

California State University, San Bernardino

CSUSB ScholarWorks

Theses Digitization Project

John M. Pfau Library

2008

Design and implementation of a multi-player role playing game

Giang Tuan Trang

Follow this and additional works at: <https://scholarworks.lib.csusb.edu/etd-project>



Part of the [Databases and Information Systems Commons](#), and the [Software Engineering Commons](#)

Recommended Citation

Trang, Giang Tuan, "Design and implementation of a multi-player role playing game" (2008). *Theses Digitization Project*. 3493.

<https://scholarworks.lib.csusb.edu/etd-project/3493>

This Project is brought to you for free and open access by the John M. Pfau Library at CSUSB ScholarWorks. It has been accepted for inclusion in Theses Digitization Project by an authorized administrator of CSUSB ScholarWorks. For more information, please contact scholarworks@csusb.edu.

DESIGN AND IMPLEMENTATION OF A MULTI-PLAYER
ROLE PLAYING GAME

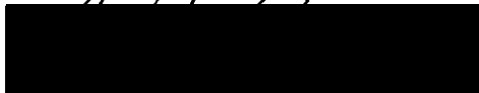
A Project
Presented to the
Faculty of
California State University,
San Bernardino

by
Giang Tuan Trang
December 2008

Approved by:



Dr. David A. Turner, Computer Science



Dr. Arturo I. Concepcion



Dr. Tong Lai Yu

11/20/2008

Date

ABSTRACT

Modern hardware had taken the gaming industry to its climax. Quad-core machines with high performance GPU and Pentium II with 16 MB graphic card will perform the same on simple Hello World program. While most software complexity does not grow at the rate defined by Moore's law (double every 1.5 year), the game applications utilize full potential of today's computer hardware. Furthermore, the availability of high speed access has increase significantly during the pass five years making multiplayer online gaming more accessible to everyone.

Despite the high volume of recent new releases, creating a good game application is not quite simple. Careful consideration must be made in hardware utilization, response time, and game logic. Furthermore, storyline, graphic and game quests must be maximized to keep player's interest. Therefore, learning to program a modern game, particularly multiplayer game, is a stepping stone to build future's state-of-the-are software solutions.

TABLE OF CONTENTS

ABSTRACT	iii
LIST OF FIGURES	vi
CHAPTER ONE: INTRODUCTION	1
The Team	2
Design Overview	3
Report Overview	8
CHAPTER TWO: CLIENT-SERVER COMMUNICATION	
Overview	9
Client Side	10
Server Side	15
Client-Server Interactions	20
Pre-login State	21
Avatar Selection State	23
Loading Map State	24
Game Play State	24
CHAPTER THREE: SERVER ARCHITECTURE	
Overview	26
Name Spaces	27
Persistence	30
Game Loop	32
Usage Scenarios	33
Load Capacity	33

Shoot a Fireball	33
Activate a Portal	34
Activate a Treasure Box	34
CHAPTER FOUR: CLIENT ARCHITECTURE	
Overview	36
Initialization	37
Game Loop	39
Game Play State	40
Shoot a Fireball	41
Memory Management	44
The Graphic Engine	44
The Approaches	46
CHAPTER FIVE: DEPLOYMENT	
Overview	50
Create Client Setup Program	50
Server Deployment Architecture	60
CHAPTER SIX: CONCLUSION	
Current Progress	62
Future Direction	63
REFERENCES	66

LIST OF FIGURES

Figure 1. Client-Server's Physical Arrangement	4
Figure 2. Project Package Diagram	6
Figure 3. Communication Project Dependencies Diagram	9
Figure 4. Client Communication System	10
Figure 5. Read and Write to Message Queue	13
Figure 6. Using a Lock to Achieve Mutual Exclusion	14
Figure 7. Server Communication System	16
Figure 8. Implementation of Listening Mechanism	18
Figure 9. Client State Diagram	21
Figure 10. Client Sends Login Message	21
Figure 11. Server Reads Login Message	22
Figure 12. Data Access Object Example	28
Figure 13. Server Entity-Relationship Diagram	29
Figure 14. Client Architecture	36
Figure 15. Create Fire Ball Method	42
Figure 16. Fire Ball Update Method	43
Figure 17. Using a Lock to Clean Up Textures	49
Figure 18. Add New Project	51
Figure 19. Add New Project Wizard	52
Figure 20. Setup Wizard 1	53
Figure 21. Setup Wizard 3	54

Figure 22. Drag and Drop Additional Files	55
Figure 23. Setup Project File Structure	56
Figure 24. Create.Uninstall.bat	57
Figure 25. Project Build Mode Window	59
Figure 26. Server Deployment	60

CHAPTER ONE

INTRODUCTION

In the field of computer science, the topic of game programming is sometimes misjudged as lacking seriousness. However, recently, it appears that the gaming industry has evolved greatly and topics such as XBOX, PS2, and World of the Warcraft are often centers of students' conversations. Coincidence or not, many computer science students are driven to universities to acquire the skills necessary to join this challenging and entertaining profession.

Although the word "game" doesn't sound too serious, game programming is among the most complex types of programming. Evidently, gaming and scientific applications are the only types of applications that utilize all the potential of today's state-of-the-art hardware. A game application is a real time application. All computations of game logic, communications and graphics must be done not only fast as possible but also as consistently as possible. Especially, in a multiplayer game environment, resource optimizations must be done

precisely or else a weak (or erroneous) node in the gaming network will slow down and cause lagging to other players. In a multiplayer role-playing game (MRPG), the setting usually involves a virtual world, playable and non-playable characters, and entities of various kinds ranging from something as large as a mountain or a building to something as small as a bullet or a coin. Therefore, the application has to solve complicated problems such as physical motion, collision detection, and graphical representation of those entities in the virtual 3D world. That is not to mention the problem of implementing storyline and the social interactions among characters in the game logic.

The Team

It was fortunate for the students who are interested in game programming that the Department of Computer Science and Engineering of CSUSB started to encourage, promote and support game programming. Led by Dr. David Turner and Dr. Arturo Concepcion, a team of approximately 15 students were formed to solve a game programming problem in the summer of 2008. The team consisted of students from various programming backgrounds and

interests. Some team members preferred to work on game logic design and some preferred to contribute to graphic design. The software architecture is mainly developed by Dr David Turner with great help of William Herrera, a student of the Department of Computer Science and Engineering. Due to conflicting schedules, team members often worked remotely; however, they met regularly to report on project status. The team's goal was to finish the first working artifact minus the art work by the end of summer 2008. This first artifact with a concrete, solid architecture will serve as a foundation for future students to learn game programming.

Design Overview

The project was designed using client-server architecture. As shown in Figure 1, the client (physical) machine consists of a virtual machine (VM) running C# code. This code is called managed code since it has a garbage collector that automatically handles memory that is no longer referenced. The VM also calls routines from external libraries such as OpenGL and SDL. These libraries run on unmanaged code. Used memory must be manually reclaimed in unmanaged environment. The VM also

read configuration files in its local file system.

The server also runs on a virtual machine. It runs only managed code. The server exchanges data locally with MySQL, a database storage. It also reads its configuration files through its local file system.

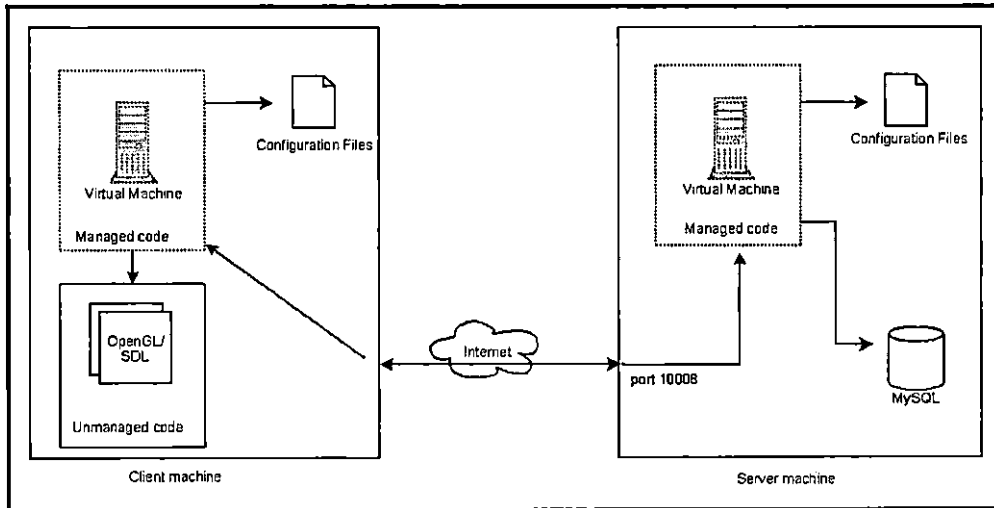


Figure 1. Client-Server's Physical Arrangement

MRPG was designed with modularity in mind. The project is a composition of eleven smaller projects (or packages) as shown in Figure 2. This modulus design serves two main purposes. First, it allows team members to work on their parts independently without affecting others' part, given that the application programming interface (API) rule is obeyed. Secondly, the design is

highly cohesive, meaning that the responsibilities within a module are strongly related. It also implies that the separations of modules are clear and unambiguous.

Therefore, MRPG will be easy to maintain, test and debug.

A good example for this trait of the project is the ServerDatabase sub-project. It deals only with data reading and writing. Thus, when debugging a communication error, the developer can save time by leaving ServerDatabase last on the list to check.

In the Project Dependencies diagram, the sub-projects:

Gui, Mobius, MobiusClient, ClientCommunication and ClientCollisionSystem reside on client machine.

ServerCommunication, ServerDatabase and Server packages run on the server. Share among client and server is CommunicationMessagesTypes acting as a protocol for exchanging messages. TestCommunication package is used for verifying the correctness in client-server communications.

On client side, Gui package handles all user interactions with the game such as logging in, selecting characters, accessing the menu and choosing weapons. Gui displays and handles mostly everything that user interact with that is not an entity in the game world. On the

other hand, Mobius package, arbitrarily named by Will Herrera, displays the graphical game world and its entities in 3D. It processes a user's interaction with game entities such as selecting an object, choosing a target and navigating the character. It's mostly responsible for all 3D aspects of the game including animations of weapon firing and character movements.

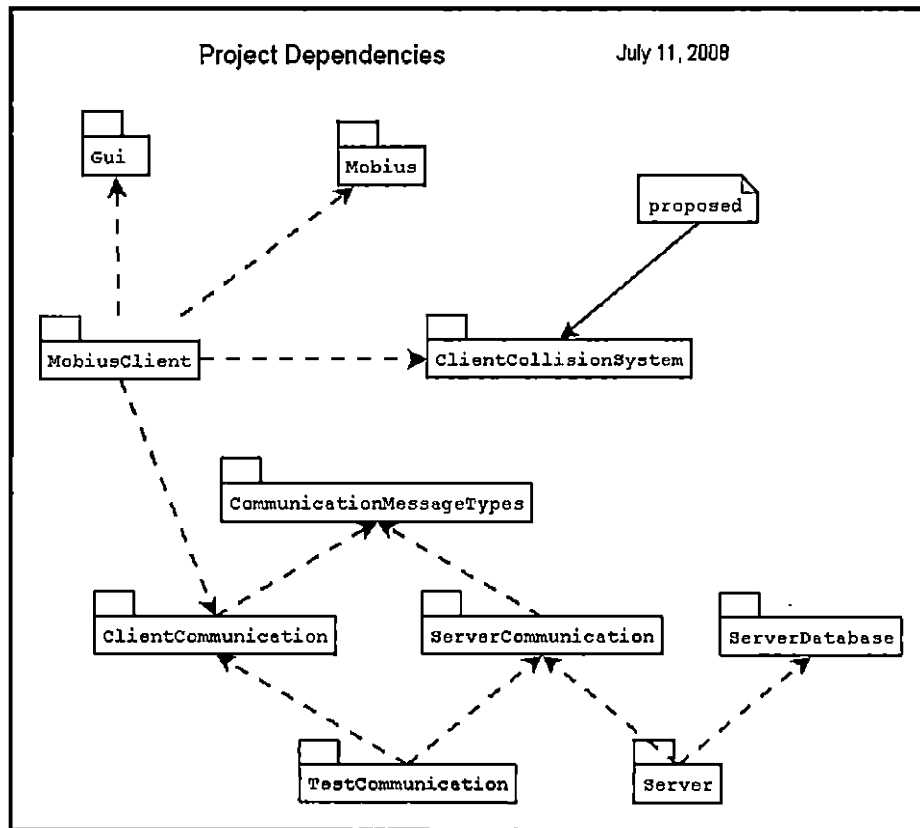


Figure 2. Project Package Diagram

The ClientCollisionSystem detect physical interactions of objects in the game world. For example, it needs to notify the Mobius when an entity accouters a wall in its path or to detect whether a fireball actually hit the target.

