

國立交通大學

資訊科學系

碩士論文

高 彈 性 高 延 展 性 易 用 之
M M O G 中 介 軟 體 之 設 計



Design Issues of a Flexible, Scalable, Easy-to-use
MMOG Middleware

研究生：陸振恩

指導教授：袁賢銘 教授

中華民國九十三年六月

高彈性高延展性易用之 MMOG 中介軟體之設計
Design Issues of a Flexible, Scalable, and Easy-to-use
MMOG Middleware

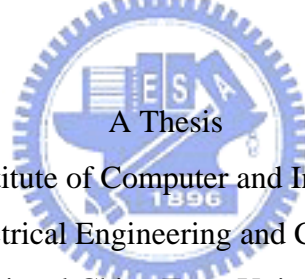
研究生：陸振恩

Student：Chen-En Lu

指導教授：袁賢銘

Advisor：Shyan-Ming Yuan

國立交通大學
資訊科學系
碩士論文



Submitted to Institute of Computer and Information Science
College of Electrical Engineering and Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer and Information Science

June 2004

Hsinchu, Taiwan, Republic of China

中華民國九十三年六月

中華民國九十三年六月

高彈性高延展性易用之 MMOG 中介軟體之設計

研究生: 陸振恩

指導教授: 袁賢銘

國立交通大學資訊科學系

摘要

MMOG(Massively Multiplayer Online Game)是一種可以支援上千人同時在一個虛擬遊戲世界互動的遊戲類型。然而，開發一個 MMOG 的難度遠超過我們所想像，很多議題都是遊戲開發者可能會遭遇到的，例如高效能的網路、分散式的技術、負載平衡等。同時，要開發一個具有競爭力的遊戲，Time-to-Market 也是十分重要的考量。因此，MMOG 中介軟體解決方案的需求大量增加，而 MMOG 中介軟體的研究也逐漸受到重視。

在本篇論文中，我們分享在開發 MMOG Middleware 的經驗。一方面，我們試圖解決開發 MMOG 所會遇到的共同問題。另一方面，我們嘗試的簡化設計遊戲的 API 以減低開發 MMOG 的難度。除此之外，我們也保留彈性給開發者以部署各種不同類型的遊戲在平台之上。

Design Issues of a Flexible, Scalable, and Easy-to-use MMOG Middleware

Student: Chen-en Lu

Advisor: Shyan-Ming Yuan

Department of Computer and Information Science

National Chiao Tung University

Abstract

MMOG (Massively Multiplayer Online Game) is a game genre, which allows more than 1,000 players to play in the same game through a network concurrently. However, it is much harder than we think to develop a MMOG. There are plenty of issues that MMOG developers may encounter, such as high performance network, distributed technology, load balancing, and so forth. In the meantime, time-to-market is also an important issue to develop a competitive MMOG. Therefore, the demand of middleware solutions for MMOG rises by degrees and the research of MMOG middleware gains more and more respect.

In this paper, we would like to share our experience on designing our MMOG middleware. On one hand, we try to overcome the common issues arisen in the MMOG environment. On the other hand, we attempt to simplify the API in order to reduce the complexity of developing MMOGs. In addition, our platform leaves developers the flexibility to adopt the solution and deploy it to any kind of MMOGs.

Acknowledgement

首先我要感謝袁賢銘教授給我的指導，在我的研究領域裡給予我很多的意見，並且給予我最大的空間來發揮我的創意。也感謝所有幫助我的學長蕭存喻、高子漢、葉秉哲、鄭明俊、邱繼弘、李宣峰，在我研究的過程中給我不少的指導跟建議。還有感謝蘇科旭跟葉倫武，在這一年來跟我一起做了很多研究討論，也激盪出不少的想法。還有所有實驗室內提供電腦給我做實驗的同學，沒有你們我無法完成實驗。也感謝我的心愛女友彭品勻，在我最無力的時候給我打氣，讓我一直有動力來完成我的研究。最後我要感謝我的爸媽，給予我這個良好的環境讓我求學生涯毫無後顧之憂，專心於學業，謹以這篇小小的學術成就來感謝您們的養育之恩。



Table of Contents

Acknowledgement	iii
Table of Contents	iv
List of Figures	vii
List of Tables	ix
Chapter 1 Introduction	1
1.1. Preface.....	1
1.2. Motivation.....	1
1.3. Research Objectives.....	2
1.4. Research Contribution	4
1.5. Outline of the thesis.....	4
Chapter 2 Background	5
2.1. MMOG Design Concept.....	5
2.2. Communication Architectures	6
2.2.1. Client-Server Architecture	6
2.2.2. Peer-to-peer Architecture	6
2.2.3. Client-Gateway-Server Architecture.....	6
2.3. Communication Protocol	7
2.3.1. Message-based Protocol.....	7
2.3.2. RPC-based Protocol.....	8
2.4. Related Works	8
Chapter 3 System Architecture	10
3.1. Game Servers	10
3.2. Gateways.....	11

3.3.	Coordinator	12
3.4.	Client.....	12
3.5.	Other Components	12
3.5.1.	Proxy	12
3.5.2.	Database.....	13
3.5.3.	Update Server.....	13

Chapter 4 Implementation Details14

4.1.	Network Engine	14
4.2.	Message wrapping and Message routing	15
4.3.	Channel State Synchronization	16
4.4.	Avatar Migration.....	17
4.5.	Login Issues	19
4.6.	Plugin Framework.....	19
4.7.	Region Migration.....	20
4.8.	NPC Support.....	22



Chapter 5 Design of Programming Interface.....24

5.1.	NetEngine Library	24
5.2.	DOIT API.....	27
5.2.1.	Game Protocol and Game Logic.....	28
5.2.2.	GameContext and GameSpace	30
5.2.3.	AvatarChannelListener	32
5.2.4.	Avatar Migration.....	32
5.2.5.	NPC.....	32
5.2.6.	Mapping the features to the API	33
5.3.	Game Deployment	34
5.4.	Plugins Framework	37

Chapter 6 Experiment and Evaluation.....	39
6.1. Round 1: NetEngine Performance	39
6.1.1. Hardware Configuration	39
6.1.2. Software Configuration.....	40
6.1.3. Experiment Result.....	41
6.1.4. Evaluation	41
6.2. Round 2: Performance baseline of DOIT Platform	42
6.2.1. Hardware Configuration	42
6.2.2. Software Configuration.....	43
6.2.3. Experiment Result.....	44
6.2.4. Evaluation	45
6.3. Round 3: Simulate the Real Game	45
6.3.1. Hardware Configuration	45
6.3.2. Software Configuration.....	46
6.3.3. Experiment Result.....	48
6.3.4. Evaluation	53
6.4. Summary	55
Chapter 7 Future Works and Conclusions	56
7.1. Conclusions.....	56
7.2. Future Works.....	57
Bibliography	59
Appendix A System Protocol.....	61

List of Figures

Figure 2-1: Client-Gateway-Server Architecture.....	7
Figure 2-2: System Architecture	10
Figure 3-1: Message Wrapping.....	15
Figure 3-2: Message Routing.....	16
Figure 3-3: Avatar Migration, source region and destination region are in the same server.....	18
Figure 3-4: Avatar Migration, source region and destination are in different servers.. ..	18
Figure 3-5: Login Region.....	19
Figure 3-6: Server Component-based Framework.....	20
Figure 3-7: Region Migration	22
Figure 4-1: Class diagram of NetEngine library	25
Figure 4-2: class diagram of the DOIT API.....	28
Figure 4-3: GameContext	30
Figure 4-4: GameSpace.....	31
Figure 4-5: NPC	33
Figure 4-6: mmog.xml	35
Figure 4-7: vwlogic.properties.....	36
Figure 4-8: npc.properties.....	36
Figure 4-9: MyPlugin.properties.....	38
Figure 5-1: Communication architecture of round 1	40
Figure 5-2: Experiment result of round1	41
Figure 5-3: Communication architecture of round 2	43

Figure 5-4: Experiment result of round 2	44
Figure 5-5: The move logic.....	46
Figure 5-6: Communication architecture of round 3	47
Figure 5-7: Average response time – total	48
Figure 5-8: Standard deviation total	49
Figure 5-9: Average response time – migration.....	50
Figure 5-10: Standard Deviation – Migration	51
Figure 5-11: The total amount of messages forwarded per second in a gateway	53



List of Tables

Table 5-1: API for our MMOG Middleware.....	34
Table 6-1: Hardware configuration of round 1	39
Table 6-2: Experiment result of round 1	41
Table 6-3: Hardware configuration. of round 2	42
Table 6-4: Experiment result of round 2	44
Table 6-5: Hardware Configuration of round 3	45
Table 6-6: Average Response Time – Total	48
Table 6-7: standard deviation – total.....	49
Table 6-8: Average response time – migration.....	50
Table 6-9: Standard Deviation – Migration.....	51
Table 6-10: CPU load.....	52
Table 6-11: The amount of commands forwarded per second in a gateway	52
Table 6-12: The total amount of messages forwarded per second in a gateway	52

Chapter 1 Introduction

1.1. Preface

MMOG (Massively Multiplayer Online Game) is a game genre, which allows more than 1,000 players to play in the same game through a network concurrently. However, it is much harder than we think to develop a MMOG. There are plenty of issues that MMOG developers may encounter, such as high performance network, distributed technology, load balancing, and so forth. In the meantime, time-to-market is also an important issue to develop a competitive MMOG. Therefore, the demand of middleware solutions for MMOG rises by degrees and the research of MMOG middleware gains more and more respect.



1.2. Motivation

Although there are some middleware solutions proposed in the market, such as Terazona [1], Butterfly.net [2], it seems that most commercial MMOGs didn't take them into consideration. We think that there are three reasons. First of all, game is a product of creativity. It is not acceptable that the creativity is limited by a given middleware. Therefore, an inflexible MMOG middleware will be knocked out from the choices of MMOG developers. Secondly, the programming interface should be as easy as possible. The steep learning curve increases the risk and cost of developing a MMOG. Usually, game logic developers do not have to be experts of distributed technology. So the middleware providers need to provide exception-free, thread-safe, clean, and simple API for developers. The third pertains to the scalability of the middleware. The number of serving clients can be linearly scaled up by adding new

machines to the cluster. In term of the need, it may require some load sharing mechanism to deal with the unbalanced loading among clusters. Therefore, we developed our MMOG middleware, named DOIT platform. We wish to overcome all these issue in this platform.

1.3. Research Objectives

There are four objectives in this paper: scalability, flexibility, easy-to-use, and high performance.

Scalability

Scalability is always the issue on building MMOGs. The traditional single server architecture is relatively harder to serve thousands of players concurrently. There is no doubt that distributed technology plays an important role in the MMOGs. Scalability can be view as two parts. First is the number of players served can scale in a linear way. The second is the game objects can be scaled up in a linear way. Most people only emphasis the first point, but it is a good practice to separate the scalability of real world view and virtual world view.

Flexibility

The wired protocol must not be limited by the Middleware. Even if the middleware requires some information in the header so that it can properly route and dispatch the messages, the defined protocol should be as simple as possible. This is because the MMOG middleware should preserve the resilience to MMOG developers. Although some middlewares provide client libraries to have interaction with servers, there are still various targets which clients run at, such as game consoles, mobile devices, and so on. The client libraries provided by provider are never enough to meet

MMOG developers demand. A good MMOG middleware should provide the maximum resilience for the wired protocol.

Moreover, the game objects in the virtual world are also as flexible as possible. A MMOG middleware should not assume the form of game object. For example, the player should have the attributes of hp, mp, name, and so on. It will limit the creativity of a game.

Easy-to-use

Because MMOG platform is very comprehensive, a middleware has the responsibility to hide the underlying complexity. The API should be designed as a game view instead of the implementation details.

High Performance

Networked games demand fast response time, and MMOGs is no exception. In [3] we understand that the latency more than 300ms can become aware by players. Poor performance will destroy the mood to play. The performance issue can be divided into two parts:

1. Efficient protocol and game logic. The protocol should be decided as efficient as possible and the game logic should not send unnecessary message to unrelated client.
2. High performance underlying network and routing technology. Serving thousands of players is not a naïve task. If the underlying network is not designed well, it could be the bottleneck of the whole system. So is the routing strategy of messages. Bad-decided routing strategy will exponentially increase the load of system.

From the middleware view, the first part is beyond the task the middleware can handle,

but the second one is the responsibility of good MMOG middleware.

1.4. Research Contribution

To achieve these objectives, we encounter many problems when designing the DOIT platform. In this paper, we address these problems and explain our solution. There are four major contributions of this paper.

1. We craft a scalable MMOG middleware based on client-gateway-server architecture.
2. We introduce an easy-to-use programming model
3. We discuss the flexibility issues in the MMOG middleware and explain our solution
4. We design a high performance underlying network facility and experiment the performance of MMOG platform.



1.5. Outline of the thesis

In Chapter 2, we discuss the background of developing the MMOG. In Chapter 3, we introduce the components in the platform and their responsibility. In Chapter 4, we discuss the implementation details for each function. It will include how the components cooperate with each other for a given function and the issues we will face and how to overcome them. In Chapter 5, we turn to developers' perspective. We can see how easy to build up a MMOG logic. In Chapter 6, we build up three test beds to experiment and evaluation the DOIT platform. In Chapter 7, we give the conclusion and future works.

Chapter 2 Background

2.1. MMOG Design Concept

MMOG is a virtual world in which there are thousands of players interact with each other simultaneously. Within the virtual world the player is represented by an “avatar”. The avatar could be a human, animal, or whatever. To design a mmog, there are two principles we should hold:

1. Command/Update principle
2. Never trust the clients.

The first explain that the MMOG is run in the command/update manner, that is to say, clients control avatars by sending command messages to server, and the server returns clients the update message to indicate the changes of state in the server. The reason we should design in this way is because the synchronization of game states is very difficult in a huge virtual world. To centralize the game states and the game logic is the only way to overcome the synchronization problem. Hence, clients send the command message to server rather than process the game logic at local. The server is responsible for updating the states of virtual world clients are interested in.

For the second one, never trust the clients. Although we can leave some game logic processed in the client-side, we should take the risks that the client should falsify the game states such that the fairness of virtual world will be threatened. Therefore, we should leave all the major game logic in the game servers. But we also can leave some game logic which is no matter of fairness in the client, such as path finding.

