A Distributed Server Architecture for Massively Multiplayer Online Games

Nadeem Khan

Master of Science

School of Computer Science

McGill University

Montreal,Quebec

2006-08-31

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Master of Science.

Your file Votre référence
ISBN: 978-0-494-32729-6
Our file Notre référence
ISBN: 978-0-494-32729-6

DEDICATION

This thesis is dedicated towards the effort for spreading peace and tolerance among all human beings.

ACKNOWLEDGMENTS

I would like to thank my supervisor, Professor Bettina Kemme, for all her guidance, ideas and continuing support throughout my research without which the completion of this thesis would not have been possible. I am grateful to my family and especially my parents for their complete and undying love, support and belief in my abilities and to whom I owe all my academic achievements. I would also like to thank the entire Mammoth group and especially Jean-Sebastien Boulanger, for his valuable input and assistance during my research. A special thank you to Omar, Miriam and Namir for their friendship and moral support during the good as well as the difficult times. I also want to thank the staff at the School of Computer Science help desk for providing me necessary technical support to run my elaborate experiments. Finally, I would like to thank Professor Athena Voulumanos and Professor Kris Onishi, at the McGill Infant Research Group for providing me an opportunity to work with them which gave me the vital financial support during my stay at McGill.

ABSTRACT

There has been a tremendous growth in the popularity of Massively Multiplayer Online Games (or MMOGs) with millions of players interacting in their virtual game space at the same time. However, the centralized server architecture of most modern day MMOGs is unable to cope with this increase in the number of participating players. Hence, there is a need for a scalable network architecture which can support these large number of players without affecting the overall gaming experience for each player. In this thesis we propose a scalable distributed server architecture which divides the virtual game space in smaller sub spaces and assigns them across a cluster of server nodes thereby reducing the overall load per server. It is based on a distributed publish/subscribe architecture which takes care of client-server as well as server-server communication. We discuss the implementation of this architecture in a real MMOG and experimentally prove that it shows better scalability than the centralized server architecture.

ABRÉGÉ

La popularité des jeux en ligne massivement multijoueurs (MMOGs) a augmenté de façon phénoménale et ces espaces de jeu virtuel comptent maitenant des milliers de joueurs qui interagissent en temp réel. Cependant, l'architecture centralisée des serveurs de la plupart des MMOGs modernes est incapable de supporter cette augmentation constante du nombre de joueurs. De ce fait, il y a un besoin pour une architecture réseau extensible qui peut supporter un nombre croissant de joueurs, sans toutefois affecter leur expérience de jeu individuelle. Dans ce mémoire, nous proposons une architecture serveur distribuée qui s'adapte pour supporter un nombre accru de joueurs. Notre architecture distribue la charge globale du serveur en subdivisant l'espace de jeu virtuel en plusieurs sous-espaces qui sont attribuś à différent noeux de réseau. Notre approche est basée sur une architecture distribuée publication/souscription qui prend en charge les communications client-serveur et serveur-serveur. Nous présentons l'implémentation de cette architecture dans le contexte d'un vrai MMOG et nous démontrons expérimentalement que le serveur distribué que nous proposons s'adapte mieux à une population croissante de joueurs que le serveur centralisé.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGMENTS iii

ABSTRACT iv

ABRÉGÉ v

LIST OF TABLES x

LIST OF FIGURES xi

1 Introduction 1

2 Massively Multiplayer Online Games 4

2.1 Introduction to Massively Multiplayer Games 4
2.2 Network Architecture 6
2.3 Interest Management 7
2.4 Introduction to Mammoth 8
2.4.1 The Game 9
2.4.2 Interest Management in Mammoth 10
2.4.3 Software Architecture for Mammoth 10
2.5 The Mammoth Network Architecture 12
2.5.1 Server Messages 13
2.5.2 Logical Channels 13
2.5.3 Communication Protocols 14

3 Scalability via Distribution 18

3.1 Limitations of the Client/Server Architecture 18
3.1.1 Scalability and Latency 18
3.1.2 Effect of Latency on Playability 18
3.1.3 The Scale of Present Day Games 19

3.2 Efforts to Improve Performance 21
3.2.1 Multiple Game Instances 21
3.2.2 Mirrored Server Architecture 21
3.2.3 The Distributed Architecture 22
3.2.4 Distributed server architecture vs. Peer-to-Peer based approach 31

4 The Distributed Server Architecture - An Overview 34

4.1 Architecture 34
4.1.1 Overview 34
4.1.2 Major Components 35
4.1.3 Administrative Components 37
4.2 Communication Protocol 39
4.2.1 The Challenge 39
4.2.2 Distributed Publish/Subscribe System 40
4.3 Game Communication Protocols 42
4.3.1 Player Entry 42
4.3.2 Player activity within sub space 42
4.3.3 Player migration across sub spaces 42

5 Implementation in Mammoth 44

5.1 Architectural Components 44
5.1.1 Server Nodes 44
5.1.2 Clients 47
5.1.3 Cluster Management with CAS 49
5.2 Communication Mechanisms 51
5.2.1 Distributed Publish/Subscribe System 51
5.2.2 Description of the publish subscribe system 51
5.2.3 Game State Retrieval 56
5.2.4 Player Migration - A detailed explanation 57
5.2.5 Consistency during player migration and startup 61

6 Experimentation and Results 65

6.1 Introduction 65
6.2 Experimental Setup 65
6.2.1 Physical Setup 65
6.2.2 Player behavior 66

6.2.3 Game Setup and Experimentation Technique 67
6.2.4 Measurement of Parameters 68
6.3 Single Server Experiments 69
6.3.1 CPU Utilization 70
6.3.2 Latency 71
6.3.3 Bandwidth Utilization 72
6.4 Distributed Server Experiments 72
6.4.1 CPU Utilization 73
6.4.2 Latency 74
6.4.3 Bandwidth and Message Statistics 75
6.5 Effect of Varying Cluster Size 80

7 Integration into the Distributed Object Model 85

7.1 Introduction 85
7.2 Publish-subscribe mechanism 86
7.3 Implementation in Mammoth 87
7.3.1 Matching and subscription 88
7.3.2 Invalidation 88
7.4 Cell based distribution 88
7.4.1 Cell Match Function 89
7.4.2 Example 90
7.5 Integration of the distributed server architecture. 92
7.5.1 Sub spaces 92
7.5.2 Match policy 92
7.5.3 Cells and the Cell Match function 93
7.6 Communication Scenarios 93
7.6.1 Scenario 1 - No cell match 94
7.6.2 Scenario 2 - Cell match 95
7.6.3 Scenario 3 - Match Policy 96
7.6.4 Scenario 4 - Player Migration 96
7.6.5 Player Disconnection 98
7.7 Dynamic Objects 98
7.8 Refresh Interval 98
7.9 Formal Algorithms 99
7.10 Applicability of the distributed server architecture 103
7.11 Performance Discussion 103
7.12 Advantages/Disadvantages over the Mammoth Publish/Subscribe Scheme 104

8 Conclusion and Future Work 106

8.1 Conclusion 106
8.2 Future Work 107
8.2.1 Integration of Distributed Server Architecture in the Distributed Object Model 107
8.2.2 Load Balancing and Dynamic Sub Space Management . . . 107
8.2.3 Fault Tolerance and Backup 108
8.2.4 Extension to Peer-to-Peer architecture 108
8.2.5 Wide Area Network (WAN) Experiments 109

References 110

LIST OF TABLES

Table page

6–1 Statistics for messages received from clients. 77

6–2 Statistics for messages received from other server nodes. 78

6–3 Player Migration statistics. 78

LIST OF FIGURES

Figure		page
2–1	Client/Server Communication Paradigm	7
2–2	Interest Management using sub spaces	8
2–3	The Mammoth Game World	9
2–4	Mammoth Software Architecture	11
2–5	Logical Channels in the Mammoth Network Engine	13
2–6	Game Startup Communication Protocol	15
2–7	State Update Dissemination	16
2–8	Player transition across static zones	16
3–1	A Generic Distributed Server Architecture	23
3–2	Flocking of players to a game hot spot	24
3–3	The Matrix Architecture	25
3–4	P2P Based Architecture	28
3–5	Voronoi Partitioning	30
4–1	The Distributed Server Architecture	35
4–2	Working of Rendezvous Server	38
4–3	Simplified Overview of Distributed Publish/Subscribe Mechanism	41
5–1	Distributed Network Engine for Mammoth	46
5–2	Cluster formation procedure	50
5–3	Cluster Management Layer	50

5–4 Loss of updates leading to incorrect game state 62
5–5 No loss of updates 63
6–1 Latency Measurement 69
6–2 Single Server CPU Utilization and Latency 71
6–3 Single Server Bandwidth Consumption 73
6–4 Mapping Configurations 73
6–5 Scalability Experiments for CPU Utilization 74
6–6 Direct and Indirect Latency for 4 Server Node Cluster 75
6–7 Scalability Experiments for CPU Utilization 76
6–8 Bandwidth Comparison 77
6–9 Sub Space - Server Mapping 80
6–10 Scalability Experiments for CPU Utilization 81
6–11 Bandwidth Comparison 82
6–12 Messages received from clients 83
6–13 Messages received from other server nodes 84
7–1 Single Server Duplication Mechanism 86
7–2 (a) Step 1 - Cell Match 90
7–3 (b) Step 2 - Duplication to Server Node 91
7–4 (c) Step 3 - Match and Duplication to Client 91
7–5 Scenario 1 - No Cell Match 94
7–6 Scenario 2 - Cell Match 95
7–7 Scenario 3 - Match 96
7–8 Scenario 4 - Player Migration 97

CHAPTER 1
Introduction

Computer games have become a widely popular source of entertainment among all age groups. The advent of the World Wide Web ushered in a new genre of computer gaming called *online multiplayer games* which provides a platform for gamers all over the world to come together and play with each other. During the recent years, the popularity of online multiplayer games has reached unprecedented heights and approximately 100 million people are expected to be playing online games by 2008 [22].

Massively Multiplayer Online Games or MMOGs are a popular form of online multiplayer games where hundreds or even thousands of players interact with each other in a virtual game world. World of Warcraft [12], a popular MMOG, has over 2 million registered users with a recorded maximum of half a million players interacting at the same time. With the widespread availability of broadband internet connectivity, the number of participating players are increasing rapidly further raising the popularity of these games.

The rapid growth in the number of players participating in an MMOG poses various technical challenges to their network architecture. The most important is that of scalability, i.e., the ability to support increasing number of players without adversely affecting performance. Traditionally, MMOGs have used a centralized server architecture with players connecting to a single game server which handles

the entire game world. However, due to the massive scale of these games, a single machine is not able to support the load generated by the hundreds or even thousands of participating players. Thus, there is a need for an efficient architecture which can support the load generated by these growing number of players without affecting the gaming experience offered to the player.

We propose a distributed server architecture that divides the virtual world into smaller sub spaces and distributes them across a number of server nodes (known as the *hosts* of these sub spaces). A client connects only to the server node which hosts the sub space the player currently resides in and sends all the updates directly to this server node (known as *home server*). A server node processes all the received updates and sends these to its clients which are interested in them. A client switches home server as its player moves over to a sub space hosted by another server. A client can also be interested in updates occurring at sub spaces that are not hosted at its home server. A distributed publish/subscribe mechanism is implemented that guarantees that a client receives all updates it is interested in.

In contrast to many other research proposals in this area, which have used simulations to test their approaches, we have implemented the distributed server architecture into the Mammoth MMOG Prototype. The experimentation performed on our architecture is done over a real network using complete client implementations to gather performance results. The parameters that we measure are the CPU and bandwidth utilization at the server nodes and the latency experienced by the players during game play. In each instance, the results show that the distributed server architecture shows the desired scalability properties that it was designed for.

Finally, we introduce the distributed object model which was adopted as an alternative scheme for state and interest management in Mammoth during the course of the development of our architecture. We show how the concepts of our distributed server architecture can be integrated into the new version of Mammoth.

This rest of the thesis is organized as follows. Chapter 2 introduces MMOGs, explains a typical single server architecture and the details of Mammoth's client/server architecture. Chapter 3 explains the problem of scalability in the single server architecture and discusses existing solutions to solve it with the help of distribution. Chapter 4 introduces the new distributed server architecture and gives a brief overview of its components and communication mechanisms. Chapter 5 then explains each of these components and their implementation in Mammoth in detail and gives formal algorithms for the communication protocols used. The results of the experiments performed on our architecture are presented and discussed in Chapter 6. Chapter 7 discusses the distributed object model and shows how our distributed server architecture can integrated in it. Chapter 8 gives the conclusion and outlines the proposed future work on the distributed server architecture.

CHAPTER 2
Massively Multiplayer Online Games

2.1 Introduction to Massively Multiplayer Games

Massively Multiplayer Online Games (MMOGs) are a genre of computer games in which a large number (usually thousands) of players share the same game world at any instant of time. This sheer scale of the number of players involved in MMOGs distinguish them from other network based online games.

A sub-category of MMOGs are MMORPGs or Massively Multiplayer Online Role Playing Games. Players in an MMORPG can take control of the characters in the game and perform actions such as move around in a *virtual world*, interact with other players, pick up objects and take part in missions or quests. Most MMORPGs also allow their characters to grow, trade currency or points with other players, have an inventory of items and accumulate experience points (based on their expertise in the game). Players can also form teams which work together and/or compete against each other. Everquest [13], UltimaOnline [4] and World of Warcraft [12] are examples of popular MMORPGs.

Another sub-category of MMOGs are *first person shooter (FPS)* games which are characterized by an on-screen display based on the point of view of the character currently playing the game. Examples of popular FPS games are Doom [19], Quake [20] and Unreal [17].

The *game world* or *virtual world* of a MMOG is generally made up of a large number of various types of objects. Objects in the game world can be broadly classified into the following types (according to [23]):

1. **Static Objects** : These are *immutable* objects, i.e., their properties do not change during the entire course of the game. A common example of static objects are stationary terrain elements such as trees, lakes, buildings, walls and roofs.
2. **Dynamic Objects** : Also known as *mutable*, these kind of objects have a *state* associated with them. A *state* of a mutable object can be any property which can be subject to modification during the course of a game. For example, the state of a bottle of wine can be the amount of wine present in it. In many games, items such as apples, books and flowers can be picked up or dropped by players and can change location.
3. **Player Characters (PCs)** : These are the characters controlled by the players. The state of a player is usually determined by its position in the game world. However, there can be other game dependent properties which can determine the state of a player such as experience, health and inventory items.
4. **Non Player Characters (NPCs)** : These are characters which are not controlled by the players but rather by automated algorithms. Most NPCs are similar to Player Characters in terms of role and serve as either opponents or alliances to the Player Characters.

The state of the game at any instant is decided by the state of all the individual objects present in the game world at that instant. These states can be altered by actions performed either by or on these objects. Such actions can be categorized as :

1. **Change in Position** : When a PC or NPC moves around the game world, it alters its position as well as the state of the world.
2. **Player - Player Interaction** : Two PCs or NPCs interacting with each other (for example, talk, engage in a fight and exchange currency) can lead to the change in either of their states.
3. **Player - Object Interaction** : Actions of players such as picking up, dropping or consuming objects modify the states of both the player as well as the object involved.

2.2 Network Architecture

The most common network architecture to support MMOGs so far has been the *client/server* architecture (used in popular MMOGs such as Quake 1 [20] and StarCraft [11]).

Here, the server (commonly referred to as the *game server*) acts as centralized component which maintains the state of all the objects present in the game world (known as the *game state*), while a client hosts the user controlled in-game player object (or a PC) and manages its state updates. In order to start playing, a client connects to the game server to retrieve the latest game state and *caches* it locally. After that, all actions performed at the client are converted into state updates and are sent to the game server in the form of messages. The server receives such updates from all the clients connected to it, serializes them, processes them to create

a response and multicasts this response back to all the clients. A client updates its local cache by applying the responses received from the game server and renders the game state using this cached information. Figure 2–1 shows the messaging scheme in common client/server based multiplayer games.

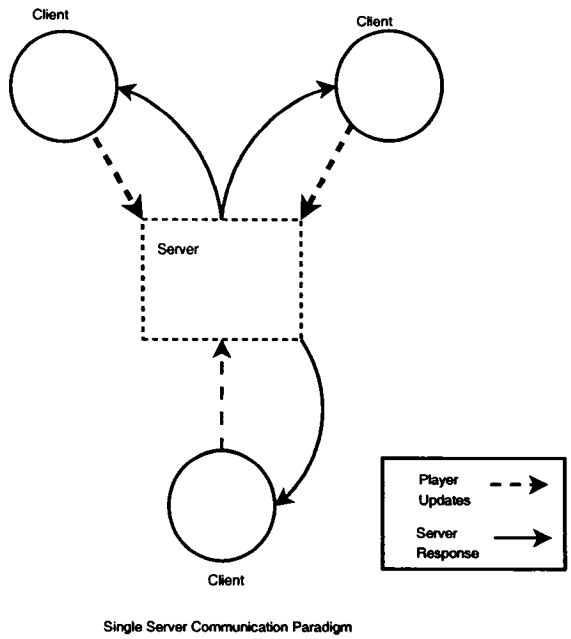


Figure 2–1: Client/Server Communication Paradigm

2.3 Interest Management

In most MMOGs, a player has limited visibility of the entire game world. This is known as its *area of interest* or 'sphere of interaction' [18] within the game. Therefore, the game server needs to send a client the state updates that are only relevant to the area of interest of its player. This technique is known as *interest management.* Interest management reduces unnecessary resource consumption (such as bandwidth consumption) while still maintaining adequate interactivity in the game. Most MMOGs

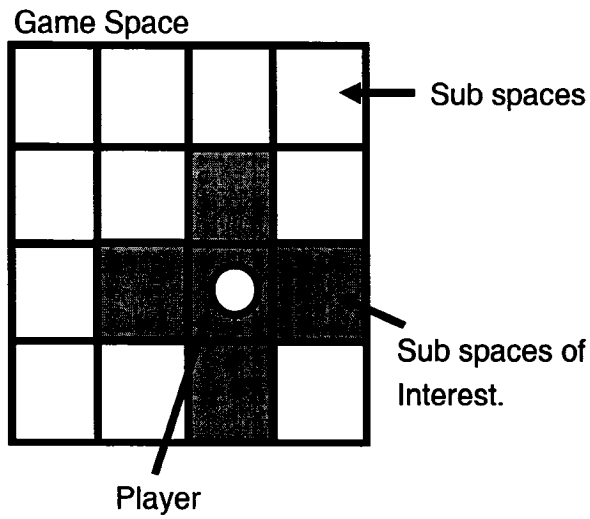


Figure 2–2: Interest Management using sub spaces

implement interest management by partitioning the game world into multiple *sub spaces*. The game server sends only those updates to a client which occur on sub spaces which are under its area of interest. Figure 2–2 shows an interest management scheme where the game world is divided into rectangular sub spaces with a player interested in the sub space it is currently in and its neighbors.

2.4 Introduction to Mammoth

We now introduce the Mammoth development framework which is used to implement an MMOG based on the client/server architecture described above. The Mammoth project is an attempt to develop a MMOG in the java programming language which can be used for conducting academic research by providing a framework in which researchers can implement and experiment with different algorithms to address the issues posed by MMOGs.

2.4.1 The Game

Mammoth implements a role playing game based on the *client/server* architecture where the player sees the virtual game world from above (also known as the *god view*) (see Figure 2–3). Players can walk around the game world which consists of static objects such as buildings, trees and walls. They can also pick up, move or drop objects scattered across the world (for example food items, books or other items which can be designed and added to the world). Each player can accumulate fame or currency points and has associated with it an inventory of items that it has picked up. The Mammoth world is a *persistent* world, i.e., the world continuously evolves with time.