MoubiusClient brings Gui, Mobius, ClientCollisionSystem and ClientCommunication together for execution. It contains the main thread of execution. It handles the top level logic of the game state such as transitioning among Pre Login state, Avatar Selection state and Load Map State.

On the server side, Server makes use of ServerCommunication for communicating with client and ServerDatabase for storing persistence data. ServerDatabase keeps track of user accounts, character inventories and saved games.

This overall design was reengineered multiple times throughout the development phase. It was changed at various times to finally accommodate most constrains as they were encountered. The project is versioned to be used as a learning tool for future students; therefore, its foundation must be as solid as possible.

Report Overview

In subsequent chapters, details of project design and implementation will be discussed. Chapter 2 focuses on communication by describing ClientCommunication, ServerCommunication and CommunicationMessage project. In Chapter 3, Server and ServerDatabase are the two main projects that make up the architecture of the server. They are responsible for game logic and data storage. Client side project such as MobiusClient, Gui, ClientSound, ManagedMobius and PhysicsSystem, handles graphic display, graphical user interface, sound and physics in the game world. These subjects are the main topics of Chapter 4. Chapter 5 discusses about deployment of the server and client. Details in setting up server and installing client program will be discussed. Chapter 6 concludes the report with suggestions for future improvement.

CHAPTER TWO

CLIENT-SERVER COMMUNICATION

Overview

In this chapter, details design of communication system will be revealed. We'll discuss about key features of client side and sever side designs. We'll begin by describing the overall structure of the involving packages and we'll go into details of different type of messages.

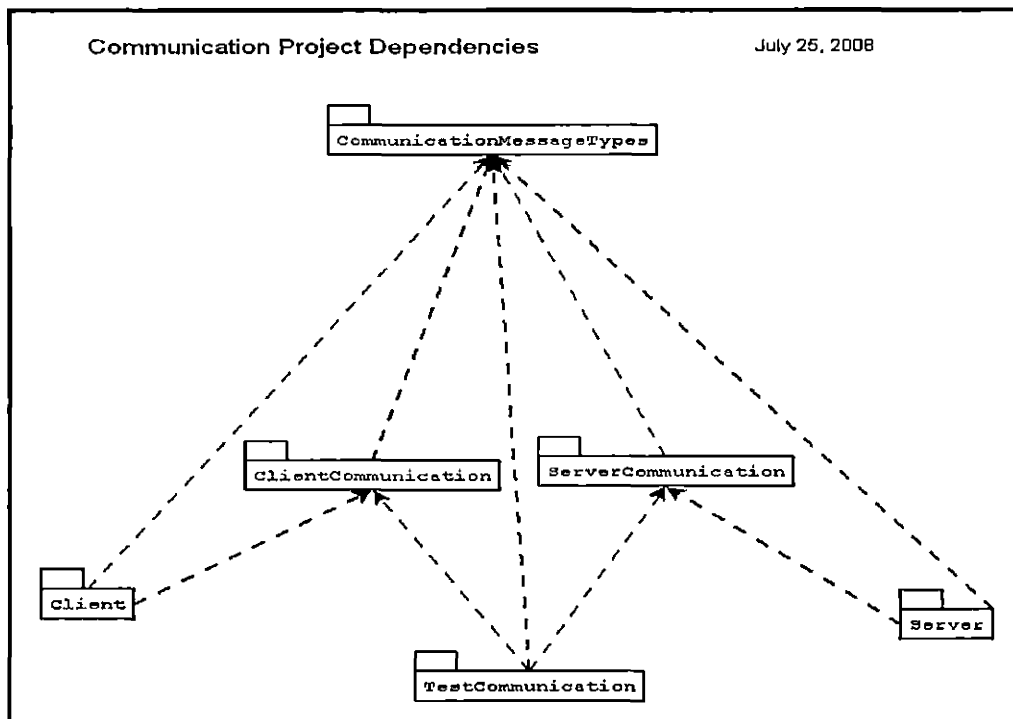


Figure 3. Communication Project Dependencies Diagram

MRPG follows client-server architecture. In Figure 3, the communication between the client and the server is handled by three packages: ClientCommunication, ServerCommunication and CommunicationMessageType. As their names suggest, ClientCommunication runs on client side and ServerCommunication runs on the server side. Both client and server use CommunicationMessageType as the protocol for exchanging messages. Therefore, CommunicationMessageType instances must be identical.

Client Side

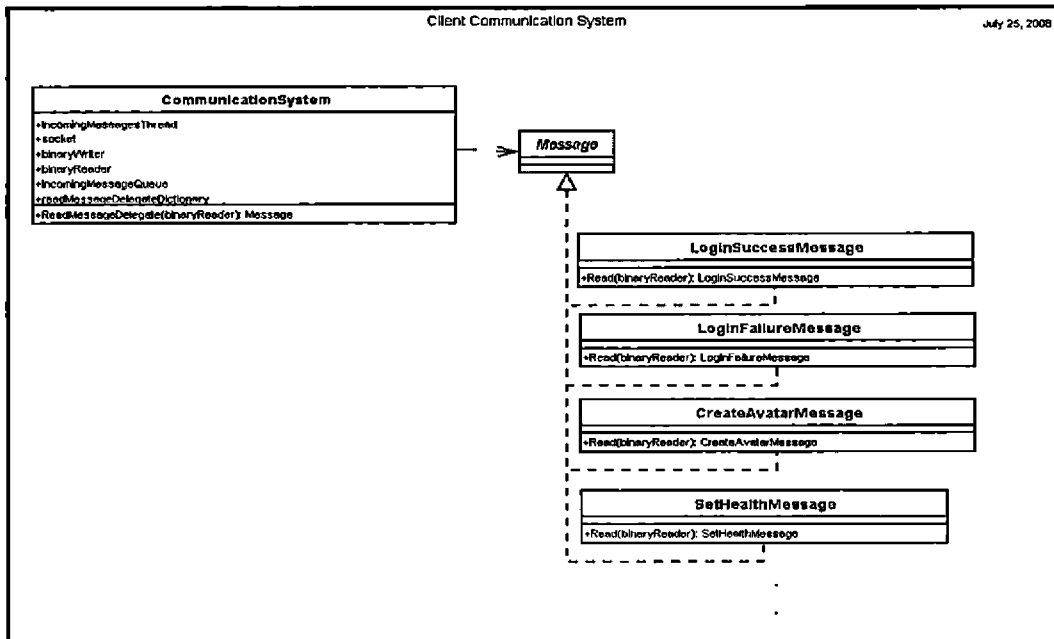


Figure 4. Client Communication System

In ClientCommunication package (Figure 4), CommunicationSystem class is the heart of the package. CommunicationSystem class has static reference to a Socket, a BinaryWriter, a BinaryReader, a queue of messages and a Dictionary of all acceptable message types such as login success message, login failure message, and create avatar message. The message queue contains implementation instances of Message interface. All message types must implement Message interface. A typical message type such as LoginSuccessMessage has a Read method that uses the static BinaryReader mentioned to interpret the message received from the server. To send a message such as login message to the server, the client's CommunicationSystem provides the SendLoginMessage that uses the BinaryWrite to write to the Socket object. The Socket class, according to Microsoft's C# documentation [1], allows program to perform both synchronous and asynchronous data transfer using communication protocol such as IP, TCP or UDP. A socket instance represents a channel of communication between two end-points.

When the client program first starts, no communication exists between the client and the server. When a user clicks on the login button, the client will

read the server's information from its configuration file creating a socket to initiate a connection. Reference to the socket is kept as a static variable for future usage. Immediately after the socket creation, Communication System will fork a static thread that is strictly designed for listening to messages receiving from the server. All messages received are queued up and processed in first-in-first-out (FIFO) manner by the main thread. This design follows the producer-consumer pattern to achieve concurrency in receiving messages and processing game logic. It's very important that the messages are received as soon as possible in a real-time system. If it was done in a single thread, the game play will be on paused every time it makes a blocking call to the socket. Consequently, the game play would not run as smoothly in user's view. Furthermore, perhaps it will introduce the complexity of handling messages that had arrives late.

Since the Communication System has its own thread for queuing up the messages and the main thread processes the messages, it's extremely important to make sure the read and write operations to the queue are thread safe. Base on Microsoft's documentation [1], object of List class is not thread safe. Therefore, to achieve mutual

exclusion, we must make use of the lock when accessing the queue. The implementation of the message queue is as follow:

```
static void queueUpIncomingMessages()
{
    while (true)
    {
        Message message = null;
        try
        {
            message = ReadMessage();
        }
        catch (Exception)
        {
            lock (incomingMessageQueue)
            {
                incomingMessageQueue.Clear();
            }
            return;
        }
        lock (incomingMessageQueue)
        {
            incomingMessageQueue.Add(message);
        }
    }
}

static Message ReadMessage()
{
    string messageType = binaryReader.ReadString();
    ReadMessageDelegate readMessageDelegate =
        readMessageDelegateDictionary[messageType];
    return readMessageDelegate(binaryReader);
}
```

Figure 5. Read and Write to Message Queue

Method `queueUpIncomingMessages` is called by the `IncomingMessagesThread`. The thread enter an infinite loop

that checks for new messages and stores them in IncomingMessageQueue. To ensure thread safety, keyword "lock" was used to make sure that only one thread can access incomingMessageQueue at a time. In the event of connection failure, the thread will clear out the queue and returns.

```
public static Message GetNextReceivedMessage()
{
    if (incomingMessageQueue.Count == 0)
    {
        return null;
    }
    Message message = null;
    lock (incomingMessageQueue)
    {
        message = incomingMessageQueue[0];
        incomingMessageQueue.RemoveAt(0);
    }
    return message;
}
```

Figure 6. Using a Lock to Achieve Mutual Exclusion

From Figure 6, the GetNextReceivedMessage method is called by the main thread during each update cycle. The method is called repeatedly until all messages are processed. Again, to avoid multithread conflict, the keyword "lock" was used to ensure exclusive usage.

Server Side

The `CommunicationSystem` of the server, in contrary to client side, need to handle multiple connections simultaneously. Therefore, it maintains a list of `CommunicationChannel` as shown in Figure 7. Each `CommunicationChannel` instance corresponds to a connected client. Similar to client side, the server has a dictionary of acceptable message type encapsulated by `CommunicationSystem`. The dictionary maps a message type to its appropriate message reader. `CommunicationSystem` also has a static `listenSocket` which strictly handle new connections.

