Architecture for a Massively Multiplayer Online Role Playing Game Engine

Sergio Caltagirone
scaltagi@up.edu

Bryan Schlief
bschlief@up.edu

Matthew Keys
mkeys@up.edu

Mary Jane Willshire, PhD
willshir@up.edu
503-943-6788

5000 N Willamette Blvd.
School of Engineering
The University of Portland
Portland, OR 97203-5798

Abstract

Faster networks, faster processors and 3D accelerator cards have contributed to the push for a new genre of online games, the Massively Multiplayer Online Role Playing Game, MMORPG. This paper presents a high-level software architecture for building a MMORPG engine. Six goals for the architecture are set, the architecture is presented and then examined to determine how well the goals have been met. The architecture blends in a unique way several classical software architecture patterns, using the strength of each to achieve the goals for the system.

Section 1 Introduction

Computer games form a large portion of the computing industry. Previously, games were either only single player or multiplayer in a turn-based format. These styles did not require an architecture that provides performance, wide-ranging functionality, or significant reliability and security; for example the game MiMaze [7]. The advent of faster networks, faster processors, and 3D accelerator cards has made possible real-time network-based multiplayer games. These games require performance, reliability, and advanced features in order to perform well in the marketplace [8]. To attain these qualities in a game, an architecture is needed which maximizes both network and client side efficiency [6]. The rising popularity of the Massively Multiplayer Online Role Playing Game (MMORPG) genre means that performance, reliability, security, and functionality now must be brought to a new level because of the high number of concurrent users, and the necessity of real-time graphics. This paper outlines an architecture that delivers these features. The architecture will fulfill six goals: minimize network traffic, provide opportunities for load balancing, provide a secure game playing environment, provide a high level of

scalability and maintainability, and maximize client side performance for real-time graphics.

This paper begins by addressing some of the challenges that are present in the problem of designing a MMORPG engine. Next, an overview of the architecture for the game engine is presented followed by sections that present lower level architecture details for the server and client-sides of the system. The paper concludes by showing how each of the six design goals has been addressed by the proposed architecture.

Section 2 Challenges Presented by Massively Multiplayer Online Role Playing Games

Massively multiplayer online role-playing games, MMORPG, represent the state of the art in computer game applications. Building the software for this class of game represents a collection of interesting technical problems. First, role-playing games require the creation of a customizable world. Secondly, these games are played online where one player's actions affect another player's game state. Thirdly, the world and the player characters must be persistent over time, which may consist of several years. Lastly, but by no means the least, the game must be playable by a large number of concurrent users. This last point requires that communication be very efficient and secure [8].

The user plays the game in a worldview. Each world has a consistent model of physical laws and rules that apply to all players within that world. During play, each player takes actions that have consequences that are appropriate to that world. Other players in the same game must be aware of the effects of actions taken by the rest of the players in the world. This constraint is true of all multiplayer games. One difference between these games and other multiplayer games is the fact that these are role-playing games. This means that each player takes on a customizable role within the game's world. The role adopted by a player defines and/or constrains what a given player may do during the game. Thus an architecture for a MMORPG must provide a world; the rules for that world (basic physics, the universe of possible actions and so on); and customizable roles for the players.

This genre of game is designed to remain active for long spans of time; a good game may be played for years. Players will enter the game, play for some period of time, and then temporarily leave the game. When a player returns to the game, his or her game state must persist. This includes the role that has been adopted, the player's position in the world and other state information that is pertinent to play.

Since the game is played online, users must identify themselves and be authenticated as legitimate players [2]. Only then will they be allowed to join the game. Online play makes it imperative for the game to act and react very quickly while minimizing the amount of network traffic generated in doing so. To avoid cheating, all communication must be secure [2, 8]. To achieve this speed of play, the architecture must specify which parts of the game will be resident on the player's machine and which parts will reside elsewhere (such as on one or more server machines). The need for performance and security is further complicated by the desire to allow large numbers of concurrent players. The architecture must be scaleable and easy to distribute over multiple processors [6].