Figure 2–3: The Mammoth Game World

2.4.2 Interest Management in Mammoth

The Mammoth game world is divided into sub spaces called *static zones.* Although the shape of these static zones can be arbitrary, they are typically planes in the world that have a shape of a polygon. A static zone can be a floor of a building or a large part of a landscape. Static zones are useful for implementing interest management schemes and are transparent to the player. They are connected together with the help of two contiguous *transition gates.* A player can only move between two zones by moving through the transition gates. Interest management is implemented in Mammoth using a publish/subscribe system where the client *subscribes* to the static zone its player is currently in as well as its neighbors (see Figure 2–2) and the server publishes state updates made on these zones to their appropriate subscribers. This way, the client only receives updates of events that occur in its own as well as neighboring static zones. As a client moves across static zones, it subscribes to the new static zones which are now of its interest and unsubscribes from all the ones it is no more interested in.

2.4.3 Software Architecture for Mammoth

Figure 2–4 gives an overview of the Mammoth software architecture. The architecture follows a *layered* approach where the major functionality is divided among layers and different *components* encapsulate specific concerns or features of the system. Layers interact with only their adjacent layers through well-defined interfaces and the implementation of one layer does not affect that of the other. The major layers in the Mammoth architecture are :

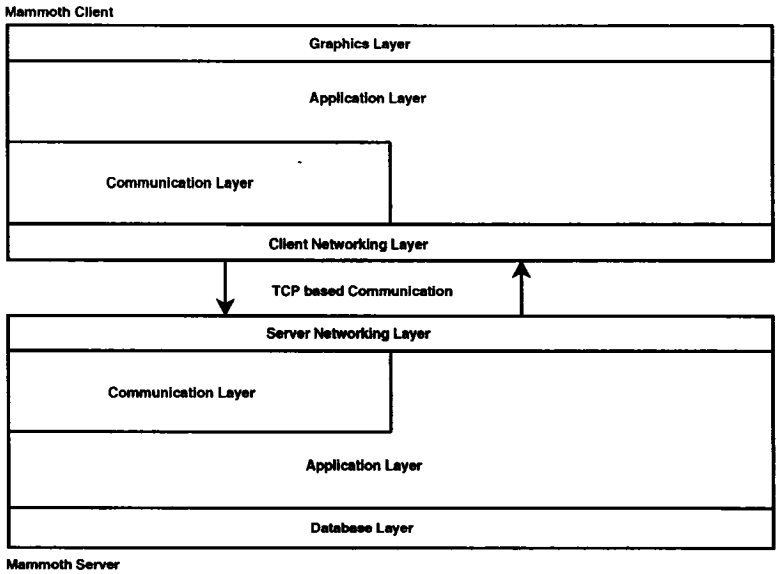


Figure 2–4: Mammoth Software Architecture

1. **Graphics Layer**: The graphics layer in Mammoth is used to render the graphical display to the end user playing the game using the Minueto GL graphics library [28].
2. **Application Layer**: The application layer implements all the game dependent logic such as game physics, session management, collision detection and path finding.
3. **Network Layer**: The network layer implements the core primitives for communication between the client and the server. Currently, the state updates between the clients and the server are transferred in the form of serializable messages using TCP (Transmission Control Protocol).
4. **Communication Layer**: Acts as an intermediary between the network and the application layer. It receives messages from the network layer and transfers

them to the application layer for processing. Similarly, it receives updates from the application layer and converts them into messages which are then passed on to the networking layer to be sent over the network.

5. **Database Layer** The database layer is used by the application layer to store or load the state of in-game objects from a persistent storage.

2.5 The Mammoth Network Architecture

This section gives an overview of the existing network architecture of Mammoth and its communication methodology. Mammoth currently follows the *client/server* architecture discussed in Section 2.2, where clients connect to a single server to transfer state updates and receive response. It is implemented in the networking layer as the *network engine.*

The network engine is present at both the client and the server side and is responsible for communicating game related data between them in the form of serializable *messages* in an asynchronous manner. A state update for a particular static zone in the game is generated by the application layer of the client, converted into a serializable message by the communication layer and passed on to the network engine. The network engine at the client sends the message directly to the server. The networking layer at the server receives the messages and sends them to the upper layers where the game state for the corresponding static zone is updated. A response message is generated which is multicast by the network engine to all the clients which have subscribed for updates in that particular static zone.

2.5.1 Server Messages

The client and the server network engine communicate with each other with the help of messages. The mammoth network engine supports two basic categories of messages from the server to the client:

1. **Multicast Messages** : Messages which need to be multicast to a number of clients. These are generally state updates which need to be disseminated to interested clients. Examples of multicast messages are player position updates and object pickup/drop updates.
2. **Direct Messages** : Messages which are meant only for a specific client. Messages containing complete states for static zone(s) are sent as direct messages to clients.

A client maintain a single connection to the Mammoth server and therefore, all messages from the client to the server are direct messages.

2.5.2 Logical Channels

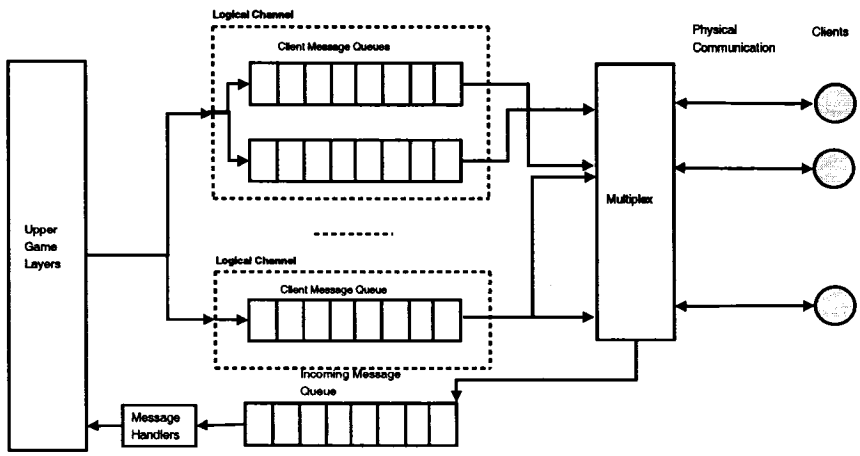


Figure 2–5: Logical Channels in the Mammoth Network Engine

Figure 2–5 illustrates the logical channel architecture for Mammoth. Logical channels are an abstraction above the actual physical socket channels of communication. In the current implementation of the mammoth network engine, each static zone is *served* by one logical channel which is usually assigned the same name as that of the static zone. Clients subscribe to logical channels serving the zones they are interested in order to receive state updates. Any update occurring on a static zone is sent to the logical channel serving it where it is multicast (or published) to all the subscribed clients.

Logical channels are implemented as objects having a list of *subscribers* which contain the objects representing the client nodes subscribed to them. Each of these client node objects has a FIFO message queue associated with it and primitives to send the messages in the queue over the network to the node the object represents. A state update received at the server is processed by upper game layers and a response message is produced which is sent to the object representing the appropriate logical channel. At the channel, an iterator goes through all the client node objects in the subscribers list and adds the message to their corresponding message queues. A multiplexing mechanism working as a separate thread in the network engine removes messages from the head of the message queues and sends them over the network to the corresponding nodes. Access to the client message queues is done in a synchronized manner as they are also used to send direct messages to clients.

2.5.3 Communication Protocols

This section outlines the protocol of communication between the client and the server during three important game scenarios:

Player enters game.

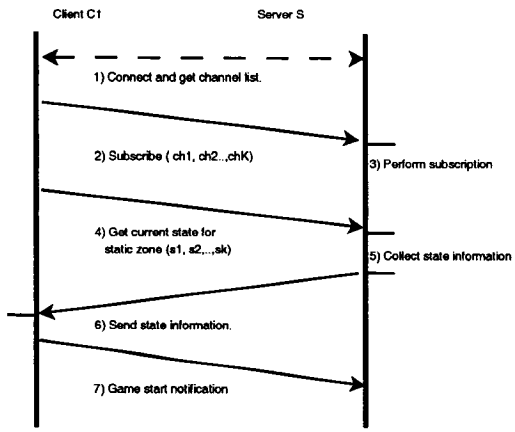


Figure 2–6: Game Start Communication Protocol

Figure 2–6 shows the steps needed to be performed when a player actually enters the game. The client connects to the server to receive the list of logical channels available. It then sends subscription requests for all the static zones it will be interested in based on its player's starting position in the world. The server subscribes the client to the logical channels serving those static zones. The client then requests the current state of all these zones from the server to start the game.

Player performs action

Figure 2–7 shows the steps needed to be performed when the player performs an action at the client. An action performed is converted into a state update message which is sent to the server. The server processes the message and multicasts the response to all clients subscribed to the logical channel serving the zone where the action was performed.

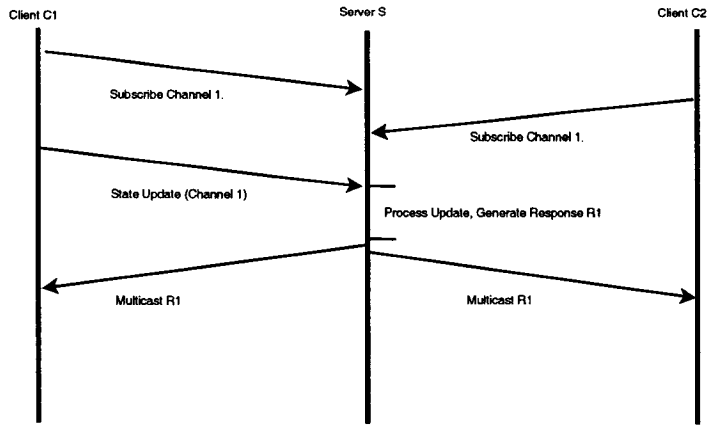


Figure 2–7: State Update Dissemination

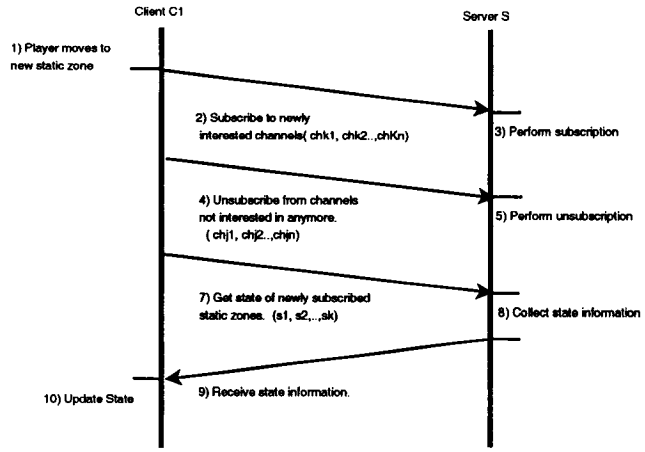


Figure 2–8: Player transition across static zones

Player changes static zone

Figure 2–8 shows the steps needed to be performed when the player moves from one static zone to another. During transition, the client machine subscribes to all logical channels serving the zones it is now interested in and requests for their current

state. It also unsubscribes from those channels which it no more wants to receive updates from.

CHAPTER 3
Scalability via Distribution

This chapter discusses the limitations of the client/server architecture described in the previous chapter and discusses various existing solutions to solve these limitations.

3.1 Limitations of the Client/Server Architecture

Although the client/server architecture may seem straightforward and relatively simple to implement, it suffers from the serious lack of *scalability.*

3.1.1 Scalability and Latency

The ability to simultaneously handle an increasing number of players without compromising *playability* is referred to as the scalability of a MMOG. Playability (or game play) is the overall experience offered by the game to a player. The most crucial factor which determines the overall playability of a MMOG is *latency*, also referred to as *response time* or *lag.*

Latency is the difference in time between the action performed by the player and its observation by all the other players in the game. The latency values should be kept within reasonable bounds (depending on the nature of the game) in order to provide the user a better gaming experience.

3.1.2 Effect of Latency on Playability

A qualitative study on the effect of latency on game play in [15] shows that the required maximum latency for good playability in first-person-shooter (FPS) games

should be between 100ms-200ms, and anything above this limit can seriously affect the playability of the game due to the real time nature of its in-game interactions. A study [3] on Quake 3, a popular FPS game, showed 150-180ms as the preferable bound for latency for sound game play.

However, studies show that MMORPGs can tolerate higher amount of latency than FPS games as their playability hinges more upon strategy rather than real time interactions. A study [16] on the effect of latency on playability of a popular MMORPG called Everquest2 [14] showed that role playing games can tolerate a maximum of 1250ms for movement based updates while 500ms to 1000ms for scenarios involving large scale player interactions such as combat situations.

3.1.3 The Scale of Present Day Games

Early network-based games supported only a small number of players (around 10-20), and scalability was not such a big issue since a single server could handle the computational and communication load generated by these players. However, with the internet providing strong global connectivity, MMOGs have become increasingly popular with thousands of players participating in a game session.

World of Warcraft [12], a popular MMORPG has around 2 million registered users and over 500,000 players interacting at the same time [30]. Lineage [29], another MMORPG has over a million registered users and recorded up to 180,000 concurrent players at one time. Second Life [26], an upcoming MMOG which focuses on social interactions between players sharing a common virtual world recorded an increase in its in-game objects to thousands in a short period of two months.

Further, with advances in the field of artificial intelligence and computer graphics, game designers try to inculcate various complex algorithms for purposes such as path finding and game physics, push towards newer kinds of interactions, design complex objects having a greater array of properties (e.g., combinability) and large diverse game worlds. All done with an aim to provide the player with a more exciting and immersible game experience.

However, the traditional client/server based architectures are not able to cope with these rapid developments due to some inherent performance limitations, the most prominent of which are described below:

1. **CPU Load** - Processing a new game state based on the updates received from a player requires processing power at the game server which gets depleted as more and more players join the game. As a result, the server gets overloaded and more time is spent processing each state update received from the connected clients. This leads to an increased response time for participating players resulting in poor game play.
2. **Bandwidth** - As the number of players joining a game increases, there is an increase in the number of update messages received by the game server from the clients. Furthermore, these updates need to be relayed to all other clients connected to this server. Since the bandwidth capacity at the game server limits the amount of data flowing in and out of it, it can act as a bottleneck for the number of updates that can reach or leave the server at any instant of time. This can cause increased response times at clients sending/receiving these updates affecting playability of the game.

Bandwidth and CPU load thus present a bottleneck in the scalability of the single server architecture.

3.2 Efforts to Improve Performance

There has been considerable efforts in academia as well as industry to come up with alternative network architectures and communication schemes for MMOGs to improve their scalability and performance. This section discusses such efforts.

3.2.1 Multiple Game Instances

A common tactic used by most commercial MMOGs is running more than a single instance of a game on separate servers. When a particular server gets overloaded in capacity, the newly registered players are then redirected to the next available server running another instance of the game. Ultima Online [4] refers to these instances of game worlds as *shards*. The drawback with this approach is that it places users into disparate non interacting worlds, limiting them to only interact with the players in their shard. Since the popularity of most games is based on the social interaction with other players, less populated game worlds can lead to a poor gaming experience.

3.2.2 Mirrored Server Architecture

In the mirrored server architecture [10], the game state is mirrored over several game servers which are geographically distributed over the internet. A client connects to the closest mirror server to transfer state update messages. The message latency is therefore reduced as an update messages from a client (and a responses from the server) only needs to travel a shorter distance to the closest mirror server as opposed to the single server architecture where the central game server can be potentially

distant. A mirror server multicasts incoming updates from its clients to the other mirror servers which compute their own copy of game state and send state updates to their directly connected clients. Special synchronization mechanisms are used to deal with keeping these multiple copies of game state consistent at all mirrors. However, due to the complex nature of these mechanisms this approach does not scale well as it becomes increasingly difficult to keep the game state at a larger number of mirror servers consistent.

3.2.3 The Distributed Architecture

There are several research efforts which partition the global game space into sub spaces and allow multiple machines (or nodes) to maintain the state of these sub spaces. Special communication mechanisms are used between these nodes to cooperatively keep the global state consistent. Since each node only has to handle updates pertinent to the sub space it is maintaining, the bandwidth consumption at these nodes is reduced to a fraction of that in the single server architecture. CPU consumption also drops as each node has to process less updates. As a result, the architecture can handle a greater number of clients without getting overloaded. There are two dominant multiplayer game architectures that support such a scheme, namely the *Distributed Server* and the *Peer-to-Peer* architecture. This section highlights existing research efforts on both of these architectures.

The Distributed Server Architecture

This architecture involves the distribution of the partitioned sub spaces of the MMOG game space over multiple game servers (see Figure 3–1) each of which is known as the *host server* of a particular sub space. The host server for a particular

sub space maintains the state for that sub space, i.e., it collects and disseminate updates occurring on that sub space.

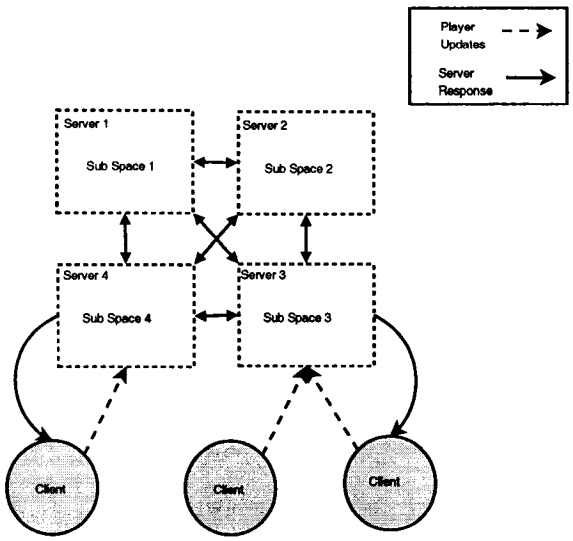


Figure 3–1: A Generic Distributed Server Architecture

The client maintains a direct connection with the host server of the sub space its player is currently in (also known as the client's *home server*) to send state updates. The server serializes these updates and generates a response which it sends to the clients connected to it. A client dynamically switches connections once its player migrates to a sub space hosted by a different server. Depending on the interest management scheme used, a client may be interested in state updates occurring on a sub space hosted by a server other than its home sever. In such a case, the server hosting such a sub space might need to forward updates to the client's home server. Hence, server nodes are connected to each other in this architecture in order to share such updates.

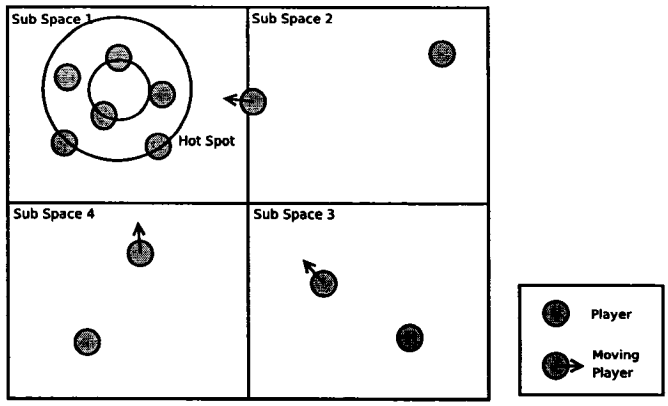


Figure 3–2: Flocking of Players to a game hot spot.

One of the main challenges of the distributed architecture is to maintain the computation and communication overhead at each server below a safe threshold. This becomes important in situations such as *flocking* where players move or flock to a specific area in the game space due to an in-game event (such as a party or a battle), thus creating a *hot spot* (see Figure 3–2). This leads to greater number of updates being sent to the host server of the sub space where the hot spot occurs, potentially overloading it.

To meet this challenge, some games introduce game level restrictions such as to limit the number of players that are allowed in a certain area. This technique ensures that only a certain number of players are maintained in each sub space and hence controls the amount of load at each server. However, this technique suffers from the disadvantage of limiting the player's ability to explore the game world and interact with other players.

