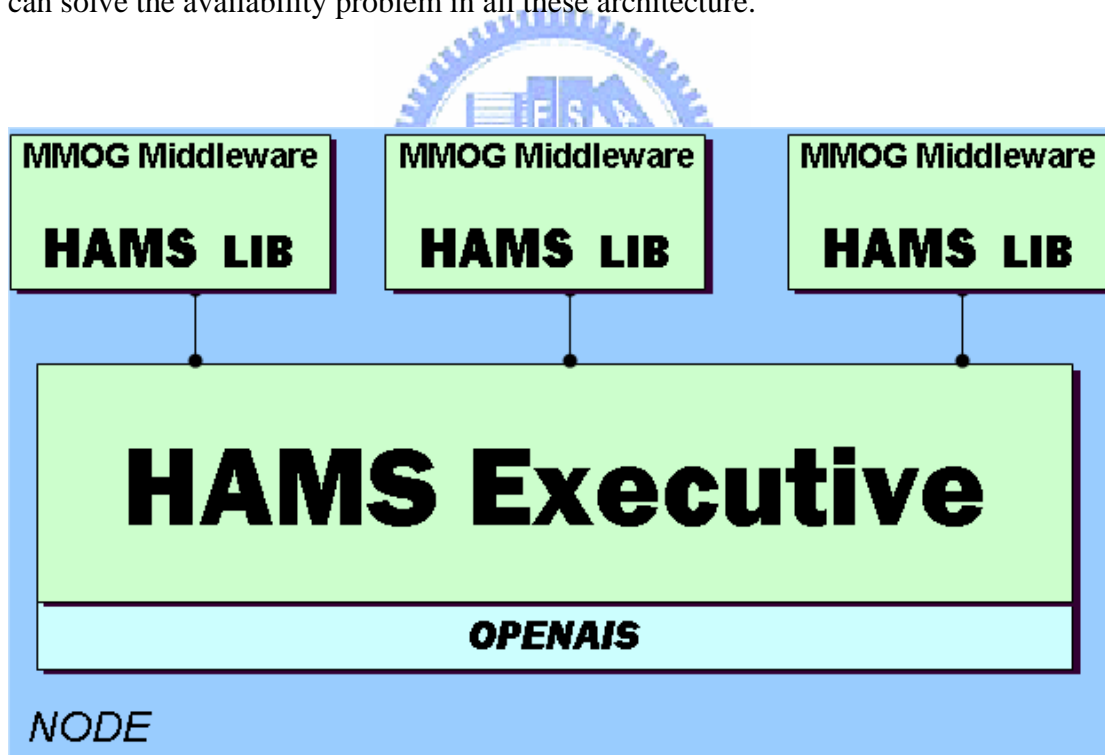


Figure 4-1 HAMS

There are two important parts in HAMS, Library Part and Executive Part, see Figure 4-1. The HAMS Library Part is linked with the MMOG middleware and

provides API for users. HAMS Executive Part starts as a demand in each node and process the request from HAMS Library Parts.

In HAMS, each node will be only one Executive Part, but we allow multiple Library Parts connect to each Executive. Executive Part binds with OPENAIS and uses its services to provide high availability to MMOG middleware. The MMOG middleware links with Library Part and can be thought as a “Server” in HAMS system. The HAMS servers can obtain the information such as node failure, node overloaded and etc from Executive Part. The middleware also use the Library to monitor their load and store game data that must be protected. The details of functions in HAMS will be introduced later. The lowest layer of HAMS is OPENAIS. HAMS Executive Part uses OPENAIS as its base network. We use many services provides by OPENAIS to design our own high availability functions in HAMS.

The main advantage of splitting HAMS into two parts is flexibility. MMOG middleware are sometimes written in different languages. To link with these varies systems; we only need to rewrite the Library Part but not whole HAMS. Below this consideration, we think that keep the HAMS Executive independent from any language will be a better idea than write whole HAMS together. Additionally, we allow many Library Parts to connect to one Executive Part to increase the usage flexibility. Through this functionality, HAMS can support multiple MMOG middleware servers in a single node. This concept was proposed by SUN named utility model which can increase the hardware utilization and improve the management. However, the server membership control will be more complex. We will introduce you how HAMS maintain the membership information of servers in Chapter 5.

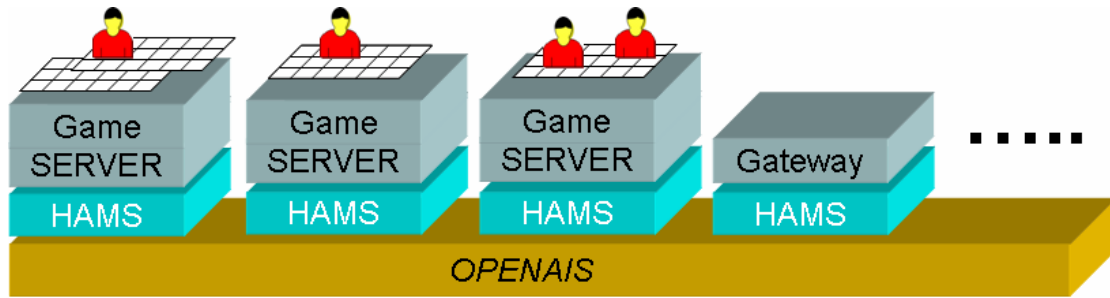


Figure 4-2 High Availability MMOG middleware concept

Figure 4-2 shows you the concept of high availability MMOG middleware with HAMS. It depicts the relationship between traditional MMOG architecture and HAMS. Whenever a component of MMOG (server, gateway ...etc) hopes to use the high availability service of HAMS, it links with HAMS and join the HAMS membership. Usually each component represents a MMOG middleware. As long as servers link HAMS Library Part, it can use the APIs defined in HAMS and develop their own high availability MMOG. HAMS is tightly bound with OPENAIS. Each HAMS Executive Part can be known by other and communicate with each other through OPENAIS services. OPENAIS makes HAMS to manage the cluster easily and provide HAMS an efficient communication network between all nodes.

All control mechanisms were designed in distributed way. There is no central controller in HAMS. Each HAMS plays the same role in the system, so there will be no single point of failure. Whichever node fails, the other node will be notified by HAMS and able to recover the game data through HAMS APIs. The similar scenario occurs while a node overloads.

## 4.2. HAMS Executive Part

The Executive Part is the kernel of HAMS. There are many components in

Executive Part. It receives the request from local servers and dispatches them to the correct handlers and response the request with necessary information. There is a lot of data information should be maintained in Executive Part such as membership, game data and load. The Executive Part must keep modifying the information frequently. The Executive Part is a bridge of MMOG middleware and OPENAIS. HAMS use OPENAIS for its base network and the other services in OPENAIS to achieve high availability. The components in Executive Part are in Figure 4-3.

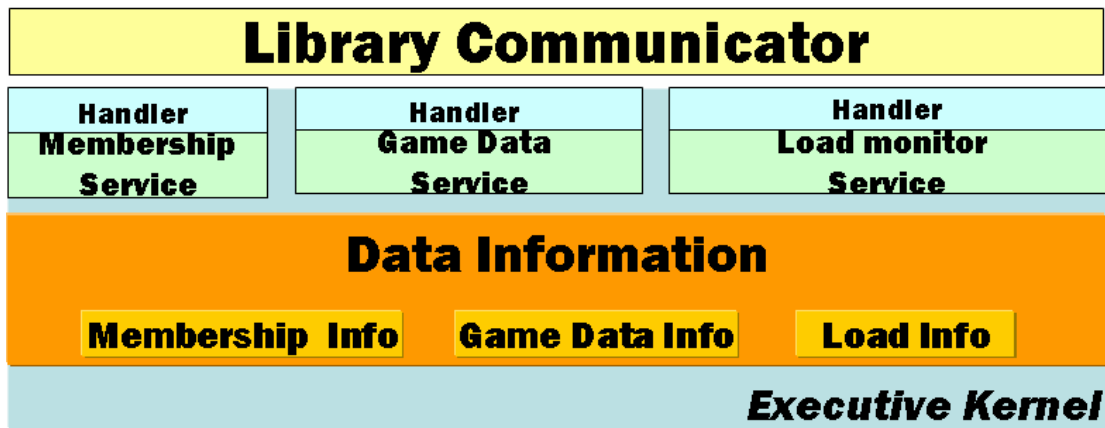


Figure 4-3 HAMS executive part

#### 4.2.1.Data Information

The data information actually is a set of share data that we must keep in the executive kernel. It includes membership information, game data information and load information. Local servers can request the data information through the three services and help users to control the system.

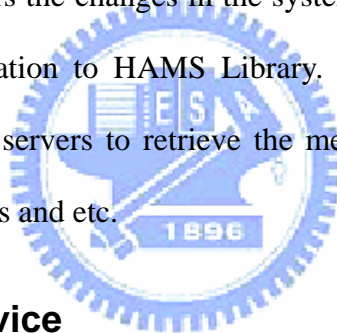
Data information will be updated by Executive Part. For example, when there is a new server connected to HAMS Executive Part, the server information (name, id...) will be saved in all Executive Part. So, whenever a node failed, its information will

not lose and can be recovered by other servers.