The `CommunicationChannel`, similar to client's `CommunicationSystem`, has reference to a socket, a message queue and its own `Thread`. The thread independently queues up the messages sent by some particular client.

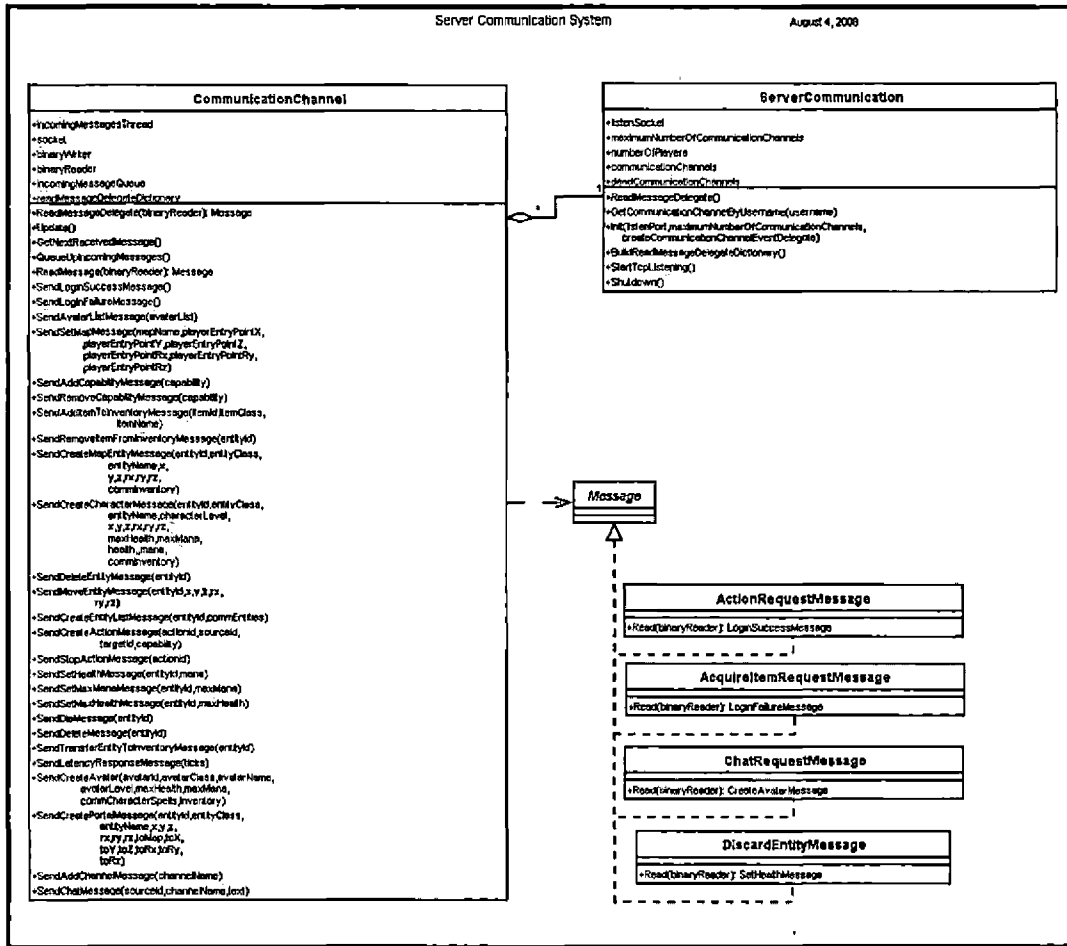


Figure 7. Server Communication System

During start up, the main thread initializes the CommunicationSystem by calling its Init method. The Init method build the message dictionary, make note of the delegate passed by the main thread. The delegate is a pointer to a function to be called upon creation on a CommunicationChannel when a new client connects. The Init method then calls StartTcpListening method marking its

readiness for accepting new client. Upon receiving a request for new connection, the CommunicationSystem creates a new Socket, instantiates a new CommunicationChannel with the newly created socket and add the new CommunicationChannel to its CommunicationChannel List. Finally, the CommunicationSystem call the delegate that was passed earlier by the main thread during start up. This call will construct the server's version of the PreLoginState. Thus, a new client session begins. The actual implementation of server listening mechanism is as follow:


```

static void StartTcpListening(int port)
{
    IPEndPoint localEndPoint = new IPEndPoint(IPAddress.Any, port);
    listenSocket.Bind(localEndPoint);
    listenSocket.Listen(port);
    listenSocket.BeginAccept(AcceptCallback, null);
}
static void AcceptCallback(IAsyncResult ar)
{
    Socket socket = listenSocket.EndAccept(ar);
    CommunicationChannel communicationChannel = new
        CommunicationChannel(socket);

    lock (communicationChannels)
    {
        communicationChannels.Add(communicationChannel);
    }
    CreateCommunicationChannelEvent(communicationChannel);
    listenSocket.BeginAccept(AcceptCallback, null);
}

```

Figure 8. Implementation of Listening Mechanism

The method `StartTcpListening` is called once during start up. It binds the `listenSocket` to the `localEndPoint`. The attribute `IPAddress.Any` specifies that the server must listen to any network activities through all networks interfaces on the server machine. The `listenSocket` is then told to listen to the port specified and begin accepting connections. It important to note that

BeginAccept method takes a function name (or delegate) as its parameter. BeginAccept is a non-blocking call and will return upon receiving client request, during which the method AcceptCallback is called. While in AcceptCallback, EndAccept returns the socket representing the incoming connection. The connection is then wrapped in CommunicationChannel as mentioned above. listenSocket then resumes its usual listening state by calling BeginAccept again.

In general, the server always runs on a main thread, which constantly loops through its client list to update their game status. For every connected client, there is a thread that runs concurrent with its peer threads and the main thread. For every iteration in the game loop, the main thread will loop through its CommunicationChannel List to check on the communication status. If a connection is terminated, it will be added to the deadCommunicationChannels list and eventually removed from communicationChannel list. Of course the thread safe rule must be obeyed using the locking mechanism when adding and removing items from communicationChannel list.

The key feature that allows the client and the server to understand each other's message is the

CommunicationMessageType package. The package has a MessageTypes class define the message type constants. For instance, the string "login" define the message that is sent by the client requesting to login to the server. CommunicationMessageType also define classes that are shared between client and server such as: CommEntity, CommAvatar and CommCharacterSpell. These shared classes contains attribute that are relevent to both the client and the server. For example, the client has a Avatar class that beside its name, it also has data that map to its graphical representation. The server also has an Avatar class but it doesn't need to know the graphical attribute of an avatar. The CommAvatar class is to define the intersection between both attribute sets making communication more effectively be removing extraneous attributes.

Client-Server Interactions

The folowing sub-section will describe all message exchanging during all states of the client, Figure 9.

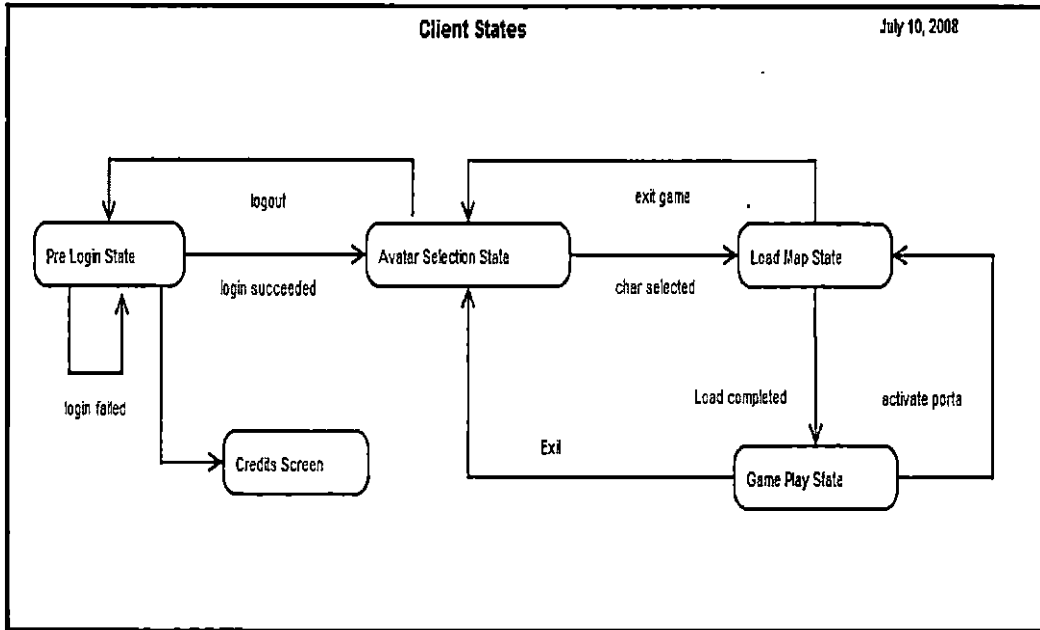


Figure 9. Client State Diagram

Pre-login State

When the client program starts, after initialization, it enters immediately to Pre-login state, Figure 9. While in Pre-login state, the client can only send one type of message to the server, the LOGIN type. The implementation of the login mechanism is as follow:

```

public static void SendLoginMessage(string username, string password)
{
    binaryWriter.Write (LOGIN) ;
    binaryWriter.Write (username) ;
    binaryWriter.Write (password) ;
}
  
```

Figure 10. Client Sends Login Message

On the sever side, the server uses class LoginMessage to read the message as follow:

```
internal static Message Read(BinaryReader binaryReader)
{
    LoginMessage loginMessage = new LoginMessage();
    loginMessage.username = binaryReader.ReadString();
    loginMessage.password = binaryReader.ReadString();
    return loginMessage;
}
```

Figure 11. Server Reads Login Message

The sever then validate credential provided and send back to client either a LOGIN_FAILURE message or a LOGIN_SUCCESS message. Client will need to use LoginFailureMessage class or LoginSuccessMessage to read the message similarly to the code snippets above. If the received message was a failure, client terminates the connection and resume Pre-login state. On the other hand, if it was a success message, the client will transition to AvatarSelectionMode. Also, the server will transition to its version of the AvatarSelectionMode. There it will fetch a list of avatars and send to the client immediately. At any moment during the Pre login state, the server and the client do not maintain communication.

Communication is initiated upon the login action and terminates when first transiting back into Pre-login state either through a logout action or when login fails. Otherwise there will always be a connection in AvatarSelectionMode, LoadMapState and GameplayState.

Avatar Selection State

Upon transitioning into avatar selection state, the client will immediately receive an AVATAR_LIST message from the server. The client will process the avatar list and remain in avatar selection state for further instruction from the user.

While in avatar selection state, the client has two choices: to send a LOGOUT message to logout or to send SELECT_AVATAR to inform the server about the chosen avatar. If it is a logout message, client will immediately transition back to Pre-login state. Otherwise, if it is a select avatar message, the client will remain in AvatarSelectionMode and wait for further instruction from the server.

After receiving the SELECT_AVATAR, the server will fetch avater's attributes from the database and send them back to the client using CREATE_AVATAR message. Sever will transition to its MapLoadingState. It then will

fetch the map attributes and send to client the SET_MAP message. On the other end, after receiving avatar's attributes through the SELECT_AVATAR message, the client stays in its current state and expect the SET_MAP message from the server. When the message arrives, client will transition to MapLoadingState state. At this point, both the server and the client are in MapLoadingState, also known as Loading Map State.

Loading Map State

After transitioning in to MapLoadingState, the client will start loading the graphical representation of the map into the memory. Loading map is a time consuming process; therefore, user has an option to force an EXIT_GAME if cannot wait for map loading to finish. If a user takes this option, both end-points will return back to AvatarSelectionState. Otherwise, after the map is loaded, a MAP_LOAD_COMPLETION message will be sent to the server and the client will transit to GameplayState. The server will do the same upon receiving the message.