Section 3 Architectural Overview for the MMORPG Engine

Two architectures are possible with multiplayer games (with some variations), centralized server and peer-to-peer (distributed). The architecture being presented is a centralized server, and was designed as such for five reasons, world consistency,

security, avoiding the game clock problem, simpler implementation, and a viable business model. The distributed architecture claims that it creates less network traffic [2], and prevents the server bottleneck. However, Cronin et al [5] claims the network traffic for a centralized architecture is roughly the same as that of a distributed architecture, whereas Diot and Gautier [6] claim the centralized architecture creates twice as much traffic. The difference may lie in network topology and an application "optimized for the client-server architecture" [5]. In addition, the server bottleneck problems can be overcome using a scalable architecture, as presented here. Therefore, the centralized server architecture presented here overcomes the opposition to a centralized architecture and provides all the advantages of the centralized architecture that are important in today's gaming.

The following sections present the major architectural patterns for building a game engine that would appropriately address the technical concerns discussed above and would fulfill the six goals of the project. The discussion moves from the highest level of abstraction to lower levels of abstraction.

3.1 Overall Architecture

At the highest level, the basic architectural pattern for the proposed MMORPG engine is a simple client-server model [1, 3, 4]. A small version of the game may consist of a single server connected over the Internet to one or more clients. This small implementation would limit the number of players and the size of the world to those whose speed requirements can be met with a single server model. A large version of the game would require multiple CPUs.

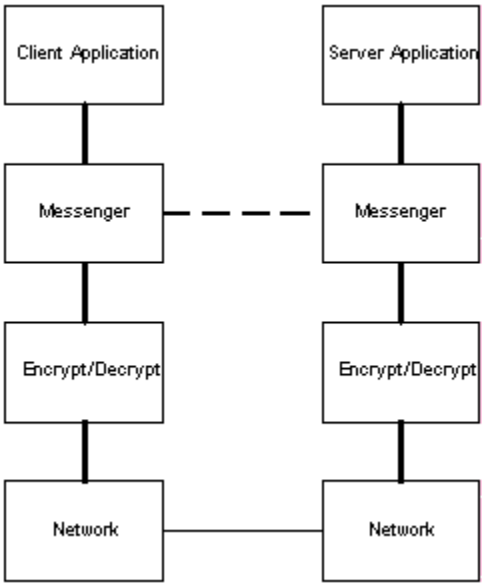

Figure 1. Layered System Architecture

The server and the client each have a layered architecture with the lowest layer handling communication over the Internet using standard Internet communication protocols. For security, the next layer is an encryption/decryption layer. To hide the complexity of the encryption layer from the game application layer, a communication or message layer is placed between the application layer and the encryption layer, as illustrated in figure 1. As might be expected, the top layer in this layered architecture

is the game application layer. It is by far the most complex layer and is discussed in more detail in the following sections.

3.2 The Game Application Layer

3.2.1 Publisher/Subscriber Pattern

Conceptually, the client-side application layer interacts with the server-side application layer in a publisher subscriber pattern [1, 3, 4]. Due to the nature of MMORPGs, a single server must service many clients; but to minimize the amount of communication traffic, the server will not automatically update all clients with all changes. Instead, clients will register as subscribers to the server. The server will receive changes from a given client, and then notify all other clients who are on the notification list. Those clients who are interested in the change will then request an update. Those clients who currently do not require the information, may opt to ignore the notification or to retrieve the changes at some later point in the game. The primary advantage of this pattern is that the publisher does not require detailed state information for each of the clients on a notification list. The clients maintain their own detailed state and it is their responsibility to decide if a given change is pertinent. This simplifies the logic and increases performance on the publisher side.

For example, suppose clients A, B, and C are currently active in the game and are on the same notification list. Client A makes a play that changes the state of the world. This change is sent to the server, the server makes the appropriate changes to its worldview and notifies B and C. B is currently playing in an overlapping part of the world so the change made by A is pertinent to B's worldview. B then requests the update from the server. Suppose, C determines, based on the content of the notification, that the change made by A is not currently pertinent to C's play. Therefore, C will not request the update. However, if in the future, A's play does become pertinent to C, then at that point the changes will be incorporated into C's worldview. As pointed out by Smed et al in [8], "..expression of data interest is called the aura or the area of interest, and it usually correlates with the sensing capabilities of the system being modeled."