### **4.2.2.Membership Service**

To achieve a high availability MMOG, we must overcome the server crash. Hence, we need a mechanism to help us to manage the servers and tell us while there is any server failed. Also, each server needs to retrieve the information of other servers.

Membership service connects local servers in HAMS libraries to the whole membership. It allows local servers to register themselves through membership manager. It also stores servers' information in other node while they register. Membership service monitors the changes in the system, for example server join and leave, and send the notification to HAMS Library. Besides, membership services provides interface for local servers to retrieve the membership information such as server numbers, server names and etc.



### **4.2.3.Game Data Service**

Except the membership, we also need to protect the important game data in HAMS. While there is a server failed, we need a mechanism to recover its regions, game logics, and avatars information. Also, if there is a server overload, we would like to move some of its game data to other servers and decrease its load.

Game data service provides interface for local server to register there data and store them. It also allows local server to retrieve the game data information in other servers. Game data service also maintains the game data information, while the game data is moved from one server to another, the game data service must aware this change and modify the game data information in Executive Part on each node.



#### 4.2.4. Load Monitor Service

Load sharing is also important for availability. To solve the server overloading issue, we have to dynamically monitor the loading in each server and provide the interface for users to retrieve the load information.

Load monitor service is simpler than the other two services. In HAMS, the types of load can be defined by users and the threshold can also be set by developers according their demand. The major work of load monitor service is updating the local server load and than sending to other servers. While it receives the load updating message from other servers, it modifies the load information in the HAMS Executive Part. It also provides interface for local servers to know what the current load in each server and also the load types.

#### 4.3. HAMS Library Part



*Figure 4-4 HAMS library part*

In this section we are going to introduce the components of HAMS library which is used with the MMOG middleware. Because the HAMS library usually link with MMOG middleware server, we can see the library part as a logical server called Local

Server. The detail of HAMS library is in Figure 4-4. All components in Library Part are corresponding to one service in Executive Part. It allows the developers use HAMS easily and hiding some detail implementations in Executive Part. Basically, the Library Part only stores the local information such as server name, id, and the game data it registered. If the server wants to retrieve the information of other servers, it uses the APIs in Library Part and obtains it from Executive Part.

Local Server is the basic object of HAMS Library. Each local server includes three important components, load monitor, membership manager and game data manager. The middleware start using HAMS by initialize local server and its components. Local server will connected to Executive Part by communicator and register itself to join the membership.

Membership manager provides simple APIs for servers to retrieve information of all nodes such as name, descriptions and etc. Developers can implement their own fail-over mechanism through membership manager. While there is a server failed, the Executive Part will notify the membership manager and let developers to run the recovery algorithm they want. Also, it defines some useful structure for users to processing the membership information such as *HAServerInfo*.

Servers can register there game data in game data manager. Game data manager protect these important data by storing them to HAMS Executive Part periodically and allow users to retrieve the game data information in other nodes. Another important job of game data manager is game data migration. Each Library Part has the capability of migrating game data from one server to another by providing some simple APIs. Also it has to notify the users while the local game data have been migrate to the other nodes. *HAGameData* is a structure which is defined in game data manager and useful for game data processing.

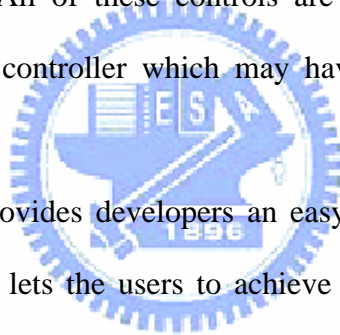
Load monitor provides a flexible way for users to define their load information.

Users can register their own load object and retrieve other servers' load information through the APIs defined in load monitor. It allows developers to define their own sharing algorithm. When the load is over the threshold, users will be notified and may start their load sharing mechanism.

## 4.4. Summary

In this chapter, we describe the HAMS system and how it connects with traditional MMOG middleware. In order to provide the fail-recovery and dynamic load sharing, we need a membership control component and load monitor mechanism in HAMS. Also, the important game data also need to be protected through game data service in Executive Part. All of these controls are done by a totally distributed technology without central controller which may have the risk of single point of failure.

HAMS Library Part provides developers an easy-to-use approach to link their middleware with HAMS. It lets the users to achieve their availability demand in a flexible way.



---

# Chapter 5 Design Detail and Issues

---

## 5.1. Chapter Overview

In this chapter, we are going to discuss the implementation detail and the design issue of HAMS. In section 5.2 and 5.3, we will talk about two basic data structures in HAMS, game data and load object which are widely used in HAMS. We will show you how we maintain load and game data information between servers and also the design consideration of them for flexibility issue. For game data, there must be a protecting approach which is necessary when we need to recover the failed server. Additionally, we will introduce the data migration mechanism which is very important in both recovery and load sharing protocol. In section 5.4 you will see the more detail of membership control in HAMS. In order to manage all servers and detect the server failure, an efficient membership control system is necessary. First, we will explain how OPENAIS provides the membership service by TOTEM, and then show the HAMS membership implementation detail which is based on OPENAIS. After the data structure and basic management system, we are going to show the most important protocol in this paper, recovery and load sharing. The recovery protocol in section 5.5 is actually the composition of the management system we mentioned in previous sections. However, there are some design issues in the user interface which improves the flexibility of HAMS. In order to make it clear, we use a simple case to explain the recovery protocol. Like recovery protocol, the load sharing protocol also invokes all data structures and the control system in HAMS. The load sharing solution we proposed demand less network transmission than we talked about in related work. Also, the game data migration may be faster because we always keep the data replication in all nodes. The last part is a summary to organize this chapter.

## 5.2. Game Data in HAMS

The first data structure you will see is game data. In HAMS, the game data can be anything that developers want through implementing the *HAGameData* object. This section also includes how we protect the game data information and game data migration protocol in HAMS. All of these will be widely used in recovery and load sharing protocols.

### 5.2.1. Game Data Information Management

Each HAMS server can register game data which are important and must be protected. Game data could be a region, a cell, a set of avatars or any kind of information. For efficient issue, we always keep track all data information on every node, because it helps us to reduce the network burst when server failed or server overloaded and users need to retrieve all information from each server. That means, whenever a game data has been registered or removed, all of the servers should be notified. There are some basic changes of *HAGameData* and the corresponding actions we should do in HAMS.

#### **Game Data Register / Deregister**

Users register game data and give it a name for other server to identify what does the data includes. While game data is registered, HAMS notifies all other servers by sending a *register\_gamedata* message. Likewise, when a server deregisters the game data, HAMS sends *unregister\_gamedata* message.

#### **Game Data Migrate In / Out**

The detail of game data migration will be talked about later. Whenever the game data is moved to local, the game data service should register this game data to local, opposite when game data is moved out.

## 5.2.2. Game Data Protection

The game data information that maintained by game data service does not include the data itself. The actual data is usually much larger than game information and stored by OPENAIS checkpoint service in HAMS. In checkpoint service, it divides each checkpoint into several sections. The read/write method actually happens on the sections. In HAMS, every server opens a single checkpoint and the game data will be corresponding to one section. Figure 5-1 shows the relationship of HAMS game data and OPENAIS checkpoint.

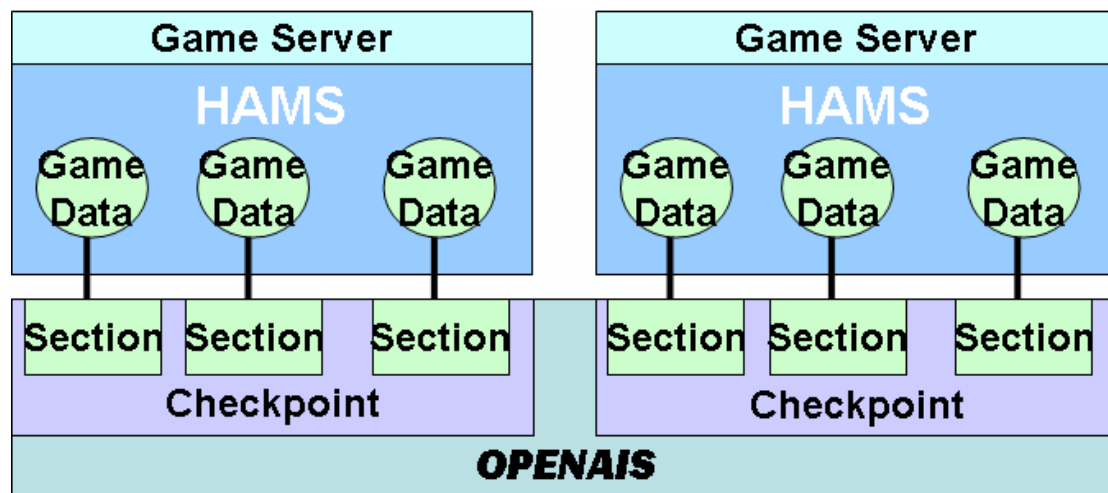


Figure 5-1 HAMS Game Data and OPENAIS Checkpoint

Each server can set its store rate to store their game data. The store rate is a trade-off on the availability degree which is demanded by MMOG middleware developers and the overhead of storing data. While the server stores data into checkpoint, it simply overwrites the corresponding section in the checkpoint. The checkpoint service provides All-or-None transaction which guarantees the store operation will not partial overwrite the section, so the game data correctness can be

promised.