Game Play State

During the GameplayState, the server expect the following type of message from the client: MOVE_REQUEST, ACTION_REQUEST, STOP_ACTION_REQUEST, INTERACT_REQUEST,

ACQUIRE_ENTITY_REQUEST, RELEASE_ENTITY_LIST, DISCARD_ENTITY, LATENCY_REQUEST, JOIN_CHANNEL_REQUEST, LEAVE_CHANNEL_REQUEST and CHAT_REQUEST. to carry out game logic among players. These types of request will not result in state transitioning and therefore will be discussed in Chapter 3, Server Architecture, and Chapter 4, Client Archichture instead. However, a EXIT_GAME message will take client and server back to AvatarSelectionMode if user choose to exit game. Also, if the game play result in a map transition, the client will transition back to MapLoadingState temporarily to load the new map and then resumes GameplayState shortly.

Although the communication systems of both client and server are relatively simple, they serve the project effectively. They handle concurrency issues very well. For maintainability, it's very easy to add new type of message if needed to. It virtually doesn't affect the existing codes. The new message just needs to implement the Message interface and it new message send/receive handlers.

CHAPTER THREE

SERVER ARCHITECTURE

Overview

In this chapter, we'll discuss in detail the architecture of the server. We'll talk about how persistent data are handled and the typical interactions between the clients and the server within the game loop.

The server, besides being the center of all communications, is the authority on game state. It decides if a requested action from the client is permitted and succeeds. Thus, clients send requests, which the server either accepts or rejects. The server makes decisions based on its knowledge of the overall system at the given moment. For instance, player 1 can request that a fire ball attack be carried out on player 2, but the server will decide whether the fireball attack is permitted under the game play constraints (such as minimum distance to target, completion of cool downs and mana restrictions). After deciding that the attack is permitted, the server then decides the amount of resulting damage to player 2. There is one exception to the rule that the server decides on the game state: the

server accepts all requested player positions. This exception was allowed in order to relieve the server of performing the computationally expensive collision detection with the terrain. During this initial programming iteration, the server was not designed to maintain the details of the terrain therefore cannot enforce collision with the terrain.

Name Spaces

The server consists of five packages, or namespaces; namely: Server, ServerDatabase, ServerCommunication, CommunicationMessageTypes and MathUtil. The Server namespace is the execution entry point. It contains the server-side game loop that, as mentioned above, controls the game states and processes client requests. As a note for future development, game AI will be calculated in this namespace. As shown in the UML diagram in Figure 2, the Server namespace depends on the ServerDatabase namespace; it uses data access objects from SserverDatabase in order to retrieve and save game account information such as user accounts, game characters and inventories. The ServerDatabase uses the Data Access Object (DAO) design pattern to hide database

access details. For example, the LoginCredentialsDao class has a static method FindByUsername that retrieves user account credentials from the database, constructs a LoginCredentials object and returns it back to caller (See Figure 12). This design pattern separates classes that handle business logic from those that query the database. It abstracts all the details of database access into DAO objects, hence increasing cohesion in the design.

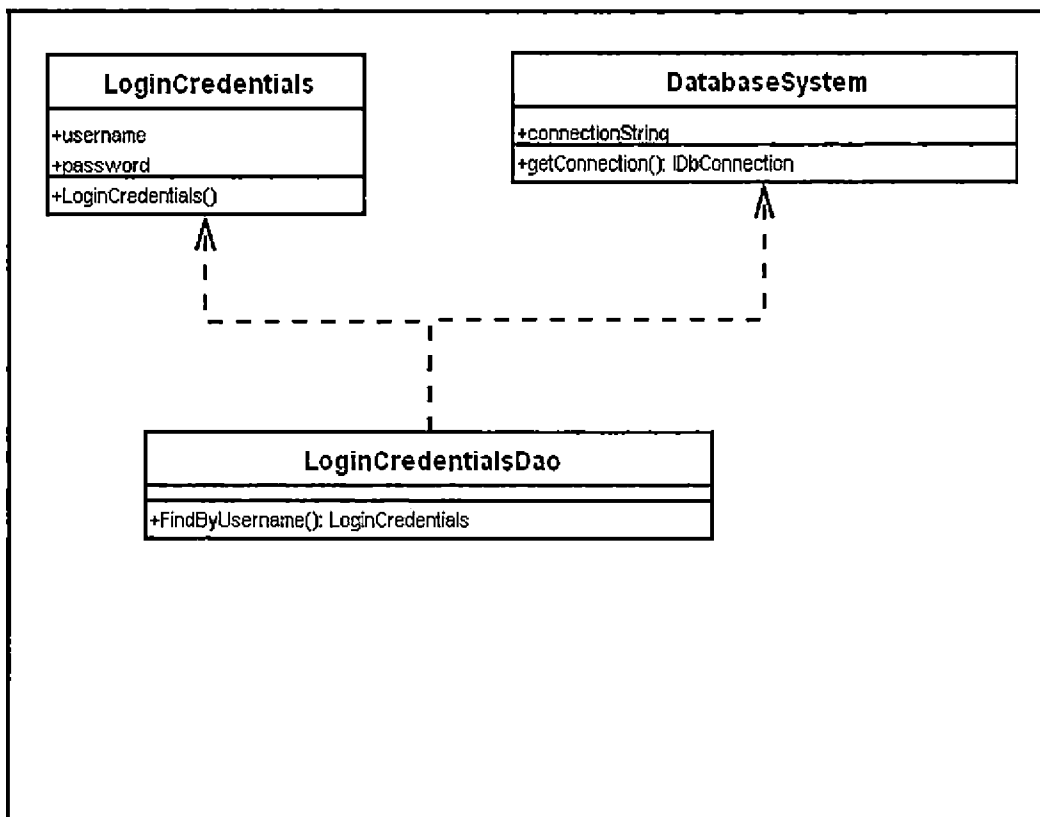


Figure 12. Data Access Object Example

The packages ServerCommunication and CommunicationMessageTypes provide the mechanisms for exchanging messages between client and server. Detailed descriptions of these packages were given in Chapter 2. The utility package called MathUtil provides user defined methods and mathematical data types.

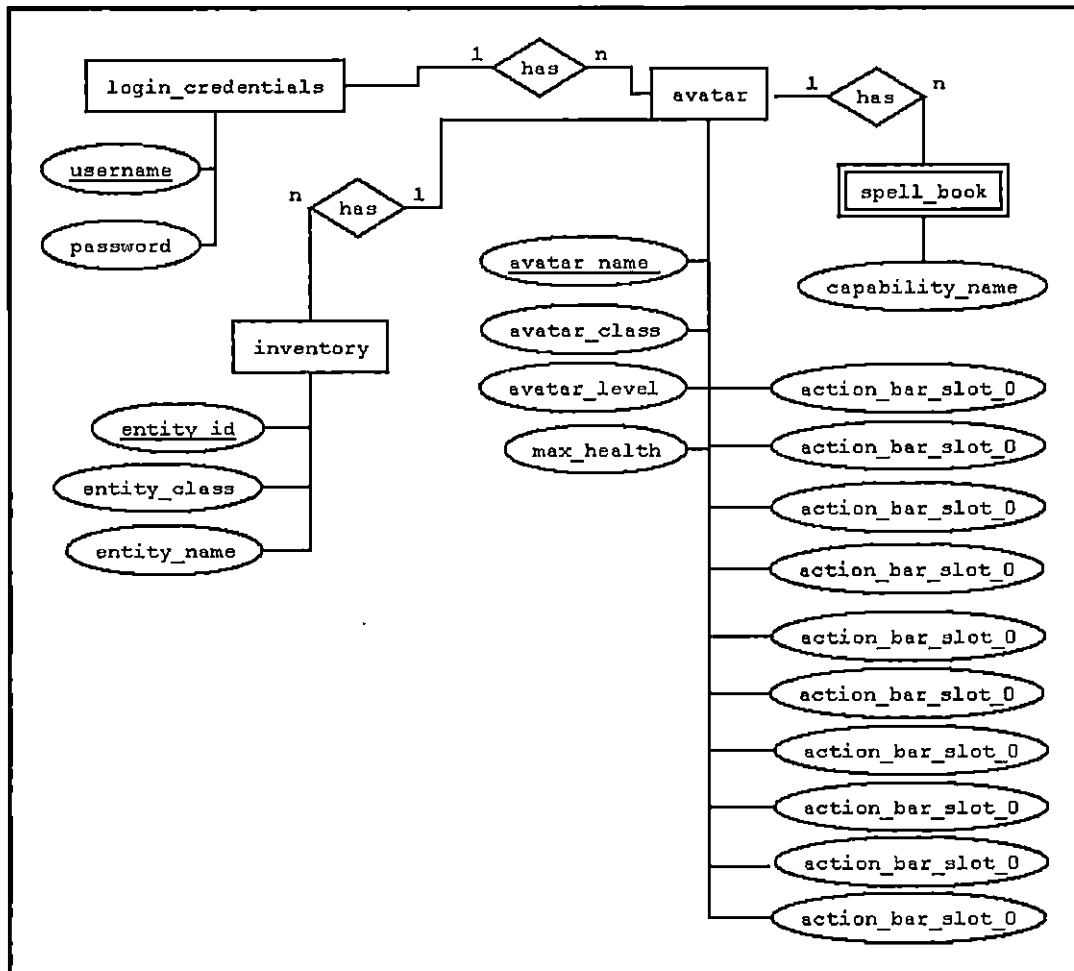


Figure 13. Server Entity-Relationship Diagram

Persistence

Persistent data is data that survive server reboots. In this project, persistent data appears in two different formats: database format and XML file format. From Entity-relationship diagram in Figure 13, the following tables are created: login_credentials, avatar, inventory, and spell_book.

The user account table, login_credentials, has simply a username column and a password column. Username is the primary key of the table. Password is in plain text for testing purposes, but can be encrypted during production operating mode.

A record in the avatar table links the avatar to a user account. An avatar has a unique avatar name and belongs to a predefined avatar class. The avatar table also keeps track of levels, maximum mana, maximum health, and ten action bar slots. The value of a level corresponds to the experience and skills of the avatar. The mana value allows the avatar to perform special tasks such as healing or attacking. When the health value decreases to zero, the avatar is considered killed. Health value decreases when avatar is hit and increases when taking a health potion, herb, or being healed by

another avatar.

The `spell_book` table records the avatars' capabilities. Capabilities can be the ability to heal or to perform a special attack tactic.

An avatar can obtain items by interacting with the world. Items that are obtained by the avatar are referred to as inventory, and are stored in the inventory table. Some items can be equipped, such as weapons and armor, which is the ability for the avatar to hold, wear or use the item. When a player equips an item, it is associated with an equipment slot. Only one equipable item at a time may occupy an equipment slot. The equipment slots include weapon, hand (for gloves), head (for hats and helmets), chest (chest armor), legs (leg armor), and feet (boots and shoes). The state of the equipment slots are kept in the inventory table.

Note that items are considered entities in the game world along with other objects, such as portals, avatars, or treasure boxes. An entity has a unique id and belongs to a predefined entity class.

The second type of persistent data, XML, is stored in text files. These files store map information such as entities and their locations.

Game Loop

A game loop is an infinitely repeating sequence of operations that the server code enters after initialization. During an iteration of the loop, the server has certain tasks to perform, such as processing client requests and synchronizing all clients' information. However, before entering the loop, the server must preprocess some tasks during initialization.