Consider the following real-world example. Suppose the subscriber logic to update is whether A's play is in B or C's line-of-sight or directly impacts B or C. Say A is in the line-of-sight of B, but A is not in the line-of-sight of C (because of some mountains, etc.) then B requests the update, but C does not. Later, if either A or C move into the line-of-sight of the other, then A and C both request an update.

This approach is an attempt to make message content as small as possible, and to minimize the number of messages that must be passed to the clients by the server(s) by eliminating unnecessary updates.

3.2.2 Shared components

In order to minimize network traffic and exploit load balancing, some components of the game will be replicated in each client (i.e. the real time graphics), and some will reside on the server. Components such as player authentication will be completely on the server side of the architecture. These components are discussed below.

Section 4 Server Application Layer Architecture

We begin our discussion of the detailed MMORPG game layered architecture by considering the modules on the server side of our client-server model. Refer to figure 2 for the relationships among the components.

4.1 Governor, World Database, and World Module Relationship

If we were to imagine the Governor as the CPU of a Von Neumann Model computer, the World would be the RAM and the World Database would be the disk drive; the World module keeps a subset of the World Database in a direct access form. There are two reasons for separating the World and the World Database in this way: speed, and scalability. The information about the world that is relevant to the current server is kept in the World module. This allows for quick access and updates to the server's current game-state because function calls are much faster than database calls. The second reason is scalability. In the MMORPG genre, it is typical to have games that