### 5.2.3. Game Data Migration

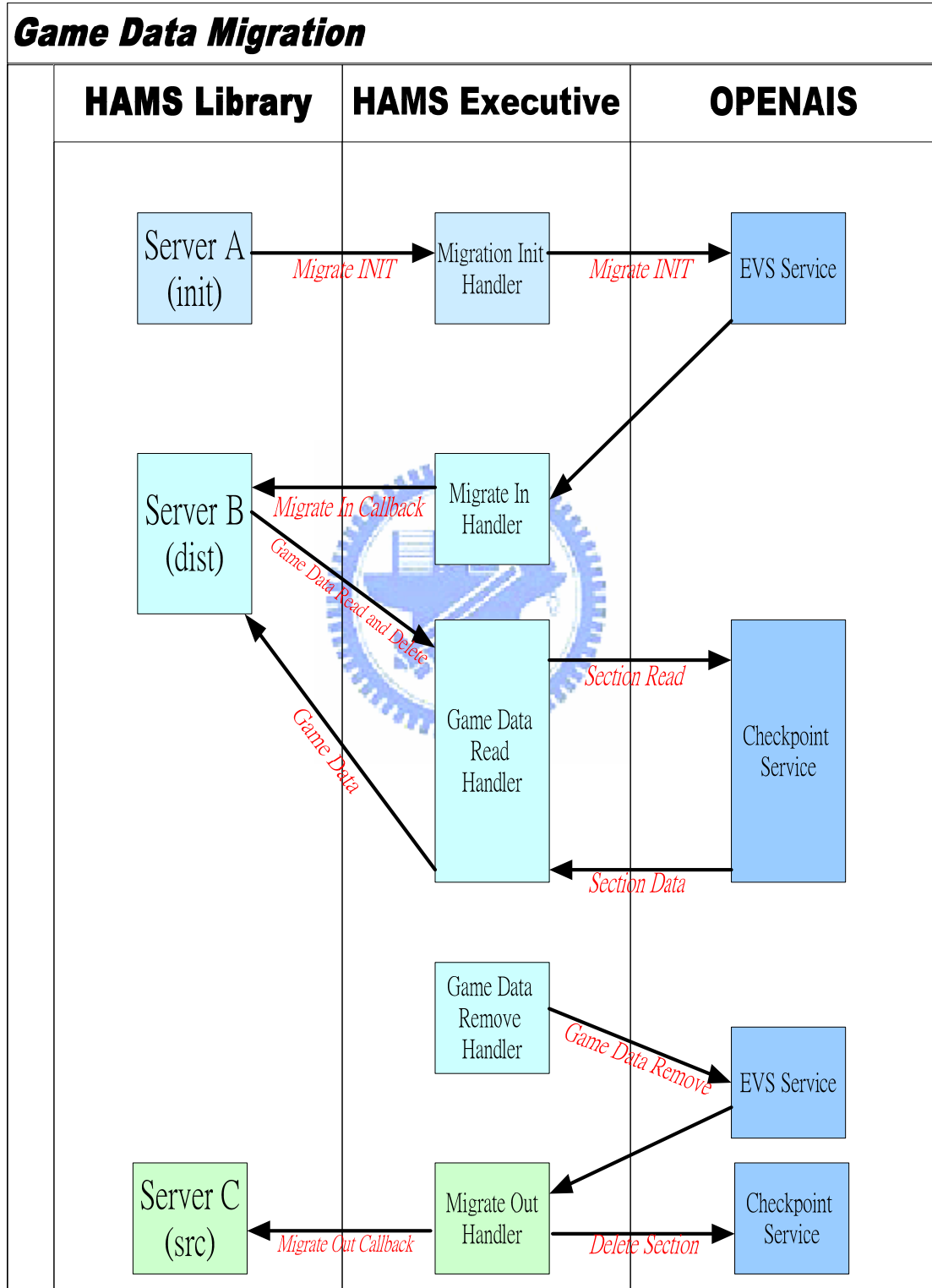
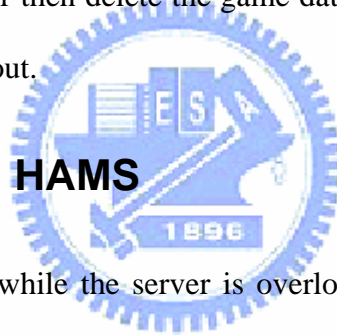


Figure 5-2 Game Data Migration

Game data migration happened in both fail recovery and load sharing. We move avatars from server to server to share their load, or send the important game data from a failed server to an active server for recovering. Basically, game data migration invokes three roles init, destination and source. We can use an interact diagram in Figure 5-2 to explain the details.

The server who wants to start a migration should send a *migrate\_init* message to HAMS executive part. HAMS uses OPENAIS Evs. to multicast the message to other servers. If the server finds that it is the destination of this migration, it must read the game data from OPENAIS checkpoint service. The HAMS Executive Part will automatically multicast a *game\_data\_remove* message after the read operation has been done. The source server then delete the game data and user will be notified that the game data has migrated out.

### 5.3. Load Object in HAMS



In order to share load while the server is overloaded, we need an approach to monitor the system load in MMOG middleware. Load object helps HAMS to monitor the server load. There are some design issues in flexibility. Also, the management mechanism will be introduced in this section.

#### 5.3.1. Load Object Design

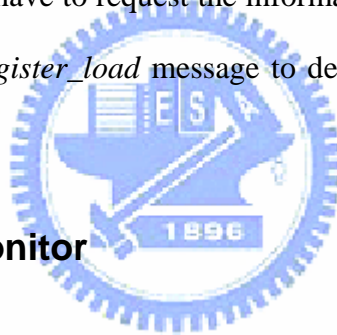
In HAMS, we provide a flexible way for users to define their load object. In traditional distributed system, a load monitor may allow users to monitor their CPU utilizations, network utilization, etc. But in the MMOG middleware, the load monitor should not only control the hardware information but also other game specific info. Sometimes, we would like to monitor the “avatar response time in region 1”, or “avatars migration frequency between regions”. This is impossible to predefine all



load types for every MMOG middleware. To realize a flexibility load monitor, HAMS define an abstract object called *HALoadObject*. Users create their own load object by extending *HALoadObject*.

### 5.3.2. Load Object Information Management

The management of load object is very similar to game data. We also keep the load information in all nodes. While a server create a new load object, it use *register\_load* message to tell all other servers. Also, the current load will be sent to all servers by *save\_load* messages. Although, the load saving messages will occupy some network bandwidth, it can improve the performance of load sharing and fail-recovery algorithm since we will not have to request the information from remote server again. Of course, we can use *unregister\_load* message to destroy this load object and stop monitoring it.



### 5.3.3. Load Object Monitor

HAMS allows users to monitor load object they defined by implementing a method named *calcLoad()* in *HALoadObject*. Developer can set the monitor rate and the threshold of *HALoadObject*. The calculation method updates the load value according the monitor rate and check if it exceeds the threshold. If it is overloaded, monitor will notify the users. The load sharing protocol will be introduced in section5.6.

## 5.4. HAMS Membership Control

In Chapter 1, we mentioned the design objective of distributed technology. Because we want to overcome the single point of failure, there must be distributed membership management without any central controller. The HAMS membership

service connects all HAMS together and maintains the servers' information in every node. Before dive into HAMS membership service, we will introduce you how TOTEM control the membership of physical nodes first. This helps HAMS to be aware of the underlying membership changes such as physical node join and leave. However, the OPENAIS membership is not enough for HAMS because there may be more than one server connect to each node. So we extend the OPENAIS membership to HAMS membership shown in Figure 5-5.

### 5.4.1.TOTEM Membership Protocol

HAMS uses OPENAIS for its base network, because it provides a high reliable membership control protocol depends on TOTEM. The TOTEM multicast was first proposed in 1995 and has been improved several years. It is widely accepted as an effective approach to realize reliable group multicast. The membership protocol of TOTEM helps it to overcome the unstable network configuration. In TOTEM network, all processors form a logical ring as Figure 5-3. Whenever a processor fails or network separate, the old ring breaks and the remaining processors will form a new ring through membership protocol. There are four states in TOTEM membership shown in Figure 5-4. The more detail you should see the reference [7].