One of the main problems is that hosting a large sized sub space per server cannot guarantee an even distribution of load on each server as players are not

evenly spread across the game world. [30] proposes breaking down the sub spaces into smaller *microcells* and allow each server to host a set of these microcells. In the event of a hot spot, microcells can be moved from an overloaded server to a lightly loaded server resizing the overall region a server hosts and spreading the load more evenly across servers. There can be a number of techniques in which microcells can be allocated across servers so that the maximum load per server is reduced. The paper focuses on comparing the performance of each of these techniques using results from simulations.

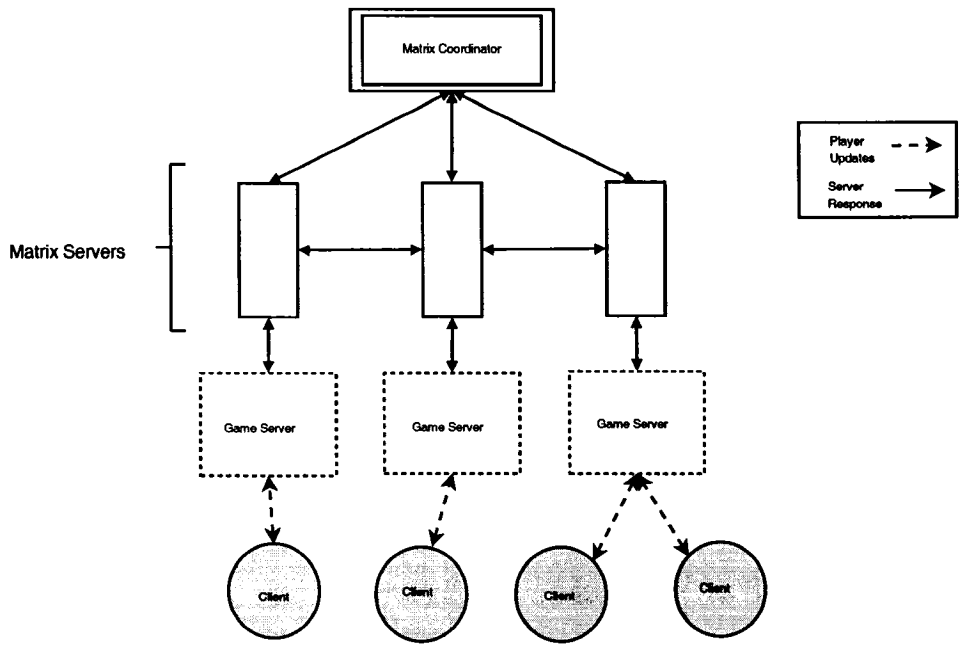


Figure 3–3: The Matrix Architecture

However, hot spots can also migrate across sub spaces causing the load on sub spaces to temporarily change. In the Matrix Architecture [5], hot spots are handled by dynamically adjusting the size of a sub space handled by a server. Further,

each player has associated with it a radius of visibility which defines its area of interest in the game. Based on the radius of visibility and the sub space partitioning information, *overlapped* regions are determined and all players in a particular overlap region receive updates from all the sub spaces involved in the overlap. Figure 3-3 shows how this architecture is organized. Clients connect directly to a game server to transfer updates tagged with their spatial coordinates in the game world. The game servers process these updates and forward them to their respective *Matrix Servers*. A Matrix Server keeps track of the sub spaces that its game server is in charge of and the load it is experiencing. It uses the overlap information to forward relevant updates to other interested game servers which then relay them to their connected clients. In the event of overload, the matrix server removes a portion of the sub space from its game server and transfers it to another lightly loaded game server and all the concerned clients are redirected to this game server. The overlaps due to this new partitioning are recomputed by a special server called the matrix coordinator which then updates all the matrix servers with this information.

In the architecture described in [9], the sub spaces are very small and each server is responsible for more than one sub space. In the event of overload, an overloaded server migrates some of its sub spaces to another server. The load balancing algorithm takes into consideration the locality of the sub spaces and attempts to keep sub spaces which are adjacent to one another (in the game world) on the same server. This technique favors localized communication within a single server and prevents excessive inter-server communication which can increase overhead. Therefore, the algorithm sheds load off an overloaded server and at the same time fixes any locality

disruption caused by load shedding by aggregating adjacent sub spaces during normal load conditions. The performance of the algorithm is evaluated under flocking conditions using a simulator.

A server can also be overloaded by the load incurred by message forwarding rather than processing. In [31], a set of lightly loaded server nodes are always maintained in the form of a backup queue. In the event of an increased message forwarding overhead at a server, the overloaded server selects one of the backup nodes to act as an intermediate node between its connected clients and itself. This relieves most of the forwarding overhead since now the responsible node only needs to forward updates to the intermediate node which takes charge of further disseminated them to all the interested clients.

Peer-to-Peer Approaches

A Peer-to-Peer (also known as P2P) approach differs from a client/server approach as there is no central point of authority, i.e., each node in a P2P based architecture can act as both a client as well as a server. P2P based architectures are more scalable than client/server ones since the computation, communication overhead and resources are shared by all participating nodes. The self organizing capabilities of P2P overlays can be used to create systems which can be dynamically scaled up and down with the number of nodes thus making them suitable candidates for the design of scalable MMOGs network architectures.

[23] introduces an approach to scale MMOGs by distributing the state of sub spaces over participating player nodes which form a P2P based network. All the players in the same sub space use the self-organizing mechanisms of P2P networks

to form multicast groups where only state updates pertinent to that sub space are disseminated. A player node joins groups corresponding to sub spaces which overlap its area of interest. Players change their multicast group as their area of interest changes, i.e., as they move around in the game space. Figure 3–4 illustrates the P2P based architecture for an MMOG.

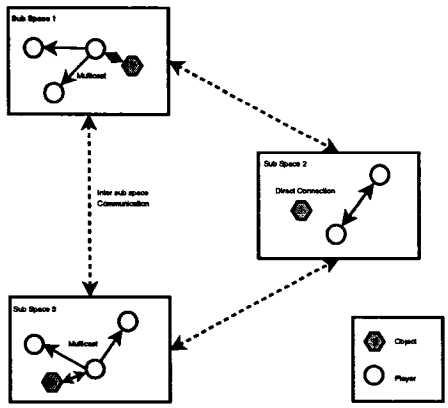


Figure 3–4: P2P Based Architecture

For each multicast group, a player node is designated as a *coordinator* for that group. A coordinator is responsible for maintaining the states of the shared objects of the group. It acts as the root of the multicast tree and provides the connecting player nodes with the current game state when they join the group. For fault-tolerance purposes, a replica of the coordinator is maintained so that in case of a failure, the backup replica takes over and the state updates are automatically forwarded to it. Player-to-player interactions are handled by establishing connections between the participating player nodes for direct exchange of state information. The architecture uses the Pastry [27] P2P overlay which maps both participating nodes

and the application objects to random, uniformly distributed identifiers from a circular 128-bit name space. Objects are mapped to live nodes whose id is numerically closest to the object-id. A distributed hash table (DHT) is used to lookup which node a particular object resides in. Scribe [8], a scalable application level multicast infrastructure build on top of pastry is used to disseminate game state. It leverages the existing Pastry overlay by using its identifier scheme and routing mechanisms and is able to form a large number of multicast groups having arbitrary number of members with highly dynamic membership.

A similar concept is employed in a P2P based architecture described in [21] with a few optimizations to reduce latency. The role of the distributed hash table is limited to that of a backup data storage of object states and the local cache at the coordinator is used by the connecting players to retrieve the latest game state. Since the access of data directly from the coordinator is faster than that from the DHT, the time required to receive state updates is reduced considerably. When another node becomes the coordinator, it loads the state information from the DHT into its local cache and provides updates from it to the connecting player nodes.

The dissemination of updates among the player nodes in most of the architectures described above can be modeled as a *publish/subscribe* system, where publisher nodes (servers or peer nodes) multicast events and player nodes only subscribe to events that they are interested in. The Mercury publish/subscribe [6] system provides features which allow player nodes to express their subscriptions using a rich *subscription language* which provides greater flexibility to describe what a player is

interested in. It is designed to work over a distributed system such as a P2P network allowing matching of subscriptions with publications to take place over multiple nodes. It also provides a scalable and efficient routing mechanism which routes publications to subscribers within the real time requirements of game play by distributing routing responsibility across several group of nodes.

The main issue faced by nodes in most P2P systems is that of understanding the overall topology of the P2P network and creating connections with the right nodes to get relevant data. In MMOGs, this is even more crucial since each node must receive only relevant state updates for proper decentralized resource consumption and improved scalability. Another fully distributed P2P based architecture [18] uses the mathematical construct of the *Voronoi Diagram* to help player nodes discover neighboring nodes which they can connect to in order to receive the required state updates.

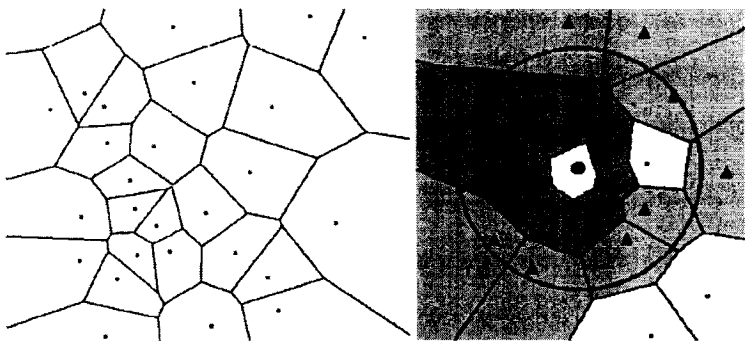


Figure 3–5: (a) Voronoi Partitioning (b) Neighbor discovery using Voronoi partitioning and area of interest. Source : [18]

A Voronoi diagram is constructed by partitioning the world space into n non overlapping sub spaces having one node per subspace. A sub space contains all the

points closest to that sub space's node than to any other node (Figure 3–5a). This way the entire game world is divided into arbitrary sizes in a deterministic way. Each participating node in a P2P network maintains a voronoi diagram of the nodes present in its area of interest and maintains a P2P connection with all such nodes (Figure 3–5b). As the player node moves in the virtual world, it readjusts its voronoi diagram to create connections with newly discovered nodes and break connections with the ones no longer in visible. Voronoi diagrams are also recomputed at relevant nodes as other player nodes join or leave the P2P architecture.

3.2.4 Distributed server architecture vs. Peer-to-Peer based approach

Peer-to-peer architectures are a good alternative since they provide increased scalability by balancing the load for managing the game state across nodes in a P2P network. However, in our opinion, there are a few drawbacks in using P2P based network architectures for MMOGs:

1. **Cheating and Fairness**: Since the game state is distributed among participating player nodes it becomes vulnerable to game hackers who can alter or view this information to gain unfair advantage in the game.
2. **No Administrative Control**: There is no central authority to manage the whole P2P based system, which can present a problem in scenarios such as monitoring resource usage among player nodes for load balancing or detecting failure in the system.
3. **Searching and Topology Maintenance**: Deciding and forming a topology in a P2P based network has been a challenge in P2P networks and although there are techniques for node discovery and connection maintenance (as mentioned

in [18]), there is still a considerable overhead incurred in forming connections among nodes which might cause problems for real-time game play. Searching in P2P based systems also induces latency due to a considerable number of network hops incurred while searching for or relaying game state.

4. **Fault Tolerance and Availability**: Since there is no control over the availability of the nodes in a P2P system they can fail or leave the system without warning, leading to the loss of the game state associated with them. This can leave the overall game state incomplete or inconsistent rendering the entire system fault- intolerant.

In contrast, the distributed server architecture offers a number of advantages which overcome the shortcomings of the P2P based approach :

1. **Central Point of Authority**: The collocation of the server nodes in a distributed architecture provides greater administrative control over them. This is advantageous for many reasons:
   - Since the game state is only distributed on the server nodes, its is easier to exercise control over their access and avoid unwarranted modification.
   - Consistency of the overall game state is easier to manage since it is now localized over server nodes that are members of the architecture.
   - It is easier to monitor resource usage at all server nodes making it simpler to deploy load balancing techniques.
2. **Faster inter server communication**: The nodes in a distributed server architecture are generally inter-connected through a high speed local area network

(LAN) resulting in faster inter-server communication giving lower response times for game actions.

3. **Fault Tolerance and Availability**: Nodes in a distributed server architecture can be centrally monitored for failure or supplemental backup nodes can be maintained to take over in case of failure.

Therefore, we consider a distributed server based architecture is the best alternative for improving the scalability of MMOGs.

CHAPTER 4
The Distributed Server Architecture - An Overview

While many sophisticated distributed architectures, including advanced features such as load-balancing, have been proposed (as discussed in Section 3.2.3), only few are implemented in real systems, and it remains unclear what are the challenges when transferring such ideas to a real MMOG. This thesis addresses this issue. It proposes a practical distribution approach and describes its concrete implementation and integration into the Mammoth protoype.

The primary goal of this approach is to improve the scalability of an MMOG, enabling it to support a larger number of clients than the single server architecture without overloading. Therefore, ability to measure the increased performance of an implemented distributed server architecture over the single server architecture is important.

4.1 Architecture

4.1.1 Overview

The distributed server architecture follows a cluster-based approach where a group of server nodes connect with each other to form a cluster where they can communicate with each other. As discussed in Chapter 2, most MMOGs have a notion of game space associated with them and the state of the entire game is the state of all objects present in this game space. In the distributed approach the game space is divided into sub spaces and each server in the cluster hosts a set of sub spaces.

Clients only maintain connection with their *home* server, i.e., the server hosting the sub space where their player is currently in, and receive all relevant updates through them. All the server nodes in the cluster always maintain a physical connection with each other in order to forward relevant state updates they receive from their clients through specialized distributed publish/subscribe mechanisms discussed later in this chapter.

4.1.2 Major Components

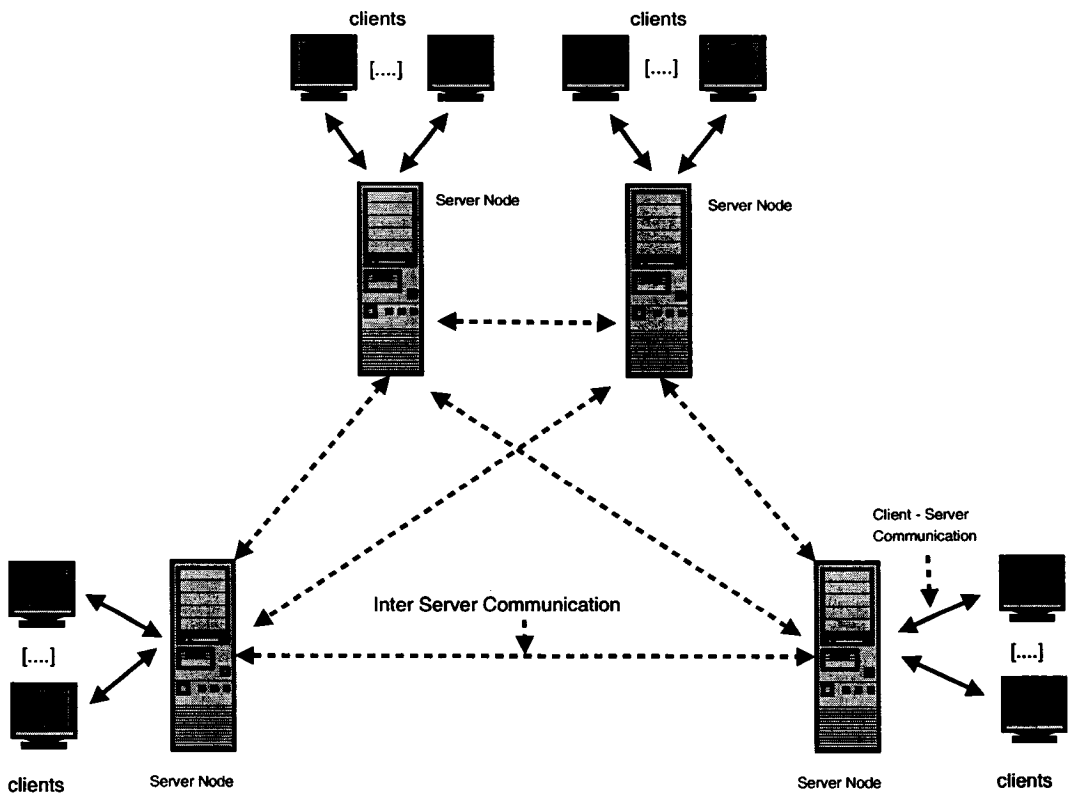


Figure 4–1: The Distributed Server Architecture

Figure 4–1 provides a broad overview of the distributed server architecture illustrating its major components. This section summarizes how these components work:

Server nodes

Server nodes *host* sub spaces, i.e., they maintain the game state for these sub spaces and they are known as their *host servers.* In our current implementation, the sub spaces that a server is meant to host are specified at start up. However, in general, such sub spaces could be assigned dynamically and might change over time, e.g., for load-balancing purposes. A server node can host more than one sub space. The main responsibilities of the server node can be outlined as follows:

1. To accept client connections for players entering the sub spaces they host.
2. Receive and process state updates corresponding to those subs paces.
3. Multicast processed state updates to all interested clients.

Each server node maintains a physical connection with all the other server nodes within the cluster. This allows inter server node communication useful for transferring state updates it receives from its clients to the other interested server nodes which can relay these updates to the clients connected to them. More details on how this communication scheme actually works is given shortly.

Clients

The end user plays the game on the client node which renders the game state using a graphical display. A client maintains a direct connection with its home server in the cluster. Actions performed by the player at the client are converted into state updates and are transferred to the home server where the state is processed and the

updated response is multicasted to all other interested clients in the architecture. A client caches a local copy of the game state which is kept up-to-date by applying the state updates it receives from its home server.

4.1.3 Administrative Components

Apart from the server nodes and the clients, there exist two more components which help in administering and managing the overall distributed architecture.

Cluster Administration Server

The cluster administration server (CAS) is the central administrative entity of the distributed server architecture. Currently, the CAS is used during the startup to setup the cluster. Server nodes upon startup connect to the CAS and provide it with important information such as 1) their host address and 2) the sub spaces they are in charge of. The CAS builds a table using all this information and consecutive connecting server nodes use this table to discover other nodes in the cluster and establish physical network connections with them.

Being an administrative entity, the functionality of CAS can further be extended to perform other important tasks such as:

1. Monitoring load at servers nodes to make sure they are not overloaded.
2. Act as a central entity to administer load balancing procedures in case of server overloads.
3. Monitor server nodes for failure and switch to backup if necessary.

Rendezvous Server

The Rendezvous Server (RS) acts as an initial point of contact for a client trying to connect to the distributed server architecture. The RS contains information which

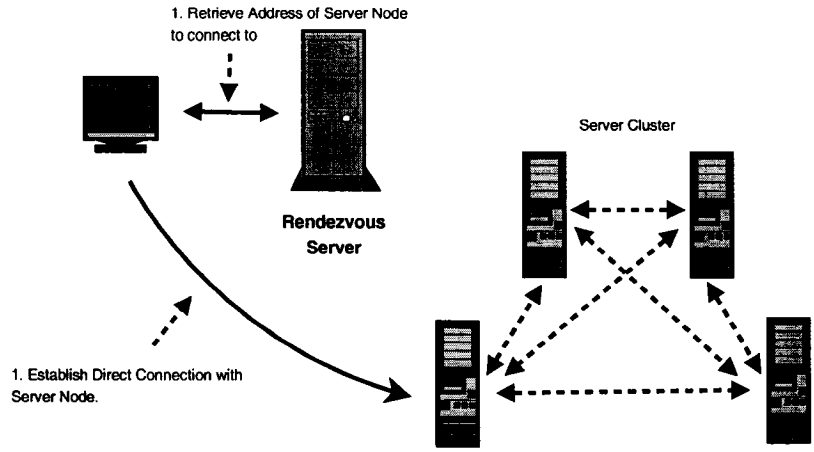


Figure 4–2: Working of Rendezvous Server

helps the client locate its home server in the cluster. A client connects to the RS and supplies player information to retrieve the host address of its home server so that it can connect to it in order to start playing the game. Figure 4–2 illustrates the working of the RS. The RS also authenticates a client before it can start playing the game.