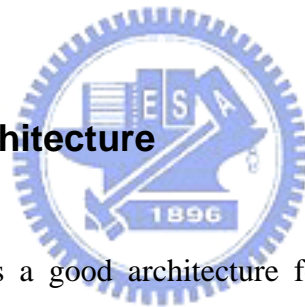
2.2. Communication Architectures

This subsection discusses the choice of communication architectures that MMOG can apply. We address three types of communication architectures and discuss the advantages and disadvantages.

2.2.1. Client-Server Architecture

Certainly, the traditional client-server architecture can apply to MMOG platform. But it suffers from the scalability issue. One server can only serve limited number of players. If we want to serve more players than the limit of one server, we should open two separated virtual worlds.

2.2.2. Peer-to-peer Architecture



Although peer-to-peer is a good architecture for multiplayer game, there are many problems for MMOG. The first one is the states consistency problem. How to share the states over thousands clients? The second and the most serious problem is the cheating problem. In peer-to-peer architecture, the game logic is process in clients' machine. It is very easy to falsify the game states by mean of some hacking tool. The fairness of virtual world is very hard to hold. Therefore, the peer-to-peer architecture is not suitable for MMOG.

2.2.3. Client-Gateway-Server Architecture

The Client-Gateway-Server architecture has been proposed and proven as an effective architecture for MMOG [4]. In this architecture, clients connect to the

MMOG through gateway. Then, a gateway is responsible for dispatching messages to the corresponding game server (see Figure 2-1). The benefit of this architecture is that the gateway can increase when we need to serve more players. Furthermore, the virtual world can also span to the servers. When the virtual world extends, we also can add more servers to the server cluster. Undoubtedly, we adopt the client-gateway-server architecture in the DOIT platform. In this architecture, we can achieve the scalability objective.

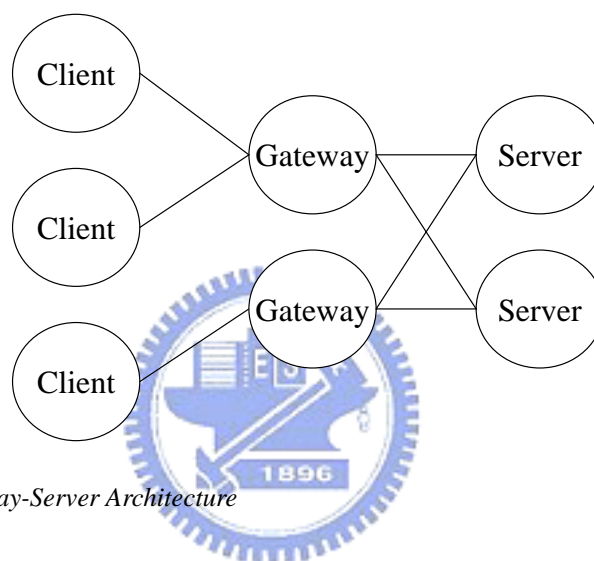


Figure 2-1: Client-Gateway-Server Architecture

2.3. Communication Protocol

Servers and clients communicate with each other by means of some communication protocol. However, we may adopt different kind of communication protocol in different situation. We address two kind of protocol and compare their advantages and disadvantages.

2.3.1. Message-based Protocol

All data are encapsulated as messages in this kind of protocol. The protocol is defined as a list of message types. The game developers should define the message

format for each type. Messages can be text-format or binary-format. The protocol with text-format is easy to deal with and easy to debug, but the performance is relatively poor. On the contrary, binary-format protocol will gain more performance. In the DOIT platform, we select message-based binary-format protocol as our communication protocol. To choose message-based is because it is easy to extend and program. The asynchronous property of message-based protocol is also suitable for most game genre. Concerning the binary-format, the package size and computation overhead are not practical for high performance MMOG middleware.

2.3.2.RPC-based Protocol

Some games are designed with RPC-based protocol. They regard game objects as remote objects. The clients control these game objects in the manner of RPC-like call. The advantage is clients can control the game objects in a high-level view. But it has poor performance and poor flexibility due to the synchronous programming model. In Wish Engine [5], they use RPC-based Protocol as their transmission protocol [6].

2.4.Related Works

In fact, the research middleware platform has been evolved for two years in our lab (DCSLab of CIS NCTU). The first work is purposed last year by Lee[7]. In this work, he proposed a scalable MMOG platform. However, the easy-to-use and flexibility issue were not taken into consideration.

There are some MMOG middleware products in the market. The first one is Terazona[1] by Zona Inc. Terazona use JMS[8] as their underlying communication facility. It also exploits multi-tier architecture to achieve scalability. The other one is

Butterfly.net[2]. Butterfly.net adopts the grid computing technology as their MMOG solution. They span the regions to multiple servers. They use IP Multicast to communicate between nodes. Besides, BigWorlds[9] and Wish Engine[5] are also the commercial MMOG middleware products in the market.

As for the academic domain, there are few researches special for MMOG middleware. However, there are many researches relative to this domain. In [10, 11], they discuss the dynamic loading issues in distributed virtual environment (DVE). In [10], they propose a scalable load balancing scheme for a large-scale DVE system. In [11], they propose an adaptive load balancing technique of global scope for solving the partition problem in DVE systems.

In [12, 13], they discuss the relevance filtering issues in MMOGs. In [12], they present a relevance filtering scheme for a two-tier server architecture optimized for MMOGs. They distinguish between interest management of server tier game state and bandwidth adaptation of concentrator tier client link thresholds, making the concentrator tier totally application independent. In [13], they propose a completely distributed publish-subscribe system supporting a content-based publish-subscribe model of communication and performs distributed matching using a novel content-based routing protocol.

As for the research of the Networked Virtual Environment (NVE), the book “Network Virtual Environments”[14] provides much information about this domain. There are many concepts of NVE valuable for the design of MMOGs.

Chapter 3 System Architecture

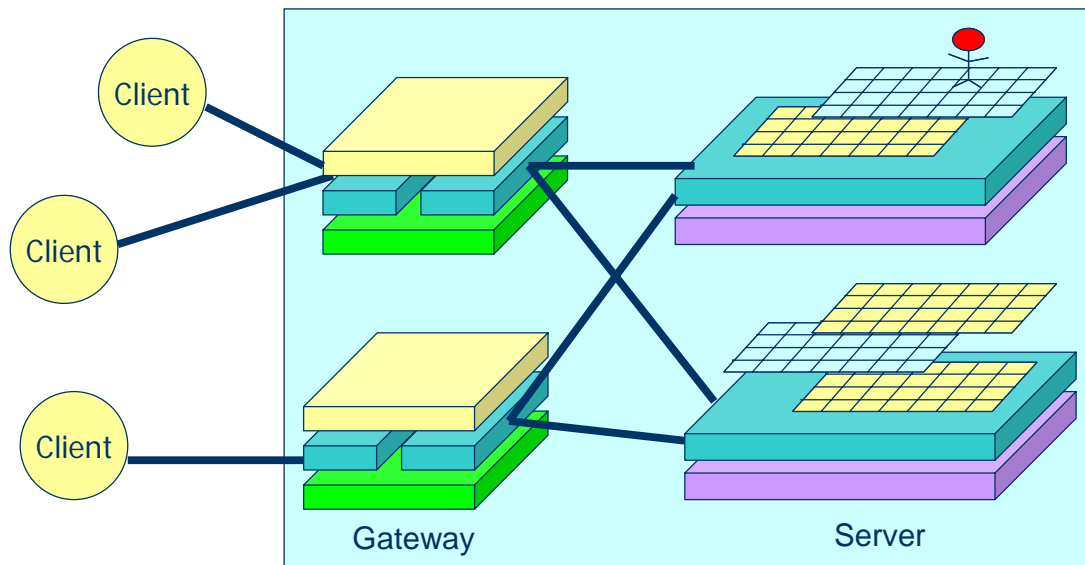


Figure 3-1: System Architecture

In DOIT platform, the client-gateway-server architecture is selected as our architecture. This section explains each component in our platform. It contains: Game Servers, Gateways, and Coordinator. The Figure 3-1 depicts the architecture.

3.1. Game Servers

Game Servers are the most important components in our platform. Game server passively accepts the Gateway connection for receiving messages from client and actively connects to Coordinator for participating in region migration. Note that there is no connection between each server when the system starts up, but one server may dynamically connect to another one on the process of region migration.

A Game Server is capable of computing the game logic and keeps the game states. It receives messages from external (mostly sent from client), changes the states of virtual world, and then returns the update to the corresponding client and

component. The game logics are provided by game developers and deployed on Game Servers.

Virtual world is divided into multiple regions. Every Game Server may contain zero to many regions and a region can't span across more than one server. A region is the concept of a single virtual world. In the same region, the game logics share the same game context, the same game space. In addition, an avatar in a region may migrate to another region. The detail is taken up in chapter 4.4. For the load sharing purpose, a region in a Game Server may migrate to another Game server. The detail will be also discussed further in chapter 4.7.

3.2. Gateways

Gateways can be seen as the interface between MMOG platform and the internet. On the internal side, they actively connect to game servers; on the external side, they passively accept connections from clients. The major responsibility of a Gateways is to forward control message from client to the server and forward update messages from server to the client. Actually, a gateway does not simply forward messages but assembles messages as an internal form. The detail will be discussed further to chapter 4.2.

We can also add some services on a gateway. For example, we can provide several kinds of protocol at the external interface. If a client is not under a firewall, he or she can communicate with server in UDP/IP in order to gain more efficient communication. On the contrast, for clients under a firewall, we provide TCP/IP as underlying protocol in order to pass through the firewall. Provided that a game demands high security, it can also expose SSL as its communication protocol. Another example is that a gateway can also provide hack protection mechanism. The most

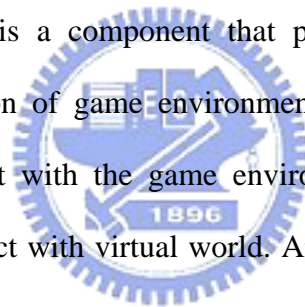
famous hack technology is known as “acceleration program.” It makes sense to add the facility at a gateway to protect against this kind of game plugin.

3.3. Coordinator

Coordinator is the component that coordinates the process of region migration. It monitors the load of game servers and initiates the process of region migration. In the current implementation, coordinator is only a console application. The process of region migration can only be initiated by a human.

3.4. Client

The Client component is a component that player directly interact with. It provides graphics presentation of game environment and interaction interface, etc Keyboard, mouse, to interact with the game environment. The Client component connects to gateway to interact with virtual world. And the protocol is defined game by game.



3.5. Other Components

The components mentioned above are the core components in the DOIT platform. But for a commercial MMOG, these components are not sufficient yet. We recommend some components to collaborate with the DOIT platform.

3.5.1. Proxy

The Proxy component is a component that forwards messages between client and gateway. The difference between gateway and proxy is that a gateway will

assemble/disassemble messages but a proxy won't. A proxy is not necessary in the whole architecture, but it is a good idea to deploy this component for each ISP in order to improve the communication quality from each ISP.

3.5.2.Database

Persistence is also an important part for a MMOG. But in our platform, we didn't define any component about persistence. The reason is that there are many O/R mapping solution in the real world. It is unnecessary to reinvent the wheels. To integrate the persistence mechanism, we can design it as a game logic or a server plugin.

3.5.3.Update Server

Update server is the component that updates the latest version of client program before clients' logging in the server. Web server is a good idea to implement an update server. We can provide a configuration file to indicate which version is the latest one and describe where the latest version client program locates at. The configuration file is put on a given URL. The client program retrieve the configuration file and compare the version with the configuration file version. If the version is old, get the latest version from the location specified in the configuration. If the client is written by java, Java Web Start technology is a good idea to facilitate the update mechanism.



Chapter 4 Implementation Details

In this chapter, we begin to dive into the implementation details for developing a MMOG middleware. We make use of java as our programming language. We first introduce the underlying network and the basic message routing. Then, we will explain the channel state synchronization issue that we will face in client-gateway-server architecture. Next, we discuss the process of avatar migration and login issue. Furthermore, we will introduce the plugin framework in our Game Server and explain the process of region migration, which enable the game server cluster to share the load in the runtime. Finally, we talk over how to implement the NPC feature in the DOIT platform.

4.1. Network Engine



MMOGs communicate with clients in a game-specific protocol. Generally speaking, the protocol is defined as message-oriented, which consists of message type and payload. Thus, we also define a message-oriented facility called NetEngine. In this library, all data are wrapped as messages with two bytes for type, two bytes for payload length, and payload with the length specified before. NetEngine abstract the implementation by mean of java interface. We can implement it by TCP, UDP, IP Multicast and so on. In current work, we only implement the TCP version, but we implement in two different models. First is event-driven model, by selector in java NIO. The other is threaded model. The reason is that for difference number of connection, the difference model gain better performance [16]. For example, we use event-driven model in the external interface of Gateway for the purpose that it demands to serve hundreds of client concurrently. The event-driven model is preferred

in this condition. Relatively, the game server use threaded model to gain more efficiency.

4.2. Message Wrapping and Message Routing

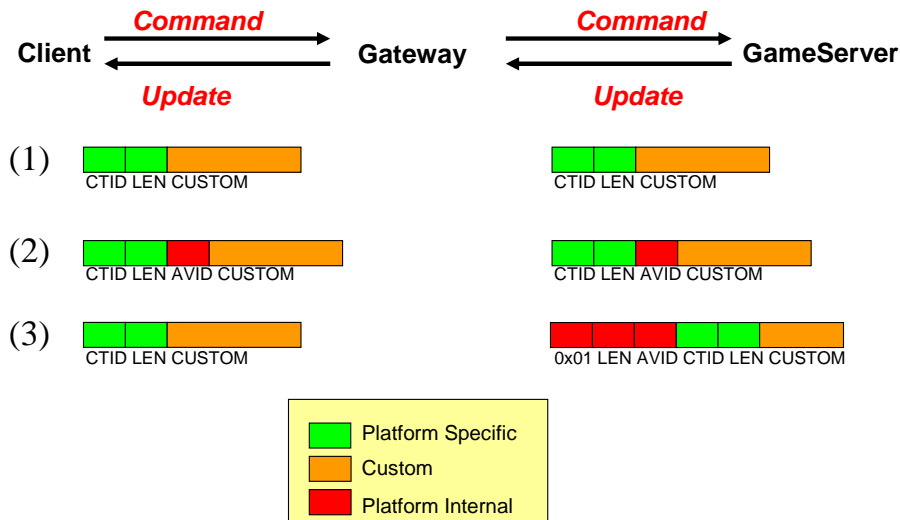


Figure 4-1: Message Wrapping

The DOIT platform adopts the client-gateway-server architecture. The gateway should forward the message between Client and Server. However, there are some problems when directly forwarding messages from client and server. The first policy in Figure 4-1 depicts the circumstance of directly forwarding. The game server will receive a message without the information of which client the message is sent. Therefore, we may consider that we can add the client identifier (or avatar identifier) for each message (see (2) of Figure 4-1). But where is the identifier generated? It is impossible that the identifier is generated by the game logic because the middleware can not depend on the game-specific data. So the identifier is generated by middleware itself. But the platform internal information exposes to the client is also a bad design. Therefore, we finally decide that the gateway injects the avatarid into a message when forwarding it to server (see (3) of Figure 4-1). The avatarid is generated by gateway when a client connects in. Throughout the connection, the

avatarid is used to represent the client. So when a command is sent from client, the gateway will first wrap the message and forward to server. When an update is sent from server, the update message is a wrapped message with the avatarid. Then the gateway will unwrap the message, retrieve the avatarid, forward the message to the corresponding client.