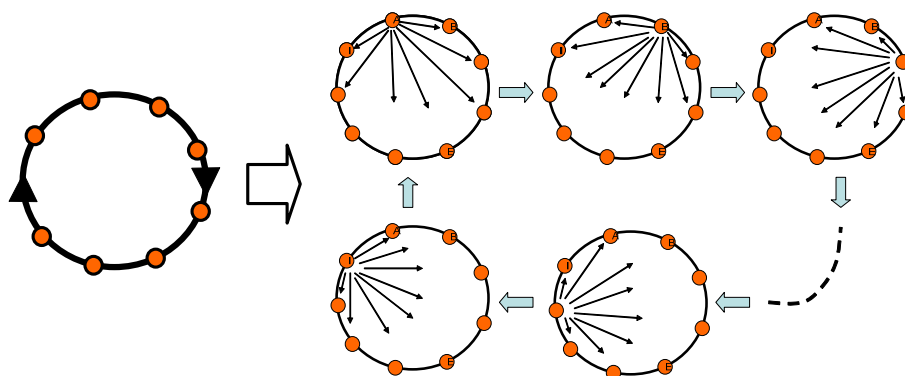


Figure 5-3 Token passing on the Totem network

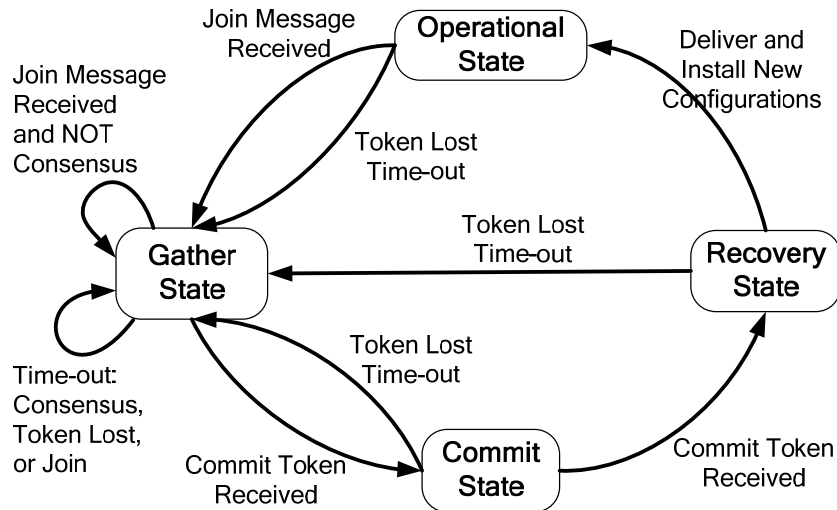


Figure 5-4 State diagram in TOTEM membership protocol

### 5.4.2.HAMS Membership Protocol

We have briefly introduced the TOTEM multicast network and its membership protocol which is used in OPENAIS membership service. Although we can easily manage all hardware nodes by OPENAIS, but this is not enough for HAMS membership. Because HAMS allows multiple local servers operate in one single node. We need an extend membership control system in HAMS executive part shown in Figure 5-5.

#### ID of Server

There are many servers connect to one HAMS Executive as Figure 5-5. The most important information of the HAMS servers is server identification. The identification of a server must be global unique. Although the OPENAIS membership service provides a unique identification for each physical node, we can not simply use this node id, because there may be several servers running concurrently on a node within HAMS. The unique identification generated by HAMS is shown in Figure 5-6. The server register time is the system timestamp that the MMOG server which was linked

with HAMS library connects to HAMS executive part. Because the IP address is a unique identification of all nodes and the server register time was unique within one node, the global unique can be guaranteed.

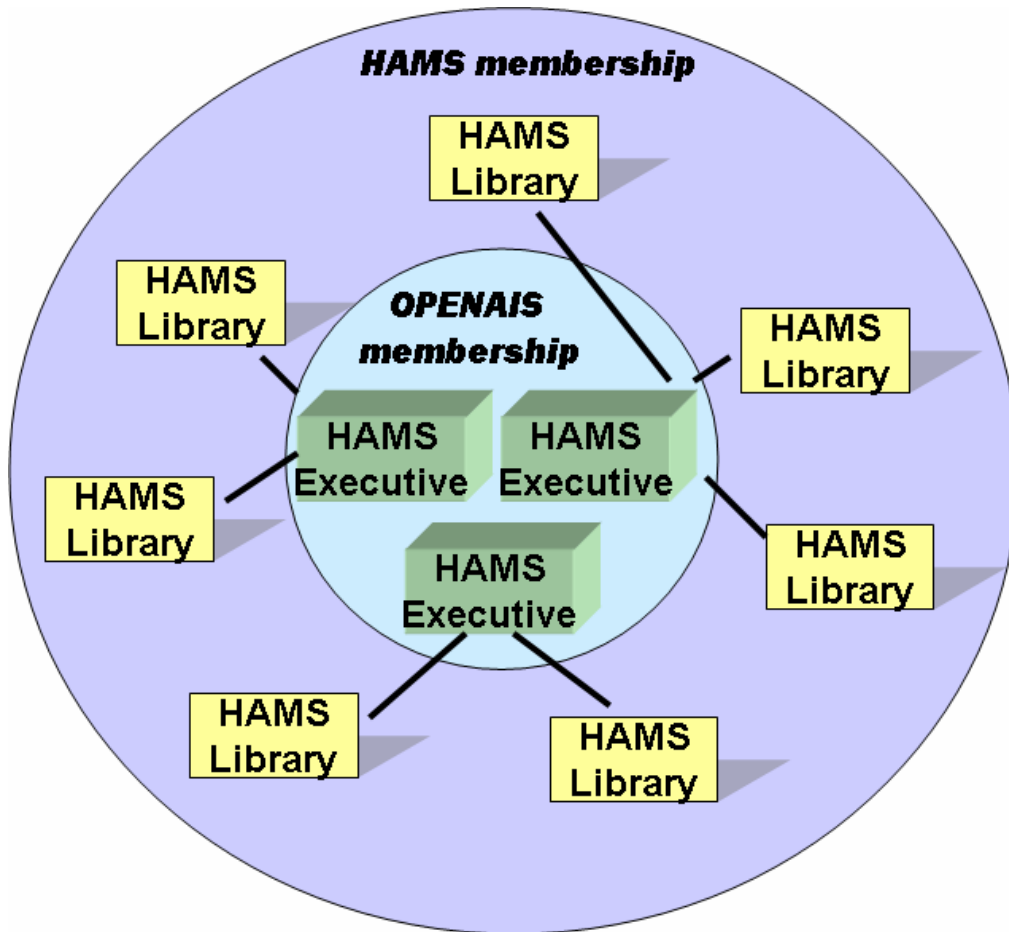


Figure 5-5 HAMS membership

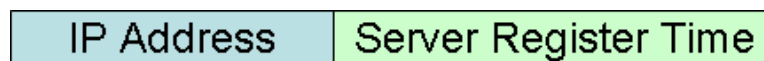


Figure 5-6 HAMS server unique id

### Membership Information Management

At the time a node boot-up and HAMS executive part is started, HAMS joins the OPENAIS membership immediately. At this time, HAMS does not know any server at

all. It starts initialization protocol as in Figure 5-7. First, HAMS sends an *init* request message to OPENAIS multicast service. Second, the OPENAIS will deliver the message to all others. While HAMS receives an *init* messages in (3), it simply sends *register\_server* messages with its local servers' information. Finally, the original initializer builds whole membership according the *register\_server* messages.

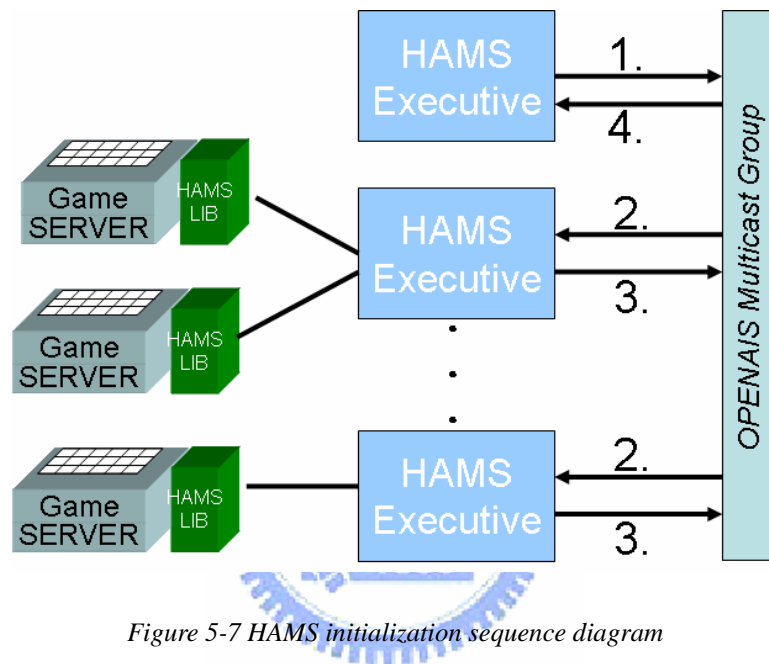


Figure 5-7 HAMS initialization sequence diagram

When there is a server hope to join the HAMS membership. It uses *register\_server* message just like what we did in Figure 5-7 (3) to (4). Likewise, when the server is going to leave the HAMS membership, it sends the message *unregister\_server*, and the other HAMS executive parts will delete the server data in local.

The most important part of membership protocol is failure handle. HAMS receives the node failure notification deliver by OPENAIS membership service and then invoke the failure handler method. The recovery protocol is implemented by user through HAMS library. The detail is in section 5.5.

## 5.5. Recovery Protocol

We have seen the detail implementation of HAMS membership protocol and the control system of game data and load object. In this section, we will introduce you how HAMS recover the server when it failed. We divide this protocol into detection phase and recovery phase.