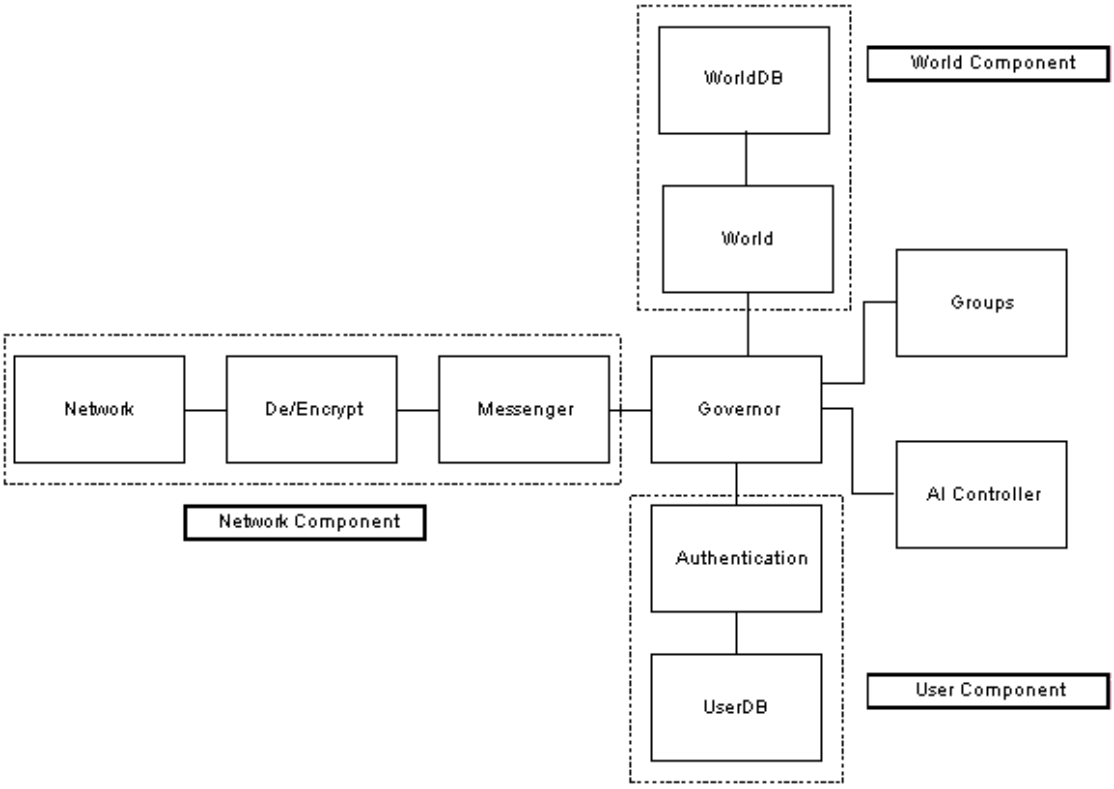

Figure 2. Server-side Architecture

require many servers; by abstracting the World away from the World Database, it is possible to have a dedicated server for each portion of the World. This allows for map extensions by adding more servers and database shadows. A cached blackboard architecture serves as the model for this module in order to increase performance for the system.

4.1.1 Governor - The central module for the server is the Governor. It is the blackboard controller [1, 3, 4]. The Governor is responsible for relaying messages between the different modules on the server side based on some logic. In addition, the Governor is in control of the primary game timer. The Governor can also act as a broker, in that it handles message routing to the players as required during game play.

There are five components that stem from the Governor: World, User, Groups, AI, and Network components.

4.1.2 World Component – The World component contains both the World Database and World modules. The World Database module holds all information about all objects in the entire world. The database connects to the Governor through the World. The World is a subset of the World Database information. This is the information that is kept in the cached blackboard. It allows the Governor to have quick and easy access to the world information without making database calls, which is costly for performance concerns. For increased maintainability as well as performance, this module abstracts all database calls from the Governor.

4.1.3 User Component – The User component contains both the Authenticator and the User Database modules. The Authenticator connects directly to the Governor. Based on business rules and information in the User Database, the Authenticator decides if the server should reject or accept a user. The module draws on information from the User Database. The User Database holds user data such as the username, password, real name, and game character information.

4.1.4 Groups Component – The Groups component contains the Groups module that stores information about the groups that a particular user belongs to. This module makes it possible for the Governor to send message to entire groups of users. The Governor is the publisher component in the publisher-subscriber architecture [1, 3, 4]. The users are represented in the notification lists maintained in this module. An alternate way of doing this is to keep the information in a database, however the database calls would slow the server down, therefore the optimal choice is to keep this information in the publisher's notification lists.

4.1.5 AI Component – The AI component contains the AI Controller module that is responsible for launching processes that are the AI Players. These AI players reside on the server. The AI player uses many of the same modules as does the client; however, instead of a View, Input, and Graphics module – there is an AI module. In addition, the De/Encrypt and Network modules are removed because the AI players are kept on local servers and therefore the packets can be trusted and secured by the server. This module allows for the abstraction of the maintenance of AI players from the Governor.

4.1.6 Network Component – The Network component contains the Messenger, De/Encrypt, and Network modules. The Messenger module is in charge of forming and sending messages. After the Governor has decided that it wants to send a message to the subscribers on its notification list, which ideally would happen as rarely as possible to avoid the bane of sending packets, a message is formed in the Messenger module. The Messenger module then passes this message on to the De/Encrypt Module. The De/Encrypt Module connects directly to the Network module. It encrypts and decrypts messages for travel across the Internet. This abstracts all De/Encryption algorithms and schemes from any other modules for ease of maintenance.

The Messenger also accepts decrypted messages and translates them to Governor calls. This module has the added responsibility of authenticating the validity of the form and data within the incoming message. The purpose of the Messenger is to provide a uniform interface for the server and client systems, as well as minimizing the number of packets sent over the network. The Network is

responsible for opening and closing connections, accepting connections, etc. This abstracts all communications from the Messenger and Governor modules. The details of this module will depend on the final selection of communication protocol.

Section 5 Client Application Layer Architecture

Next, we look at the components of the game that will be resident on each client machine. Efficiency is gained by delegating as much computation as possible to the client machine. See figure 3 for the relationships among the components of the client side architecture.