The CAS and the RS are kept as separate machines mainly to logically separate both their functionalities. RS's main task is to authenticate and redirect clients to their home server which can be very intensive given the scale of participants in present day games and the dynamic nature of their entry and exit into the game. Further, since all the task of an RS are on a separate machine, it is possible to have several RSs places at a number of geographical locations in order to allow faster, localized connectivity to the cluster. Being the only point of entry to the cluster, the RS also becomes vulnerable to attacks from hackers, therefore it may not be advisable to perform critical CAS related operations on the same machine.

4.2 Communication Protocol

4.2.1 The Challenge

The single server architecture simplified communication as the state for the entire game was centrally located at a single node and clients only needed to connect to this node to transfer updates. However, in a distributed architecture the state of the entire game is now distributed across the server nodes in the cluster with clients sending updates only to their home server. Further, each client may not only be interested in updates occurring on the sub space its player is currently in but also in other sub spaces (depending on the interest management technique). These sub spaces may be hosted on the client's home server or on other server nodes in the cluster. Therefore, it becomes a challenge as state updates must reach all interested clients in an efficient manner.

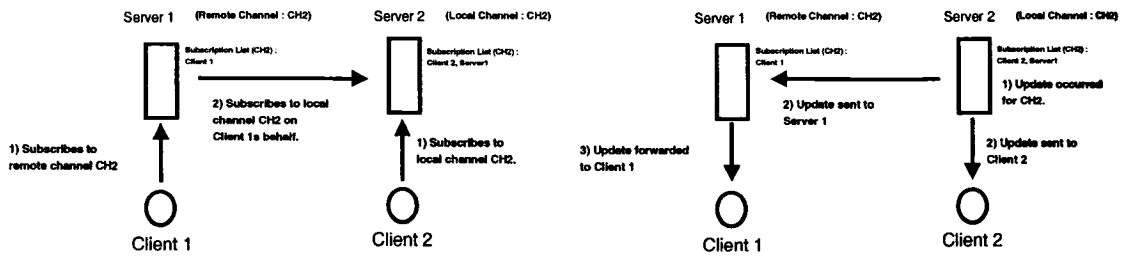
One of the possibilities is for the client to maintain connections with all server nodes hosting the sub spaces of their interest. The client sends its own update requests only to its home server, but all server nodes that host sub spaces the client is interested in, send updates on these sub spaces to the client. However, this approach has several shortcomings: firstly, the client must maintain several connections with different servers. This adds complexity and overhead to the client which might have to be connected to many servers. Futhurmore, each server has now connections not only to the clients for which it is home server, but possibly many more clients, limiting the scalability. Also, our aim is to minimise the open points of vulnerability at the cluster. With the presented scheme, malicious clients can exploit the fact that they

have open connections with multiple server nodes at the same time compromising the security of the cluster to a larger extent.

We also looked whether a group communication system could be used. We analyzed the Spread Toolkit [2]. Spread provides a mechanism to form multicast groups for disseminating updates either in LANs or WANs. Using Spread, the server nodes form multicast *groups* with clients and then send updates along these multicast groups. Basically, a multicast group can be built for each sub space. Clients can join or leave these multicast groups (depending on which sub space they are interested in). The communication between client and server nodes in Spread is via User Datagram Protocol (UDP) with an additional reliable delivery mechanism implemented to guarantee message delivery and ordering. However, experiments with Spread revealed that although it was easy to use, its communication mechanism did not scale well with a large number of clients resulting in poor latency values. This was mainly because of the token ring architecture of Spread which required acquisition of a token by the sending entity before a message transfer could take place. Also, the number of groups can become larger, which might become a problem for the group communication system.

4.2.2 Distributed Publish/Subscribe System

Based on the challenges presented above and the alternatives evaluated, there was a need for a mechanism which could provide the communication scheme we required without making connection management at the client too complex, minimising points of vulnerabilities at the cluster and scaling well with increasing number of connecting clients.



(a) Remote Subscriptions **(b) Update Dissemination**

Figure 4–3: Simplified Overview of Distributed Publish/Subscribe Mechanism

We came up with a publish/subscribe mechanism which could accomplish all the above goals by distributing the subscription mechanism across the cluster. Using this system, the client is only connected to its home server. It can subscribe to any sub space it is interested in irrespective of the server which hosts this sub space. We take from Mammoth the concept of a logical channel. Each sub space is associated with a logical channel. Each logical channel has a subscription list of all clients subscribed to it. In our implementation, each server node has now two types of logical channels : *local* and *remote*. The local channels serve the sub spaces hosted by the server and the remote channels serve sub spaces hosted by other server nodes in the cluster. Clients can subscribe to any channel, only indicating the sub space they are interested in, by sending a subscription messages to their home server S.

If the subscription is for a local channel LC, S simply subscribes the client to LC and sends it updates occurring in the sub space through LC. Figure 4–3 illustrates in a simplified manner the working of remote subscription. If S receives a request for subscription to a remote channel RC, it adds the client to the subscribers list for RC locally but then forwards this subscription to the remote server R which actually

hosts this sub space and has a local channel serving it. The remote server R then adds S to its subscription list for that channel and sends updates occurring on that channel to S. When S receives these updates it disseminates them to all the clients who are subscribers to the corresponding remote channel RC.

4.3 Game Communication Protocols

This section gives a high level overview of the communication protocols used in the distributed server architecture during three common game scenarios:

4.3.1 Player Entry

When a player enters the game, the client machine first connects to the Rendezvous Server to retrieve the address of its home server. Upon receipt of this information, the client directly connects to this server node and subscribes to all the sub spaces it is interested in (based on the interest management scheme). The server node uses the distributed publish/subscribe mechanism to subscribe itself to the sub spaces the client is interested in (if it is not yet subscribed due to another client). It also retrieves the current game state of these sub spaces for the client to cache.

4.3.2 Player activity within sub space

As the player moves or interacts with other players/objects within a sub space, it transfers its state updates to its home server. The server serializes and processes these updates and multicasts responses to clients subscribed to the corresponding sub space as well as to the other servers which have connected clients also interested in these updates.

4.3.3 Player migration across sub spaces

When a player moves across sub spaces, one of two scenarios can happen.

1. **Player migrates to sub space managed by existing server**: In this case, the client maintains connection to its home server but subscribes to newly interested sub spaces and unsubscribes from uninterested ones.
2. **Player migrates to sub space managed by another server**: In this case, the client disconnects from the server node it is currently connected and establishes connection with the server in charge of the sub space it is migrating to, i.e., the client changes its home server. Before disconnection, the player unsubscribes to all sub spaces it was subscribed to earlier and makes fresh subscriptions when it connects to the next server. This is needed since it now has to receive all updates through its new home server.

CHAPTER 5
Implementation in Mammoth

The previous chapter introduced the Distributed Server Architecture for game scalability and gave an overview of its components and communication mechanisms. This chapter gives a more detailed explanation of each of those components and provides a deeper insight into the communication mechanisms involved. It also discusses the implementation of crucial components in Mammoth.

5.1 Architectural Components

Chapter 4 provided an overview of all the major components in the distributed server architecture. This section explains them in greater detail describing their internals and working.

5.1.1 Server Nodes

The functionality of server nodes in the distributed server architecture is similar to that in the single server, except that in the distributed case, each server node hosts a subset of all sub spaces (i.e., a part of the game state) and maintains connections with other server nodes in the cluster to send/receive state updates to cooperatively manage the entire game state.

Upon startup, a server nodes is assigned the sub spaces it is supposed to host and local channels are created to serve each of these sub spaces. A server node connects to other server nodes in the cluster to form inter-server connections. During this process, server nodes inform each other about the local channels they host. Once

all server nodes know about the local channels hosted by every other server node in the cluster, they build a hash table (referred to as `remoteChannelMap`) which maps channel names to the server nodes that host them. They use this table to forward updates received from their connected clients to other servers. More details about how this process actually takes place are given shortly. A client only sends updates occurring in the sub spaces hosted by its home server. A server node receives and processes these updates, generates a response and sends them to all clients and server nodes that subscribe to the channels serving these sub spaces.

Although a server node hosts sub spaces, it loads the entire game state at start up. In our current implementation, a server node does not process the messages its receives for a sub space that it does not host. Rather it simply relays it to its connected clients that subscribe to the remote channel serving that sub space. Not processing these messages locally (i.e., not updating the local game state for these sub spaces) saves considerable processing costs, and hence, can achieve better scalability. However, keeping the entire game state at the server node could be beneficial for fault-tolerance purposes. For example, a server node S_1 can be designated as a backup node for server node S_2, i.e., S_1 receives state updates for sub spaces hosted by S_2 and applies them to its local state. In the event of the failure of S_2, S_1 can take over and accommodate S_2's clients since it has the up-to-date state of the sub spaces that S_2 hosted.

Implementation in Mammoth

According to Section 2.5.2, which describes the logical channel architecture of the single server mammoth network engine, each logical channel is maintained as

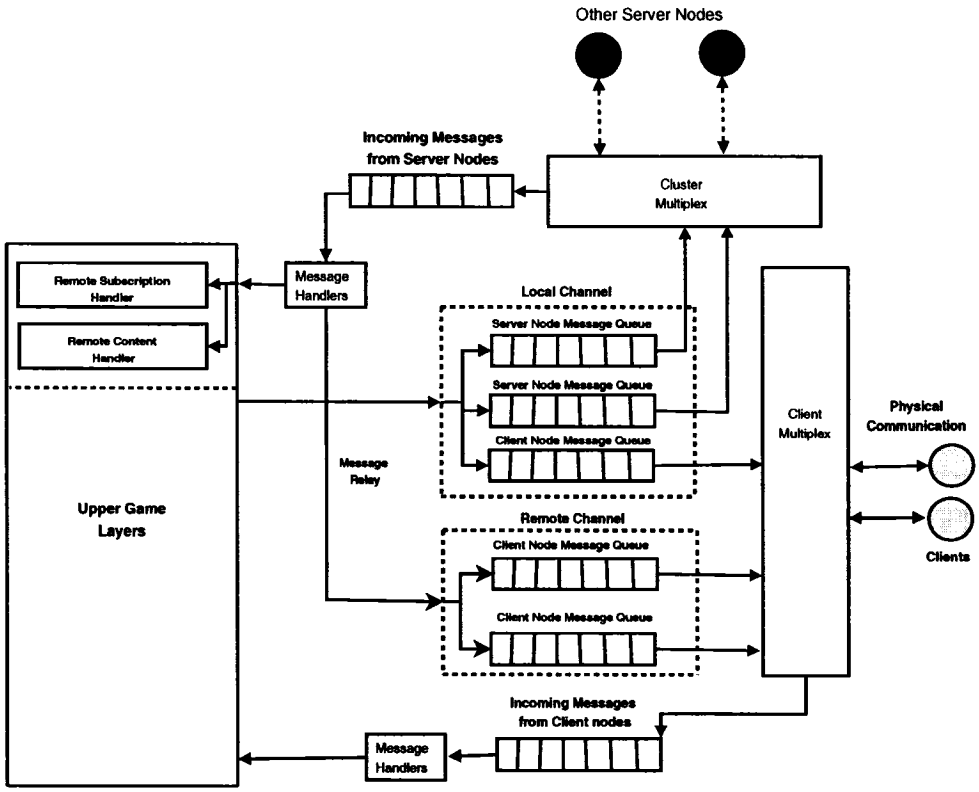


Figure 5–1: Distributed Network Engine for Mammoth

an object having a *subscribers* list containing the objects representing all the clients subscribed to it. Each of these client node objects has a FIFO message queue associated with it and primitives to send the messages in the queue over the network to the node it represents. An iterator goes through all the client node objects in the subscribers list and adds the message to their corresponding message queues. A multiplexing mechanism removes messages from the head of the message queues and sends them over the network to the corresponding nodes.

In the distributed server implementation of Mammoth, the network architecture has been extended to support all the additional communication capabilities and subscription mechanisms of server nodes. Figure 5–1 shows the internals of the server node in the Mammoth implementation. The local channel objects have a subscribers list which consists of both server node and client objects representing the subscribers of that channel. The remote channel object have a subscribers list which consists of only the client objects which subscribe to that channel. Another multiplexing unit is added to send/receive messages to/from other server nodes in the cluster. Incoming messages from the other server nodes in the cluster are received by a separate message handler. Messages consisting of state updates are not processed but are relayed to the remote channels where they are added to the message queues in order to be dispatched to client nodes that have subscribed for those updates. Messages requesting subscription or requesting state for a sub space hosted by the server node are processed by the *remote subscription handler* and *remote content handler* respectively. Similar to client message queues, the server node message queues are also used to send direct messages to other server nodes in the cluster.

5.1.2 Clients

A client in the distributed server architecture only maintains connection with its *home* server. When its player migrates to a new sub space, the client switches connections to the server hosting this sub space, subscribes to all the sub spaces it is now interested in and unsubscribes that are no more of its interest. Although, during migration of a player across servers, it unsubscribes from all and resubscribes again.

The primitives for all communication remains unchanged for the client in the distributed server architecture, since complex processing such as subscription management and game state management is handled among the server nodes. The client remains oblivious to the distributed publish/subscribe system and uses the existing primitives to send/receive updates and subscriptions. However keeping in mind the fact that the client might change home servers, its implementation in the distributed architecture has been extended with certain capabilities:

1. **Rendezvous** When entering the game, the client connects to the rendezvous server instead of any particular server on the cluster. The rendezvous server supplies the client with the `remoteChannelMap`[1] which it uses to locate and connect to its home server.
2. **Connection Management** A connection manager is implemented at the client which manages dynamic connection switching as the player migrates to a subpace hosted by another server. During the migration process, the connection manager disconnects the client from the current server, looks up its new home server using the `remoteChannelMap` and connects to it.

State Management

Clients upon startup only load the static objects in the game (for e.g., map information). Upon connection to their home server, they retrieve dynamic state of all the sub spaces they are interested in, this procedure is known as *world management* in the Mammoth implementation. Clients also (re)load state information of

[1] A hash table which maps a channel to the server node which hosts it.

sub spaces the sub spaces they are interested in after they migrate across servers. Detailed protocol for world management in the distributed server architecture is given in later sections.

5.1.3 Cluster Management with CAS

The cluster administration server (CAS) is in charge of forming the server cluster. Server nodes which intend to join the cluster first connect to the CAS where they are provided with all the information required to establish connections with other existing server nodes in the cluster. The CAS also builds up the `remoteChannelMap` using the information it receives from the connecting server nodes. Once all the nodes have joined the cluster, the rendezvous server connects to CAS to receive the complete `remoteChannelMap` to help redirecting connecting player nodes in the appropriate server nodes in the cluster. Figure 5–2 illustrates the cluster formation procedure.

Cluster Management in Mammoth

The implementation of the cluster formation and management in server nodes is separated from the existing distributed network engine in the *cluster management* layer. Figure 5–3 illustrates the cluster management layer. This layer sits on top of the networking layer and performs all the cluster management procedures such as connecting to the CAS, retrieving connection information and connecting to the other server nodes.

Discussion

As explained in the previous sections, each server nodes maintains a physical network connection with every other server node in the cluster. Although it may seem

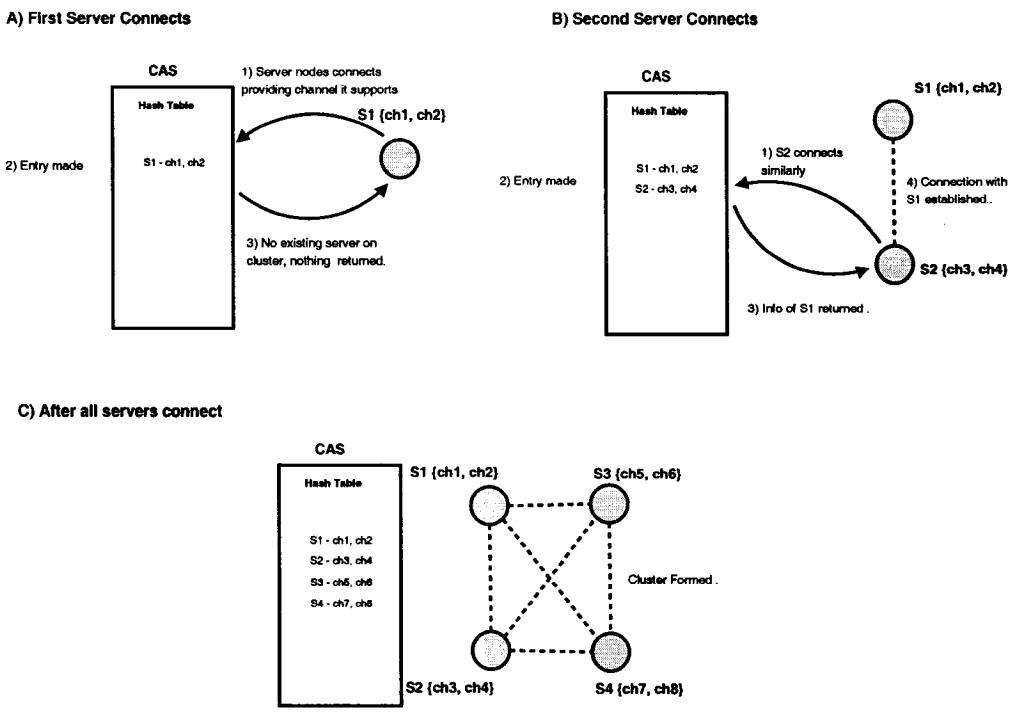


Figure 5–2: Cluster formation procedure

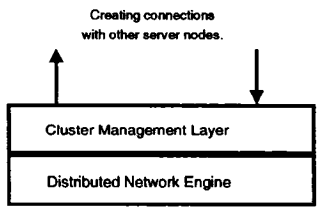


Figure 5–3: Cluster Management Layer

unnecessary to maintain such a number of connections, this scheme actually turns out to be more efficient. For example, consider a cluster consisting of 4 nodes with the game space equally distributed among them. During an active game scenario with a considerable number of players, there will be a need for constant inter-server communication as players move across sub spaces or subscribe to sub spaces hosted

by servers other than their home server. If a physical connection is maintained only on need basis, there will be a considerable overhead in establishing and tearing down these connections repeatedly, increasing the time taken for a state to be transmitted across servers. However, if a permanent physical network connection is maintained with proper interest management, then this overhead can be avoided.

5.2 Communication Mechanisms

This section discusses in detail the communication mechanisms and the protocols used in the distributed server architecture.

5.2.1 Distributed Publish/Subscribe System

The distributed publish/subscribe system provides a means for state updates to be disseminated across the distributed architecture to all interested client nodes. The detailed explanation of the protocol is described in the following section

5.2.2 Description of the publish subscribe system

The logical channels at the server node can be classified as that of two types - *local channels* and *remote channels.*

1. **Local Channels** correspond to the sub spaces that the server node is actually in charge of and are specified in the `localChannels` linked list.
2. **Remote Channels** correspond to sub spaces that reside on other nodes on the server cluster. The `remoteChannelMap` hash table maps these channels to the servers which host them.

As explained earlier, each logical channel has client and server nodes associated with them. Local channels have both clients and server nodes associate with them

while remote channels have only client nodes associated with them. Using the information of these two types of channels, we will now explain how the distributed publish subscribe system works in two different scenarios with the help of examples.

Assume a game world divided into four sub spaces - sub_1, sub_2, sub_3, sub_4 each hosted by server nodes S_1, S_2, S_3 and S_4 respectively, which form a cluster.

Example 1 - Local Subscription

Two clients C_1 and C_2 are both connected to S_1 with their players in sub_1. Naturally both of them will be interested in state updates that occur in sub_1. Hence, both send subscription messages to S_1 requesting subscription to the local channel ch_1 serving sub_1. Upon receipt of these subscription messages, S_1 adds both C_1 and C_2 to the subscribers list of local channel ch_1 (referred to as $subscribers(S_1, ch_1)$).