### Detection Phase

HAMS rely on the membership control in section5.4 to detect the server failure. The detection time is really short. In [12] we do much effort on measuring the OPENAIS membership failover time. In the worst case, the new TOTEM ring can be reformed in one second, and best could be 50~60 millisecond. The result is HAMS can detect the hardware node failure very fast. The membership control invoke the *failure handler* in HAMS after it discover there are some servers failed. The main job of *failure handler* is to choose a representative server and deliver the fail message to its HAMS library. The representative was chosen by the unique identification which is the smallest. Because the id is global unique, there will be only one representative which will execute the recovery algorithm.

### Recovery Phase

HAMS actually just provides the tools for user to recover the failure servers in recovery phase. We leave the algorithm (how to recover) to users by implementing the *HAMemberShipFailListener* themselves. While there is a server failed, membership protocol will deliver the *fail\_msg* to *HAMemberShipFailListener* and start running the recovery algorithm. The reason why we allow the users to implement recovery algorithm is that it helps to design MMOG middleware more flexible. Developers can use Game Data Manager and Load Monitor to retrieve the game data information and load information in all servers. Then the developers can choose their own policy to

decide which data of the fail server would be migrated to whom. Following are some examples that you can do through HAMS while a server failed.

*You can always choose the servers named “repairing server” to take over the failed server’s data.*

*You can choose the servers which have game data named “region x”, because it is closed to the failed region.*

*You can choose the servers which have a load object named “CPU utilization” and the value is lower than 50%.*

These are very common decisions that a MMOG developer may think. Because there are so many you can do in the recovery algorithm, we must leave the algorithm for users. In HAMS, we can do all these easily.

### **Simple Scenario**

Figure 5-8 is a simple scenario which can help you to understand the recovery protocol more clearly. In (a.), there is a node crash, and two servers operated on it will also be failed. Then, OPENAIS membership service finds the configuration changes and notifies the HAMS (b.). At the same time, HAMS choose a representative to handle this failure and deliver the *fail\_msg* to representative. In (c.), the representative runs the recovery algorithm which was implemented by developer, and decides how to recover the game data in the failed servers by sending *migrate\_init* messages. After the game data migrated, the failed server will be deleted and their game data (in this case are two regions) will be took over by other servers (d.).

In Figure 5-8, you should understand that the failed servers’ information actually has been stored in other nodes by HAMS membership manager. So after the node crashes, the servers will be marked as “Failed” in all HAMS Executive Parts. The reason why we do this is to confirm the failure will surely be handled. If the representative fails during the recovery algorithm, the server will still be marked as

“Failed” in each node and we may choose another representative after a while.

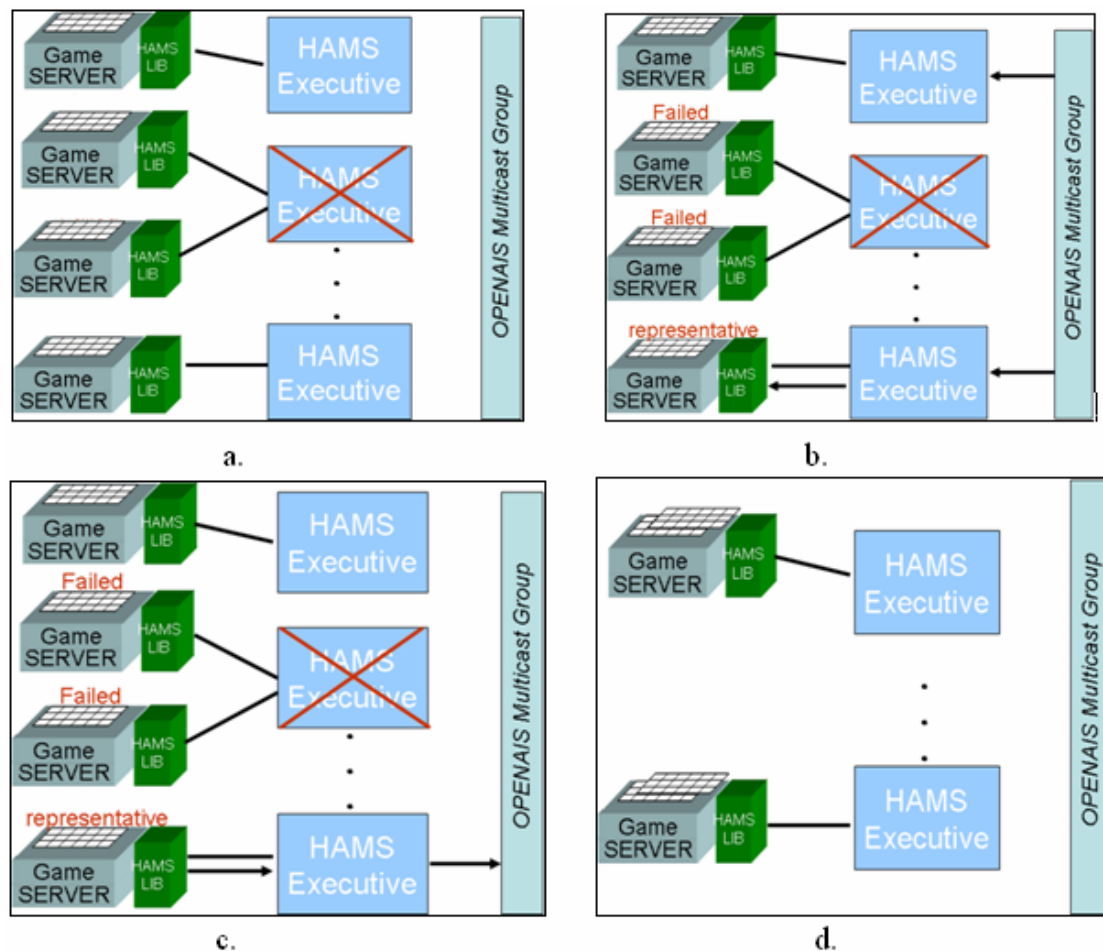


Figure 5-8 A scenario of Recovery Protocol

## Efficiency

The HAMS recovery protocol is very efficient. First, the node failure detection time is short because there is a good membership service in OPENAIS and HAMS only needs search which server belongs to the failed node and choose the smallest unique id for representative. Developers decide how to migrate the game data by retrieving information HAMS provides from local node. All game data migration action can be sent by one *migrate\_init* through multicast. The destination servers can recovery the game data parallely. Moreover, in HAMS we let the checkpoint replication of OPENAIS be distributed in all physical nodes, so destination servers



actually read the game data from local and this makes a big improvement in recovery speed.

## 5.6. Load Sharing Protocol

We have seen how HAMS management the load objects in section 5.3. In this section, we will explain the detail of how HAMS sharing the servers' load.

The concept of load sharing mechanism in HAMS is similar than recovery HAMS also leaves the load sharing algorithm for user to implement. Users can implement the *HALmOverLoadListener* which will be notified by HAMS load monitor while local serve is overloaded. Likewise, users can retrieve the information of game data and load in all servers and make their decisions about how to share the load. Because the information are retrieved in local and *migration\_init* message use multicast, the global sharing algorithm will be faster then recursively sharing from neighbor nodes to remote. Because load sharing also use game data migration to distribute to load, it is as efficient as we talked in recovery protocol.

## 5.7. Summary

### Main Contributions

In this chapter, we explain what we did in HAMS and how it accomplished the research objectives. For distributed technology, the membership protocol of HAMS give you a no central control mechanism to management the game server cluster, and duplicate the game data into several replicas for protecting issue. We make many design consideration to improve the performance in recovery protocol and load sharing protocol. The multicast technology helps us to implement an efficient communication mechanism between servers. Finally, we did much effort to provide developers a flexibility interface in HAMS. Users can design their own game server

management mechanism through implementing the algorithms of recovery and load sharing, storing game data they hope to protect and monitoring load objects which are important for their middleware.

### **Extra advantages**

Except the main objectives, HAMS also brings another advantages for developers. Because users can register there own game object and load in each server and retrieve these data in any other servers. HAMS can also be used as a middleware monitoring system which allows users to watch current system status such as resource utilization, avatars distribution, load distribution and etc.



---

## Chapter 6 Experiment and Discussion

---

We are going to evaluate HAMS in this chapter. First we will measure the game data store throughput. The store throughput affects the avatars status rollback in recovery protocol. While a server failed, we need to recover its game data from other game data replication stored in different node. The faster we can store the game data, the shorter rollback avatars will be, because the replication will contain the “latest” status of avatars. Second, we will evaluate the game data read throughput. The read throughput affects the efficiency of recovery and load sharing protocol in HAMS. Finally, we will give some testing of failure detection time in HAMS. It also plays an important role when we are going to recover the servers after they fail.

### 6.1. Hardware Environment

There are most five computers in our test environment. Each computer uses 3.4G CPU and supports hyper-threaded. The memory is 512MB and use 1000M network interface. All of these computers connected with a GIGABIT switch (SMC tiger) in a private LAN.