When first started, the server enters an initialization phase. First, it reads in the configuration files for the database's location, port, and login credentials. It then initializes the database system. Second, it initializes a catalog of actions, which is a list of supported actions such as fire ball attack or healing. Similarly, it fills up a catalog of equipment, which describes which equipment slot certain items may occupy. Third, it initializes the game world by reading the XML configuration files mentioned above. All the maps are read and populated with entities. Portals that link maps are also resolved. Finally, the server reads in the communication configurations, sets up the communication system as described in Chapter 2, and starts listening for client connection requests.

Once initialization is done, the server enters the game loop. Within each iteration, the server processes its incoming message queues, creates outgoing messages to clients, and updates the database with the latest changes.

Usage Scenarios

Load Capacity

One type action user can perform is to load a capability to the action bar slot. After client code has determined that play wanted to load a capability, it will construct a `SET_ACTION_BAR_ENTRY` message and send to the server. This message type takes two parameters, a spell name and an action bar slot index. The server acknowledge the request and update the avatar's `actionBarEntries` at requested index with the spell name specified.

Shoot a Fireball

When client code has determined user's choice of target and type of action to be performed (firing a fireball), it sends an `ACTION_REQUEST` to the server. The server will determine the target based on the id provided and confirmed that it's a `Character` type of target. The server will load the `FireBallAction` and call `Activate` on

the requesting Character (source) and targeting Character (destination). Server will determine if the source Character has enough mana and the is both entities are within acceptable distance. If conditions satisfied, server will deduct mana from source, deduct health from destination, and adjust readyDeadline to provide cool down period for next action. It also send SendAnimationMessage message to both source and destination for graphical display of the fireball shooting.

Activate a Portal

When user activates a portal by clicking it, client sends an INTERACT_REQ message to the server. If the message contains a entity_id that points to a Portal, the server will send a SET_MAP message to client with details of position and orientation along with the name of the map.

Activate a Treasure Box

Similar to activating a portal, when the entity is the treasure box, server will determine if it's currently opened by another player. If not, it will prepare a list of CommEntity and send a OPEN_TREASURE_LIST message back

to client. It marks the treasure box as opened to lock out other player from accessing the box.

The server while making many interaction decisions among entities, it does not handle entity to terrain collision. The server, during this development state, is not to process graphical data other than 3-dimesional coordinate system that is use for positioning calculations.

CHAPTER FOUR
CLIENT ARCHITECTURE

Overview

In this chapter, we discuss the design of the client, which is separated into the packages: Mobius, MobiusClient, ClientSound and Gui. We'll see how MobiusClient makes use of other packages to carry out the graphics, sound and user interactions. Also the topic of memory management will be discussed.

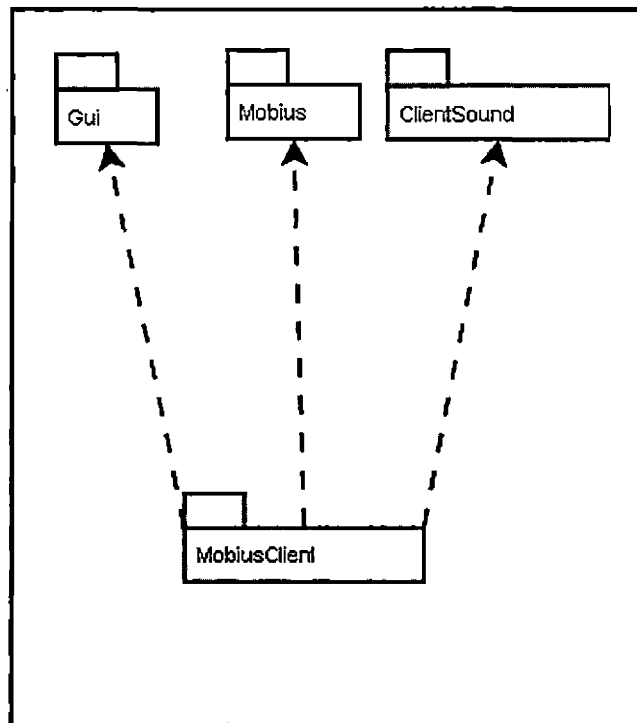


Figure 14. Client Architecture

As shown in Figure 14, the MobiusClient package is the central component of the client architecture. It contains the main execution method and the game loop. The ClientSound package handles 2D and 3D sound. The Gui package is responsible for the heads-up-display (HUD), which is the graphical user interface that handles interactions such as choosing a spell, putting on equipment or striking with a weapon.

The Mobius package is placed into a library to enforce stronger separation from the main client code. It loads and generates the 3D terrain and renders the 3D models comprising the game world. The Mobius library provides the APIs for usage and hides all details of how the rendering works. Missing from Figure 14 is CommunicationSystem, which was described in Chapter 2. This was omitted to simplify the diagram.

Initialization

During start up, the main method first initializes its memory with the server's information, such as address and port number. The predefined server details are stored in a local XML file called App.config. From the same XML file, it reads the screen dimensions and title of the

game screen. With the screen dimensions, it initializes the Simple Direct Media Layer (SDL), which is a free open source library that enables cross-platform development. The next item to initialize is the Mobius library. The Mobius library provides an API that renders game entities and special effects such as a fire ball animation. The Mobius library also needs detailed description of the entities and special effects, which can be found in `map_entities.xml` and `effects.xml` respectively.

Next on the list is the sound system in the `ClientSound` package. The sound system needs to initialize startup criteria such as music volume, sound volume and ambient sound volume. After the sound system is loaded, the GUI system follows. The GUI system needs to set up its model-view and projection matrices, and configure the view port. The client also needs to load a list of supported spells and initialize the spell system. The spell system contains a map that associates a spell with the function that handles its creation. Similarly, the buff system is loaded with functions that handle buff creation. (A buff is an effect that influences the state of a character over a relatively long period of time.) Finally, the client enters the pre-login state, calls the

state's activate method to prepare the login screen and sets up the update method to wait for a user login message. This completes the initialization phase of the client.

Game Loop

After initialization, the client program enters the game loop. In each iteration of the game loop, the client calls the current state's update method. This method calls another method through a delegate (a function reference) in order to process the current sub-state of the state. For instance, if the delegate is set to a function that waits for a specific response from the server, the function will be called until it receives and processes the response.

Next, MobiusClient calls the communication channel's update method to send out any pending messages that may have been created by the state's update method. It will also check the global update event queue and invoke any delegates that are registered into the queue.

After all game logic is processed, the client calls the Mobius library to render the latest snapshot of the visible 3D world. Then it calls the draw method of the

GUI system in order to render the visible components of the user interface. Finally, the sound system is called to allow the sound system to adjust playing sounds to simulate 3D sound effects.

After executing the game logic, displaying graphics and updating sounds, the game loop processes all the events that generated by the user. In other words, it handles user inputs. For example, if there is a delegate that was registered to the key down event queue and there is such an event detected by SDL, that delegate will be called. These events include key down, key up, mouse move, mouse button down and mouse button up. One exception is that the program will shut down if the user clicks the window exit icon.

Game Play State

Once the client is authenticated and the user chooses an avatar, the client will transition into the load map state. This state lasts momentarily and is followed by the game play state as soon as the client loads all graphics content for the current map. Upon initialization, the game play state accepts the avatar, the current map, and the heads up display (HUD) from the

load map state. It constructs a message processor that has the details of how incoming message types are handled. After initialization is done, the activate method is called. The activate method populates the entity dictionary, and moves the avatar to the initial position in the map, and makes the HUD visible.

Shoot a Fireball

During the game play state, the player will be invoking various capabilities of the avatar that he or she controls. One such capability is to shoot a fireball. Before a fireball attack can be performed, the fireball spell must be loaded onto the spell bar. The player needs to select the target to register it to the HUD and then activate the fireball attack by either clicking on the fireball icon in the action bar or pressing its corresponding number on the keyboard. When the user does this, the client will send the server an action request message that carries the target id and the name of the fireball spell. The server responds by sending a create action message (Figure 15).


```

internal static Action Create(CreateActionMessage msg)
{
    Character sourceCharacter =
        (Character)BuffSystem.Map.EntityIdDictionary[msg.SourceCharacterId];
    Character destinationCharacter =
        (Character)BuffSystem.Map.EntityIdDictionary[msg.DestinationCharacterId];
    FireBall fireBall = new FireBall(destinationCharacter);
    fireBall.effect = Mobius.Renderer.CreateEffect(effectName);

    Vec3f sourcePosition = sourceCharacter.Position;
    Vec3f destinationPosition = destinationCharacter.Position;
    sourcePosition.y += 4.0f;
    destinationPosition.y += 4.0f;
    Mobius.Vector3 mobiusSourcePosition = new Mobius.Vector3(
        sourcePosition.x,
        sourcePosition.y,
        sourcePosition.z);
    Mobius.Vector3 mobiusDestinationPosition = new Mobius.Vector3(
        destinationPosition.x,
        destinationPosition.y,
        destinationPosition.z);
    fireBall.effect.Start(mobiusSourcePosition, mobiusDestinationPosition);
    float distance = sourcePosition.DistanceFrom(destinationPosition);
    float secondsToImpact = distance / speed;
    fireBall.timeToImpact = TimeSpan.FromSeconds(secondsToImpact);
    Program.UpdateEvent += fireBall.Update;
    return fireBall;
}

```

Figure 15. Create Fire Ball Method

In order to render the fireball animation, the FireBall class needs the source and destination that mark the fire ball's path. The fire ball object maintains a reference to the destination character. The code calls the Mobius library to generate an effect. The position of the fire ball is adjusted 4 feet in the positive y direction so that the fire balls strikes the character target in it chest area. Once the fire ball is fired, the

delay (secondsToImpact) that takes the fire ball to reach its destination will be calculated using the distance between 2 characters and the speed of the fire ball. The estimated time of arrival will also be calculated and saved. The FireBall object then registers itself to program's update event queue; so that its update method runs in the game loop.

```
internal void Update(TimeSpan dt)
{
    Vec3f destinationPosition = destinationCharacter.Position;
    timeToImpact -= dt;
    if (timeToImpact <= TimeSpan.Zero)
    {
        PlayExplosion(destinationPosition);
        Destroy();
        return;
    }
    destinationPosition.y += 4.Of;
    Mobius.Vector3 mobiusDestinationPosition = new Mobius.Vector3(
        destinationPosition.x,
        destinationPosition.y,
        destinationPosition.z);
    effect.SetDestination(mobiusDestinationPosition);
}
```

Figure 16. Fire Ball Update Method

As shown in Figure 16, during a game loop update, the fire ball's update method will be called. Here, if the fire ball has reached its destination, it will play an explosion indicating the fireball has expired. The

update method will also call the destroy method in order to perform clean up, including deregistration from the program's update event queue. Otherwise, the position of the fire ball will advance according to its speed and destination. The cycle continues until the fire ball finally expires and destroys itself.

Memory Management

One important aspect that was considered when choosing a programming language for this game was its memory management capabilities. C# with .NET framework was chosen over C++ due to its automatic memory management capabilities. The .NET framework comes with a runtime environment called Common Language Runtime (CLR). It is Microsoft's implementation of the Common Language Infrastructure (CLI), an ECMA and ISO standard [2]. The CLR is a virtual machine with its own garbage collection facilities that automatically clean up unreferenced memory. In this project, minimizing memory management in the code was a big plus when there are many higher problems needed to be solved, such as communication, camera control, and physical interaction.

The Graphic Engine

The project's vision extends beyond Microsoft Windows. The project is built with future extensibility to Linux and Macintosh in mind, which is the reason SDL and OpenGL are used. OpenGL is encapsulated inside the Tao framework which exposes an OpenGL like API in C# syntax. The Tao framework is runnable under the Mono framework, another implementation of the CLI that can be run in Linux and Macintosh. In short, all programming and choices of third party library will need to be Mono compatible.