Now, when C_1 sends a state update for sub_1, it is received and processed at S_1 and a response is generated and sent to the local channel ch_1 in the network engine. Since there are C_1 and C_2 in $subscribers(S_1, ch_1)$), the response is sent to both of them. This way, C_2 gets to see the update performed by C_1 while C_1 simply ignores the message since it was the on which actually performed the update.

Example 2 - Remote Subscription

Now, assume there is another client C_3 connected to S_3 who wishes to listen for updates that occur in sub_1. It sends a subscription message to S_3 requesting for subscription to the logical channel ch_1 which serves sub_1. S_3 now realizes that ch_1 is a remote channel and adds C_3 to $subscribers(S_3, ch_1)$. Now, it uses the `remoteChannelMap` to discover that S_1 is the server hosts logical channel ch_1. It then

creates a message and sends it to S_1 requesting a subscription to ch_1 which is local to it. S_1 upon receiving this message adds S_3 to $subscribers(S_1, ch_1)$.

Now, when a state update for sub_1 is processed at S_1 and a response is generated, it is sent to the logical channel ch_1 in the network engine. Since S_3 is in $subscribers(S_1, ch_1)$, the response is forwarded to S_3 as well. S_3 receives this update and sends it to all the nodes present in $subscribers(S_3, ch_1)$ (the subscriber list for the remote channel ch_1). As C_3 is present in $subscribers(S_3, ch_1)$, it receives the response as well. This way C_3 can see the updates performed at sub_1 hosted by S_1.

Handling multiple subscriptions

Now consider a large number of clients connected to S_3 who want to subscribe to ch_1 present at S_1. If S_3 keeps sending subscription requests to S_1 each time a client wants to subscribe to ch_1, it will result in multiple subscriptions of S_1 at S_3, and each time a message is sent over ch_1 it will be sent multiple times to S_1 causing unnecessary bandwidth usage. Plus, there is no way for S_1 to know when to unsubscribe S_3 from ch_1, since it has no means of keeping track of how many clients are subscribed to ch_1 through S_3. Therefore, even after all clients at S_3 unsubscribe to ch_1, S_1 can still end up sending updates to S_3.

A solution to this problem is to let S_3 to keep track of the number of subscribers to its remote channel ch_1. The first time S_3 receives a subscription request for ch_1 from its connected clients, it adds the client to $subscribers(S_3, ch_1)$ and sends a subscription request to S_1 for its local channel ch_1. S_1 adds S_3 to $subscribers(S_1, ch_1)$ effectively subscribing it to all updates occurring at sub_1. When another client

wants to subscribe to remote channel ch_1 at S_3, S_3 will simply add this client to $subscribers(S_3, ch_1)$ without sending another subscription message to S_1.

When all clients unsubscribe from ch_1, $subscribers(S_3, ch_1)$ becomes empty. S_3 detects this and sends an unsubscription message to S_1 which removes S_3 from $subscribers(S_1, ch_1)$ effectively unsubscribing it from all updates occurring at sub_1.

This approach solves the two issues faced earlier in the following manner:

1. There is only one message sent from S_3 to S_1 when an update occurs at ch_1 irrespective of the number of clients subscribing to that on S_3 therefore saving inter server bandwidth.
2. When there are no more clients subscribed to ch_1 at S_3, then S_1 will not send any messages to S_3, again conserving inter server bandwidth.

Algorithm 1 gives the formal algorithm for the distributed publish/subscribe mechanism. Algorithm 2 expresses the dissemination of updates across the distributed server architecture in a formal manner.

Algorithm 1 Distributed Publish/Subscribe algorithm

1: **At Server node** S_i :

2: **Data Structures**:
3: remoteChannelMap : *Hashtable*
4: localchannels : *List*
5: $subscribers(S_i, ch_k)$: *List*

6: **Upon receipt of message** ***SUBSCRIBE***$(C_m,\ ch_k)$ **from client** C_m :
7: **if** ch_k in *localChannels* **then**
8: Add C_m to $subscribers(S_i, ch_k)$
9: **else**
10: $S_j \leftarrow$ remoteChannelMap.get(ch_k)
11: **if** $subscribers(S_i, ch_k)$ == EMPTY **then**
12: Add C_m to $subscribers(S_i, ch_k)$
13: Send ***SUBSCRIBE***$(S_i,\ ch_k)$ to S_j
14: **else**
15: Add C_m to $subscribers(S_i, ch_k)$
16: **end if**
17: **end if**

18: **Upon receipt of message** ***SUBSCRIBE***$(S_j,\ ch_k)$ **from server** S_j:
19: Add S_j to $subscribers(S_i, ch_k)$

20: **Upon receipt of message** ***UNSUBSCRIBE***$(C_m,\ ch_k)$ **from client** C_m :
21: **if** ch_k in *localChannels* **then**
22: Remove C_m from $subscribers(S_i, ch_k)$
23: **else**
24: Remove C_m from $subscribers(S_i, ch_k)$
25: **if** $subscribers(S_i, ch_k)$ == EMPTY **then**
26: Send ***UNSUBSCRIBE***$(S_i,\ ch_k)$ to S_j
27: **end if**
28: **end if**

29: **Upon receipt of message** ***UNSUBSCRIBE***$(S_j,\ ch_k)$ **from server** S_j:
30: **if** S_j in $subscribers(S_i, ch_k)$ **then**
31: Remove S_j from $subscribers(S_i, ch_k)$
32: **else**
33: print 'ERROR : No earlier subscription'
34: **end if**

Algorithm 2 Message Transfer Algorithm

1: **At Server node** S_i :
2: $subscribers(S_i, ch_k)$: *List*

3: **Upon receipt of state update** MSG_t **from client** C_m :
4: Process MSG_t
5: Generate repsonse $RESP_t$ to be sent over ch_k
6: **for all** Servers/Clients SC_i in $subscribers(S_i, ch_k)$ **do**
7: send $RESP_t$ to SC_i
8: **end for**

9: **Upon receipt of state update** MSG_t **from server** S_j :
10: **for all** Client Ci in $subscribers(S_i, ch_k)$ **do**
11: send MSG_t to C_i
12: **end for**

5.2.3 Game State Retrieval

As discussed in Chapter 2, clients retrieve and store complete up-to-date state of the sub spaces they are interested in from the servers which host them. This process of requesting for and retrieving relevant state information is referred to as *world management* in Mammoth. In the single server architecture, the protocol for retrieving state information by the clients was relatively simple since all sub spaces resided on one node. The client simply sends a state retrieval message for the sub space whose state is required. The server collects the state for that sub space and returns the information to the client which updates its local cache.

However, in the distributed server architecture, this becomes a challenge since the sub spaces are distributed over a number of servers. Therefore there is need for a more detailed protocol which can manage retrieval of state for the client over this

architecture. This section describes this protocol using the same example used in the previous section and gives a formal algorithm for it.

Protocol Overview

Assume client C_1 is connected to its home server S_1 and is interested in events occurring on sub space sub_2 hosted by S_2. Apart from subscribing to the remote channel ch_2 at S_1, it also needs to retrieve the current state of sub_2. It therefore sends a state retrieval request message for sub_2 to S_1 attaching its client identifier with the message. Client identifiers help server identify the clients connected to them. When S_1 receives the requests, it realizes that it does not host sub_2, therefore it looks up the server hosting sub_2 using `remoteChannelMap` hash table (since sub space sub_2 corresponds to logical channel ch_2) and finds out that it is S_2. S_1 then forwards the state retrieval request to S_2. When S_2 receives this request, it collects the state for sub_2 and sends it in the form of a state information message to S_1 attaching the client identifier for C_1 that was present in the state retrieval request. When this message is received by S_1, it simply relays it to C_1. C_1 upon receipt of the state information message, uses this information to update the state for sub_2. Algorithm 3 formalizes the game state retrieval protocol.

5.2.4 Player Migration - A detailed explanation

Player migration is referred to as the movement of players across sub spaces within the game world. Since a server can host more than one sub spaces, player migration can be of two types, *local* and *remote* migration. The following sections discusses both of them in greater detail.

Algorithm 3 Algorithm for Retrieving Game State

1: **At Client node C_i :**

2: **Received request of state for sub space k from upper layers. :**
3: Create ***GET_STATE***(C_i, ch_k) message.
4: Send ***GET_STATE***(C_i, ch_k) to Home Server S_i

5: **Upon receipt of message *STATE_INFO*$(state_k)$ from Home Server S_i :**
6: Send state to upper layers for processing.

7: **At Server node S_i :**

8: **Received *GET_STATE*(C_i, ch_k) from client C_i :**
9: **if** ch_k in *localChannels* **then**
10: Collect state information $state_k$ for sub space k
11: Send message to ***STATE_INFO***$(state_k)$ to C_i.
12: **else**
13: $S_j \leftarrow$ remoteChannelMap.get(ch_k)
14: Send ***GET_STATE***(C_i, ch_k) to S_j
15: **end if**

16: **Received *GET_STATE*(C_i, ch_k) from Server S_i :**
17: Collect state information $state_k$ for sub space k
18: Send ***STATE_INFO_RESP***$(C_i, state_k)$ ***to*** S_i)

19: **Received *STATE_INFO_RESP*$(C_i, state_k)$ from Server S_i :**
20: Send ***STATE_INFO***$(state_k)$ ***to*** C_i)

Local Migration

When a player migrates across sub spaces hosted by the same server, it is known as local migration. As a result, the client maintains connections with its current home server. However, as it has moved over to a different sub space, the interest management algorithm will require it to subscribe to newly interested sub spaces and unsubscribe from sub spaces that it is no longer interested in. It also requires the client to retrieve the current state of new sub spaces it is now interested in. Since these sub spaces can either be local or remote to the client's home server, the client uses the distributed publish/subscribe system and the game state retrieval protocol described earlier to manage the subscriptions and to retrieve the game state. Since the client does not change home servers, the session of the client remains the same on its home server.

Remote Migration

A player undergoes remote migration when it moves across sub spaces which reside on different server nodes. This section outlines the various steps that occur as a player remotely migrates across sub spaces using the continuing example of the distributed architecture.

Assume, player P_1 represented by client C_1 migrates from sub_1 (hosted by S_1) to sub_2 hosted by S_2. As it moves across the sub space boundary, it performs the following steps:

1. When C_1 detects transition across sub spaces, it looks up which server is responsible for the destination sub space (i.e., sub_2), It realizes that it is not the home server (using `localchannels` list it receives from S_1 upon connection).

2. It looks up the server which is responsible for sub space sub_2 (i.e., S_2) using `remoteChannelMap` hash table it had received from the rendezvous server when it entered the cluster.
3. It unsubscribes from all the sub spaces it has currently subscribed to.
4. It disconnects from S_1.
5. It establishes a TCP connection to S_2 and exchanges control information (such as retrieving S_2's `localChannel` list and a client identifier).
6. It subscribes to the sub spaces that the player is now interested in (dictated by the interested management scheme) and retrieves game state of all these sub spaces (using the same procedure as that used in local migration). The order in which these two operations occur is essential in order to avoid inconsistencies. More details about this is discussed later in this chapter.
7. Send a *start* message containing the identifier of the player migrating. When S_2 receives this message, it establishes a game session for this player associating it with this client.
8. At this point, player migration is complete and C_1 can start sending state updates to S_2.

State Management during Player Migration

As explained in Section 5.1.2, clients do not have any dynamic state information at startup. They get the state of the sub spaces they are interested in from server nodes that host these sub spaces. Using this state information, clients create local copies of dynamic objects present in that sub space. They then subscribe to the sub space and receive updates which they can apply to these objects to keep them

up-to-date. If a player enters a sub space now and performs some updates, the clients receiving these update would not be able to apply them as they don't have the local copy of this newly entered player's dynamic object.

Therefore, before P_1 migrates onto sub_2, all the clients which are already subscribed to this sub space will not have a local object for P_1 and therefore will not be able to apply the state updates that they receive for this player. Hence, S_2 must send a copy of the P_1 object to all the clients subscribed to sub_2 so that they can create a local copy of P_1. The following steps illustrate how this actually takes place. These steps occur when S_1 detects that P_1 is about to migrate (i.e., along with Step 1 of the Remote Migration Protocol).

1. Before P_1 is about to migrate, S_1 creates a PLAYER_MIGRATION message containing P_1's object and sends it to S_2.
2. S_2 upon receiving the object for P_1, multicasts it across the logical channel ch_2 effectively sending it to all client nodes subscribed to this channel.
3. Upon receiving the object for P_1, all clients add it to their local state for sub_2 and are now able to see the player and process its updates.

In case of local migration, as the player moves across the sub space boundary, the server multicasts the player object across the logical channel of the sub space to which the player migrates to. Clients add the received player object to their local game state in order to see the player.

5.2.5 Consistency during player migration and startup

Whenever a client starts up or its player migrates across sub spaces, it subscribes to the newly interested sub spaces and gets the state information. However the order

in which these two operations occur proves to be very essential to the correctness of the state received at the client.

Scenario 1 - State Retrieval before Subscriptions

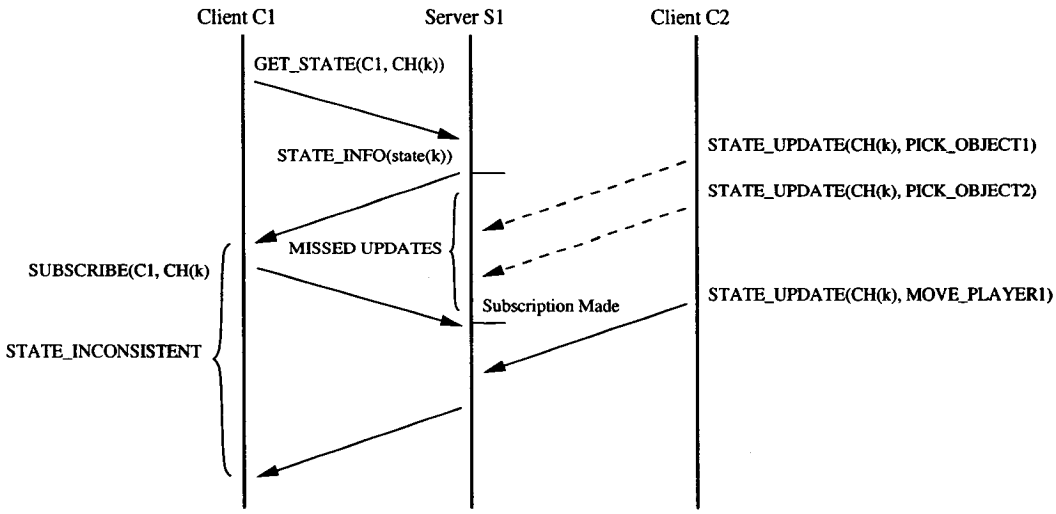


Figure 5–4: Loss of updates leading to incorrect game state

This case is described in Figure 5–4. In this case, the client C_1 tries to retrieve the state information for sub space sub_k before it subscribes to its corresponding logical channel (ch_k). As visible from the scenario described in the figure, there could be crucial state updates that may change the state of the sub space before the client subscribes to the logical channel and after the game state has been computed and sent. Therefore, C_1 completely misses these updates hence resulting in an incorrect view of the world. For example, if the missed updates correspond to picking up of some important objects by C_2, then the player at C_1 will still see them as present in the game world, which is wrong.

Scenario 2 - Subscriptions before State Retrieval

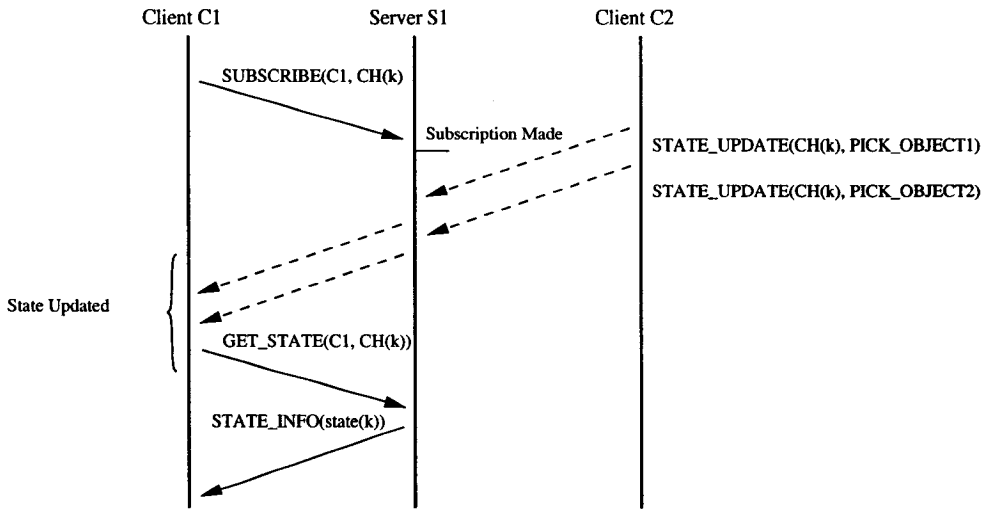


Figure 5–5: No loss of updates

This case is described in Figure 5–5. In this case, the client C_1 subscribes to the logical channel ch_k before it tries to retrieve the state information for the sub space sub_k. Lets examine this scenario at a greater level of detail. C_1 sends a subscription message (say SUB) to S_1 soon followed by a state retrieval message (say SR). Since, the connection between C_1 and S_1 is TCP, SUB is guaranteed to reach S_1 before SR. Now, if S_1 receives crucial state updates for sub_k (say U1 and U2) in between, they will be queued between SUB and SR in the incoming message queue. Hence, SUB will be processed first followed by U1, U2 and then SR. Therefore the state information collected after processing SR will include the state updates U1 and U2. Since U1 and U2 are processed before SR, their responses will be queued ahead of the state information message (say SI) in the message queue for C_1 as the same message queue for C_1 is used for sending direct as well as multicast messages. Therefore, C_1

receives U1 and U2 before SI. Since SI already contains the state updates specified by U1 and U2, applying them to C_1's local cache will be idempotent, hence C_1 discards these updates and applied state information in SI to its local cache without losing updates.

CHAPTER 6
Experimentation and Results

6.1 Introduction

Now that we have described our distributed server architecture for MMOGs in detail, we present results of the experiments performed on the implementation of the architecture in the Mammoth prototype with an aim to measure its scalability. The parameters measured were bandwidth consumption and CPU utilization at the server nodes as well as the overall latency experienced during game play. We give our initial results which we obtain by experimenting with the existing single server implementation of the Mammoth network engine explained in Chapter 3. We then present the results obtained from the distributed server architecture implementation in comparison with the single server, proving clearly in each instance our hypothesis that the system shows the scalability properties that it was designed for.

6.2 Experimental Setup

6.2.1 Physical Setup

In order to get results which reflect true performance, we ran our experiments using the complete distributed implementation using actual game messages on a real network rather than simulations. Each server node was configured to run on a different machine. The machines were connected via local area network of high bandwidth capacity. The experiments were performed on machines equipped with 3.40 Ghz Pentuim 4 processors hosting the server nodes. The machines ran the Linux

2.6.14 kernel with Sun JDK 1.5.0.04. A maximum of 16 server nodes were used to test the scalability of the architecture. The cluster administration server and the rendezvous server were also configured to run on the same local area network as the server nodes.

The clients were running on 38 different machines in different subnets of the department local area network. In our future work we propose to run the experiments with clients on different networks (preferably at a greater geographical proximity) to get more realistic network delay. However, we believe the results obtained from our local area network experiments are indicative of the scalability of our system when compared with the existing architecture.

For the purpose of our experiments we implemented a special *viewless* client without a graphical display. This allows us to run more clients per machine without the heavy graphics overloading it. Each viewless client was implemented as a thread and ten such threads were started from each machine, effectively representing 10 clients. Accommodating more clients per machine would have allowed us to have more players in the game, but that would have overloaded the machine where they were run, affecting the behavior of each client. Therefore, the number of clients we could experiment was limited by the number of machines available to us, allowing us to experiment with up to 380 clients. However, this number was large enough to provide a comparison of the scalability of our architecture under varying conditions.