### 6.2. Game Data Store Throughput

First we want to know the game data store throughput in different environments. This Section, we divide our testing into three rounds. Each round, we evaluate a single factor affects the store throughput and discuss the result.

#### 6.2.1. Game Data Size vs. Store Throughput

##### Software Configuration

There will be five HAMS Executive Parts; each of them is deployed in one computer. One single server connects to the HAMS Executive and start to store the game data with different game data size. The game data will be distributed to the all five nodes.

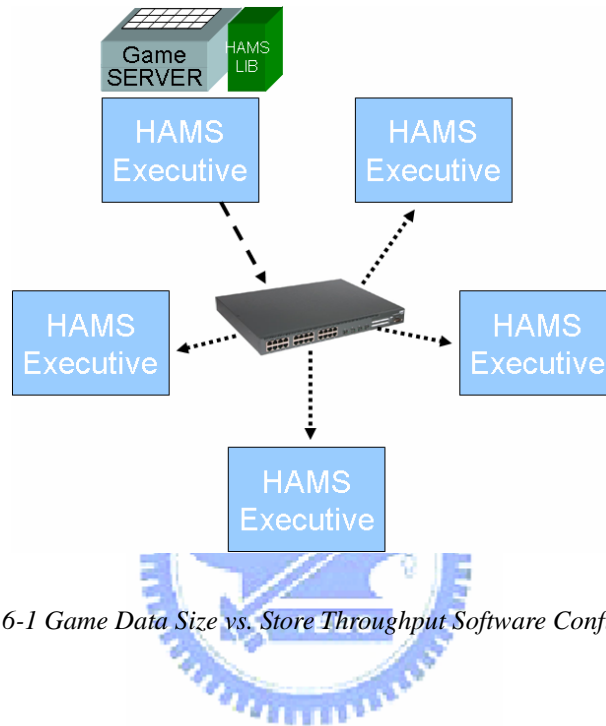


Figure 6-1 Game Data Size vs. Store Throughput Software Configuration

## Result and Discussion

Game Data Size (Bytes)	Throughput (KB / second)
20000	10854.77
40000	15799.39
60000	17407.35
80000	19916.07
100000	20661.39
120000	20806.72
140000	20848.32

160000	21718.44
180000	21928.11
200000	22305.24
220000	23294.43

Table 6-1 Game Data Size vs. Store Throughput Result

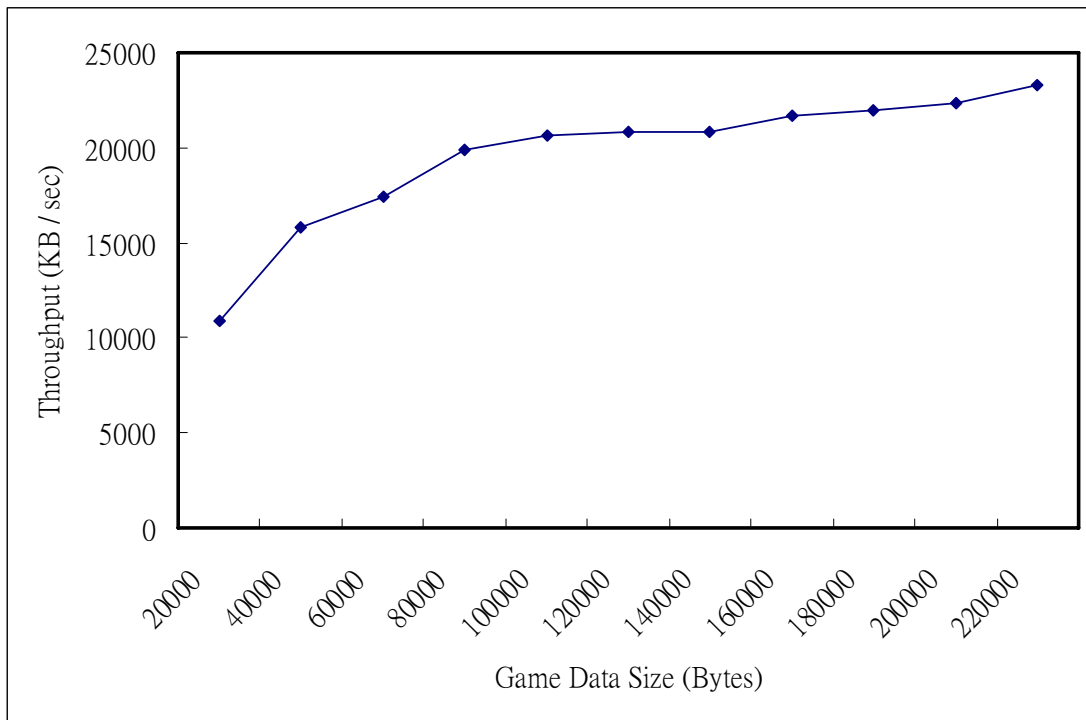


Figure 6-2 Game Data Size vs. Store Throughput Result

Initially, the game data size was 20000 bytes and the average store throughput is about 10MB per seconds. When the game data size grows to 200000 bytes, the store throughput becomes 20MB per second. This is because while the total store size is the same, small game data segment will invoke more requests in HAMS and OPENAIS kernel. Fewer requests will be more efficient. We did not try larger size than 200000 bytes, because there is a message constraint in OPENAIS. In this evaluation, we can see the maximum store throughput when there is only one single server is about

20MB per second.

### 6.2.2. Node Numbers vs. Store Throughput

Generally, there may be many computers that handle one MMOG middleware together. Obviously, we hope the game data storing should not lose its efficiency when the node numbers grows up.

#### Software Configuration

Like Figure 6-1, there is one server storing game data periodically. But the HAMS Executive will be added into the cluster one by one. The Game data size is set to 200000 Bytes which is evaluated to be the best performance in 6.2.1.

#### Result and Discussion

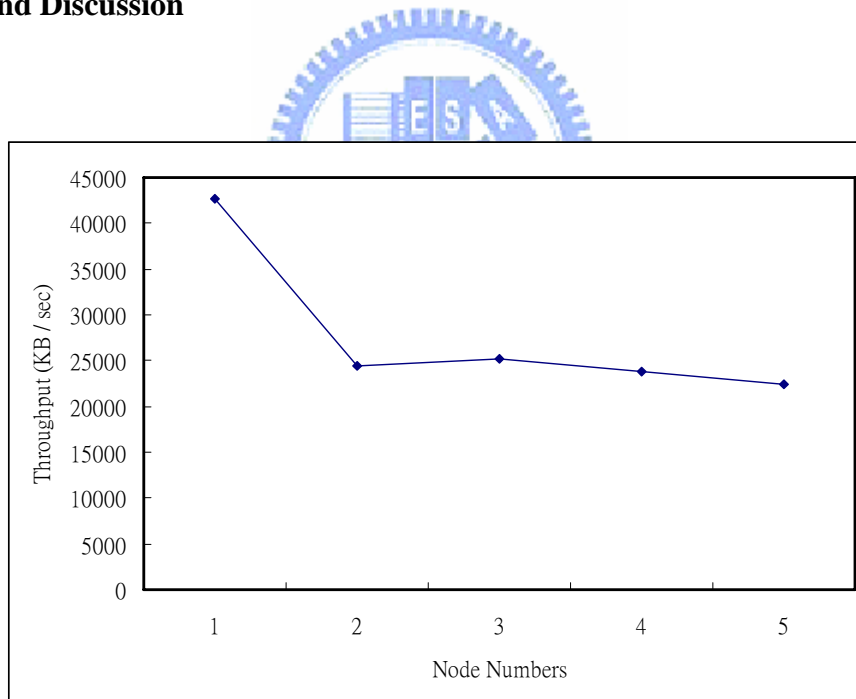


Figure 6-3 Node Numbers vs. Store Throughput

First, we can see that game data store throughput decrease from 40MB/sec to 25MB/sec when the node numbers grows from one to two. It is because while there is

only one HAMS Executive Part, the game data only store into local and require no network bandwidth. The result shows, when we add more nodes, the store throughput remained in 20MB per second. The reason is because we use multicast technology in OPENAIS to store our game data, add nodes will not increase the network bandwidth overhead in HAMS.

### **6.2.3.Store Rate vs. Store Throughput**

The game data store rate is defined by developers. It indicates the frequency of game data store in each server. When there is only one server, the store rate will not affect the throughput. However, while there are several servers, the longer store rate can lower the simultaneous game data storing. Hence, if the game data doesn't change frequently, you should set its store rate longer so that the other can use the network bandwidth.

The other point is fairness. If there are many servers store their game data currently and share the network bandwidth, we hope that each node can get equal chance to store their data.

#### **Software Configuration**

There will be five servers in five different nodes. All of them store there game data simultaneously. We set different store rate each time and measure the throughput in every server. In each store round, we send 10MB data to checkpoint and distributed to other servers. See Figure 6-4.