Switching back to memory usage, OpenGL libraries will interact with the graphic card directly [2]. Graphic card memory is not managed by the virtual machine. Therefore, graphic card memory management must be done in the code. For instance, in order to display a wooden box, it's necessary to follow these steps:

- 1) Read the graphic file that contains the wood texture.
- 2) Ask OpenGL to generate the texture and push it to the graphic card's memory.
- 3) When the texture is no longer used, OpenGL must be told to de-allocate the texture in the graphic card.

Consequently, a central concern is how to handle effectively this scheme of operation.

The Approaches

In general, there is a Texture class that is responsible for calling OpenGL and saving the texture's OpenGL handle as a member variable. During construction, a Texture object will call the OpenGL function to generate a new integer handle to identify the texture. The problem is now how to tell OpenGL that a texture is no longer needed.

In the first approach to cleaning up the texture, it relies on the Texture destructor to call to tell OpenGL to deallocate texture memory. However, this approach failed, because destructor is called by the garbage collector, which runs in a different thread. OpenGL functions need to be called from a single thread in order to work correctly. Consequently, when the garbage collector tries to deallocate texture memory, OpenGL throws an exception.

The second approach takes advantage of the IDisposable interface that C# provides. The implementor must implement the Dispose() method. This mechanism enforces a structured way of handling unmanaged objects,

which are objects that manage resources that are not automatically released by the garbage collector. More importantly, the thread that created the texture can call this method manually to delete the texture. When `Dispose()` is called manually, it flags the Texture destructor not to call `Dispose()` when being collected. This technique works as long as the programmer remembers to call `Dispose()` before the destructor does. It's cumbersome to have to remember to call `Dispose()`; therefore, another approach was developed.

In the third approach, there is a static, global list of integer handles to textures that need to be deallocated from graphic card memory. If a texture is no longer used, it will be added to the list by the texture destructors. The main thread (same thread that created the textures) will iterate through the list and deallocate each texture in each iteration of the game loop, which thus avoids violating the OpenGL policy. The benefit of this approach is focusing texture deallocation in one place. Also, the business of adding textures to the list of texture handles can be done in the destructor or at will.

The last approach raises one major concern that is

the issue of thread safety. As mentioned, the list of handles to textures to delete is accessible through the main thread and the garbage collection thread. Both threads are free to add and delete items in and out of the list at any time. The list is an instance of class List in C#. The class List is not thread safe [2]. A classic scenario of thread safety is one thread iterating through the list while the other thread is deleting items out of it. It's obviously a violation of the correctness of the program. To resolve this issue, some kind of mutual exclusion mechanism must be implemented to avoid the above scenario. Fortunately, C# provides a technique to handle such cases using locks. An example of its usage is given below in Figure 17.

```

//Thread 1 - clear the textures
:
lock(texturesToDelete)
{
    foreach (int textureName in texturesToDelete)
    {
        int name = textureName;
        Gl.glDeleteTextures(1, ref name);
    }
    texturesToDelete.Clear();
}
:
:
//End Thread 1

//Thread 2 - add texture to list
~ Texture()
{
    lock(texturesToDelete)
    {
        texturesToDelete.add(this.textureId)
    }
}

```

Figure 17. Using a Lock to Clean Up Textures

CHAPTER FIVE

DEPLOYMENT

Overview

In this chapter, we'll focus on the operating environments of the client, the server and the configurations necessary for them to communicate. We'll discuss about hardware, software, and network requirements. We'll walk through the process of compiling an installable setup package for the client program. We'll describe the organization of file structure in both client and server.

Create Client Setup Program

The following procedure applies to clients running Windows XP or a later Microsoft operating system. The procedure will generate a single MSI file that contains all the necessary libraries, executables, and data files. From the MSI file, the user can automatically install the client program. Once the program is installed, the user can uninstall the program easily using either a shortcut or Add and Remove Program user interface.

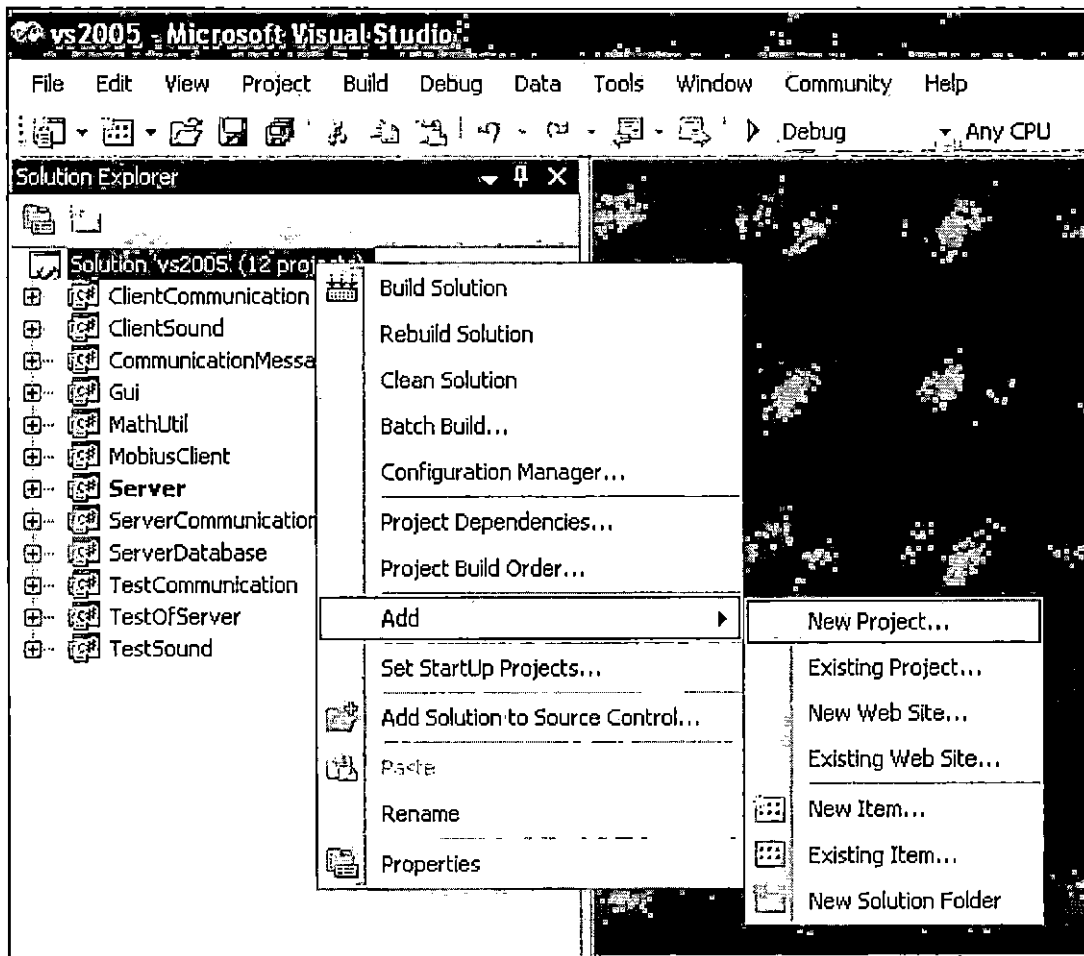


Figure 18. Add New Project

The first step in creating the setup file is by adding a Setup and Deployment project as shown in Figure 18.

1) Right click on solution icon, navigate to the label "Add" and click on the "New Project" label.

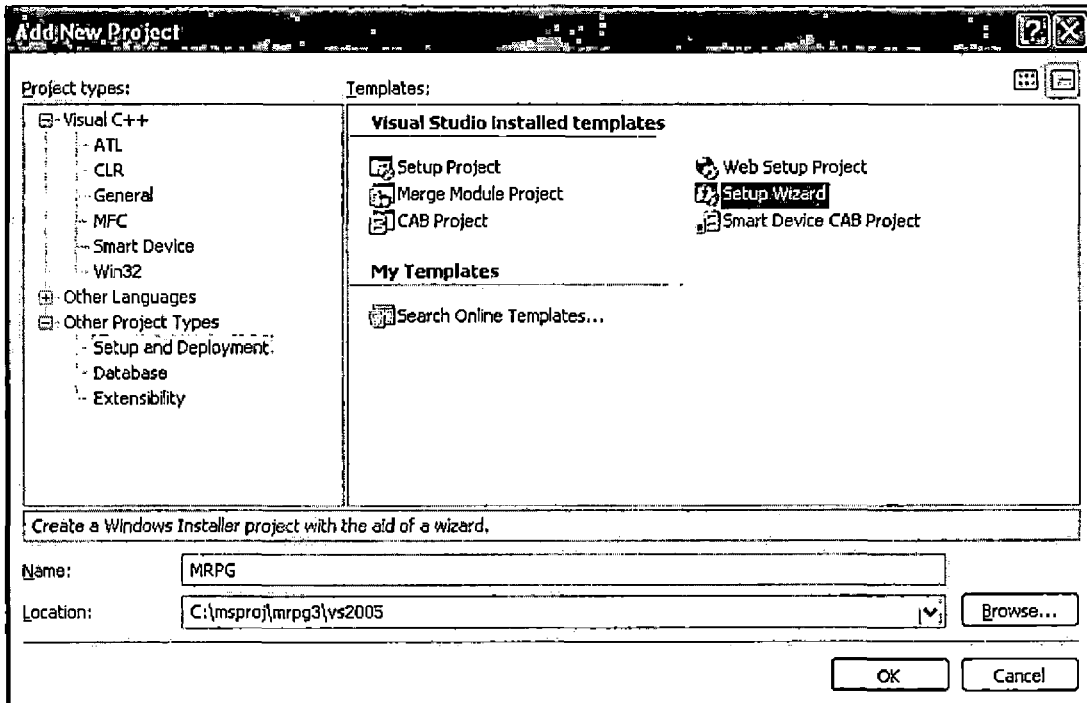


Figure 19. Add New Project Wizard

2) As in Figure 19, an "Add New Project" window will pop up. Select "Setup and Deployment" from the Project types box and "Setup Wizard" from the Template box. Enter the name of the program in the "Name" input box. In this case, it's MPRG. Click OK.