As for routing message from client to server, the gateway should be aware of the location of each server. In the DOIT platform, however, an avatar may migrate from one region to another and a region may migrate from one server to another. Hence, when a message is forwarded to server, we perform two-level-mapping to find out the location the avatar resides. The first mapping is from avatarid to regionid. This table will be updated when an avatar is migrated. The second mapping is from regions to serverid. This table will be updated when a region migration is performed. Figure 4-2 depicts the flow of message routing.

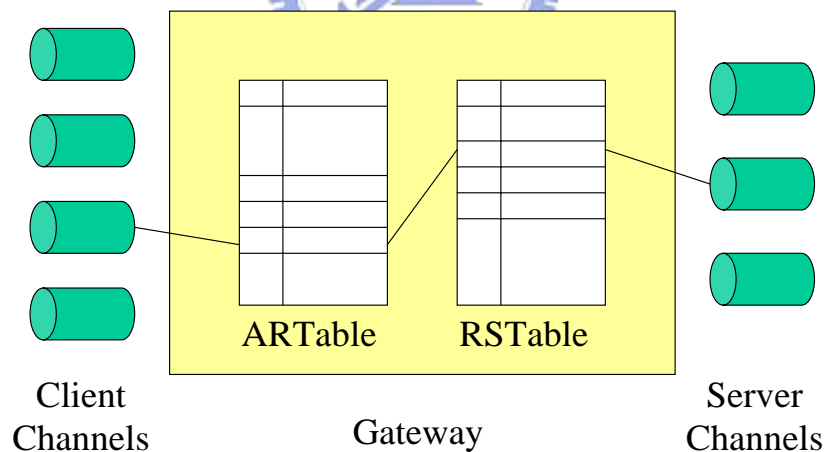


Figure 4-2: Message Routing

4.3. Channel State Synchronization

In the client-gateway-server architecture, GameServer is aware of normal disconnection by a game-specific logout message, but GameServer is not aware of the

abnormal disconnection, such as link broken, crash, and so on. Therefore, we should synchronize the states between Gateway and GameServer. Thus, we implements a listener to listen the events of channel connected and channel disconnected of the external NetEngine in the gateway. When a channel is established, the gateway will send a *AVACONN* message to the server and when a channel is closed, the gateway will send a *AVADISC* message to the server. In addition, the channel can be closed in the game logic, the server will send a *AVACLOSE* message to the gateway if a channel is closed by game logic.

4.4. Avatar Migration

Due to the virtual world is divided into multiple regions. An avatar may migrate from one region to another region dynamically. We call it *Avatar Migration*. To implement this feature, the following should be considered. First, what data should be moved to destination? Secondly, the gateway should be aware of this migration in order to let gateway dispatch messages to new region for the successive messages. Thirdly, we should consider the two conditions that source and target region in the same server and they are in the different servers.

We use *AVAMIG* sent between gateways and servers. *AVAMIG* consists of avatarid indicating which avatar to migrate, source region id, target region id, and avatar data. We must note that the data are provided by developers himself through the API in the form of byte array. Figure 4-3 explains the condition that source and destination regions are in the same region. The game server migrate the avatar data to destination directly. In the meantime, it sends an *AVAMIG* to notify the gateway the update of avatar location. Figure 4-4 explains the condition that source and destination regions are in different server. The source game server sends an *AVAMIG*

to the gateway, and then the gateway forwards this message to the game server where the destination region resides.

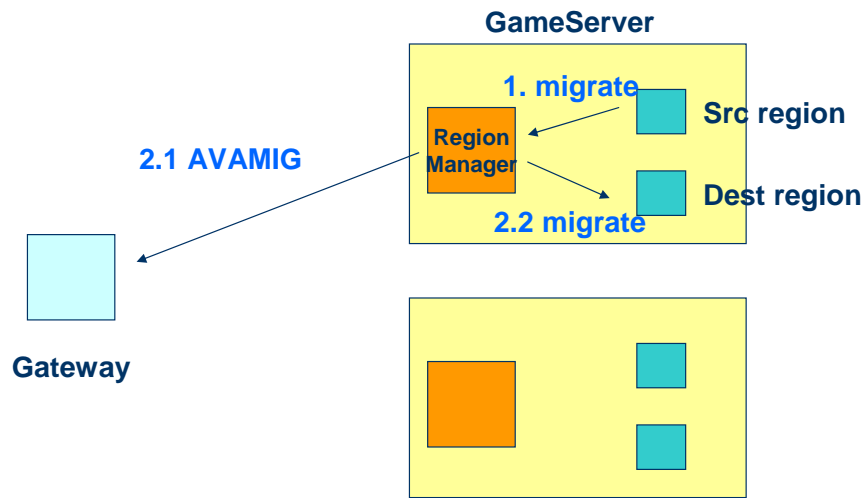


Figure 4-3: Avatar Migration, source region and destination region are in the same server.

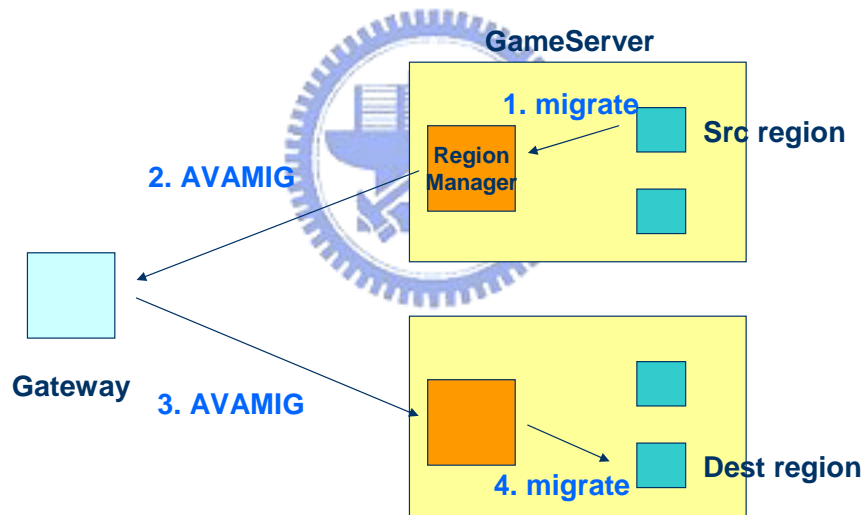


Figure 4-4: Avatar Migration, source region and destination are in different servers..

4.5. Login Issues

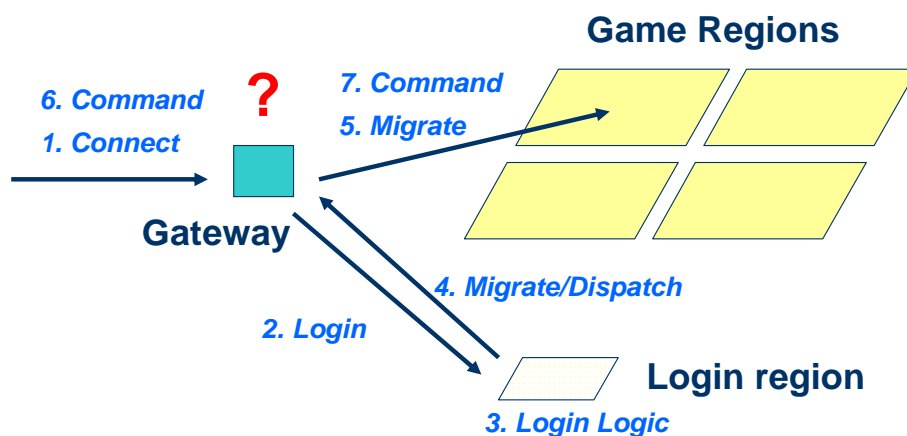


Figure 4-5: Login Region

Most MMOGs allow players to continue their adventure when they login. If one player was logout at the region A previously, the player should login at region A. It is not easy to implement in the MMOG middleware. The fact that which region the player logs in is a kind of game logic. Therefore, our gateways never know which server the first message to route before a player login.

To solve the question, we simply let developer to specify which region is login region. This region is responsible for processing the login logic, maybe query the database. Then, it migrates the avatar to the correct region. Thus, the login region can be considered as a kind of dispatcher. It processes the login logic and dispatches the avatar to correct region. The Figure 4-5 depicts the flow of login process.

4.6. Plugin Framework

In our sever implementation, we build up the server in a component-based architecture. We separate components into server scope components and region components scope. Besides, For the sake of flexibility, we allow developers to

provide their own plugin. The Figure 4-6 depicts the architecture. A server scope component, as the name implies, lives with the game server. It can be seen as a service on a server, such as timer, thread pool, and so on. A region scope component lives with the region. A region scope component always tightly couple with a region, such as NPC engine. There should be some dependencies between components. A region scope component can depend on a server scope component, but a server scope component can not depend on a region scope component.

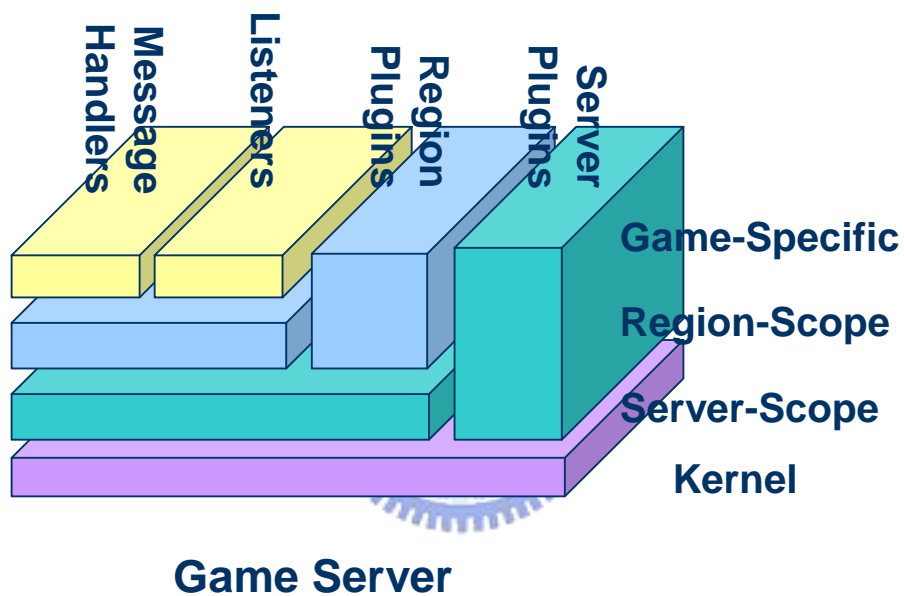


Figure 4-6: Server Component-based Framework

4.7. Region Migration

In order to share the load between game server clusters, a region can migrate from one server to another dynamically. We call it *Region Migration*. Here come the problems: the way to collaborate between several components, the way to send migrating data, the data to migrate, the way to migrate game-specific objects, the way to migrate region scope plugin, and the way for region scope components to bind

server scope services.

We use the component coordinator to coordinate the process of region migration. The migrating flow is explained in the Figure 4-7. When a coordinator broadcasts an *RMINIT* message to initiate a region migration, the destination sever opens a port and passively waits for the connection from source region. When the destination server is ready for accepting the socket from the source server, it sends *RMREADY* message to the coordinator. The coordinator forwards this message to source server. As source server receives this message, it connects to the destination and begins to send migration data as stream. As all data are sent successfully, the destination server sends the *RMCOMPLETE* to notify the completion of migration. Then the coordinator broadcasts this result to all participants. Otherwise, the *RMFAIL* message is sent to notify that some failure occurred when migrating.

However, it is important to know what data are sent when migrating. First, the system data of region should be migrated. Secondly, the game specific data which is created and managed by developers' code should be migrated. Thirdly, the region scope plugin should be migrated. The dependencies are crosscut between these objects. It will be a big problem to recover all the data and dependency in the new server. Fortunately, java serialization mechanism solves all these problems. Region in our system is designed as a *Serializable capsule*. The complex dependency can be serialized and restored to the origin form by mean of java serialization.

The other issue is that the region scope components and plugin may depend on server scope components resided on server. Therefore, before a region's migrating from a server, these region components should be unbind from server, as well as after a region's migrating to a new server, they should bind to the new server. Thus, the base class of region scope component provide `bind()` and `unbind()` abstract

method. Through these callback methods, the component itself can handle the logic of bind and unbind logic.