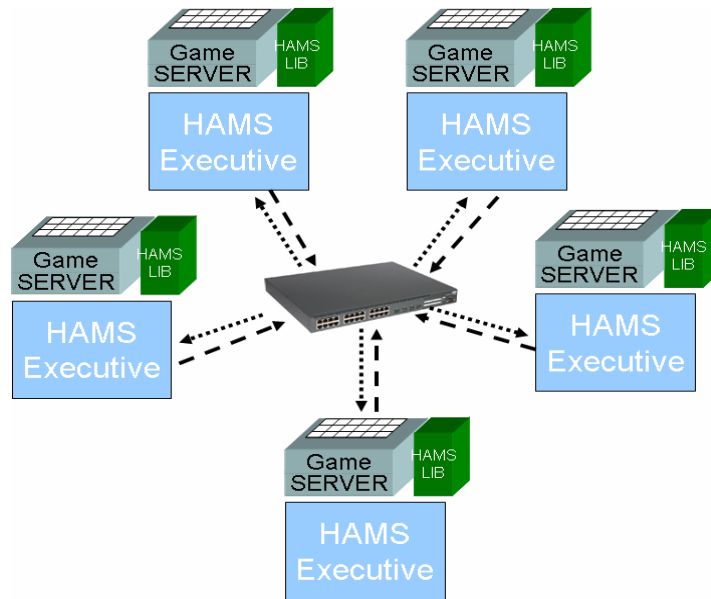


Figure 6-4 Store Rate vs. Store Throughput Software Configuration

## Result and Discussion



Store Rate (ms)	Server1 throughput	Server2 throughput	Server3 throughput	Server4 throughput	Server5 throughput
500	7722.52	7718.88	7709.54	7709.45	7705.72
1000	7700.43	7717.93	7726.46	7683.91	7684.42
1500	12991.73	16423.70	13422.08	14636.68	15059.39
2000	13449.91	17954.36	18011.32	13289.34	13135.01
2500	16796.68	22565.44	20174.74	16194.18	22626.47
3000	17514.06	22691.99	22642.37	22322.41	17498.43
3500	22465.64	22783.72	22008.20	21767.43	22681.87
4000	22367.85	20284.12	20354.02	22372.28	22518.08

Table 6-2 Store Rate vs. Store Throughput (KB / sec)



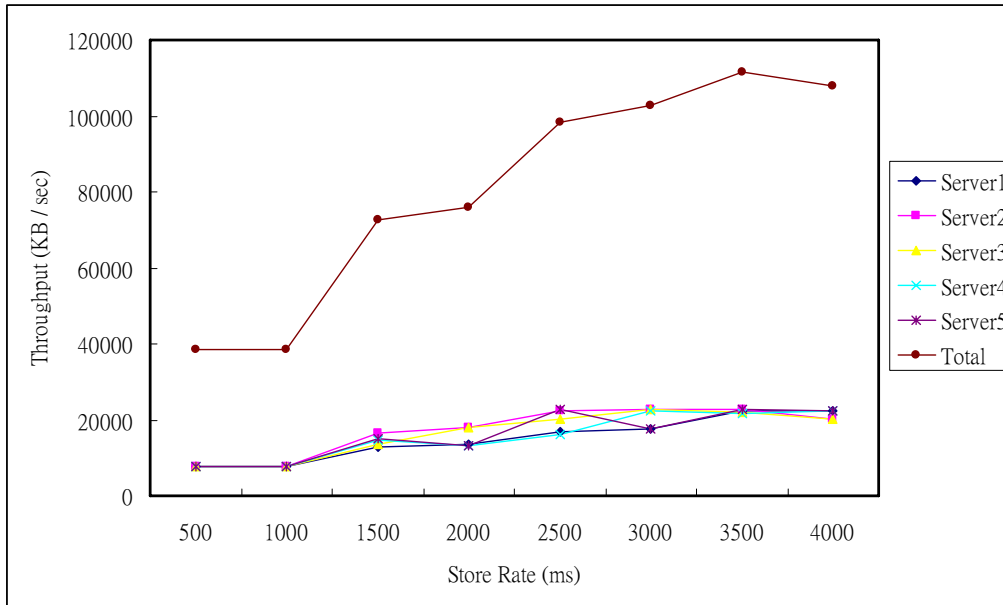


Figure 6-5 Store Rate vs. Store Throughput

First of all, we can see when the store rate was smaller than 1000 milliseconds; the total throughput is almost the same. The network is always busy and the full throughput is about 35MB per second. The result is different from which we measured in 6.2.1 because single server can not fully utilize the network bandwidth.

While the store rate becomes larger, the throughput of each server increases as well. This is because some servers store all data before the next store rate expired. When they are waiting for next store round, the network resource can be used in other servers. Hence, the store throughput arises. When the store rate is larger than 3 seconds, each server's reaches the highest throughput of 20MB in single server.

Finally, the result shows whatever the store rate is, the game store throughput in each server are close. So the store chance is almost the same.

### 6.3. Game Data Read Throughput

When we request game data migration in recovery protocol or load sharing

protocol, the destination server has to read the game data. The performance of read operation will affect the time of these protocols.

### Software Configuration

There are five nodes, and two servers. One server store its game data into HAMS and the other read the game data. We adjust the game data size and measure the read throughput.

### Result and Discussion

Game Data Size (Bytes)	Throughput (KB / second)
20000	13396.94
40000	24119.08
60000	31094.84
80000	37140.59
100000	43841.81
120000	47270.53
140000	44945.32
160000	53482.22
180000	56565.76
200000	58135.56
220000	54208.68

*Table 6-3 Game Data Size vs. Read Throughput*

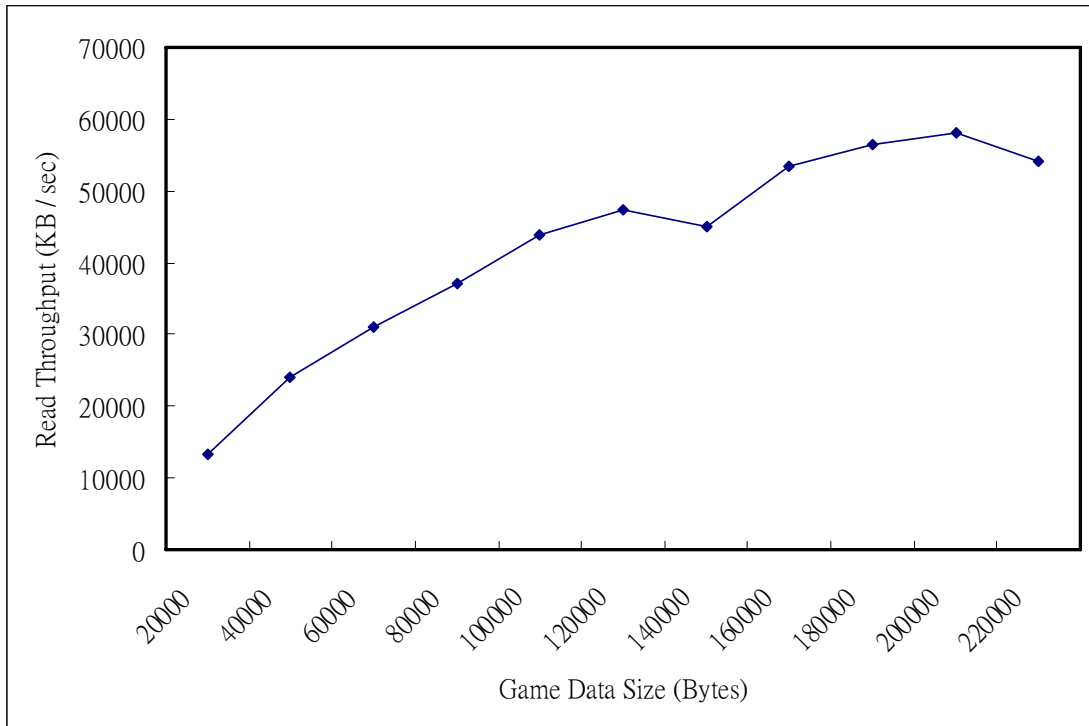


Figure 6-6 Game Data Size vs. Read Throughput

In HAMS, we replicate the game data in all other nodes for availability issue. When a server read the game data from other server (either live or failed), it actually read from local space. So it is much faster than store game data. However, when the game data size is too small, the throughput is still lower than the larger game data size. The maximum read throughput is about 50MB per second.

## 6.4. Failure Detection Time

We start to evaluate the failure detection time in HAMS. Our purpose is to measure the time from one node failed to the representative start running recovery algorithm. The failure detection time includes two parts. First, the OPENAIS membership finds there is a node failed. Second, the HAMS membership finds all failed servers and delivers the *fail\_msg* to the representative.

In section 5.4.1, we showed how OPENAIS detects the node failure by token rotation. We adjust the *token\_lost\_timeout* in OPENAIS and see the different performance of failure detection. Remember, different *token\_lost\_timeout* actually affects the membership of OPENAIS, but not the HAMS membership.

### Software Configuration

Five computers run HAMS. Each of them hosts one server. We let one of the computer leave the membership and log the time. And then, we log the time after the representative is been notified.