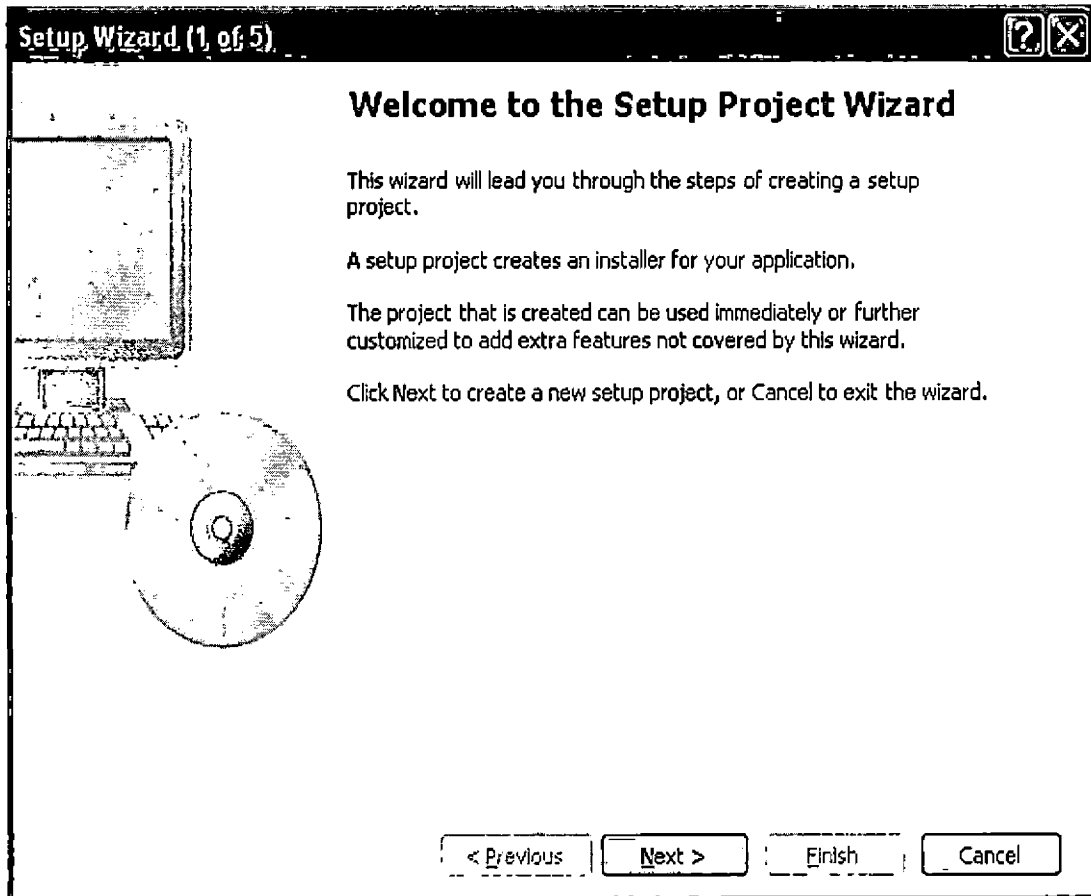


Figure 20. Setup Wizard 1

3) The "Setup Wizard" screen will popup as in Figure 20. Click "Next" to go to Setup Wizard step 2. In step 2, select "Create setup for a windows application" and click "Next."

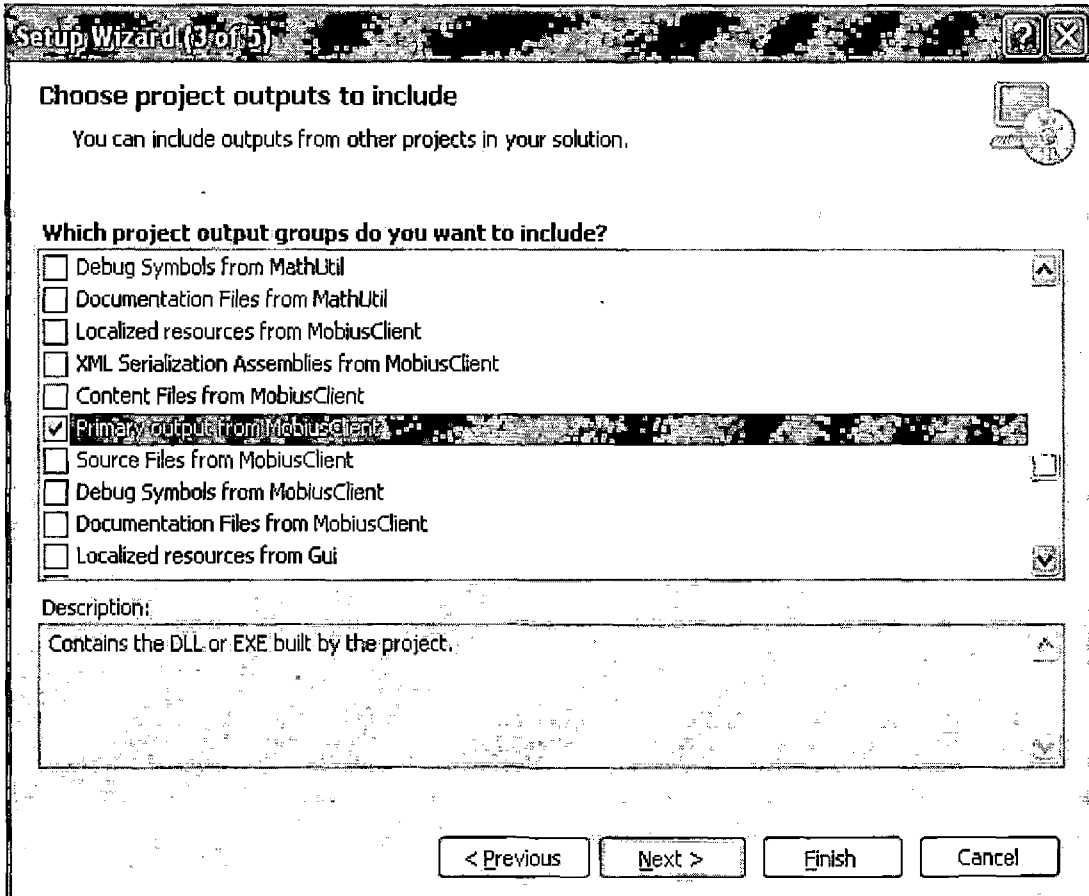


Figure 21. Setup Wizard 3

4) In Setup Wizard step 3 (Figure 21) check "Primary output from MobiusClient." Click "Finish". We skipped Setup Wizard step 4 (add additional files) because it's more efficient if done manually without the help of the wizard. At this point we have a File System window with 3 directories: Application Folder, User's Desktop, and User's Programs Menu. These directories present the client's installation directory, Desktop, and All Program

menu, respectively.

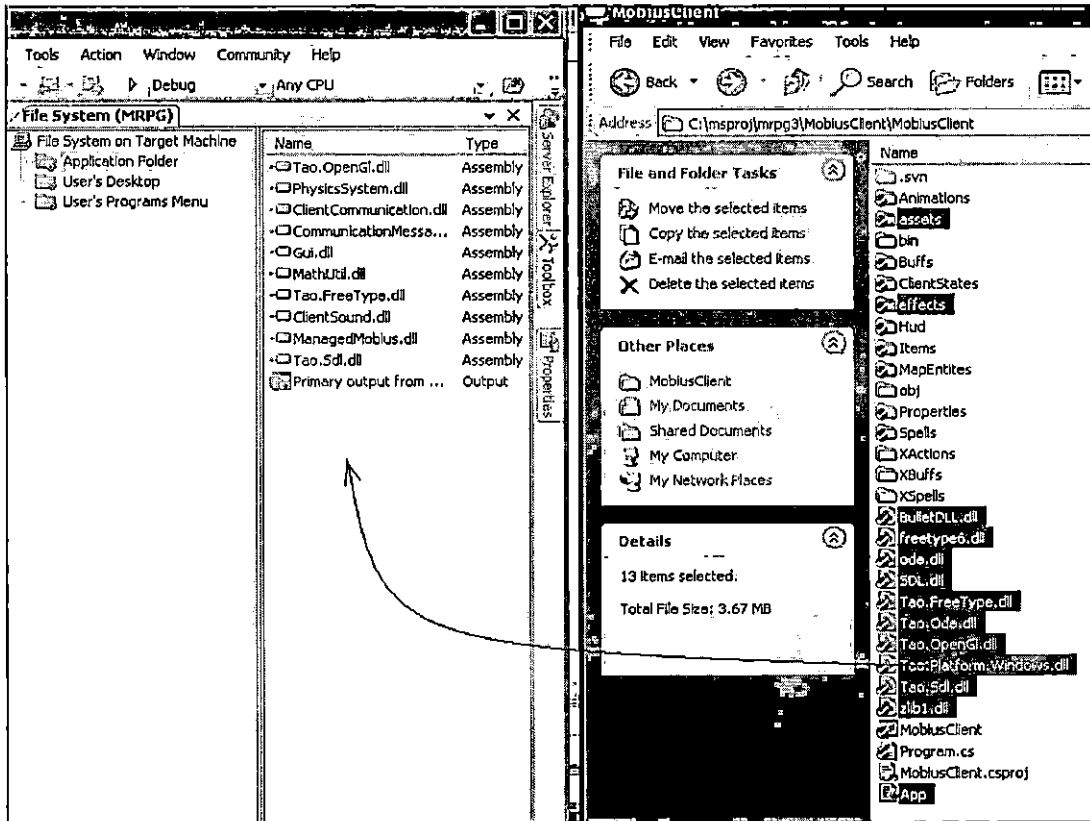


Figure 22. Drag and Drop Additional Files

5) In this step, as mentioned earlier, we'll add the necessary files manually from MobiusClient directory. Note that under Application Folder directory, there is a "Primary output from MobiusClient" file. This file is the execute file generated from MobiusClient project. Visual Studio 2005 also automatically imported 10 libraries, namely {list imported file names} that it thinks the

program will need. However, the program also needs {list file additional file names} libraries. Also, the program needs data files. Therefore, we need to add the assets and effects directories along with App.config file. We need to drag-and-drop these files to the Application folder as shown in Figure 22. It's a good idea to sort the Name column in the File System window to remove duplicates. Once done, the application folder should look like the one shown in Figure 23.

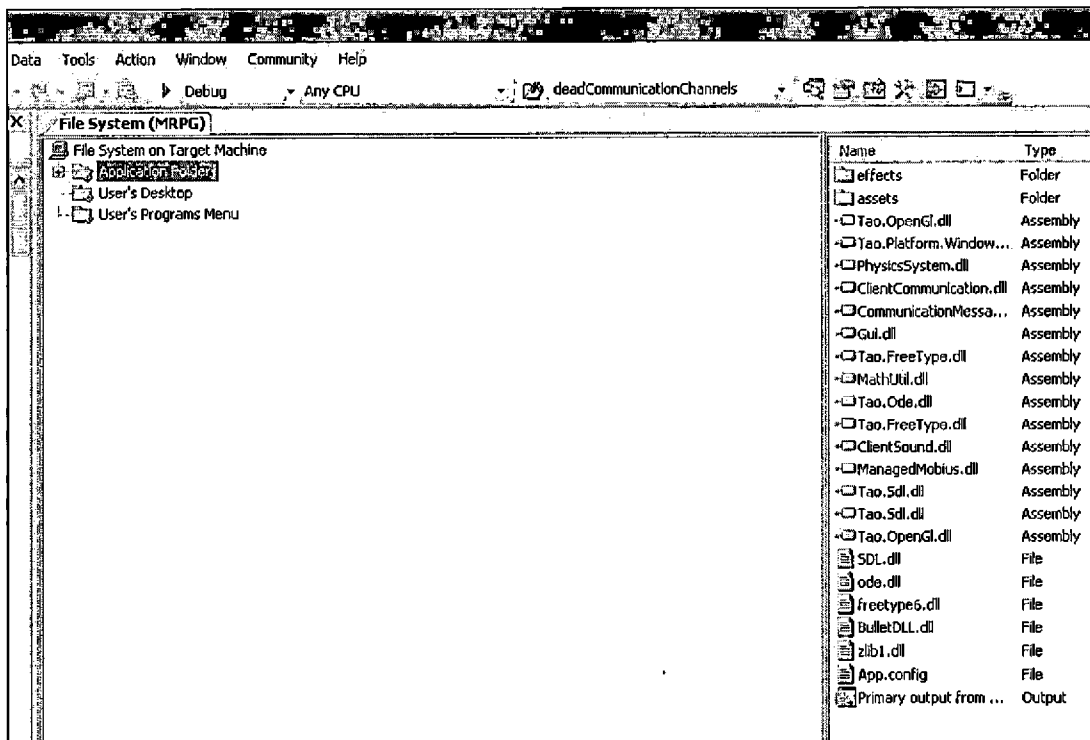


Figure 23. Setup Project File Structure

6) In this step, we'll create shortcuts to the executable and drop them to the user's desktop and Start Menu. Right click on file "Primary output from MobiusClient" and click on "Create Shortcut to Primary output from MobiusClient (Active)" menu item. This will result a new file called "Shortcut to Primary output from MobiusClient (Active)". Rename this file to a user friendly name such as MRPG. Copy the newly created file to User's Desktop directory in File System. Also, add a directory to User's Programs Menu called MRPG. Repeat the above technique to add another shortcut to directory MRPG under All Program menu.