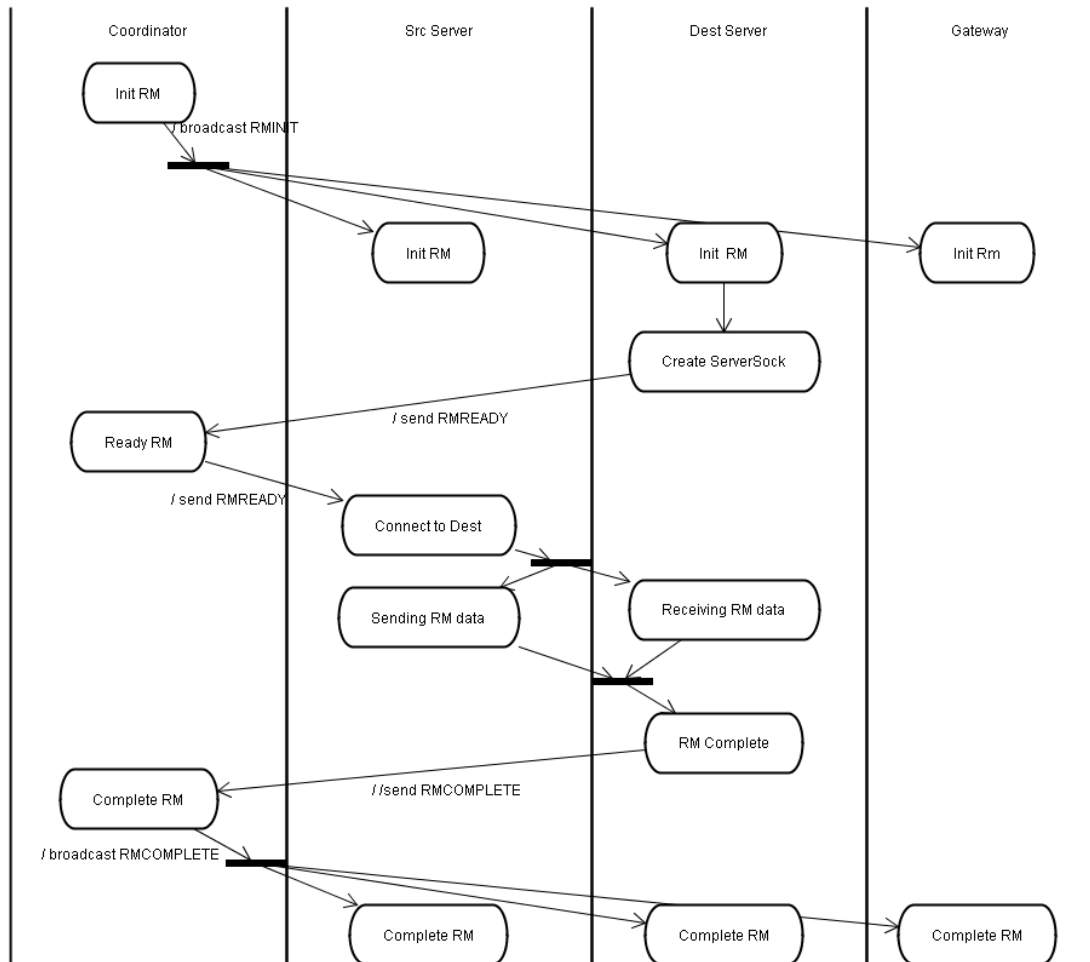


Figure 4-7: Region Migration

4.8. NPC Support

NPC is an abbreviation of Non Player Character. In most MMOG platform, NPCs reside in a special standalone component. It is a good idea to separate the game logic and AI logic, but it increases the communication overhead. In most cases, NPCs reside in only a region. It is more efficient to communicate between the game world if the NPC and virtual world in the same server. Thus, we tend to implement the NPC

feature on the game server.

The NPC feature is designed as a region scope component. It makes use of the timer component, which is a server scope component, to wake up NPCs in a fixed rate. The NPC logic is divided into two parts: AI logic and game logic. The NPC component is responsible to collaborate between these two parts. The discussion of design detail is postponed to Chapter 5.2.5.

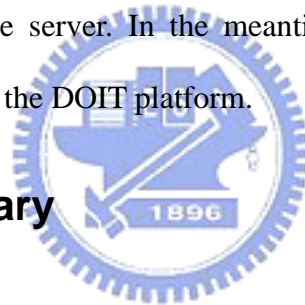


Chapter 5

Design of Programming Interface

In this chapter, we will shift the emphasis away from implementation of DOIT platform to use of it. In order to simplify the development, there are three layers of library introduced in this chapter. The first is the NetEngine library, which is the underlying network engine of DOIT platform. We also can make use of it at the client-side to communicate with DOIT platform. The second is the DOIT API. It provides the classes and interfaces to implement the game logic. The last one is the DOIT plugin framework. If we hope to extend the DOIT platform, we can customize a plugin and plug it onto the server. In the meantime, we will introduce how to develop a game application to the DOIT platform.

5.1. NetEngine Library



NetEngine is a message-oriented lightweight service. Namely, all data are sent as a concept of message. The Figure 5-1 presents the whole picture of NetEngine library. The NetEngine class is the core façade class. We also start to introduce this library from this class.

NetEngine can be used as client or server. To be a client, we call the `connect()` method with the specified address to open a connection. This method will return a Channel object. To be a server, we call the `listen()` method with the specified address. The NetEngine will listen to a specified port. It is necessary to register a listener implementing the interface `ChannelListener`. When a channel connects in, the NetEngine will call back the `channelConnected()` of

ChannelListener. In the same way, when a channel disconnects, the system will call back the channelDisconnected() of ChannelListener

To send a message, we make use of send() method of Channel. This method should pass in an object implementing Message class. For each type of message, developers should define a class extending the Message class. It must override the encode() and decode() abstract methods. The system will call back these two methods in order to encode/decode the message data to/from byte buffers. In addition, we recommend that these message classes adhere the JavaBeans [17] convention, that is, if a class has a property named id, it should provides the setId() and getId() methods in this class. It will make it easier to get use of the message objects.

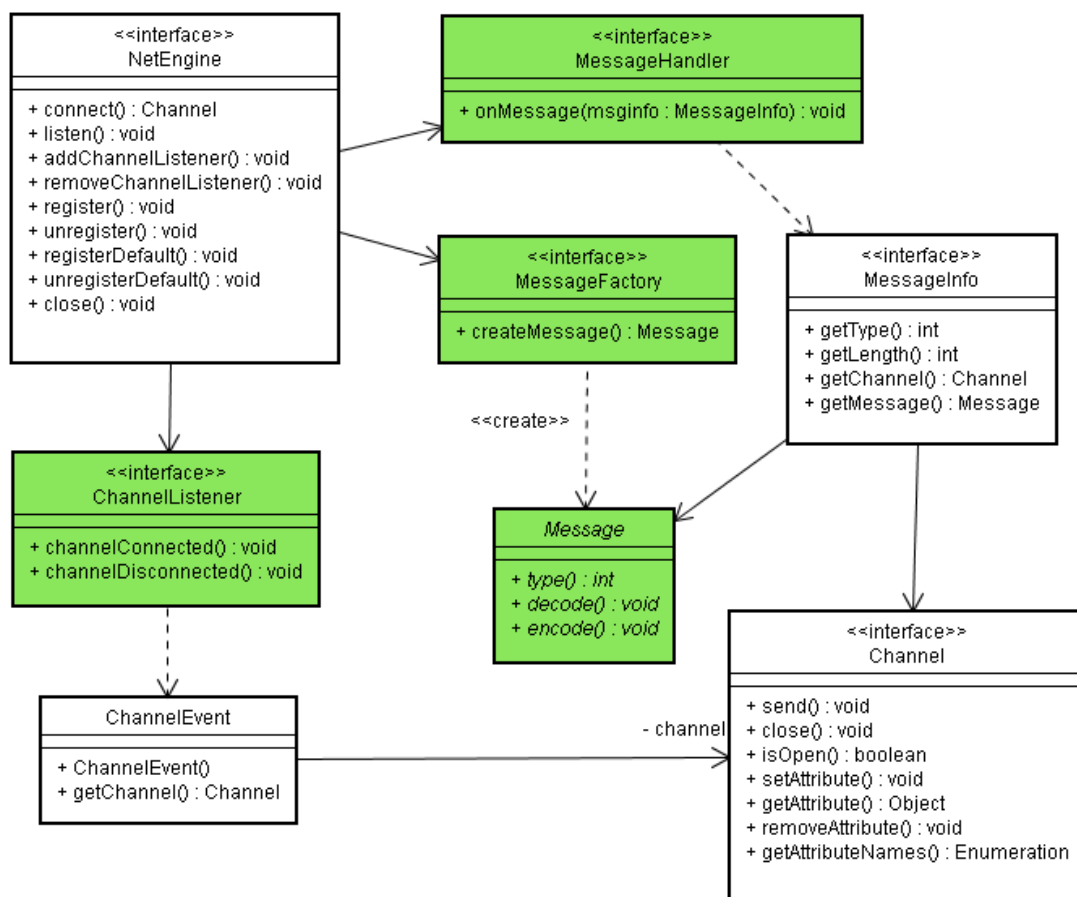


Figure 5-1: Class diagram of NetEngine library

To receive a message, we should first register a message handler and a message factory for a given type of message. The factory should implement the `MessageFactory` class and overrides the `createMessage()` method. The handler should implement the `MessageHandler` class and overrides the `onMessage()` method. When a message comes, the `NetEngine` will first analyze the type of message from its header. Then it creates the message from the corresponding message factory and call the `decode()` method such that it can read the message from binary stream. Subsequently, it dispatches the message to the corresponding message handler.

`Channel` is abstraction of connection. We can send a message by `send()` method. In addition, the channel can be seen as a session to the remote peer. We can store some attributes associating to this session. `NetEngine` supports two channel establishment methods: active connecting by `connect` method and passively listening by `listen`. And when a connection is established, we can get the connection handle `Channel` by `ChannelListener`.

The following is some example to use `NetEngine`.

Server-side Example:

```
import java.net.InetSocketAddress;
import mmog.net.*;
import mmog.net.tcp.TCPNetEngine;
//...
NetEngine engine = new TCPNetEngine();
InetSocketAddress sockaddr =
    new InetSocketAddress(13579);
engine.register(MyMessage.TYPE,
               new MyMessageFactory(),
               new MyMessageHandler());
//...
engine.listen(sockaddr);
```

Client-side Example:

```
import java.net.InetSocketAddress;
import mmog.net.*;
import mmog.net.tcp.TCPNetEngine;
//...
NetEngine engine = new TCPNetEngine();
InetSocketAddress sockaddr =
    new InetSocketAddress("localhost", 13579);
engine.register(MyMessage.TYPE,
               new MyMessageFactory(),
               new MyMessageHandler());
Channel channel = engine.connect(sockaddr);
MyMessage msg = new MyMessage();
channel.send(msg);
//...
```

5.2. DOIT API



DOIT API provides all the classes and interfaces to develop a game on the DOIT platform. In this API, developers can totally develop the game in view of game without the complexity of networking, distributed system, and multi-thread issues. The DOIT API is somewhat similar to NetEngine because it also have the message-oriented property. But the DOIT API is specially designed for game development. It will provide some features aimed at game. The Figure 5-2 is the class diagram of the DOIT API.

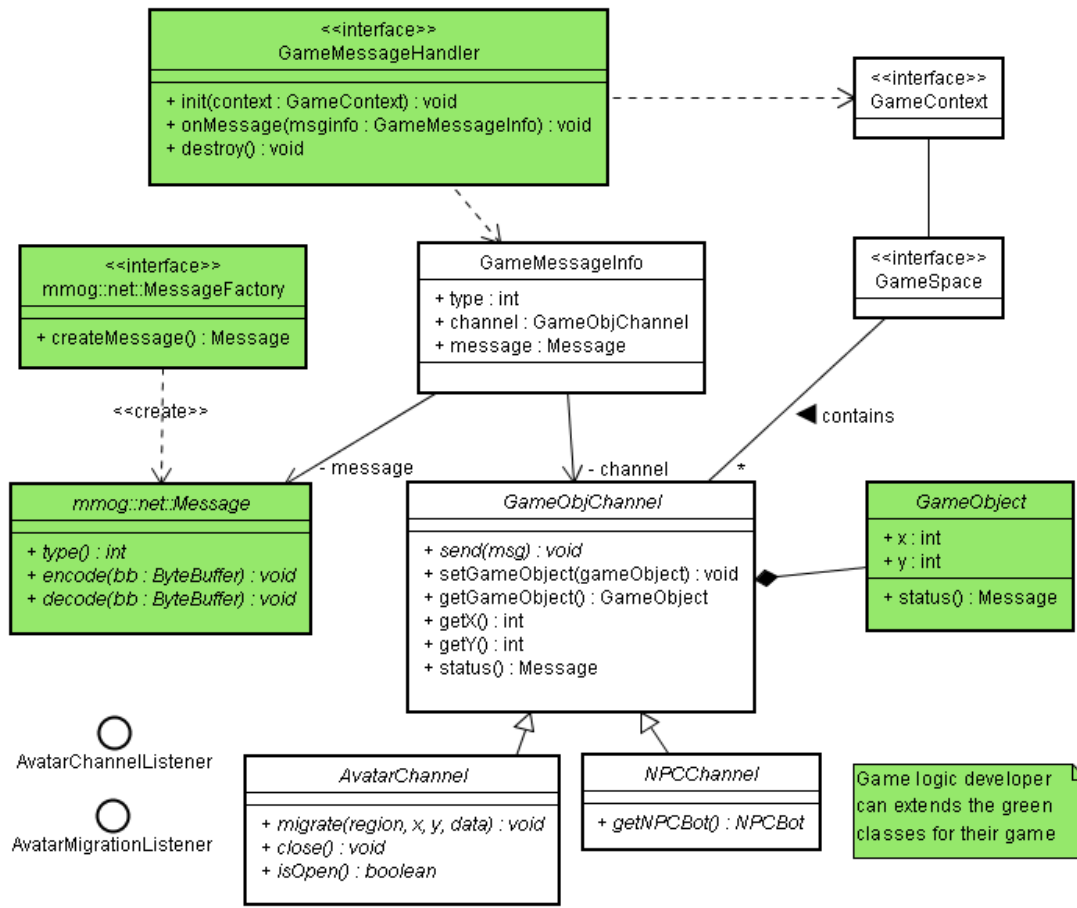


Figure 5-2: Class diagram of the DOIT API

5.2.1. Game Protocol and Game Logic

Our platform inherits from NetEngine. So we define the game protocol in the same manner as using NetEngine library, that is, we should define a message class extending the Message class for each type of message. Similarly, we also need to provide a message factory for each message class. To write a game logic, we should provide a message handler extending GameMessageHandler for each type of message. This design is inspired from Java Servlet API [18]. The developers should override the onMessage () method to handle the message. When a message comes, the system will call back the onMessage () method. It will pass in a

GameMessageInfo object. From this object, we can get the message, message type, and game object channel. In particular, the game object channel represents the source of the message to be handled.

The game object channel which is an instance of GameObjChannel class is the similar concept of Channel in NetEngine library. The difference is that the game object channel represents the channel to communicate a game object. In the current version of DOIT API, we provide two kind of GameObjChannel. They are AvatarChannel and NPCChannel respectively. As the name implied, the AvatarChannel is a channel to an avatar. We can send a message to a client indirectly through this class. As for the NPCChannel, it is a channel to a NPC, we can send a message to the NPC bot through this class.