6.2.2 Player behavior

The players at each client had an automated behavior described by a random algorithm. For the purpose of simplicity, we only allowed movement actions for

experimental purposes. Therefore, each player made a movement of a specific displacement at specific time intervals in a randomly selected direction. The seed for the random number was computed from the player identifier. This way the same set of random numbers were selected for each experiment (with same number of clients). Thus, we had the same traces of random movements making the scenarios identical. In our experiments, we used an interval of one second between generation of every movement update. We decided on this interval after observing traces obtained from actual gaming scenarios. Since we were using actual client implementations rather than just dummy message senders, the responses received back from the server nodes were processed at the clients and helped in deciding their next action, providing us with a more realistic behavior.

6.2.3 Game Setup and Experimentation Technique

For the purpose of testing our system performance, we created a special map consisting of 16 sub spaces (or static zones) of equal size and shape in the form of a 4x4 rectangular grid. The density of players was even across all sub spaces in the map so that no part of the entire game could be unevenly concentrated with players.

The cluster administration server was started initially and all server nodes connected to it to form the cluster. The rendezvous server was then started which obtained the cluster information. A remote script simultaneously started all 10 clients at each machine each connecting to the cluster through the rendezvous server to start sending state updates corresponding to their in-game movements. The player identifiers decided the starting position of a player in the game map were randomly

selected and assigned by the rendezvous server in order to have a more even distribution of players starting off in different areas of the map. However, the same set of random numbers were used for each experiment, so that the experiments remained identical.

6.2.4 Measurement of Parameters

The CPU consumption at each server node was measured at an interval of 250 ms during each experiment run and an average was computed. Both incoming and outgoing bandwidth was measured at the server nodes at a regular interval in order to measure the communication overhead and the average value was recorded. A latency monitoring utility was developed for estimating latency. This utility had a sending and receiving component. The sending component connects to a server node in the cluster and sends *probe* messages for a particular channel recording the time at which it was sent. The server node relays this message to all subscribers of that channel. The receiving component also connects to a server node and subscribes to the same channel to receive these probe messages. The latency is estimated as the difference between the time of receipt recorded by the receiving component and the sending time recorded by the sending component.

There were two ways in which we estimated latency:

1. **Direct Latency** This is an estimate of the amount of latency experienced by the clients for state updates occurring at sub spaces hosted by their home server. Figure 6–1(left) shows how direct latency is measured. The sending component connects to a server node and sends probe messages for a local channel. The

receiving component connects to the same server, subscribes to a local channel to receive these probe messages and calculates direct latency.

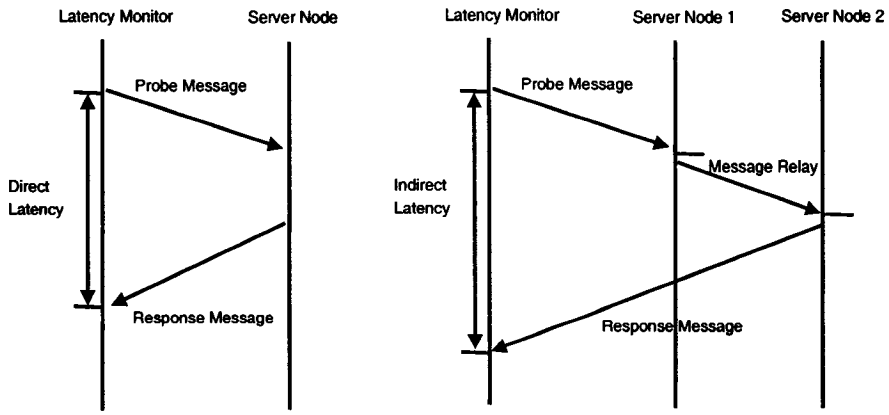


Figure 6–1: Latency Measurement, Left: Direct Latency, Right: Indirect Latency

2. **Indirect Latency** In a distributed architecture, it is imperative to measure the latency experienced by clients for state updates occurring at sub spaces hosted by servers other than their home server. This is estimated by indirect latency. Figure 6–1(right) shows how indirect latency is measured. The sending component connects to a server node and sends a probe message on a local channel. The receiving component connects to another server node and subscribes to the same channel (which is remote at this server) to receive this probe message and calculates the indirect latency.

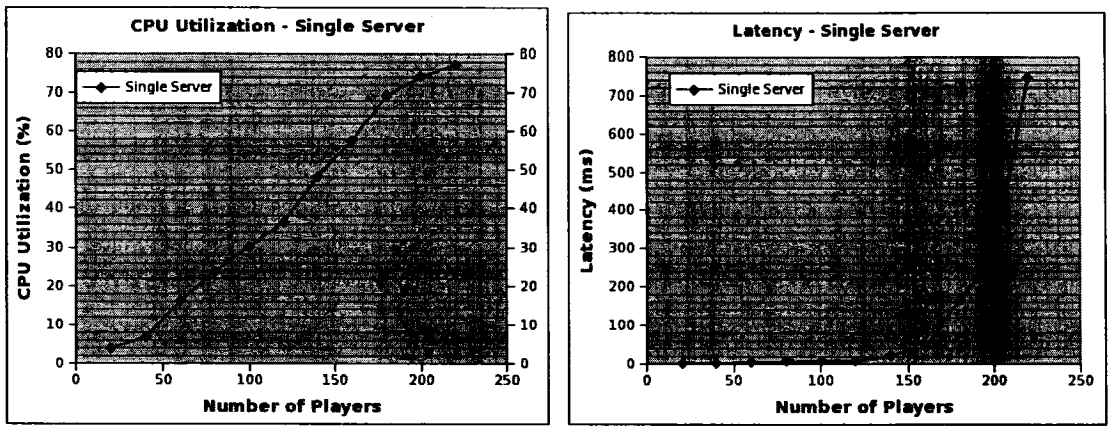
6.3 Single Server Experiments

This section presents results for experiments performed on the existing single server network architecture of the Mammoth prototype. Since the aim of our experiment was to evaluate the scalability of the system, we naturally took our measurements of CPU, bandwidth and latency with increasing number of clients. In order to

get the most accurate results, the measurement of parameters was commenced only after the required number of players were connected to the single server and had started sending updates. We stopped taking measurements before the first player stopped sending updates and disconnected.

6.3.1 CPU Utilization

This experiment was conducted with the aim to measure the CPU utilization at the single server with increasing number of clients. The experiment was repeated with increasing number of clients and the measurement of CPU utilization was taken when all clients were connected and were transferring state updates. Figure 6–2 shows the result of the experiments. As visible from the graph, the CPU consumption increased almost linearly with increasing number of players in the game. This is due to the increasing number of messages that needed to be processed from the rising number of players. Also, another major overhead was the serialization and deserialization of messages sent and received by the network engine. With more messages being sent and received, this overhead increased leading to an increase in CPU consumption. We could only take measurements for up to 220 simultaneously connected clients as the latency at this point became intolerable for proper playability (see next section).



(a) CPU Utilization (b) Latency

Figure 6–2: Single Server CPU Utilization and Latency

6.3.2 Latency

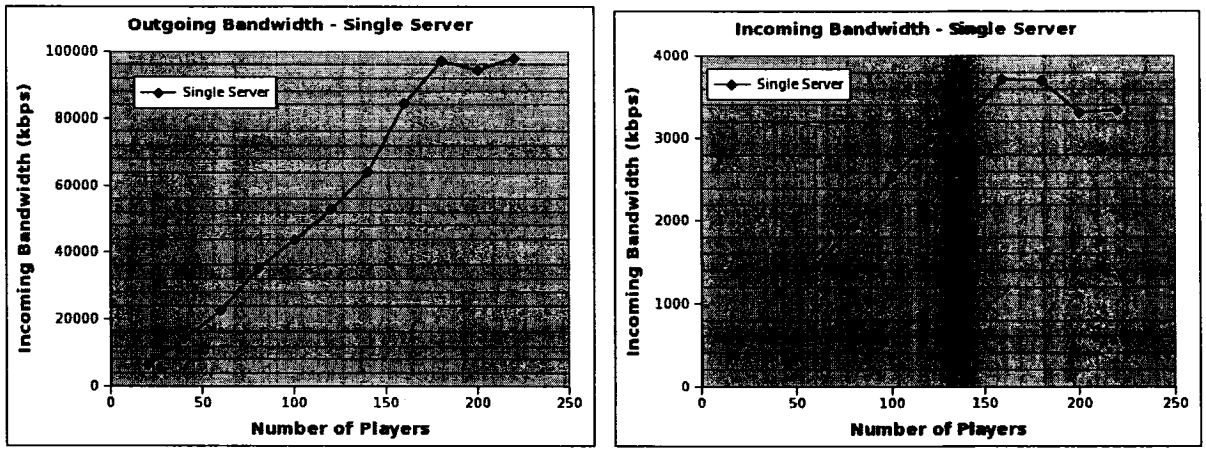
The latency of the single server architecture was measured under varying loads by experimenting with increasing number of clients. Since the experiment was on the single server architecture, we measured the direct latency. A certain number of probe messages were sent during the experiment and the time taken for them to reach the server and relayed back to measured. As we can see from Figure 6–2, the measured latency is between 3ms - 50ms for up to a 150 players in the game. However, there is a sharp increase in latency values when more than 160 players join the system. The main reason for this behavior is that the multiplexing unit at the network engine cannot process the large number of messages in the different message queues for all the logical channels fast enough leading to clogging of messages in these message queues. Therefore, it takes longer time for response messages to reach the clients. At 220 clients, latency values reached 745 ms which we deemed as intolerable for good playability.

6.3.3 Bandwidth Utilization

Both incoming and outgoing bandwidth utilization was measured at the server to determine the communication overhead. The incoming bandwidth refers to the messages being received by the server while the outgoing bandwidth was representative of all the multicasts/direct messages sent to the clients. Figure 6–3(a) shows the outgoing bandwidth consumption while Figure 6–3(b) shows that for incoming bandwidth. As visible in both cases, the bandwidth consumption increases with increasing number of clients as both the number of incoming and outgoing messages increases. Outgoing bandwidth consumption is much higher than incoming since for every message received for a channel k with n subscribers, there are n outgoing messages. From Figure 6–3(a), it is visible that the curve starts to flatten when it reaches the peak number of players that the server can handle. This is mainly because the server cannot process the large amount of incoming messages fast enough, hence these messages start queuing up in the message queues in the network engine and only a constant amount of messages are processed and sent out over the network.

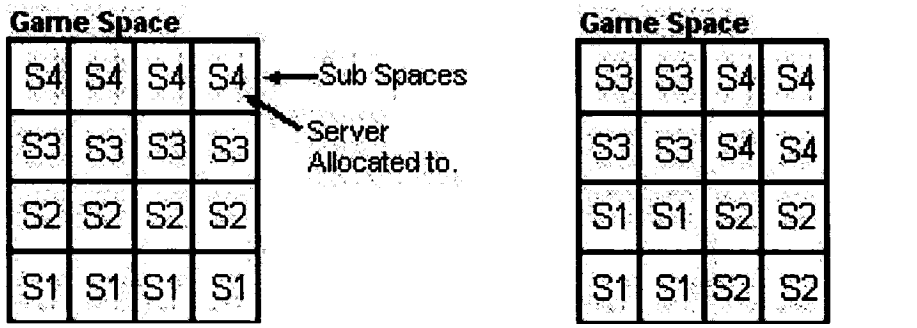
6.4 Distributed Server Experiments

This section now presents the results obtained by experiments performed using our distributed server architecture. For preliminary results, we used 4 server nodes in our cluster each in charge of 4 of a total of 16 sub spaces. Figure 6–4(a) and (b) illustrates configuration 1 and 2 respectively of mapping sub spaces onto server nodes. We used configuration 2 for comparing CPU and bandwidth utilization results with those obtained from the experiments on the single server implementation. We also analyze and compare the message statistics for both the configurations.



(a) Outgoing Bandwidth (b) Incoming Bandwidth

Figure 6–3: Single Server Bandwidth Consumption



(a) Mapping Configuration 1 (b) Mapping Configuration 2

Figure 6–4: Mapping Configurations

6.4.1 CPU Utilization

Figure 6–5 shows the CPU utilization at all four server nodes in the distributed server architecture (using mapping configuration 2) with increasing number of players in the game. From the results obtained we can see that the CPU utilization at all server nodes in the distributed architecture is much less than that for the single server scenario. Since each server node hosts only part of the game state, it receives and

processes a fraction of the total messages received by the single server. Further, the messages received from the other server nodes are not processed but relayed to the connected clients and do not contribute much to increase the processing overhead. Therefore, the CPU utilization at each server node in the distributed architecture is much lower than that for the single server. As a result, our distributed server architecture is able to scale up to 380 clients without incurring excessive CPU overhead at each server node.

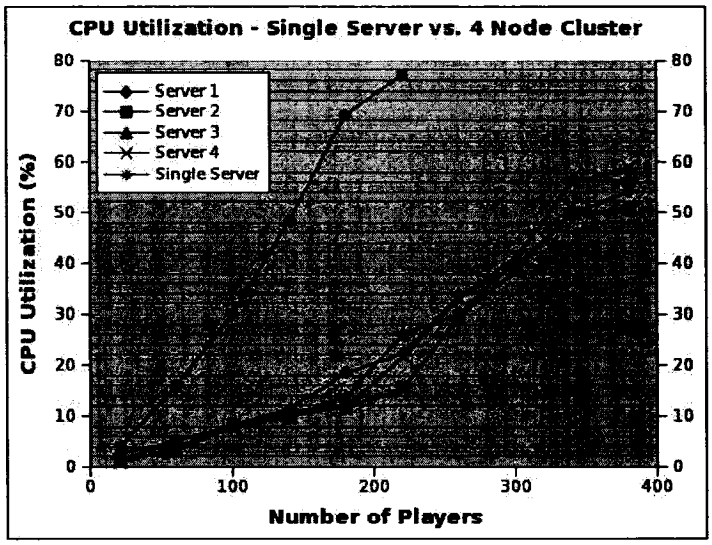


Figure 6–5: Scalability Experiments for CPU Utilization

6.4.2 Latency

Figure 6–6 shows the measurement of both direct and indirect (or across) latency values in the distributed server scenario. As we can see the latency increases steadily with increasing number of players. This is again due to longer time taken by messages to get through the message queues at the server nodes. As observed from

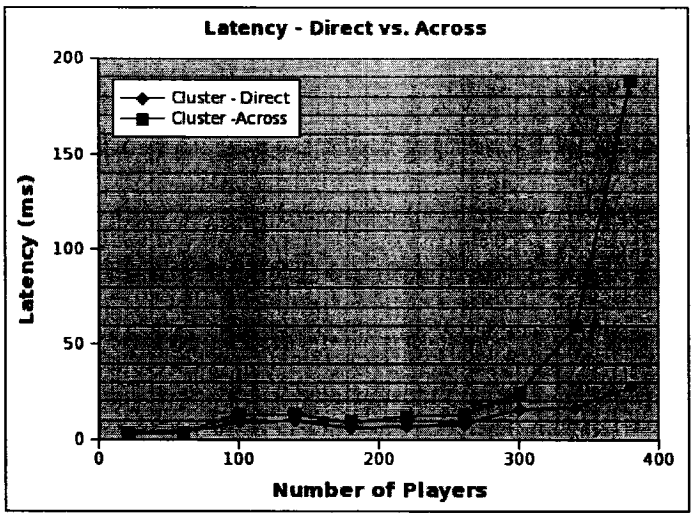


Figure 6–6: Direct and Indirect Latency for 4 Server Node Cluster

the graph, the indirect latency increases exponentially at 320 players, this is because of the large amount of time spent by the probe message in the message queues at the server node and the relay of the message between the servers. However, when compared to the latency measurements in the single server scenario (see Figure 6–7), the latency measured in the distributed server architecture was much lower for a larger number of players. The peak latency measure in the single server was 745 ms at a peak load of 220 players while the distributed architecture scales better with only 188 ms (indirect) latency with a load of 380 players.

6.4.3 Bandwidth and Message Statistics

Figure 6–8(a) compares the incoming bandwidth in the single server scenario with the average of the incoming bandwidth measured at each server node in the distributed server cluster (using mapping configuration 2). The incoming bandwidth at the single server decreases at overload as clients are unable to connect to the

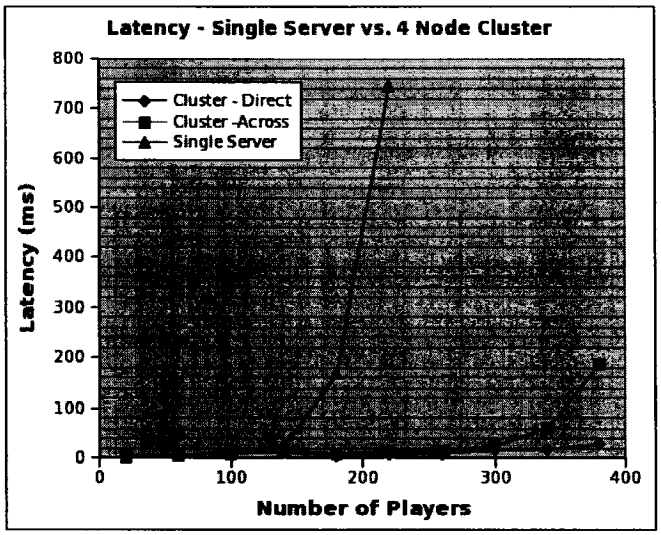
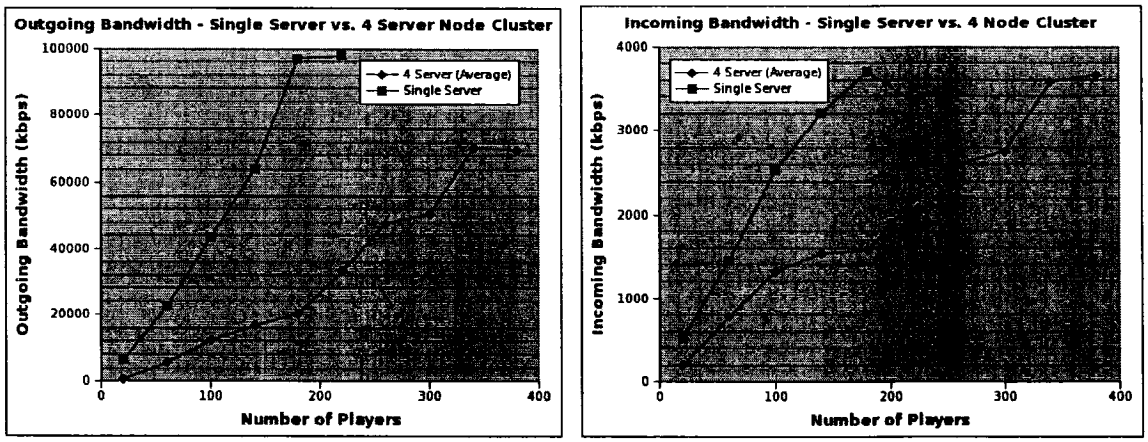


Figure 6–7: Scalability Experiments for CPU Utilization

already overloaded server and those which already are connected are not able to transfer messages as the server is too overloaded to process any more messages. In the distributed scenario however, the incoming bandwidth at the server nodes is much less than that compared to that in the single server scenario. Since each server node host sub spaces, they receive fewer updates compared to the single server. Also, due to interest management the updates received from other server nodes is just for a subset of sub spaces that its client is interested in. This leads to overall reduction of incoming bandwidth on average on each server node.