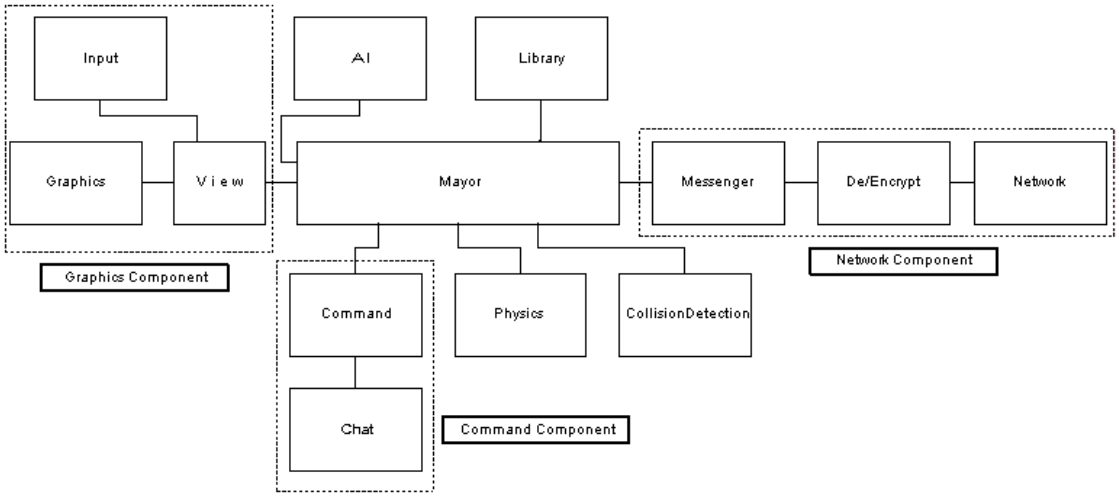

Figure 3. Client-side Architecture

5.1 Model-View-Controller Pattern

A pattern used in the architecture of the client side application layer is that of a model-view-controller (MVC) [1, 3, 4]. The model-view-controller pattern is well known in applications that rely on graphical user interfaces (GUI), user input, and logic. However, this architecture does not follow the traditional MVC model. Here, the link between the traditional model and view are eliminated to provide a persistent interface and increased speed of the system. By persistent interface we mean that if a player leaves a game at some point in the play, the interface state is preserved so that re-entry into play takes the player back to where the game left off. In this architecture, all module communication is routed through the Governor and Mayor (controller). All information, messages, and data will travel through central logic, thereby creating a more maintainable system. The details of this modified MVC model are discussed below.

5.1.1 Mayor – This is analogous to the server side Governor. A different term is used because the client-side Mayor is primarily responsible for local decision-making, where as the Governor is responsible for global decision-making. The Mayors are the knowledge sources for the blackboard on the server. The Mayor is responsible for passing messages and requesting information between all the client-side modules. It is also responsible for keeping track of the client side timer. The Mayor has seven components, which are the Library, AI, Graphics, Command, Physics, Collision and Network components.

5.1.2 Library Component – The Library component contains the Library, which holds the states of all the objects in the client's worldview. It is analogous to the World module on the server side. The Library module should only have to be loaded once, at user initialization, and the Mayor updates the module as many times as necessary when the client receives messages from the server to keep the client's worldview up-to-date.

5.1.3 AI Component – The AI component contains the AI module that acts as a player control for the pseudo-client process on the server. The AI module receives data from the Mayor as would the View, and responds to that data, based on some logic, the same way the View would respond as if the user had sent the module the input.

5.1.4 Graphics component – The Graphics component contains the View, Graphics and Input modules. The view is called by the Mayor to start setting up graphics calls for the Graphics module. The Graphics module renders the model to the screen. This module's purpose is to contain all platform or API dependent graphics calls for abstraction reasons.

The View also has the responsibility of translating the user input into messages that the Mayor can understand. This module abstracts all graphics, and input data from the Mayor. The Input module takes input from the user and passes it to the View, which determines what the input means and relays the appropriate message to the Mayor. The purpose of this module is to abstract the type of input from the user, be it keyboard, mouse, or joystick, etc.

5.1.5 Command component – The Command component contains the Command and Chat modules. The Command module abstracts certain user commands from the Mayor because they do not have any direct bearing on the game play. These commands could consist of calls to check the time (on both the client and server side), or ask how many hits the user has made, etc. The Chat module is responsible for handling specific types of commands, chat commands. The module is to parse the chat command, form the message, and pass it to the Mayor, which will send it to the server for distribution.