A game object channel can associate with a game object which extends the GameObject class. The GameObject class is the base class of all game objects. That is, developers can customize a game object by extending the GameObject class. The GameObjChannel delegate some methods call to his associating game object. They include getX(), getY(), and status() these three methods. Certainly, the getX() and getY() are the accessor methods to get the location information. As for the status() method, it return a Message instance according to its current state. They will be useful for GameSpace and GameSpaceUtil class.

Therefore, we can conclude the basic flow to design a game.

1. Defines the game protocol.
2. Define the message classes and corresponding message factory classes.
3. Define the game objects.
4. Implements the message handlers.

5.2.2.GameContext and GameSpace

In `GameMessageHandler`, we can receive a message for a given type, process message, and send updates back to the game object channel. But it is only possible to send updates to original game objects. We need more functionality to deal with game objects. In this subsection, we introduce the two classes, `GameContext` and `GameSpace`.

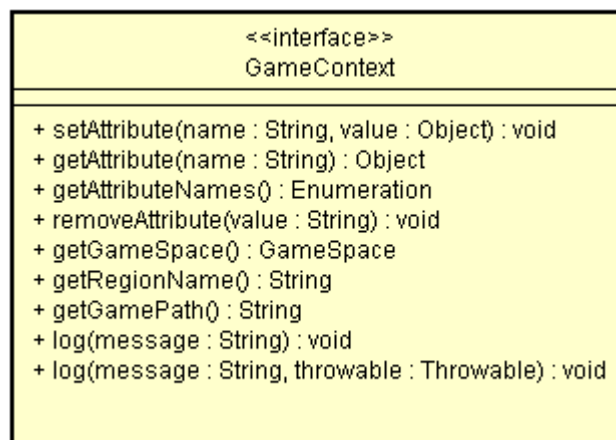


Figure 5-3: `GameContext`

`GameContext` represents the context of the game. The concept is similar to `ServletContext` in the Servlet API. But the game context only associates with the current region. We can store and retrieve attributes to and form the context. Besides, we can also obtain some information of the game. The `GameContext` instance can be obtained from the `init()` method of `MessageHandler`, `AvatarChannelListener`, and `AvatarMigrationListener`. The Figure 5-3 is the class information of `GameContext`.

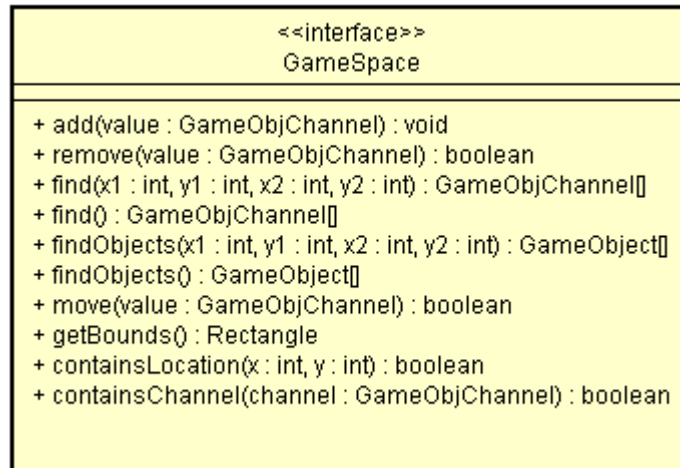


Figure 5-4: GameSpace

GameSpace can be seen as a data structure of the virtual worlds. We can *add*, *remove*, *move*, and *find* game objects from the GameSpace. The element type of GameSpace is GameObjChannel. Before adding a GameObjChannel to a GameSpace, we should first associate it with a game object. It is because GameSpace set and retrieve the location information from `setX()/getX()` and `setY()/getY()` methods of GameObjChannel. These methods delegate the implementation to the same methods of GameObj associated by the GameObjChannel. Therefore, the GameSpace prohibit a GameObjChannel without GameObj from added to it. The size of GameSpace is specified in the game descriptor (see chapter 5.3). The GameSpace instance can be obtained from the GameContext.

GameSpaceUtil is a class let us use GameSpace more conveniently. We usually need to send a message to surroundings of a game object. We can call the `sendMsgToSurroundings()` method. It will get the GameObjChannels from the given range and send the message respectively. Likewise, if we need to receive updates from surroundings, we can call the `recvUpdtFromSurroundings()`

method. It will get the `GameObjChannels` from the given range, get the updates from `status()` method of `GameObjChannels`, and send them to the given channel.

5.2.3. AvatarChannelListener

We can have greater control over the lifecycle of avatar by means of `AvatarChannelListener`. A listener implementing `AvatarChannelListener` can receive the events when an avatar connect or disconnect. Even though the client disconnected abnormally, the disconnect event also raise correctly.

5.2.4. Avatar Migration

In DOIT API, we use `migrate()` method of `AvatarChannel` to migrate a avatar to another. In this method, we should specify the region to migrate, the location, and the migration data. The migration data is type of byte array. We can put arbitrary data in it. Besides, the developers may implement the interface `AvatarMigrationListener` to get the event about some avatars' migration.

5.2.5. NPC

NPC (Non Player Character) is also supported in DOIT Platform. In our design, we separate the game logic and AI logic into two parts, `NPCBot` and `GameMessageHandler` respectively. They communicate with each other by means of sending messages. The game server is responsible for relaying messages between them (see Figure 5-5). The advantage is that we can process game logic in the same way as processing game logic of avatars. Moreover, it prevents codes of different

purpose from mixing together. To write NPC AI, we should write a class implementing `NPCBots`. There are three methods that should be overridden, `onInit()`, `onUpdate()`, `onTimer()`. `onInit()` is called when a NPC is created. `onUpdate()` is called when a message comes. The message is sent from his corresponding `NPCChannel`. `onTimer()` is called periodically by `GameServer`. The timer interval is defined in the description file.

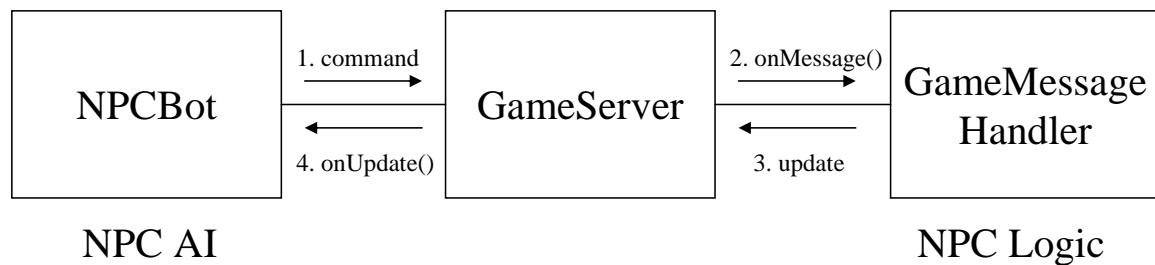


Figure 5-5: NPC

5.2.6. Mapping the features to the API

In Chapter 4, we explained the features of the DOIT platform. It is interesting to map these to the API. Table 5-1 lists the features mapping to the API. We can find that the underlying complexity is totally hidden under the easy API.

Features	Related API	Notes
Customized Protocol	Extends <code>Message</code> , Extends <code>MessageFactory</code>	Use <code>encode()</code> and <code>decode()</code> callback methods to turn underlying protocol to object form, and vice versa
Customized game objects	Extends <code>GameObject</code>	<code>GameObject</code> implements <code>java.io.Serializable</code>
Game Logic	Extends <code>MessageHandler</code>	Overrides <code>onMessage()</code> method to handle message.

Channel Event Forwarding	Implements AvatarChannelListener	Overrides avatarConnect()/ avatarDisconnected()
Avatar Migration	Implements AvatarMigrationListener, Uses AvatarChannel.migrate()	Overrides avatarMigrateIn()/ avatarMigrateOut()
Region Migration	Transparent	
NPC Support	extends NPCBot, Use NPCChannel	Overrides onInit()/ onTimeout()/ onUpdate()
Virtual World	Uses GameSpace	Provides add()/ remove()/ find()/ move() methods
Game Context	Uses GameContext	Provides put/ get attributes

Table 5-1: API for our MMOG Middleware

5.3. Game Deployment

All the game logic run on the DOIT platform, so we must deploy them onto the DOIT platform. All game logics are wrapped as a game application. It should put at the “game” directory in the DOIT platform. In this directory, we put the classes in the “classes” subdirectory, and we put all the libraries in the “libs” subdirectory. Besides, there are two files necessary in the game application: mmog.xml and vwlogic.proerties. In mmog.xml, it describes which components are in the game

platform and which regions are in the virtual world and the initial assignment of regions. The Figure 5-6 is an example. For the sake of simplicity, we don't describe it in detail.

```
<?xml version="1.0"?>
<!DOCTYPE mmog SYSTEM "mmog.dtd">
<mmog>
  <components>
    <server name="server1" host="localhost" port="8765"/>
    <server name="server2" host="localhost" port="8764"/>
    <gateway name="gateway1" host="localhost" port="5678"/>
    <gateway name="gateway2" host="localhost" port="5679"/>
    <coordinator host="localhost" port="8778"/>
  </components>
  <maps>
    <map name="map0">
      <region name="regionlogin" x1="0" y1="0" x2="0" y2="0"
login="true"/>
    </map>
    <map name="map1">
      <region name="region1" x1="0" y1="0" x2="100" y2="100"/>
      <region name="region2" x1="100" y1="0" x2="200" y2="100"/>
      <region name="region3" x1="0" y1="100" x2="100" y2="200"/>
      <region name="region4" x1="100" y1="100" x2="200" y2="200"/>
    </map>
  </maps>
  <assignments>
    <assignment region-ref="regionlogin" server-ref="server1" />
    <assignment region-ref="region1" server-ref="server1" />
    <assignment region-ref="region2" server-ref="server1" />
    <assignment region-ref="region3" server-ref="server2" />
    <assignment region-ref="region4" server-ref="server2" />
  </assignments>
</mmog>
```

Figure 5-6: mmog.xml

Concerning the vwlogic.properties, it lists the classes of game logics, including

GameMessageHandler, MessageFactory, AvatarChannelListener, and AvatarMigrationListener. Especially, the GameMessageHandler and MessageFactory must associate with a message type. The Figure 5-7 is a simple example of vwlogic.properties

```
# 1=MessageFactory
# 2=MessageHandler
# 3=AvatarChannelListener
# 5=AvatarMigrationListener
1=cis.game.common.message.LoginMessageFactory/0x01
2=cis.game.server.handler.LoginMessageHandler/0x01
1=cis.game.common.message.MoveMessageFactory/0x03
2=cis.game.server.handler.MoveMessageHandler/0x03
1=cis.game.common.message.NpcBornMessageFactory/0x06
2=cis.game.server.handler.NpcBornMessageHandler/0x06
1=cis.game.common.message.LogoutMessageFactory/0xFF
2=cis.game.server.handler.LogoutMessageHandler/0xFF
3=cis.game.server.listener.MyAvatarListener/0x00
5=cis.game.server.listener.MyMigrationListener/0x00
```

Figure 5-7: vwlogic.properties

Optionly, we may provide the npc.properties if we want to provide NPCs in our game. In this file, we must specify the NPCBot classes of NPCs, the number of NPCs for a given type of NPC in a given region. The Figure 5-8 is a simple example of npc.properties

```
mmog.npc.bot.good=cis.game.server.npc.GoodNPCBot
mmog.npc.bot.bad=cis.game.server.npc.BadNPCBot
mmog.npc.region.region1=good,100
mmog.npc.region.region1=bad,50
mmog.npc.region.region2=good,50
mmog.npc.region.region2=bad,100
```

Figure 5-8: npc.properties

5.4. Plugins Framework

In DOIT platform, we can plug service components to extend the functionality. As mention in Chapter 4.6 , plugins are divided server scope plugins and region scope plugins. To write a server scope plugin, we should write a class extending `mmog.server.ServerScopePlugin`. When the server starts up, the server will call back the `init()` method of `ServerScopePlugin`. The system will pass in the `ServerContext` and plugin initial parameters to this method. In `ServerContext`, we can get the information of server and look up other server components in this server.

As for designing a region scope plugin, we should provide a class extending `mmog.server.RegionScopePlugin`. Similarly, when a region is initiated, the server will call back the `init()` method of `RegionScopePlugin`. The difference is that there is no `RegionContext` parameter passed in the `init()` method of `RegionScopePlugin`. It will be postponed to `bind()` method. Region scope plugins live with a region. When a region is bound to a server, all the region scope plugins in this region will have the `bind()` called. Relatively, when a region is unbound from a server, all the region scope plugins in this region will have the `unbound` called. It must be noted that `init()` is only called once throughout the lifecycle of game platform, and however, `bind()/unbind()` is called once whenever a region is migrated in/out. Moreover, region scope plugins should implement `java.io.Serializable`. Because region scope plugins migrate accompany with regions, the plugin should be the serialization form. Note to add the `transient` keyword to the field which is not able to serialize. We can restore this field in the `bind()` method.

A plugin should be pack as a jar file and put in the “plugins” directory of DOIT platform. For each plugin, we should provide a plugin definition file. In this file, we can define the type classname of plugin, plugin type, plugin initial parameters. The Figure 5-9 is an example of plugin definition file.

```
# MyPlugin.properties
# put plugins config here
mmog.plugin.server.test = hello.MyPlugin
mmog.plugin.server.test.foo = bar
mmog.plugin.server.test.foo2 = bar2
```

Figure 5-9: MyPlugin.properties



Chapter 6 Experiment and Evaluation

We start to evaluate our platform in this chapter. To evaluate the performance of our platform, we experimented in three rounds. In the first round, we hope to realize the performance of the NetEngine. We implement a simple echo server by using NetEngine. In the second round, in the same way, we implement a simple echo server. The difference is that the echo server is implemented with client-gateway-server architecture. We provide a simple EchoMessageHandler and deploy it on our platform. The purpose is that we hope to know the performance baseline of our game platform. In the last round, we simulate the real online game. We implement the most significant game logic, login and move. Besides, the virtual world is divided into four regions. So an avatar may migrate from one region to another. Through this experiment, we hope to observe the performance of the real game.

6.1.Round 1: NetEngine Performance

In the first round, we tried to find out the limit of the NetEngine.

6.1.1.Hardware Configuration

For the hardware configuration, we use one machine as the server and 10 machines as clients. The Table 6-1 describes the detail configuration.