Figure 6–8(b) provides a similar comparison except now with the outgoing bandwidth consumption. Again, the architecture is shown to scale well as the maximum bandwidth consumption in the cluster at 380 players is much less than that for single server scenario where the server peaks out at 220 players. Since each server node receives fewer state updates and disseminates the response to a much less number



(a) Incoming Bandwidth (b) Outgoing Bandwidth

Figure 6–8: Bandwidth Comparison

clients/server nodes, the outgoing bandwidth on average for a server node is much less than that for the single server scenario.

	Messages from Clients					
Server	Movement Updates		State Retrieval		Subscriptions	
	Config 1	Config 2	Config 1	Config 2	Config 1	Config 2
1	3345	3722	743	943	743	943
2	4121	4873	1313	1357	1313	1357
3	3106	4332	1065	1122	1061	1122
4	3818	3035	633	842	633	842

Table 6–1: Statistics for messages received from clients.

We now look at the statistics of the messages received at server nodes in an experiment involving 120 players connecting to the cluster at the same time each making a total of 150 movements in a random direction. The statistics are compiled and compared for mapping configurations 1 and 2. Table 6–1 shows the statistics for messages received at server nodes from their connected clients. From the table, it is visible that all movement updates sent by the clients are roughly evenly distributed

	Messages from other Server nodes							
Server	Movement Updates		State Retrieval		State Info		Subscription	
	Config 1	Config 2	Config 1	Config 2	Config 1	Config 2	Config 1	Config 2
1	3764	3752	291	290	211	250	4	**4**
2	6021	2583	447	210	582	349	9	**5**
3	6943	3015	473	273	573	300	8	**4**
4	2767	4244	237	359	182	233	4	**4**

Table 6–2: Statistics for messages received from other server nodes.

	Type of Player Migration			
Server	Local Migration		Remote Migration	
	Config 1	Config 2	Config 1	Config 2
1	114	167	97	61
2	159	288	191	76
3	133	268	140	75
4	118	147	72	73

Table 6–3: Player Migration statistics.

across all server nodes in the cluster. State retrieval and subscription messages constitute approximately 12 to 15 percent of the total messages received from the clients while movement updates are 60 to 65 percent. In all instances, the number of subscription messages received from clients are equal to state retrieval messages, which is in concordance with our algorithm.

Table 6–2 shows the statistics for messages received at each server node from other server nodes in the cluster for both mapping configurations. Movement update messages refer to state updates sent to other server nodes which have subscribed to them on behalf of their clients, *state retrieval* messages are requests for state of sub spaces received from other servers, *state information* messages are sent in response to state retrieval requests and contain the requested state information. Subscription

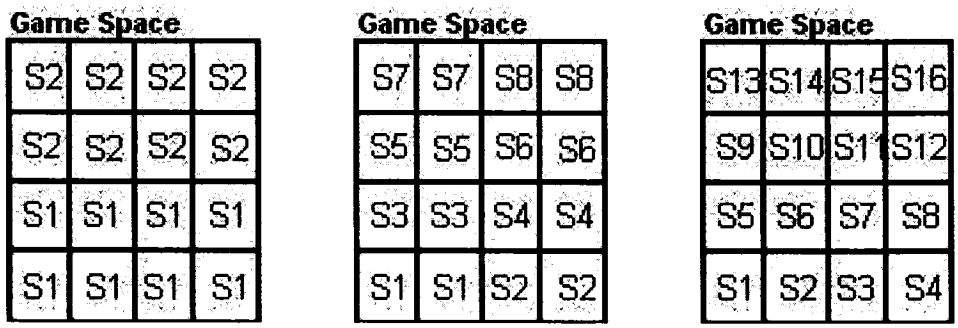
requests are messages received from other server nodes requesting subscription to local channels. From the table, we can see that the total messages exchanged between server nodes is much less for configuration 2 than for configuration 1 (approximately 30 percent reduction in overall inter server messages). In configuration 1, server 2 and 3 receive greater number of messages than server 1 and 4 since the sub spaces they host have greater number of neighboring sub spaces hosted on other nodes than server 1 and server 4. The results also show server 2 and 3 receive a larger number of subscriptions than server 1 and 4 due to the same reason. For configuration 2, variation in updates received is not as large as that for configuration 1 as total neighboring sub spaces at other servers remain same for all server nodes. Similarly, subscription requests from other server nodes is roughly the same for all server nodes.

Table 6–3 shows statistics for player migrations across sub spaces categorized by both local migration (across sub spaces hosted the same server) or remote migration (across sub spaces hosted on separate servers). As shown in the results, there were fewer remote migrations in case of configuration 2 than for configuration 4 due to the way the sub spaces were arranged. Since remote migrations are more complicated and involve a considerable overhead in switching server connections, we prefer a mapping configuration which avoids frequent remote migrations of players. Even though configuration 2 had more players migrations in total, most of them were local and the number of remote migrations are lower than that for configuration 1 (which had more remote than local migrations).

Therefore, from the above results, it is clear that the scheme in which sub spaces are mapped on to server nodes affects inter server communication. We infer from

our results that configuration 2 is a better mapping scheme of sub spaces over the server cluster as it results in lower inter server communication.

6.5 Effect of Varying Cluster Size



(a) 2 Server **(b) 8 Server** **(b) 16 Server**

Figure 6–9: Sub Space - Server Mapping

Based on the results presented in the previous section, we confirm that the 4 server node distributed architecture shows better scalability than the single server architecture. In this section, we go a step further by observing how the scalability of the system is affected as we increase the cluster size. We repeat our experiments for CPU and bandwidth utilization over clusters having 2/4/8 and 16 nodes and present our results comparing them with those for the single server architecture. For each cluster size, Figure 6–9 shows how the sub spaces are mapped on to the server nodes. We use configuration 2 (see Figure 6–4(b)) as the mapping scheme for the 4 node server cluster.

Figure 6–10 shows the average CPU utilization at server nodes for increasing cluster sizes. From the figure we can see that the worst performer is the single server architecture which can only support up to 220 clients. The increase in server nodes

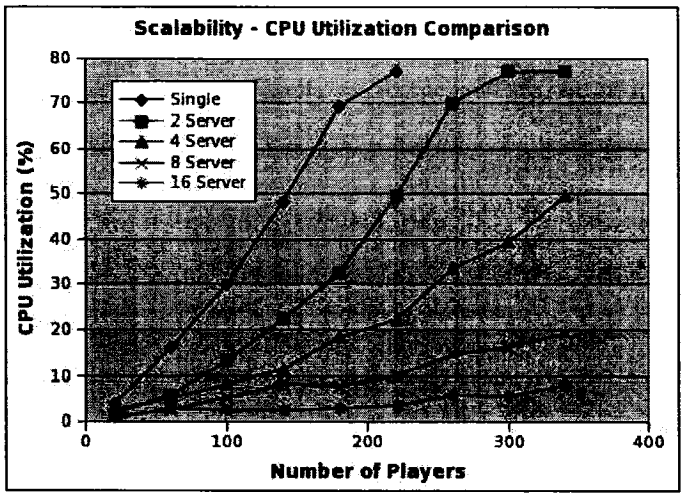
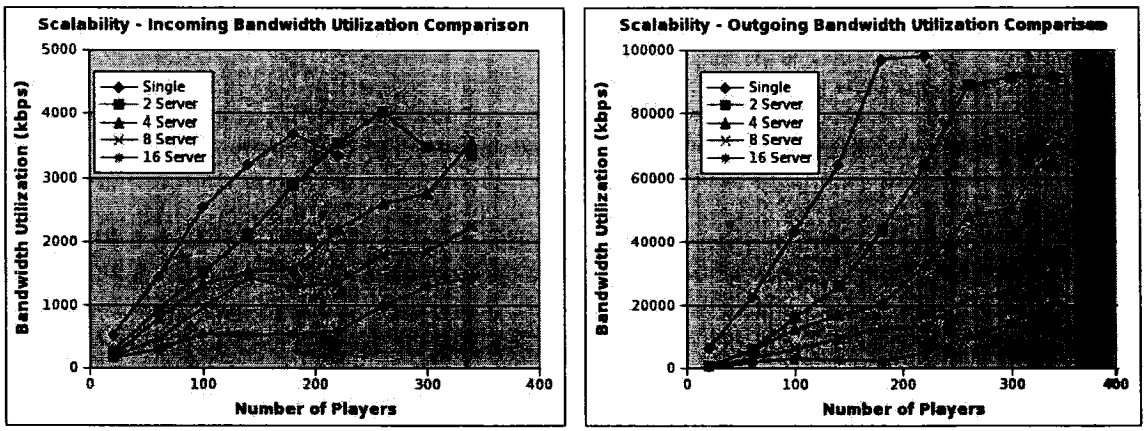


Figure 6–10: Scalability Experiments for CPU Utilization

in a cluster, decreases the number of sub spaces hosted per server node. Therefore, the number of state updates each server node is responsible to handle decreases. As a result, fewer state update messages are received and processed per server node reducing their average CPU utilization. Since messages from other server nodes are just relayed to connected clients rather than processed, they do not contribute to a considerable increase in CPU utilization.

Figure 6–11(a) and 6–11(b) show the average incoming and outgoing bandwidth utilization at the server nodes respectively. From the results we can see that the incoming bandwidth utilization per node on average decreases as the cluster size increases. This is again because, with increasing cluster size, each server nodes receives fewer state updates as it hosts fewer sub spaces. Although messages received from the other server nodes contribute to the incoming bandwidth as well, this contribution does not compensate for the sharp decrease in messages received directly



(a) Incoming Bandwidth **(b) Outgoing Bandwidth**

Figure 6–11: Bandwidth Comparison

from the client. With each server node hosting fewer sub spaces, the clients connected to them also decrease. Also, a server node only needs to send one copy of state update to other subscribed server nodes. Hence, there is a decrease in average outgoing bandwidth with increase in cluster size. We next examine the bandwidth consumption in detail with the help of message statistics collected for each scenario.

We conducted experiments with 120 players connecting to the server cluster and performing 150 movements in random directions. The experiments were performed using increasing cluster sizes and the message statistics were obtained. Figure 6–12 shows the messages received on average at each server node in the cluster from its connected client. We observed that the average messages received per server node is halved as the cluster size is doubled. This explains the reduction in CPU consumption and incoming bandwidth utilization. The figure also shows that the majority of messages received at the server nodes are movement updates. Subscription and

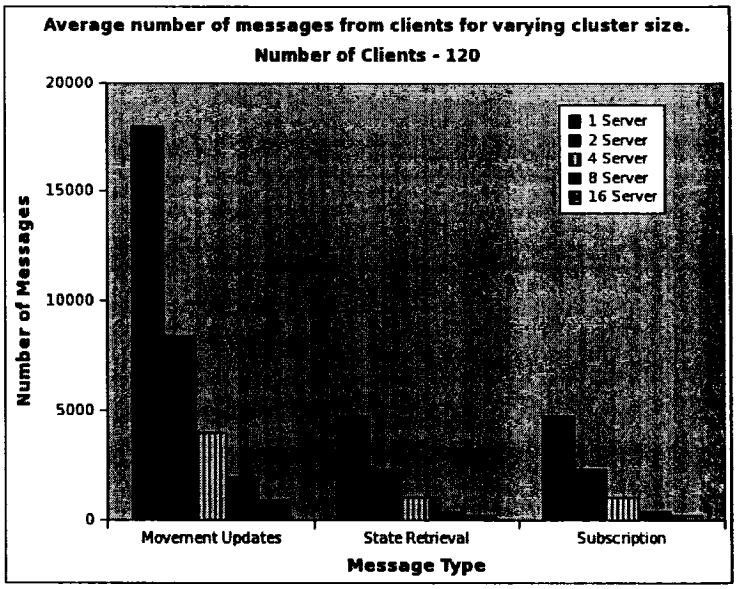


Figure 6–12: Messages received from clients.

state retrieval messages are equal in number and are comparatively much less than movement updates.

Figure 6–13 presents the number of messages received at each server node from other server nodes. The number of subscription requests from other server nodes are too few to be shown in the graph and are therefore omitted. As observed, the number of movement updates received from other server nodes does not reduce as dramatically as in the case of the updates received from the clients. Although, the sub spaces hosted by each server node decrease with increasing cluster size, it becomes increasingly dependent on other server nodes to receive updates occurring in other sub spaces. The statistics for cluster sizes 2 and 4 are similar, as in both cases, each server node subscribes to at most 4 sub spaces hosted on other server nodes. For cluster of size 8, each server node subscribes to either 3 or 5 sub spaces

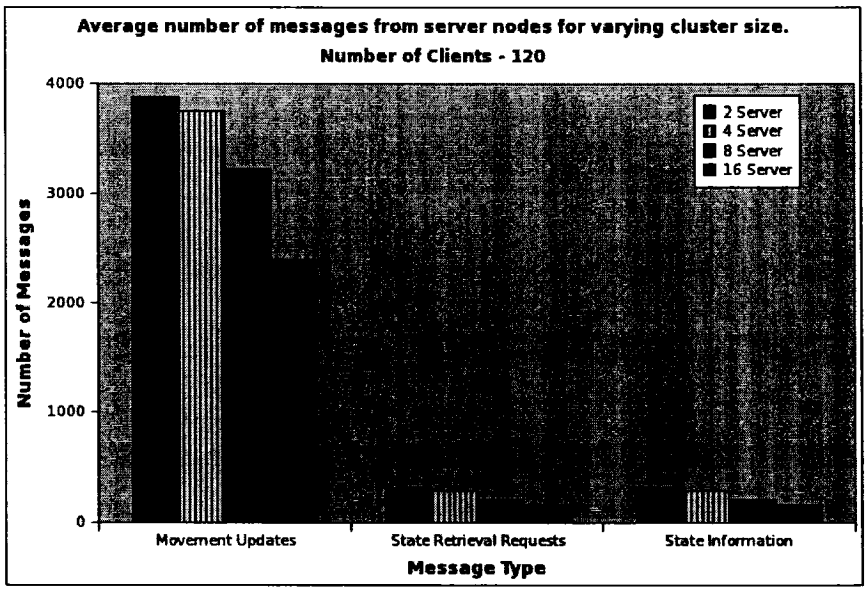


Figure 6–13: Messages received from other server nodes.

hosted by other server nodes, so the average inter server messages comes out to be slightly lower than that for clusters for size 2 and 4. Similarly for cluster of size 16, each server node subscribes to between 2 to 4 sub spaces hosted by other server nodes and hence the average movement update messages received by each node is lower.

CHAPTER 7
Integration into the Distributed Object Model

During the course of the development of the distributed server architecture, there was a change in the interest management and game state management model of Mammoth. In this chapter, we discuss these changes and describe how our distributed server can be adjusted to work with this new *distributed object model.*

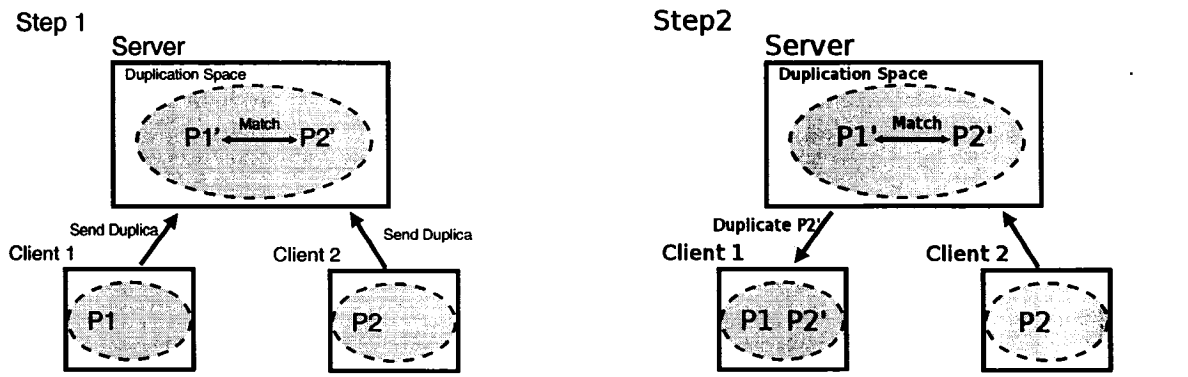
7.1 Introduction

The *distributed object model* is a paradigm created and forwarded by Quazal Inc. [1] and is implemented in the Mammoth prototype to improve its scalability. According to the description of the model given in [25], the entire game can be described as a collection of objects with each object having a certain state associated with it. These objects are distributed over all the machines taking part in the game. In multiplayer games, the change in state of an object on a machine should be made visible to other machines. Therefore, these objects need to be *duplicated* over the network to other player machines so that they can see these changes. An object can either be a *master* or a *duplica.* Master of an object is its controlling instance, i.e., where changes to its state are made. Duplicas are copies of the master object which are sent to other machines and are kept up-to-date so that their state is consistent with that of the master. In a client/server based MMOG, clients control their players and therefore they host the master copy of their player object and are

called its *duplication master*. The server contains the duplicas of all player and other dynamic objects in the game.

7.2 Publish-subscribe mechanism

For a client to see other players in the game, it needs to have a duplica of their object. However, duplicating all game objects over every station can lead to heavy bandwidth consumption and resource usage. *Duplication Spaces*, a concept developed by Quazal, provides greater control over which objects need to be duplicated over which machine, therefore providing a control over the number of duplicas for a master object. A duplication space is a space which contains all duplicated objects which are matched with each other using a *matching policy*. Each object in a duplication space can either be a publisher, subscriber or both. Publisher objects are those which publish the state updates that occur on them. Subscribers, as the name suggests, *discover* these publisher objects and subscribe to them for these updates. When a subscriber discovers a publisher in the duplication space, a duplica of the publisher object is created and sent to the duplication master of the subscriber.



(a) Player duplicas at server matched **(b) Player 2 object duplicated at client 1.**

Figure 7–1: Single Server Duplication Mechanism

In the client/server based system, the duplication space at the server contains duplicas of all player and other dynamic objects. A player is both a publisher and a subscriber while all other dynamic objects (such as food items, books) are just publishers. A user implemented boolean match function iterates over every publisher-subscriber pair in the duplication space at regular intervals to determine if a certain subscriber should discover a certain publisher. If this function returns true, a duplica of the publisher is discovered at the duplication master of the subscriber. This way, a client is able to see other objects in the game. The criteria used by the matching function to match a subscriber to a publisher is implementation specific and can be based on things such as the location of objects, the distance between them or the sub spaces in which they reside. Figure 7–1 illustrates the entire process, client C_1 and C_2 contain the master copy of player objects P_1 and P_2 respectively. In Step 1, these objects are duplicated at the duplication space of the server as P_1' and P_2'. The matching function at server 1 returns true for the pair P_1'-P_2'. Hence, P_2' is discovered by P_1' and duplicates its copy at its duplication master, i.e., client 1.(as show in Step 2).

7.3 Implementation in Mammoth

This section discusses the single server architecture of Mammoth which is based on this model [7]. A client is the duplication master for its player's object whose duplica is created at the server when the client connects to it. A state update made on the master object at the client is sent to the server where it is applied to its duplica. The server contains a duplication space object which maintains separate lists for publishers and subscribers. The server adds the duplica of player object to

both these lists while duplicas for other dynamic objects are added to the publisher list only.

7.3.1 Matching and subscription

The implementation of the matching function at the server can be user defined. At a regular *refresh rate*, the matching function is executed for every publisher-subscriber pair obtained from the lists in the duplication space. When the matching function for a particular publisher-subscriber pair returns true for the first time, the server creates a message containing the duplica copy of the publisher object and sends it to the client which is the duplication master of the subscriber object. There is a logical channel associated with each publisher object. The server subscribes this client to the logical channel of the publisher object. All state updates made to a publisher object are disseminated to all clients subscribed to its logical channel. This way, clients can see relevant players and in-game objects and receive their updates.

7.3.2 Invalidation

If the match function returns false for a previously matched publisher-subscriber pair, the server sends an *invalidation* message to the duplication master of the subscriber and removes its subscription from the publisher's logical channel. The duplication master, upon receiving the invalidation message removes the duplica of the publisher object from its duplication space. This way, a client can no more see the objects which are irrelevant to it and does not receive any updates for it.