### Result and Discussion

Token Lost Timeout (ms)	Failure Detection Time (ms)
250	536.8032
500	780.9614
750	1031.193
1000	1151.134

Table 6-4 Failure Detection Time vs. Token Lost Timeout

As the result in Table 6-4, when we set the timeout longer, it makes HAMS take more time to find out which servers were failed. This is reasonable because the token lost may be caused by network latency or noise, so sometimes we elongate the timeout to avoid the mistrial. However, in network with low packet lost rate, we can choose shorter timeout about 250ms and the HAMS failure detection can be done in 500ms (0.5 second). It means each hardware failure in MMOG middleware can be aware by HAMS in a very short time.

## 6.5. Summary

We have given you some HAMS performance testing result above. We will try to map the real MMOG game content to our services.

First, we hope to know how many avatars status can we store in HAMS per second. An avatars' status includes the positions, equipments, life and etc. The size may vary between MMOG designs. However, some experience shows, the basic information of one avatar occupy 10K to 20K memory size. According to our evaluation, HAMS store throughput is about 38MB per second when there are multiple nodes storing game data concurrently and with proper data size.

$$\frac{38 \times 1024}{20} \cong 1900.$$

It means HAMS can store 1900 to 3800 avatars every second. If there is a MMOG which support 10 thousands of avatars, the average timestamp in the game data replication is  $\frac{1}{2} \times \frac{10000}{1900} \cong 2.5$  seconds before the current game data. The result shows if there is a game server failed, the average avatars status rollback should be 2.5 seconds.

More over, when the failure occurs, according to the result in section 6.4, it takes HAMS 0.5 second to identify which server has been failed. Future more, the average read throughput is about 40MB per second. Assuming the recovery algorithm is simple and can be finished in 1 second. If there are five thousand of avatars (  $20 \times 5000 = 100000$  KB) in the failed server, the whole fail-over time is

$$0.5 + 1 + \frac{100000}{40 \times 1024} \cong 4 \text{ seconds.}$$

---

# Chapter 7 Conclusion and Future Work

---

## 7.1. Conclusion

Because the MMOG market is growing quickly recent years, the demand of MMOG middleware also becomes stronger. A good MMOG middleware must be stable and available and have to be flexible for most of the developers.

The availability issues can be defined in to two phase, fail-recovery and load sharing. Because the hardware is undependable, we must have an approach to recover the game data while there is a server failed. Central controller may cause single point of failure and repairing time would damage the players benefit. Because the unpredictable movement of avatars, the server will be overloaded sometime. We have to share the load between servers dynamically and efficiently.

HAMS (High Availability MMOG Service) provides a high availability environment for MMOG middleware to use and can be customized to most of the common architectures of MMOG middleware. It is fully distributed which have no any central controller. Developers can use HAMS to monitor the servers load and the membership of all game servers. When there is a server overloaded or failed, HAMS helps you to share the load to any other servers or recovery the fail game data. Even more, HAMS if very flexible for users, which means you can monitor your own load object and define your own algorithm to migrate game data you need.

We have also given you some experiments of the performance testing in HAMS. It shows HAMS have the ability to share load efficiently as well as recover failed servers regardless how many nodes are there in the cluster.

Even more, using HAMS in you MMOG middleware will make you utilize your

hardware resource more. Also, you can use HAMS to develop a system monitor to watching the game run time situation. We have described these in section5.7.

## **7.2. Future Work**

There still many functions can be improvement in HAMS.

### **Software failure detection**

Now, HAMS can only overcome the damage cause by hardware failure. However, the MMOG middleware is so complicate and many components may fail in the run time. In the future, we hope to provide an approach to monitor the software components in HAMS and be a more available service.

### **Transparency design**

HAMS now open the algorithms and load object type design to developers. For users that do not familiar with software engineering, they may have trouble to do use HAMS easily. We would like to design some common load objects (CPU utilization, network latency) for users or provide an easy-to-use interface for users to configure the behavior of recovery protocol and load sharing protocol.

### **Testing in real MMOG middleware**

We hope HAMS can be tested in a real MMOG middleware. Because the actual MMOG can be vary and the demand is different. Testing HAMS in a real middleware can help us find more issues that we have not seen before.

---

## Bibliography

---

- [1] David kushner heavy light data centers. EVERQUEST. IEEE Spectrum July 2005
- [2] Chen-en Lu, Tsun-Yu Hsiao, Shyan-Ming Yuan. Design issues of a Flexible, Scalable, and Easy-to-use MMOG Middleware. Proceeding of Symposium on Digital Life and Internet Technologies 2004
- [3] Daniel Bauer, Sean Rooney, Paolo Scotton. Network Infrastructure for Massively Distributed Games. Proceedings of Workshop on Network and System Support for Games (NETGAMES), Braunschweig, Germany, April 2002.
- [4] Timo Jokiahho, Fred Herrmann, Dave Penkler, Manfred Reitenspiess, Louise Moser, Service Availability Forum. The Service Availability™ Forum Specification for High Availability Middleware. The RTC Magazine, June 2003.
- [5] Manfred Reitenspiess, Louise Moser. Application interface specification by the Service Availability Forum. September 2004.
- [6] Josh Adelson, Steven A. Roldan, Paul Kingsepp, Lazar Rozenblat, Ken Grob, Anthony W. Vilgiate, Carrier-Grade Requirements. Focus on Telecom 2001.
- [7] Y. AMIR, L.E. MOSER, P. M. MELLIAR-SMITH, D. A. AGARWAL, P. CIARFELLA. The Totem Single-Ring Ordering and Membership Protocol. ACM Transactions on Computer Systems 13, 4 November 1995.
- [8] Deborah A. Agarwal. TOTEM: A Reliable Ordered Delivery Protocol for Interconnected Local-Area Network. Dissertation Submitted in Partial Satisfication of University of California Santa Barbara.
- [9] L.E. MOSER, P. M. MELLIAR-SMITH, Deborah A. Agarwal, TOTEM: A Fault-Tolerant Multicast Group Communication System. Communications of the



ACM April 1996

- [10] D. A. AGARWAL, L. E. MOSER, P. M. MELLIAR-SMITH, and R. K. BUDHIA The Totem Multiple-Ring Ordering and Topology Maintenance Protocol. ACM Transactions on Computer Systems. May 1998.
- [11] Efstratios Thomopoulos, Louise E. Moser, Peter M. Melliar-Smith. Latency Analysis of the Totem Single-Ring Protocol. IEEE/ACM TRANSACTIONS ON NETWORKING 2001.
- [12] Mu-Chi Sung, Ming-Chun Cheng, Zhi Xin Fan, Ping-Jer Yeh, Shyan-Ming Yuan, et al. An Experimental Study toward Failure Impact on OpenAIS. To appear in WSEAS Transactions on Computers 2006.
- [13] Ta Nguyen Binh Duong, Suiping Zhou. A dynamic load sharing algorithm for massively multiplayer online games. Networks 2003, The 11<sup>th</sup> IEEE International Conference on 28 Sept.-1 Oct. 2003.
- [14] Kyungmin Lee, Dongman Lee. A scalable dynamic load distribution scheme for multi-server distributed virtual environment systems. Proceedings of the ACM symposium on Virtual reality software and technology 160 – 168, 2003.
- [15] Dugki Min, Eunmi Choi, Donghoon Lee, Byungseok Park. A load balancing algorithm for a distributed multimedia game server architecture. Multimedia Computing and Systems, 1999. IEEE International Conference on, July 1999.
- [16] Laurent Gautier, Christophe Diot, Jim Kurose. End-to-end transmission control mechanisms for multiparty interactive applications on the Internet. INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, March 1999.
- [17] Christophe Diot, Laurent Gautier. A Distributed Architecture for Multiplayer Interactive Applications on the Internet. Network, IEEE, Jul/Aug 1999.
- [18] F' abio Reis Cecin, Rodrigo Real, Rafael de Oliveira Jannone, Cl' audio

- Fernando Resin Geyer. FreeMMG: A Scalable and Cheat-Resistant Distribution Model for Internet Games. Proceedings of the Eighth IEEE International Symposium on Distributed Simulation and Real-Time Applications, 2004.
- [19] Eric Cronin Burton Filstrup Anthony Kurc. A Distributed Multiplayer Game Server System. UM EECS589 Course Project Report, 2001.
- [20] Stefano Ferretti. Interactivity Maintenance for Event Synchronization in Massive Multiplayer Online Games. Technical Report BLCS-2005-05 March 2005.
- [21] Stefano Ferretti, Claudio E. Palazzi, Marco Rocchetti, Giovanni Pau, Mario Gerla. FILA, a Holistic Approach to Massive Online Gaming, Algorithm Comparison and Performance Analysis. ACM Journal of Computer in Entertainment, ACM Press, to appear, selected as Best Paper Award at the ACM GDTW'05 Conference, November 2005.
- [22] Lun-Wu Yeh, Shyan-Ming Yuan. A Research of Persistence Component on MMOG Middleware.
- [23] BigWorld Technology. [http://www.bigworldtech.com/index/index\\_en.php](http://www.bigworldtech.com/index/index_en.php)
- [24] Application Interface Specification, SAI-AIS-B.01.01, November 2004. Service Availability Forum <http://www.saforum.org>.
- [25] OPENAIS Standard-Based Cluster Framework.  
<http://developer.osdl.org/dev/openais/>
- [26] JAVA Technology. <http://java.sun.com/>