Usage	Number	Configuration
Server	1	P4 2.4GHz CPU with 1MB ram
Client	10	P4 1.6~2.4GHz CPU with 512MB ram

Table 6-1: Hardware configuration of round 1

6.1.2. Software Configuration

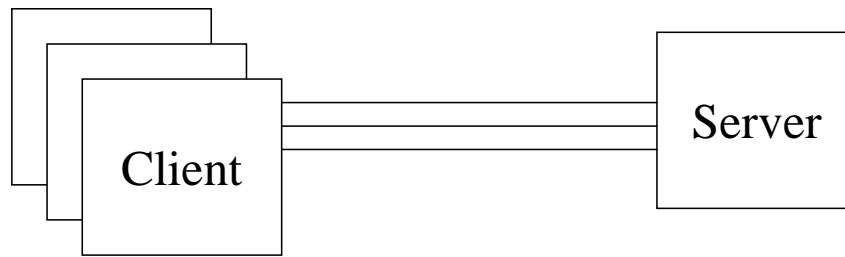


Figure 6-1: Communication architecture of round 1

In order to perform stress test, we designed a virtual client generator program to simulate multiple players in a single machine. We run this program in 10 machines simultaneously, and we increased the number of clients by 50 for each run each machine. Namely, the number of client is increased by 500 for each run. We totally test 8 runs, and the client number scale from 500 to 4000. Each client sent one message per second and lasted for 10 minutes. Clients and the server run on the same LAN and they are connected with 100 Mbit Ethernet (see Figure 6-1).

The server performed simple echo action. Such that, when the server was received a message, it simply sent back this message to client. The echo message contains a 4-bytes serial number field, which is used to identify which message is sent back. For each test, we run for 10 minutes and use the data in the medium 8 minutes as the effective data. We analyzed the log data and evaluated the average response time and standard deviation for each test.

6.1.3. Experiment Result

Client number	Average Response Time	Standard Deviation
500	0.3	3
1000	0.99	5
1500	2.68	17
2000	4.06	26
2500	9.34	75
3000	7.75	30
3500	24.9	84
4000	55.11	534

Table 6-2: Experiment result of round 1

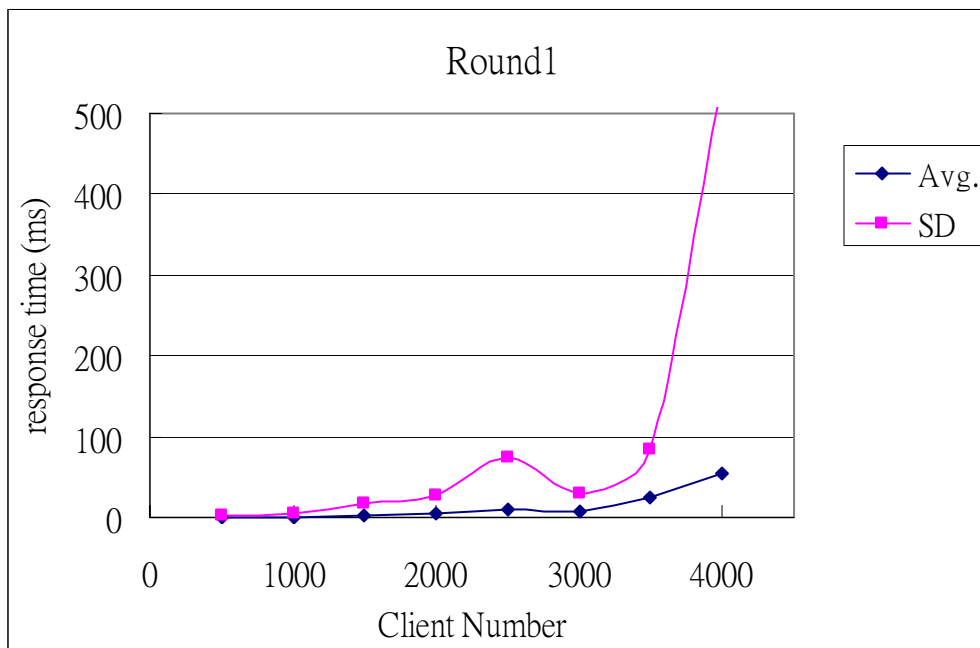


Figure 6-2: Experiment result of round1

6.1.4. Evaluation

Through this experiment result, we discovered that it got excellent performance under 3000 clients concurrently. The average response time is less than 10 ms and the


standard deviation is less than 75ms. This performance is very ideal for a game environment. When the client number reaches to 4000, the average response time is also less than 100 ms, but the deviation is more than 500 ms. It begins to have unstable behavior.

6.2. Round 2: Performance baseline of DOIT Platform

In our platform, we adopt the client-gateway-server architecture. It is desirable to realize the performance baseline under this architecture. In this round, we also use the simple echo program to experiment with the performance baseline of client-gateway-server architecture.

6.2.1. Hardware Configuration

In this round, we prepared one more machine to run as a gateway. This machine has the same configuration as the server.



Usage	Number	Configuration
Server	1	P4 2.4GHz CPU with 1MB ram
Gateway	1	P4 2.4GHz CPU with 1MB ram

Table 6-3: Hardware configuration. of round 2

6.2.2. Software Configuration

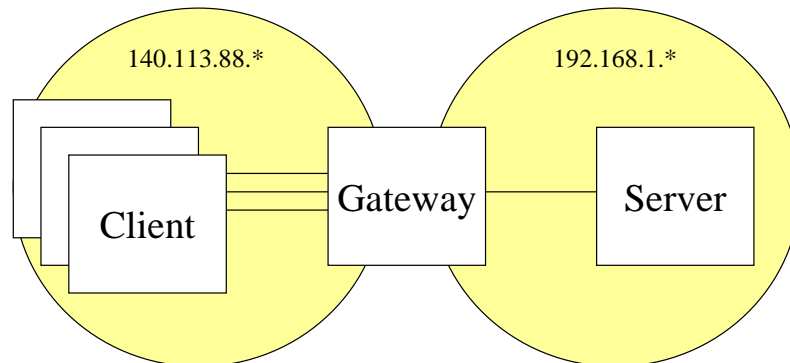


Figure 6-3: Communication architecture of round 2

Similarly, we designed a client generator program to perform stress test. The client number also scaled from 500 to 4000 between 8 runs. The most different point is that the test environment was separated into 2 LAN. Clients connected to the gateway in a subnet and the gateway connected to server in another one (see Figure 6-3). Therefore, the traffic from client to gateways didn't inference the one between the gateway and the server.

Concerning the test program, we designed an echo server as well. The difference is that the echo semantic was implemented as a game logic and we deployed it to the game platform. So this program can be considered as the simplest game and the performance can be considered as the performance baseline. Similarly, there were 8 runs and each run lasted for ten minutes.

6.2.3.Experiment Result

Client number	Average Response Time	Standard Deviation
500	17.87	43.13
1000	13.09	27.20
1500	12.18	30.28
2000	18.27	34.21
2500	17.83	37.63
3000	20.17	45.72
3500	27.66	121.29
4000	33.22	135.09

Table 6-4: Experiment result of round 2

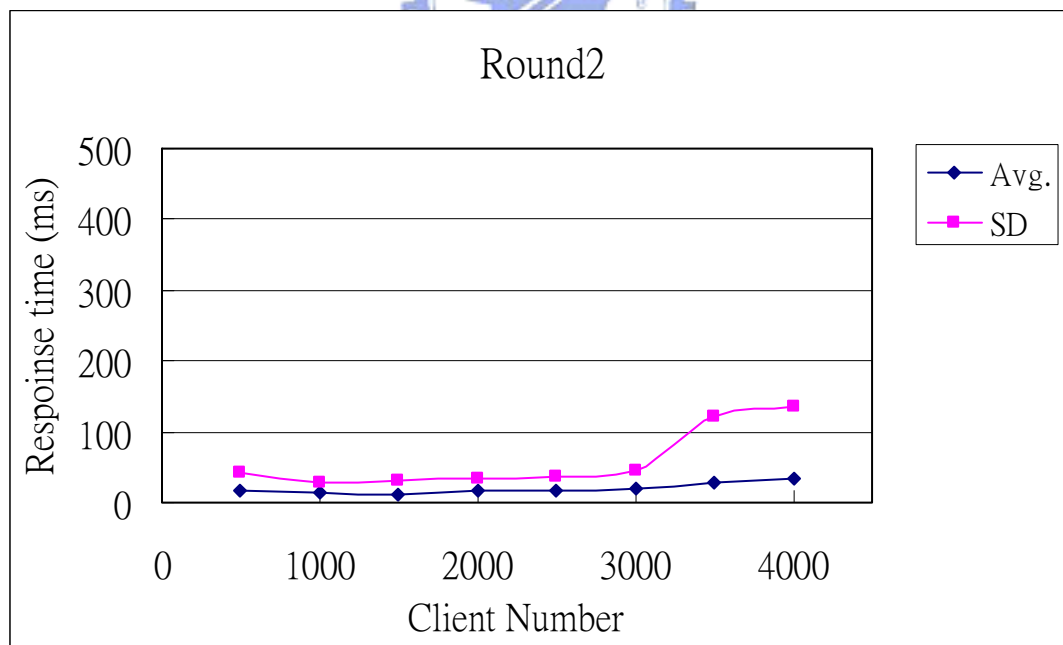


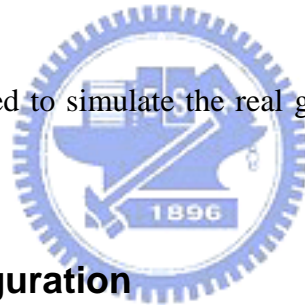
Figure 6-4: Experiment result of round 2

6.2.4.Evaluation

We observe that the result was very similar to the result of round 1. Although the average response was slightly larger than that of round 1, it was also less than 100 even if the client number reached to 4000. The standard deviation also had the level near 100 ms. It is interesting to note that the average response time in 500 clients was higher than that in 1000 clients. The reason could be caused that it fully exploited the benefit of message aggregation in test of 1000 clients such that it got the better performance than that in 500 clients. The similar result also appeared in round 3.

6.3.Round 3: Simulate a Real Game

In the last round, we tried to simulate the real game environment and observed the performance.



6.3.1.Hardware Configuration

We used two computers for servers, two for gateways, ten for clients. The detail configuration is list as follow.

Usage	Number	Configuration
Server	2	P4 2.4GHz CPU with 1MB ram
Gateway	2	P4 2.4GHz CPU with 1MB ram
Client	10	P4 1.6~2.4GHz CPU with 512MB ram

Table 6-5: Hardware Configuration of round 3

6.3.2. Software Configuration

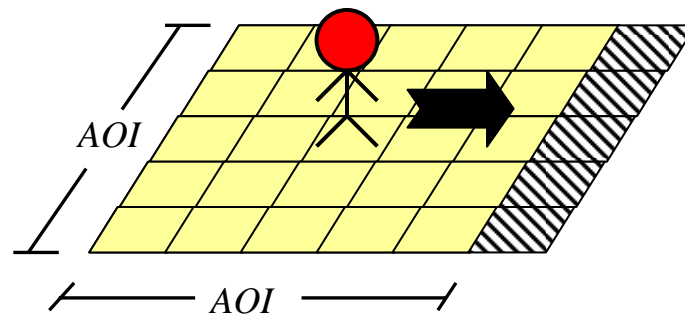


Figure 6-5: The move logic

To simulate the real game world, we implemented the most significant game logic, avatar movement, in the test program. However, the different implementation of the move logic will affect the result obviously. So we should make more effort to describe the implementation detail of move logic. We should first define the term AOI (Area of Interest) as the area in which all events are interesting to a given game object. This game object is interested in all the game objects in AOI. Then, we explain the move logic by mean of the Figure 6-5. Assume that one avatar moves right, as the figure indicates, we should send the player update message to all the game objects inside the AOI and the region of the oblique line. This is because the game objects in the AOI of new location should be interested in the new state of this avatar. The game objects in the leftmost of the AOI of the old location should receive the player update to indicate that this avatar was move out of their AOI. In addition to send updates to the surrounding objects, the avatar also needs to extend his eye sight. So the states of the game objects in the oblique line area should also be sent back to avatar.

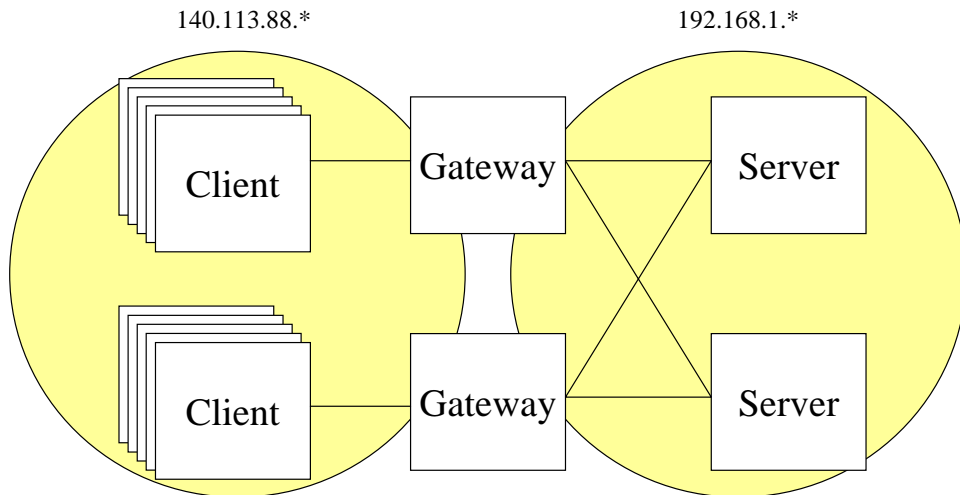


Figure 6-6: Communication architecture of round 3

The Figure 6-6 depicts the communication architecture of this round. The virtual world was divided into 4 equal size regions plus a login region. They were deployed to 2 game servers. In addition, we provided two gateways for clients to connect in. Similarly, we put the two servers in the private network. The gateway was responsible for routing messages between internal and external network. Due to multiple regions, it is possible that one avatar will migrate from one region to another, the avatar migration also be implemented in this round.

Concerning the test program, there were also 10 machines running the client generator program. We separated them into two groups. Machines in the same group connected to the same gateway. There were 8 runs in this round. The number of clients is increase with the number of 50 in each client generator program. Therefore, the gateway had 2000 clients connected concurrent at most, and the game platform had 4000 clients in the virtual world concurrently. Furthermore, we performed different tests for different map size and AOI size. The map had 500 x 500 and 1000 x 1000 two different sizes, and AOI had 9 x 9 and 16 x 16 two different sizes. We hope to realize the performance in different environment. Besides, in each test, we pick the

updates with avatar migration to do further analysis.