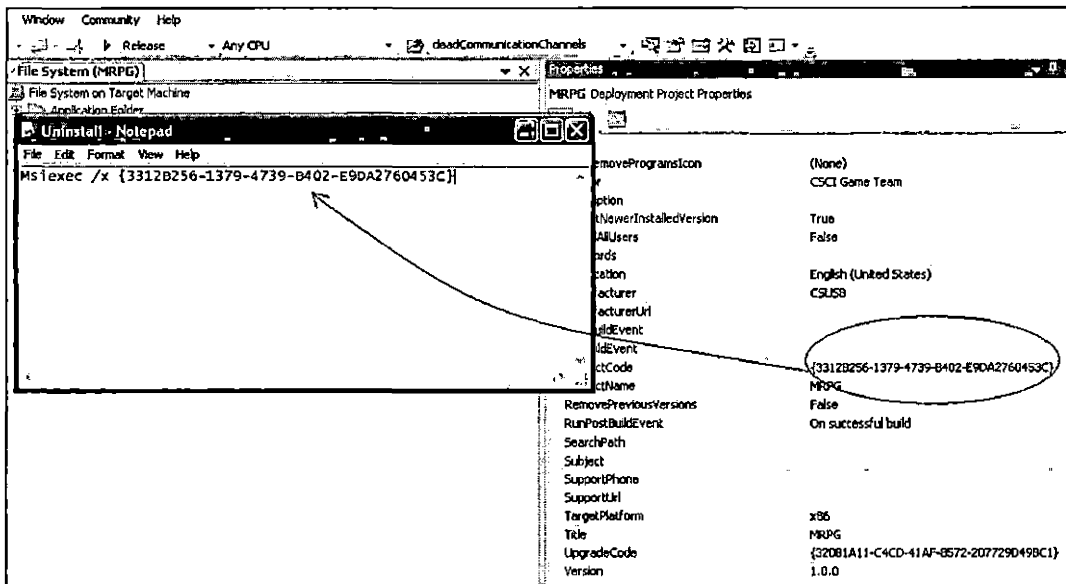


Figure 24. Create Uninstall.bat

7) This step is to create an uninstaller under the MRPG directory in the All Program menu. This is a third option to uninstall the program besides "Add and Remove Program" and the setup file. (Running the msi file while the program is already installed will prompt the option to uninstall the program.) Creating the uninstall file requires creation of a batch file; let's call it Uninstall.bat. As shown in Figure 24, use an editor such as Notepad to enter the following command without quotes and with a different 32 digit number: "Msiexec /x {3312B256-1379-4739-B402-E9DA2760453C}" As shown in the figure, The 32 digit number is in the command is the unique identifier of the program and can be obtained by viewing the Properties Window of the Setup project. When the command executes, the operating system will prompt user to complete the uninstalling process. Finally, attach the Uninstall.bat file to MRPG.

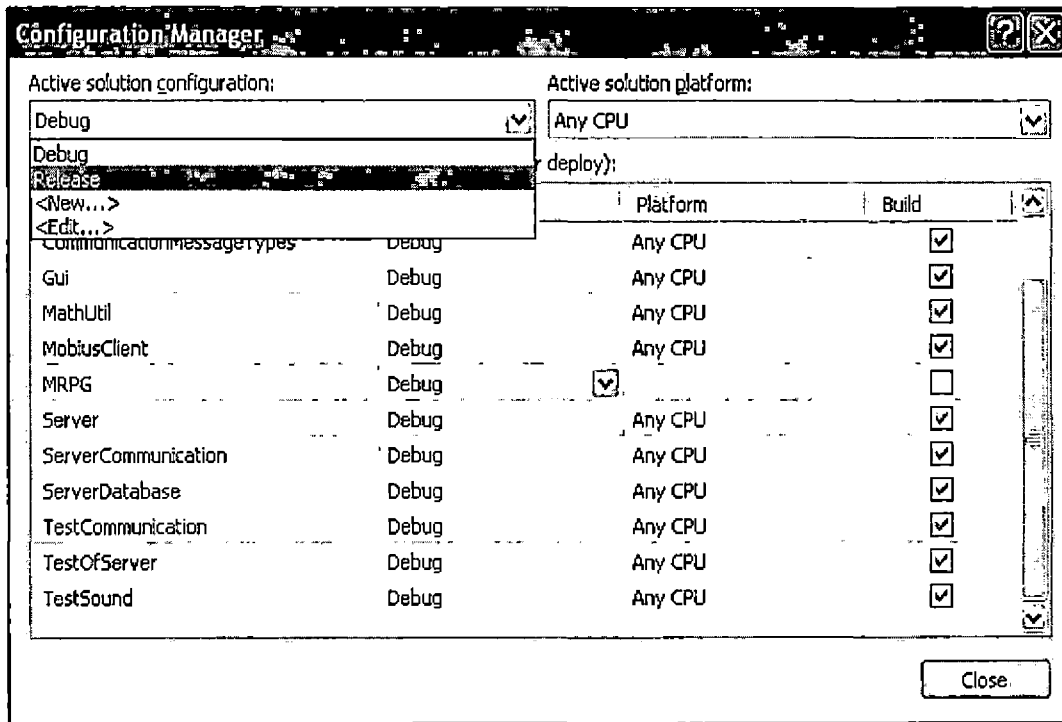


Figure 25. Project Build Mode Window

8) Before we build the Setup Deployment project, we need to configure the project to build in release mode. This can be done by accessing the Build menu and then Configuration manager. In Configuration manager window, change Active configuration from Debug to Release as in Figure 25. To complete the procedure, right click on Setup Deployment project name then click Build. Visual Studio will compile the project into MRPG.msi file under Release directory. This file is the end product of this section.

Server Deployment Architecture

Although it's possible to create a portable setup file for the server using technique described earlier, it might be uncommon, not necessary or inadequate. The server consists of a machine running server code and a remote MySQL database. As shown in Figure 26, the Sever is 2-tier. It uses an external MySQL database for data persistence data storing. The server also uses a file system to retrieve stored map files. These files reside in the server machine itself.

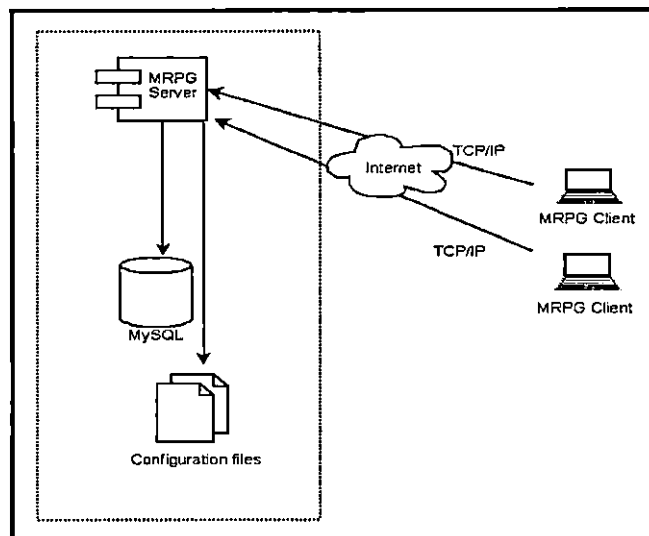


Figure 26. Server Deployment

The setting up process of the server involves first, setting up the database. For this project we used the

free MySQL distribution from Sun Microsystems. The database is named MRPG and protected with a username and password. The username and password are kept in the App.config configuration file of the server. The server communicates with the database through port 3306.

The server's executable code needs to be installed as a service. A service can be configured to start automatically after server reboot. The server machine need to open port 10008 for incoming TCP connections.

All clients communicate with each other through the Internet or Intranet via the server. The sever act as a medium for all communication during a game session. The 2-tier design is simple and straight forward. It's easy to maintain and more importantly, the database can adapt easily to other application architectures namely web-application for account creations and updates.

CHAPTER SIX

CONCLUSION

Current Progress

The latest iteration of the MRPG project includes many important features. These features together help define the backbone of the project. Generally, the game allows the user to login, to select a character, to see other users and chat with them, to move the character across maps through portals and to attack other characters. The project's internal structure is solid and well defined. Many sub-features have taken shape and are functional. First of all, the authentication system was completed to allow users to login and retrieve their game profiles that contain previously created characters from the server's database.

Furthermore, the chat system that gives users the ability to exchange text messages functions as intended. The character models are rendered correctly when examined from first-person and third-person views. The character can move anywhere in the game world and is bounded within the terrain boundary. The HUD is completed, featuring a spell book that contains spells such as a fireball attack and buffs. The HUD also gives the player an action bar of

ten available slots, the ability to view the player's current inventory, and a screen through which the player can choose equipment. The player can also interact with the items in the game world such as a treasure box. The player then can decide whether to keep the items in the treasure box. Mutually exclusive access to the treasure box instances is also carefully implemented. Players can enter portals to transit between maps. Finally, the ranged attack is also completed in conjunction with associated animations and 3D sounds. That completes the list of main features in the latest iteration.

Future Direction

Although most dominant problems in the MRPG are solved, many details were left out for the sake of completing the architecture. Furthermore, several lessons were learned during the implementation of the project. For instance, one suggestion for future development would be to rewrite the client program to use the XNA framework. Although the XNA framework has limited operating platform support (only MS Windows, XBOX and Zune), it provides a good deal of solutions to both graphics and physics problems of the virtual world.

Adapting to this framework will definitely improve programming throughput because developers can concentrate more on game logic features and less on the problem of technical barriers.

To make the game livelier, it is recommended to add animation states to the game characters, so characters can express through gesturing or facial expressions. For instance, the character can act out the gesture of throwing a fireball when using a fireball attack. Furthermore, a multiple channel chat system also increases player interactions making the game more enjoyable. The terrain also needs to be improved to contain more type of items in the game world, such as different types of buildings, plants, and animals. These are some of the small improvements that can make the game more realistic and enjoyable.

Beside the minor details that help promote game play experience, the improvements in game logic are important as well. Future development should develop a collision detection system that detects entity collision in the game world. For instance, a character should not be able to proceed if another entity or a large object is in its path; the character should step over or walk around the

obstacle to continue its path. Furthermore, there should be a system that handles AI for the non-playable characters. A non-playable character should behave close to ones controlled by a human player. It should have multiple levels of intelligence and aggressiveness. Finally, a quest system that defines the player's gaming experience through the use of game levels or point system in order for the game to quantitatively evaluate the player's accomplishment. This feature will give players a sense of being challenged thus making the game play more interesting. With these features applied, there is high probability that the project will gain greater momentum.

Comparing to other software domains, game programming is one of the more complex. Despite the lack of seriousness in the domain, game programming has very high demands. It requires thorough knowledge of the hardware platform, mathematics, physics and most importantly, it requires that the developers be always on top of the technology they are using. Therefore, besides serving as an entertainment medium, MRPG is a highly suitable project for future developers who wish to express their talents into the gaming world.

REFERENCES

- [1] Microsoft Software Development Network,
<<http://msdn.microsoft.com>>
- [2] Rabin S., AI Game Programming Wisdom 4.
Massachusetts: Charles River Media, 2008.
- [3] Rabin S., Introduction to Game Development.
Massachusetts: Charles River Media, 2005.
- [4] Shreiner D., Woo M., Neider J., & Davis T.,
OpenGL Programming Guide. Massachusetts: Addison-
Wesley Pub Co, fifth edition, 2006.