5.1.6 Physics component – The Physics component contains the Physics module that takes an object's state and determines the object's next state in the model based on programmer defined physics rules. This module abstracts all physics rules from the rest of the application.

5.1.7 Collision component – The Collision component contains the Collision Detection module that is responsible for determining that the Physics module does not juxtapose two objects. The module's purpose is to abstract all Collision Detection algorithms for maintainability reasons.

5.1.8 Network component – The Network component contains the Messenger, De/Encrypt and Network modules. Just as in the server, the client side Messenger forms messages that will be sent to the server for distributing, as well as translating messages from the server to the Mayor. It is also responsible for checking the validity of the form and data in the messages as they are received from the server after they are decrypted. This adds to the security of the system. Again, analogous to the server's De/Encrypt module, the client's De/Encrypt module is responsible for encrypting and decrypting messages that travel to and from the client. Lastly, the network module connects the client application to the physical network and handles packet transfer between the two. This encapsulates all network capabilities from the client, as the server Network module does for the server.

Section 6 Conclusion

The proposed architecture for a Massively Multiplayer Online Role Playing Game engine utilizes the blackboard, Model-View-Controller, client-server, and broker patterns [1, 3, 4] to achieve the six goals: security, maintainability, scalability, low network traffic, client application performance, and load balancing.

Security is achieved by encrypting and decrypting the packets sent between the client and server machines. In addition, the messenger ensures that messages from another machine are well formed with respect to form and data to prevent message manipulation.

Maintainability, scalability, and load balancing, are realized goals of the architecture because of the modularized components that logically separate application duties and responsibilities. Each server module could, in theory, be hosted one or more machines, which provides for excellent scalability and load balancing. Since the architecture is modularized, any module could, also in theory, change (as long as a public interface wasn't modified) without the knowledge of any other portion of the application.

However, the key to a successful networked game is connectivity. Our paradigm addresses the double-headed monster of keeping many players concurrently connected while maximizing client side performance. This is accomplished by the fact that our architecture provides for extensive load balancing because as much computation as possible is passed off to the client machine. Using a combination of the blackboard architecture and the publisher-subscriber architecture reduces the client's overall load. This combination has the clients, acting as the knowledge sources, updating the blackboard; but the blackboard controller is also a publisher that notifies the clients of changes. The clients then only get updates when necessary. The client is designed in a fashion that the Mayor will handle all intercommunications; data can be passed quickly and efficiently to the necessary modules for quick computations.

This MMORPG engine architecture addresses all the incredibly difficult technical issues that surround this genre, as well as other applications that are similar in scope, and provides a suitable framework in which to solve these problems while presenting users with an unparalleled experience; which is the goal primary goal of all MMORPG designers.

References

[1] Bass et al. *Software Architecture in Practice;* Addison Wesley, 1998.

[2] Baughman, Nathaniel and Levine, Brian. Cheat-Proof Playout for Centralized and Distributed Online Games. In *Proceedings of the Twentieth IEEE Computer and Communication Society INFOCOM Conference.* IEEE, April 2000.

[3] Bosch, Jan. *Design and Use of Software Architectures;* Addison Wesley, 2000.

[4] Buschmann et al, *Pattern-Oriented Software Architectures;* Wiley, 1996.

[5] Cronin, Eric, et al. A Distributed Multiplayer Game Server System. Not yet published. http://citeseer.nj.nec.com/cronin01distributed.html

[6] Diot, Christophe and Gautier, Laurent. A Distributed Architecture for Multiplayer Interactive Applications on the Internet. www.eecs.umich.edu/~bfilstru/quakefinal.pdf

[7] Lety, Emmanuel et al. MiMaze, a 3D Multi-Player Game on the Internet. INRIA, INSTITUT NATIONAL DE RECHERECHE EN INFORMATIQUE ET EN AUTOMATIQUE, September 1997.

[8] Smed, Jouni et al. Aspects of Networking in Multiplayer Computer Games. In *Proceedings of the International Conference on Application and Development of Computer Games in the 21st Century. November 2001.*