6.3.3. Experiment Result

Average Response Time - Total				
	500x500 9x9	1000x1000 9x9	500x500 16x16	1000x1000 16x16
500	20.13	18.69	19.84	24.79
1000	20.85	19.43	24.08	17.05
1500	25.24	15.23	24.83	17.88
2000	20.94	12.18	31.29	15.90
2500	19.37	13.28	50.89	29.08
3000	26.12	18.36	235.71	20.84
3500	63.09	19.69	489.58	43.57
4000	124.69	25.51	4186.74	52.19

Table 6-6: Average Response Time - Total

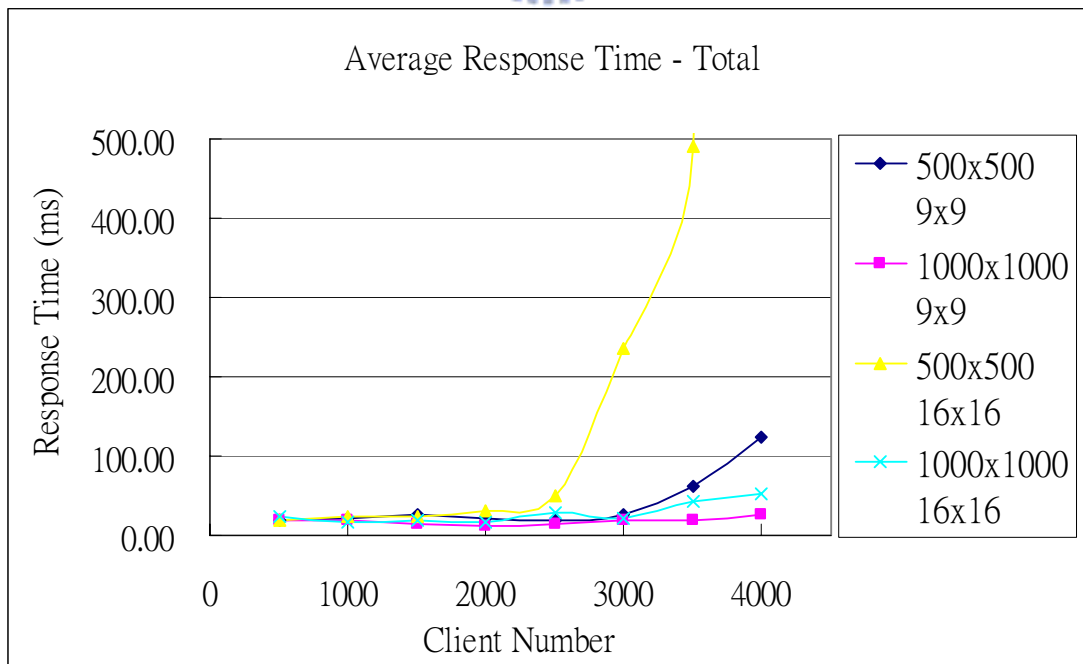


Figure 6-7: Average response time - total

Standard Deviation - Total				
	500x500 9x9	1000x1000 9x9	500x500 16x16	1000x1000 16x16
500	53.43	45.44	52.06	60.26
1000	54.95	38.69	55.53	40.60
1500	59.73	39.51	57.73	52.84
2000	52.91	32.74	73.60	39.06
2500	52.19	35.50	175.10	54.50
3000	73.72	42.80	829.66	155.52
3500	269.75	51.34	1013.89	136.74
4000	586.98	97.16	5709.61	245.59

Table 6-7: standard deviation – total

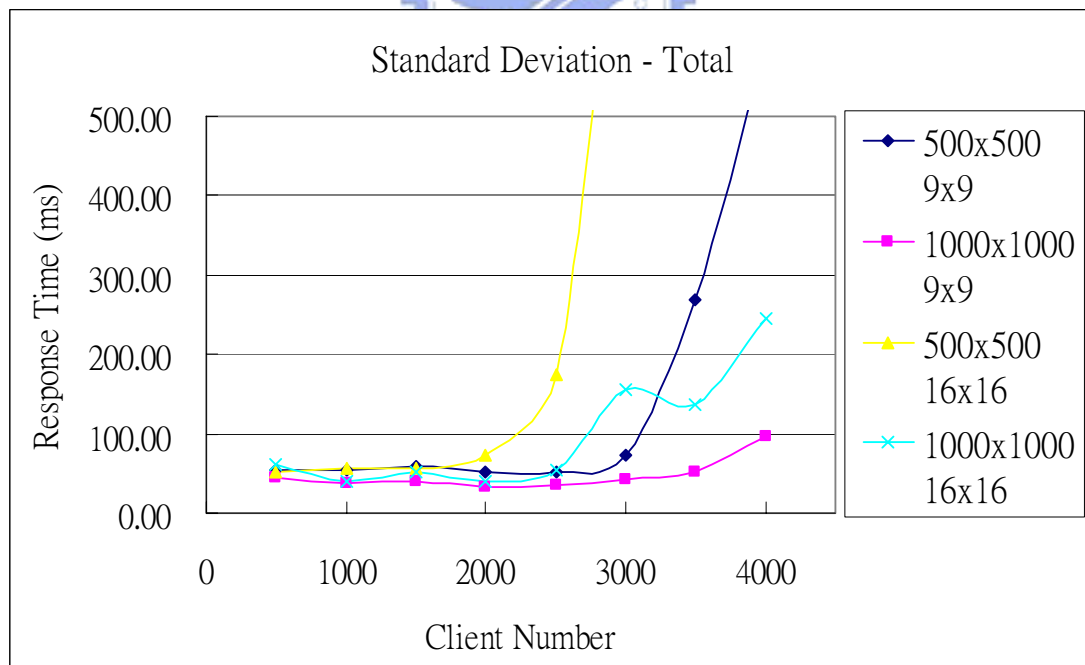


Figure 6-8: Standard deviation total

Average Response Time - Migration				
	500x500 9x9	1000x1000 9x9	500x500 16x16	1000x1000 16x16
500	43.79	27.55	35.55	47.49
1000	30.00	31.34	39.34	32.82
1500	29.34	26.70	41.59	23.08
2000	33.21	20.91	40.01	32.29
2500	27.16	22.70	74.58	39.25
3000	36.44	30.84	200.78	30.26
3500	71.17	27.21	623.84	58.70
4000	115.99	32.11	5184.81	62.83

Table 6-8: Average response time – migration

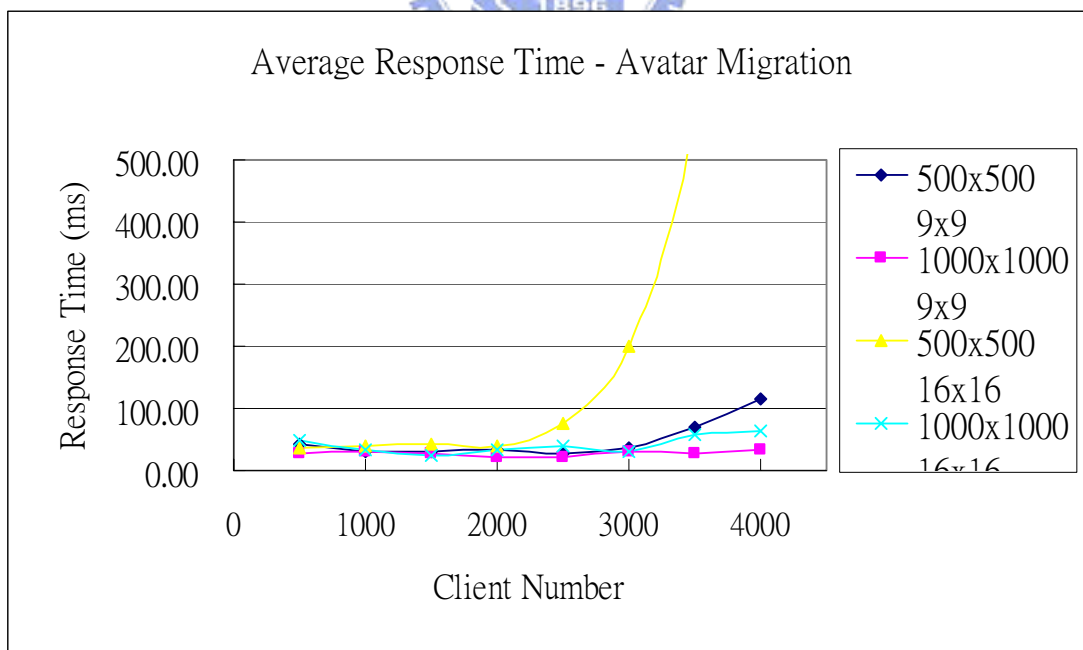


Figure 6-9: Average response time – migration

Standard Deviation - Migration				
	500x500 9x9	1000x1000 9x9	500x500 16x16	1000x1000 16x16
500	74.11	46.75	68.77	85.84
1000	58.26	52.23	70.18	61.07
1500	58.67	52.20	79.08	53.59
2000	65.85	54.53	78.22	62.56
2500	58.61	46.79	378.09	62.41
3000	74.18	58.76	911.63	100.19
3500	194.15	51.05	916.62	144.49
4000	479.83	92.61	6277.19	211.02

Table 6-9: Standard Deviation - Migration

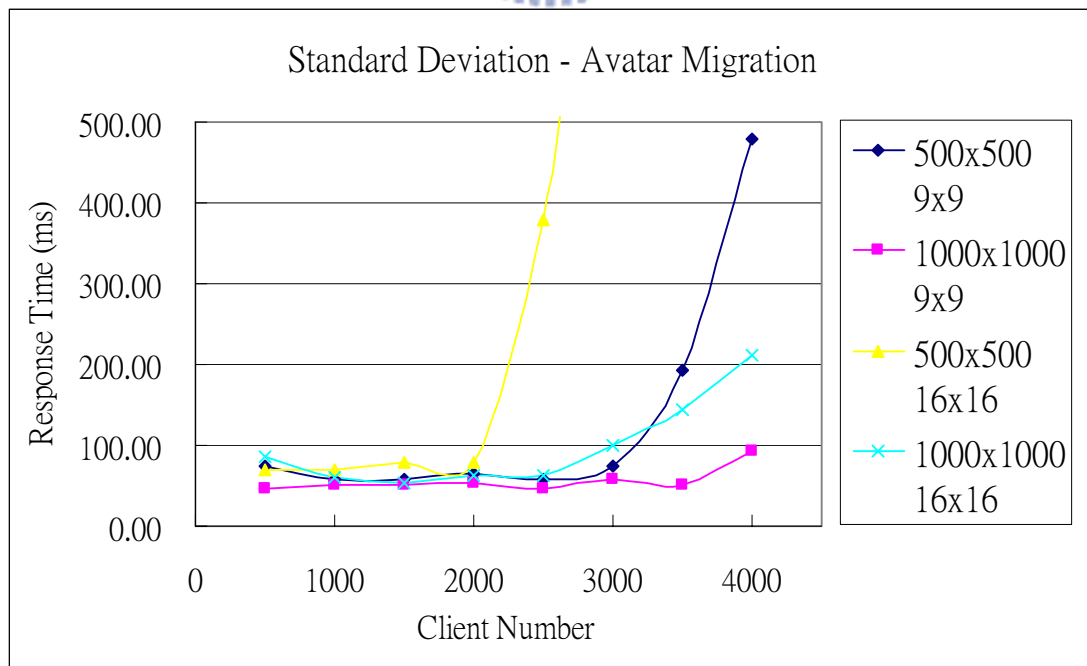
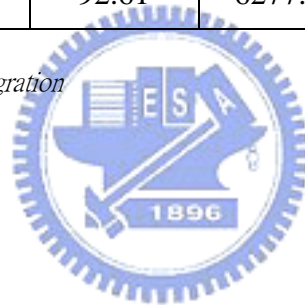


Figure 6-10: Standard Deviation - Migration

	500x500 9x9		1000x1000 9x9		500x500 16x16		1000x1000 16x16	
	Gateway	Server	Gateway	Server	Gateway	Server	Gateway	Server
500	6	2	2	2	2	2	5	2
1000	24	3	12	2	6	5	11	4
1500	32	9	17	3	10	5	22	6
2000	36	9	24	9	22	11	25	9
2500	40	10	38	10	50	14	43	10
3000	78	10	58	10	90	18	66	17
3500	95	15	86	13	98	25	88	25
4000	98	36	89	17	100	33	90	28

Table 6-10: CPU load

Total Clients	Gateway Clients	500x500 9x9		1000x1000 9x9		500x500 16x16		1000x1000 16x16	
		Prediction	Actual	Prediction	Actual	Prediction	Actual	Prediction	Actual
500	250	300	330	262.5	275	378	397	282	298
1000	500	700	925	550	578	1012	1128	628	670
1500	750	1200	1691	862.5	906	1902	2086	1038	1076
2000	1000	1800	1962	1200	1256	3048	3350	1512	1644
2500	1250	2500	2817	1562.5	1655	4450	4853	2050	2182
3000	1500	3300	3663	1950	2070	6108	6415	2652	2822
3500	1750	4200	4679	2362.5	2540	8022	7849	3318	3538
4000	2000	5200	5782	2800	3018	10192	10826	4048	4312

Table 6-11: The amount of commands forwarded per second in a gateway

	500x500 9x9	1000x1000 9x9	500x500 16x16	1000x1000 16x16
500	580	525	647	548
1000	1425	1078	1628	1170
1500	2441	1656	2836	1826
2000	2962	2256	4350	2644
2500	4067	2905	6103	3432
3000	5163	3570	7915	4322
3500	6429	4290	9599	5288
4000	7782	5018	12826	6312

Table 6-12: The total amount of messages forwarded per second in a gateway

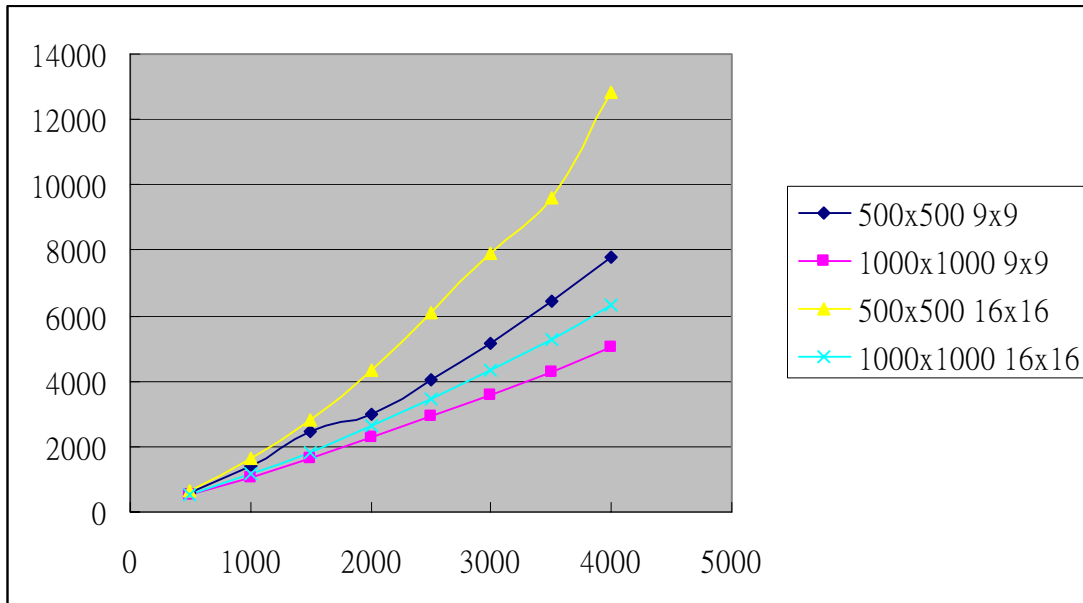


Figure 6-11: The total amount of messages forwarded per second in a gateway

6.3.4.Evaluation

In Figure 6-7 and Figure 6-8 we can observe that different map size and different AOI size will lead to obviously different result. In all the tests, we find that the data of map size 500 x 500 and AOI 16 x 16 got the worst result. We think that the result can be expected because it have the largest density in this configuration. In this configuration, the average response time grows steadily under 2500 clients concurrently. But it starts to have dramatic growth after 2500 clients. Concerning the two configuration of 1000 x 1000, 16 x 16 and 500 x 500, 9 x 9, there is no obvious growth until the client number reaches to 3500 clients. But the amount of growth is steady and slow. Concerning the configuration of 1000 x 1000, 9 x 9, the average responses time is almost under good result throughout the test.

If we observe the percentage of CPU usage in Table 6-10, we can find that the performance bottleneck is on the gateway. The CPU usage is always under 50% in the servers. We take a future look. We recorded the update sent per second in the gateway

at the Table 6-11. We consider it have tight relation to the performance. The main game logic is move message. According to the move game logic described above, the number of update messages should be a function of total clients, map size, aoi size, and clients in the gateway. We first consider how many update messages are sent when one move message is process. First of all, the update message should be sent back to the avatar himself. Furthermore, the updates from surroundings and updates to surroundings can totally seem as function of the avatar in his AOI. It can calculate by

$\frac{n_{total}}{map} \times aoi$, where n_{total} is the total client number and map and aoi are map size and AOI size respectively. So a move message can cause $(\frac{n_{total}}{map} \times aoi + 1)$ update messages. Finally, the client sent one move message per second. So the updates per second can be described as the following formula

$$update = (\frac{n_{total}}{map} \times aoi + 1) \times n_{gat}, \text{ where } n_{gat} \text{ is the number of clients on a gateway}$$

More concisely, we use d as the number of game objects in an aoi , then the updates per second is described as following

$$update = (d + 1) \times n_{gat}, \text{ where } d = \frac{n_{total}}{map} \times aoi$$

and the total messages the gateways processed is

$$total = command + update = n_{gate} + (d + 1) \times n_{gate} = (d + 2) \times n_{gate}$$

Therefore, we calculate the prediction number of update messages processed by gateway according the last formula and list it in the “Prediction” column for each configuration in Table 6-11. The prediction data is very close to the actual data. The actual data is slightly more than prediction data. The reason could be that there were some avatar migrations occurred and the avatar migration may cause much more updates. Finally, we calculate the total messages per second processed by a gateway

by the given update messages per second plus the command message sent by clients. The result is listed in Table 6-12 and depicted in Figure 6-11. According the result we get above and map it to this diagram, we can find that a gateway can bear 5000~6000 messages per second.

Finally, we observe the impact of avatar migration in Figure 6-9 and Figure 6-10. The trend of updates of avatar migration is very similar to the general message updates. The average response time is 1.5 times larger than general message. But it is acceptable because the avatar migration does not happens often.

6.4. Summary

To evaluate our platform, we use three rounds to experiment our platform. In the first round, we hope to realize the performance of underlying NetEngine. Until the client numbers reach to 4000, the average response time also have the level of less than 100 ms. But there is steep rise of standard deviation when the client number reach to 4000. In the second round, we try to find the performance baseline of client-gateway-architecture. The result is very similar to round 1 expect that the average response time is slightly larger than that of round1. So the client-gateway-server architecture doesn't downgrade the performance very much. In round 3, we simulate the real game world. We implement the most significant game logic, avatar movement, in our experiment platform. We found that the performance bottleneck lay on the gateways. Furthermore, we observed the amount of update messages forwarded by gateway. We construct a formula to calculate the number of updates for one command. The result is that the gateway can bear about 5000~6000 messages per second.

Chapter 7

Future Works and Conclusions

7.1. Conclusions

To craft a MMOG middleware is not an easy job. In this paper, we figured out the issues we may encounter when designing an MMOG middleware. We discuss them in four criteria: scalability, flexibility, easy-to-use, and high performance.

In scalability, we use client-gateway-server architecture to scale up the platform. In view of real world, we can scale up the number of clients connected by way of multiple gateways. In view of virtual world, we can scale up the number of game objects by way of spanning regions over the server cluster.

In flexibility, we reserve as few footprints as possible for the protocol. We also reserve the most resilience of game objects for developer. In addition, developer can extend the platform by designing their own plugin. There are also some flexibility concerns in designing the feature of Avatar Migration and Region Migration. We always hope the data can be migrated successfully as well as the data format is as flexible as possible. Furthermore, we use the Login Region pattern to overcome the login problem.

In easy-to-use, we simplify the underlying complexity of MMOGs by clean and thread safe API. We provide three layer of API: NetEngine library, DOIT API, plugin framework. Using DOIT API, game developer can develop a MMOG rapidly. We also suggest a easy way to send updates and receive update from an game object (by means of `GameSpace` and `GameSpaceUtil`). We also purpose a way to separate

the AI logic and game logic when designing the logic of NPC.

In high performance, we use an abstract NetEngine library, and different threading model for different demand. We deploy the event-driven NetEngine at the external interface of gateway for serving numerous clients concurrently while deploying the threaded NetEngine at the internal interface of gateway for exploiting the message aggregation feature to gain better performance. We also provided 3 experiments on our platform. We found, in some configuration, it got good performance when there are 4000 players on the same virtual world concurrently with 2 gateways and 2 servers. The bottleneck lay on the gateway component. It can hold 5000 messages per second with acceptable response time. We also can add more gateways in this platform if we need better service quality and quantity.

7.2. Future Works



In the current work, we focused on the scalable, flexible, high performance and easy-to-use issues. However, there are still plenty of issues should be taken into consideration.

Failover mechanism is absent in the current work. We hope that if a game server is done, we can recover the game states and continue to serve the clients. We may make use of the *Serializable Capsule* mechanism in Region Migration. We serialize the region state to the persistence store instead of another game server.

Cheating prevention is another place that has not been taken into consideration in the current work. The cheating prevention can be implemented as a gateway plugin. We may allow the game developers to monitor and the incoming command messages and filter out the illegal message. Besides, we may provide the SSL version of

NetEngine in order to meet the requirement for high demand of online game, such as online shopping mall.

We also hope to provide a script language framework for game developers for writing game logics. There are some good script language implementation for java platform, like Jython and Groovy. We may provide a scripting framework on top of the DOIT API.

Furthermore, the easy-to-manage is also important for MMOG middlewares. The components are distributed over plenty of machines. It is desirable to have a central management console. In fact, we have made efforts on this feature. We adopt the JMX[19] and JMX Remote technology.

The last one is the location-free communication. In the current work, it is easy to implement the feature that the avatars talk to others around him. However, we didn't provide the mechanism to communicate with a group located around the future world in the current work. We hope to provide this kind of location-free communication in our framework. Maybe the JMS[8] is a good choice to implement this feature.

Bibliography

- [1] Zona Inc., <http://www.zona.net/>
- [2] Butterfly.net, <http://www.butterfly.net/>
- [3] Nathan Sheldon, Eric Girard, Seth Borg, Mark Claypool, Emmanuel Agu. The Effect of Latency on User Performance in Warcraft III. In Proceedings of ACM NetGames at Electronic Arts Headquarters Redwood City, California, USA May 22-23, 2003.
- [4] M. Mauve, S. Fischer, J. Widmer. A generic proxy system for networked computer games, In Proceedings of ACM NETGAMES '02, pp.25-28.
- [5] Wish Engine, <http://www.mutablerealms.com/>
- [6] Massively Multiplayer Middleware ,
<http://www.acmqueue.com/modules.php?name=Content&pa=showpage&pid=115>
- [7] Xuan-Feng Lee, Tsun-Yu Hsiao, Shyan-Ming Yuan. A Scalable Middleware Architecture for Supporting Massive Multiplayer Virtual Worlds. In Proceeding of Symposium on Digital Life and Internet Technologies 2003.
- [8] Sun Microsystems Inc. Java Message Service,
<http://java.sun.com/products/jms/>
- [9] BigWorld Technology, <http://www.bigworldgames.com/>
- [10] K. Lee and D. Lee. A Scalable Load Balancing Scheme for Large-Scale Multi-Server Distributed Virtual Environment Systems. In Proceedings of ACM Symposium on Virtual Reality Software and Technology (VRST2003), Osaka, Japan, 1-3 October 2003.
- [11] P. Morillo, J.M. Ordu.na, M.Fernandez. An Adaptive Load Balancing

Technique for Distributed Virtual Environment Systems. Proceedings of the IASTED International Conference Parallel and Distributed Computing and Systems November 3-5, 2003 in Marina del Rey, CA, USA.

- [12] Lars Aarhus, Knut Holmqvist and Martin Kirkengen. Generalized TwoTier Relevance Filtering of Computer Game Update Events. Proceedings of ACM NetGames 2002.
- [13] Ashwin R. Bharambe. Mercury: A Scalable Publish/Subscribe System for Internet Games. Proceedings of ACM NetGames 2002.
- [14] Sandeep Singhal, Michael Zyda. Networked Virtual Environments, Design and Implementation. Published by Addison-Wesley, 1999.
- [15] Daniel Bauer. Network Infrastructure for Massively Distributed Games. Proceedings of ACM NetGames 2002.
- [16] Matt Welsh, Steven D. Gribble, Eric A. Brewer, and David Culler. A Design Framework for Highly Concurrent Systems. UC Berkeley Technical Report UCB/CSD-00-1108, Submitted for publication, April, 2000.
- [17] Sun Microsystems Inc. JavaBeans Architecture,
<http://java.sun.com/products/javabeans/>
- [18] Sun Microsystems Inc. Java Servlets API,
<http://java.sun.com/products/servlet/>
- [19] Sun Microsystems Inc. Java Management Extension,
<http://java.sun.com/products/JavaManagement/>

Appendix A System Protocol

ID	Name	Direction	Description
0x01	CTRL	G→S	The control message
0x02	UPDATE	G→S	The update message
0x03	AVACONN	G→S	Notify that a channel connected.
0x04	AVADISC	G→S	Notify that a channel disconnected
0x05	AVACLOSE	S→G	Notify that a avatar channel is close by game logic
0x06	AVAMIG	G→S S→G	Send when a avatar migrated.
0x07	GATHEL	G→C G→S	Say hello and tell the gateway information.
0x08	SRVHEL	S→C	Say hello and tell the server information
0x09	RMINIT	C broadcast(*1)	Send when a coordinator initiate an region migration
0x0A	RMREADY	S→C	The destination server says he is ready to receive migration data.
0x0B	RMCOMPLETE	S→C C broadcast(*1)	Send when the region migration is complete
0x0C	RMFAIL	S→C C broadcast(*1)	Send when a region migration fails.
0x0D	AVACONN_ACK	S→G	ACK of AVACONN
0x0E	AVAMIG_ACK	S→G	ACK of AVAMIG

G: Gateway, S: Server, C: Coordinator

*1 A coordinator broadcasts the messages to every participants of the region migration.

0x01 CTRL

Direction: G→S

Description:

The control message

Format:

long avatarid

short controlType

short controlLength

byte[] wrappedMessage

0x02 UPDATE

Direction: G→S

Description:

The update message



Format:

long avatarid

short updateType

short updateLength

byte[] wrappedMessage

0x03 AVACONN

Direction: G→S

Description:

Notify that a client connected.

Format:

long avatarid

0x04 AVADISC

Direction: G→S

Description:

Notify that a client disconnected.

Format:

long avatarid

0x05 AVACLOSE

Direction: S→G

Description:

Notify that a avatar channel is close by game logic

Format:

long avatarid



0x06 AVAMIG

Direction: G→S, S→G

Description:

Send when a avatar migrated.

Format:

long avatarid

int srcRegionid

int destRegionid

int x

int y

short dataLength

byte[] data

0x07 GATHEL

Direction: G→S, G→S

Description:

Say hello and tell the gateway information

Format:

int gatewayid

0x08 SRVHEL

Direction: S→C

Description:

Say hello and tell the server information

Format:

int serverid



0x09 RMINIT

Direction: C broadcast

Description:

Send hen a coordinator initiate a region migration

Format:

int rmid

int regionid

int srcServerid

int destServerid

0x0A RMREADY

Direction: S→C, C→S

Description:

The destination server says he is ready to receive migration data. This message will be forwarded to source server by the coordinator.

Format:

int rmid

int host

int port

0x0B RMCOMPLETE

Direction: S→C, C broadcast

Description:

Send when the region migration is complete.

Format:

int rmid



0x0C RMFAIL

Direction: S→C, C broadcast

Description:

Send when the region migration fails.

Format:

int rmid

0x0D AVACONN_ACK

Direction: S→G

Description:

ACK of AVACONN

Format:

long avatarid

0x0E AVAMIG_ACK

Direction: S→G

Description:

ACK of AVAMIG

Format:

long avatarid