7.4 Cell based distribution

In the single server scenario with one duplication space, all publisher-subscriber matching computations are done at one machine, i.e., the server. However, game

sessions with a large number of players and in-game objects will require a large amount of computations to match all possible publishers and subscribers leading to increased resource consumption and poor scalability. Therefore, in order to reduce the resource consumption at the server, we need a scheme to distribute the matching task over many server nodes. Quazal's duplicated object model provides a solution to accomplish this by dividing the duplication space into smaller *cells* each containing a subset of the publishers and subscribers. These cells can be hosted by different server nodes. Matching computations can be performed between publishers and subscribers in each cell, thereby distributing the computation load among the different server nodes. A cell is a partition of the duplication space which can group *similar* publishers and subscribers on a separate machine.

7.4.1 Cell Match Function

In order to determine whether a publisher or a subscriber belongs to a cell, we need to define a boolean `CellMatch` function for each cell in our system. The `CellMatch` for a given cell takes as an argument a publisher or a subscriber object and returns true if that object belongs to that particular cell. The `CellMatch` function and other pertinent cell information is encapsulated in a *cell object.* The server hosting a cell is known as the *duplication master* for the corresponding cell object. A duplica of each cell object is distributed at every server node in the system. The `CellMatch` function of each cell is executed for all publisher and subscriber objects on a server node at a regular refresh rate. If the function returns true, the publisher or the subscriber object is duplicated at the server node which is the duplication master of the cell object. Once `CellMatch` for all cells has been performed at every

server node, the match function is executed for all possible publish-subscribe pairs at the duplication master of each cell. The definition of the `CellMatch` function is also left user defined depending on the definition of a cell, i.e., the criteria by which publishers and subscribers are grouped into a cell on a server node.

7.4.2 Example

Figure 7–2 illustrates a system consisting of two cells cell 1 and cell 2 hosted by server nodes 1 and 2 respectively. Clients 1 and 3 connect to these server nodes and duplicate their player objects P_1 and P_3 on them. The duplica of the cell object for cell 1 is present on server 2 (denoted by Cell 1') and contains the `CellMatch` function for cell 1. For this example, we consider the invocation of the `CellMatch` function for cell 1 on all publisher and subscriber objects on server 2. As seen in from the figure, the `CellMatch` returns true for the duplica P_3'.

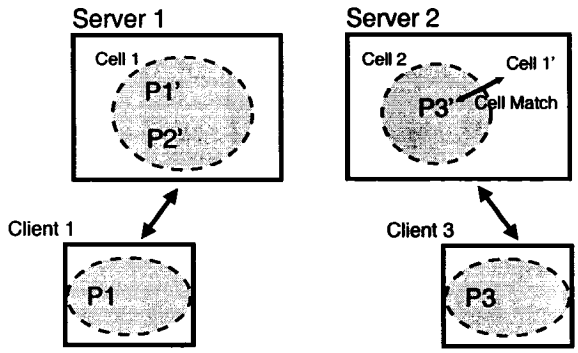


Figure 7–2: (a) Step 1 - Cell Match

Upon a cell match, the player object P_3' is duplicated on the duplication master of cell 1, i.e., server 1. This way, P_3' is discovered by server 1. Figure 7–3 illustrates this process.

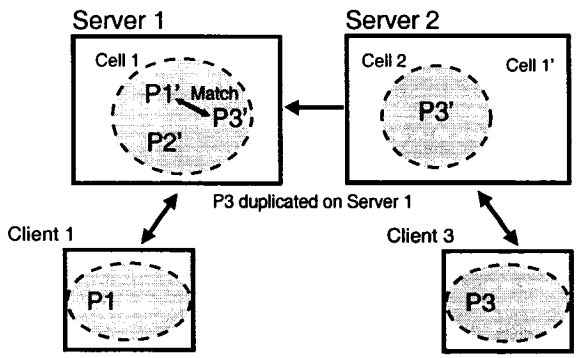


Figure 7–3: (b) Step 2 - Duplication to Server Node

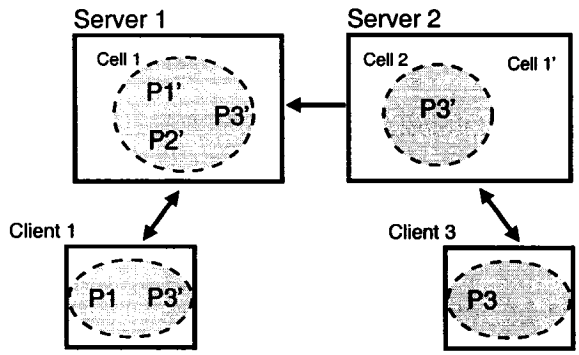


Figure 7–4: Step 3 - Match and Duplication to Client

Once P_3' has been duplicated on server 1, the match function is executed at cell 1 on this server as shown in Figure 7–4. The match function is executed for each publisher-subscriber pair in cell 1 which now also includes P_3'. As shown in the figure, we assume that the match function returns true for P_1' and P_3'. Hence, a copy of P_3' is duplicated at client 1. This way, player 1 can see player objects which previously resided on other server nodes.

7.5 Integration of the distributed server architecture.

Using the concept of cell based distribution of publishers and subscribers provided by Quazal, we can subdivide the resource intensive task of matching over several workstations allowing us to support more players in a gaming session. In this section we propose how we can integrate the cell based distribution scheme using the distributed server architecture that we have developed. We show how the server nodes in our architecture can host cells and use its existing primitives to define protocols of communication which can implement the cell based distribution scheme.

We explain our implementation using concrete definitions of the three abstractions described previously, i.e., the match policy, cells and the cell match function. However, the implementation is not specific to these definitions and can be used with alternate definitions as well.

7.5.1 Sub spaces

The game world in our MMOG is subdivided into a collection of small rectangular sub spaces of equal size. These sub spaces are much smaller in comparison to the sub spaces described in previous chapters and are larger in number for more fine grained interest management. We use the term *tiles* instead of sub spaces for them. Although there can be other shapes in which the game world can divided such as triangles [7] or hexagonal (honey comb) grids [24], we use rectangular tiles for sake of simplifying our discussion.

7.5.2 Match policy

Our match policy matches a subscriber to a publisher if the publisher is located on a tile which shares an edge with the tile in which the subscriber is present (i.e.,

adjacent tile). The match function is executed periodically for all publisher and subscriber pairs present on each server node. As stated earlier, there can be many other approaches which define the match policy. For example, a distance based approach matches two objects if they are within a specific distance from each other.

7.5.3 Cells and the Cell Match function

We define our cell as a named collection of tiles. All the player or in-game objects present on these tiles are said to belong to that particular cell. A cell has a *hard* and a *soft* boundary. The edges of the outermost tiles that form a cell form its hard boundary. When a player leaves or enters a cell when it moves across its hard boundary. A player inside the tiles of a cell can be interested in updates occurring on tiles present in other cells. Given our match policy, this is usually the case for outermost tiles in a cell which share an edge with tiles in other cells. These tiles (of other cells) are included in the soft boundary of our cell. A client's host server is the one which hosts the cell its player is currently in (considering the hard boundary).

For a given cell and an object, the `CellMatch` function for that cell returns true if the object is located in either the soft or the hard boundary of the cell. Hard and soft boundaries for all cells can be precomputed before the game starts and hence the definition of the `CellMatch` function is known at all server nodes before the game starts. The `CellMatch` function for each cell is present on all server nodes in the cluster. This function executes periodically for all objects present on the server node.

7.6 Communication Scenarios

We use an example to explain the main scenarios which can occur. Consider two cells, $Cell_1$ and $Cell_2$ hosted by server nodes S_1 and S_2 respectively. Clients C_1

and C_2 are connected to server nodes S_1 and S_2 respectively as their players P_1 and P_2 are on the cells hosted by those nodes.

A DUPLICATION_MESSAGE contains the copy of an object and is sent to a machine where a duplica of the object needs to be created. Upon connection, a client sends a DUPLICATION_MESSAGE containing their player object to their home server. Dynamic objects are created inside the cells they are located in. The servers that host these cells are the duplication masters for these dynamic objects.

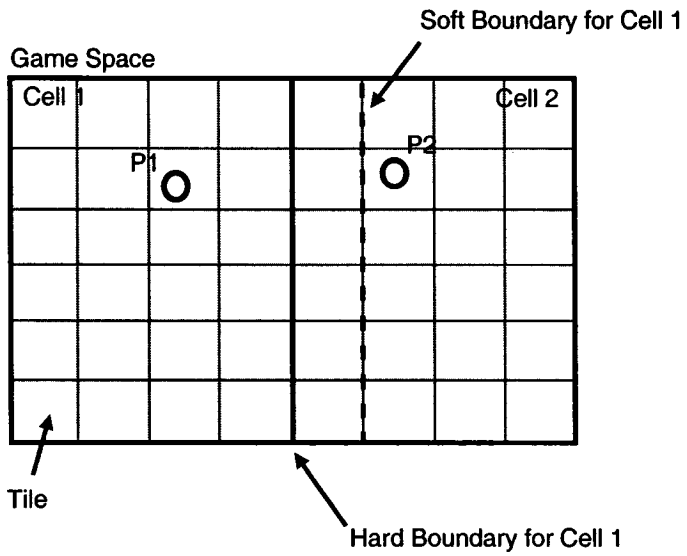


Figure 7–5: Scenario 1 - No Cell Match

7.6.1 Scenario 1 - No cell match

Figure 7–5 shows a scenario when players P_1 and P_2 are moving around in their respective cells. The match function at each server returns false as there is no match among the players present on each server. The `CellMatch` function for each cell on all nodes also returns false.

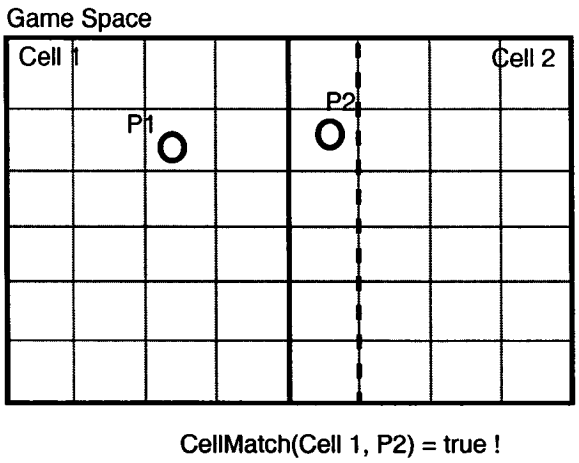


Figure 7–6: Scenario 2 - Cell Match

7.6.2 Scenario 2 - Cell match

Figure 7–6 shows player P_2 moving into the soft boundary of $Cell_1$. Now, when the `CellMatch` function for this cell executes on S_2 with P_2 as an argument, it will return true. At this point, S_2 determines that S_1 hosts $Cell_1$ (using `remoteChannelMap`). It then sends a DUPLICATION_MESSAGE containing the object for P_2 to S_1 and adds S_1 to the subscribers list for P_2. S_1, upon receipt of this message, creates a duplica of P_2 and adds it to the subscribers and publishers list. It also creates a logical remote channel for P_2 and adds it to its list of available channels.

If there is a state update made to P_2 at S_2, it will be forwarded to all the nodes subscribed to P_2 (including S_1). When S_1 receives this state update, it applies it to P_2's duplica and at the same time disseminates it to all nodes subscribed to P_2. This way, P_2 is discovered at S_1. The same procedure occurs in the opposite direction for S_1 to discover P_1 . If P_2 moves out of the soft boundary of $Cell_1$, the `CellMatch`

function at S_2 will return false and an invalidation message will be sent to S_1, S_1 will also be removed as one of the subscribers of the local channel for P_2 at S_2.

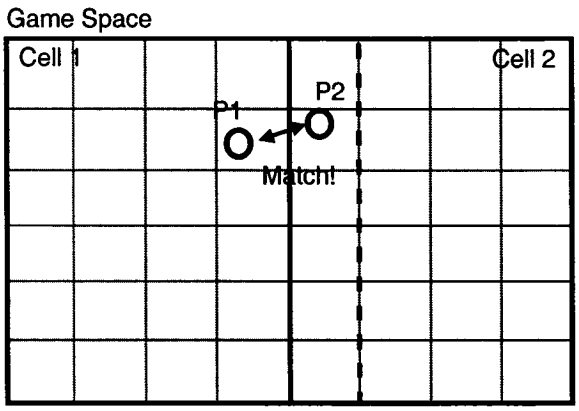


Figure 7–7: Scenario 3 - Match

7.6.3 Scenario 3 - Match Policy

Now that P_2 is duplicated at S_1, it will be included in the matching operations when the match function is next executed for all publisher-subscriber pairs at S_1. From Figure 7–7, we can see that there is a match between P_1 and P_2 on S_1, since they are in adjacent tiles. The match function returns true and a duplica of P_2 is send to the C_1 (the duplication master of P_1) and C_1 is also added to the subscribers list of P_2's logical channel. All updates that S_1 receives from S_2 for P_2 are sent to C_1 as it is subscribed to P_2. The same occurs to C_2 whose player P_2 matches with P_1 on server S_2 and hence, is subscribed to P_2.

7.6.4 Scenario 4 - Player Migration

Figure 7–8 shows the case where P_2 actually moves across the hard boundary into $Cell_1$. At this point, P_2 needs to change its home server to S_1. S_2 detects this

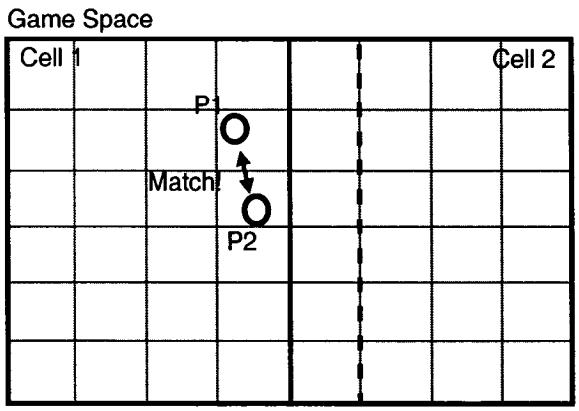


Figure 7–8: Scenario 4 - Player Migration

transition and sends a redirection request to P_2 containing the address of S_1 and disconnects it. The migration process from now on can be divided into two phases.

Phase 1 - Invalidation

S_2 then sends an *invalidation* notification to all nodes subscribed to P_2. This basically notifies all subscribers that P_2 is no longer available at S_2. S_2 then closes the logical channel for P_2 and removes its duplica.

Clients upon receiving the invalidation notification remove the duplica of P_2. When S_1 receives the invalidation notification from S_2, it forwards it to all nodes subscribed to P_2 at S_1. It also removes the duplica of P_2 and closes its logical channel.

Phase 2 - Transition

Upon receipt of the redirection request, C_2 connects to S_1 and duplicates P_2 on it (by sending a DUPLICATE_MESSAGE to S_1). S_1 adds P_2 to its publisher and subscriber lists and creates a local logical channel for it. All the required subscriptions are made when the match policy and the `CellMatch` function execute the next time. In our case, the next time the match policy executes, it matches P_1 with P_2. A

duplica of P_2 is created at C_1 and it receives all state updates on P_2 directly from S_1. Furthermore, S_2 becomes a subscriber of P_2 because the tile it resides in belongs to the soft boundary of the cell maintained by S_2.

7.6.5 Player Disconnection

A player at any point may decide to leave the game. In this case, the client sends an invalidation message for its player object to its home server and disconnects. The server removes this object from its list of publishers and forwards the invalidation message to all the subscribers of this object. It then removes the local channel for this object.

7.7 Dynamic Objects

Dynamic objects (such as food items, flowers) can also be picked up by players and migrated across servers. When an object is picked up, it is removed from the server node and is associated with the player object (e.g., added to the player's inventory). Upon removal, this server node sends an invalidation message to all nodes subscribed to that object. This way, the object is no more visible to any other player in the game. When the player drops this object on a cell, a new instance of it is created at the server responsible for this cell. This server becomes the new duplication master of this object.

7.8 Refresh Interval

The refresh interval for the execution of the match policy function and the `CellMatch` function at server nodes decides how fast a player discovers objects within the game. It is imperative that this refresh interval is set to an optimum value. A

large refresh interval can lead to late discovery of objects while a very small refresh interval can lead to unnecessary resource consumption at the server nodes.

7.9 Formal Algorithms

This section gives the formal algorithms for the communication protocols and the functions discussed above. Algorithm 4 introduces the data structures and presents the steps involved when the `CellMatch` function is executed at a server node. Algorithm 5 presents the behavior at all server nodes upon receipt of the different types of messages. Algorithm 6 presents the sequence of operations performed at the client and the server during player migration.

Algorithm 4 Cell Match refresh algorithm

1: Data Structures:
2: `duplicated` : Hash table maps objects to the machine they are replicated **to**.
3: `HostMap` : Hash table maps cells to their host machines.
4: `sub` : Linked List of subscriber objects on a server node.
5: `pub` : Linked List of publisher objects on a server node.
6: `channelList` : Hash table which maps object to its channel.
7: `cellTotal` : List of all cells in the game.
8: $subscribers(S_i, ch_k)$: List of subscribers for ch_k on S_i

9: **function** boolean `CellMatch`($Cell_k$, obj)
10: //checks if object obj is in weak boundary of $Cell_k$
11: **end function**

12: **function** `refreshCell`
13: **for all** obj in pub and sub **do**
14: **for all** $Cell_k$ in cellTotal **do**
15: **if** (obj, S_j) $\notin$ `duplicated` **then**
16: **if** CellMatch($Cell_k$, obj) **then**
17: S_j = `HostMap`.get($Cell_k$)
18: Send DUPLICATION_MESSAGE(obj) to S_j
19: ch_{obj} = `channelList`.get(obj)
20: S_j to $subscribers(S_i, ch_{obj})$)
21: Create entry (obj, S_j) in `duplicated`
22: **end if**
23: **else**
24: **if** !`CellMatch`($Cell_k$, obj) **then**
25: `invalidate`(obj, $Cell_k$, S_t)
26: **end if**
27: **end if**
28: **end for**
29: **end for**
30: **end function**

31: **procedure** `invalidate`(obj, $Cell_k$, S_t)
32: Send INVALIDATE_MESSAGE(obj) to S_t
33: ch_{obj} = `channelList`.get(obj)
34: Remove S_t from $subscribers(S_i, ch_{obj})$;
35: Remove entry (obj, S_t) from `duplicated`
36: **end procedure**

Algorithm 5 Message processing at Server node

1: Additional Data Structures:
2: `session` : Hash table maps objects identifiers to object instances.
3: **At Server node** S_i :

4: **Upon receiving DUPLICATE_MESSAGE from** S_j **for obj**
5: Create obj on S_j.
6: Add obj to pub and sub.
7: Create channel ch_{obj} for obj and add to channelList.
8: Add (objID, obj) to `session`

9: **Upon receiving STATE_UPDATE(objID, updateInfo) from** S_j **or** C_t
10: Retrieve obj for objID from `session`.
11: Apply updateInfo on obj.
12: ch_{obj} = `channelList`.get(obj)
13: **for all** Servers/Clients SC_i in $subscribers(S_i, ch_{obj})$ **do**
14: Forward STATE_UPDATE(objID, updateInfo) to SC_i.
15: **end for**

16: **Upon receiving INVALIDATE_MESSAGE(objID) from** S_j
17: Retrieve obj for objID from `session`.
18: Remove obj from pub and sub.
19: ch_{obj} = `channelList`.get(obj).
20: **for all** Servers/Clients SC_i in $subscribers(S_i, ch_{obj})$ **do**
21: Send INVALIDATE_MESSAGE(objID) to SC_i.
22: **end for**
23: Remove ch_{obj} from `channelList`.
24: Remove (objID, obj) from `session`.

25: **Upon receiving INVALIDATE_MESSAGE(objID) from** C_1 **for obj**
26: Retrieve obj for objID from `session`.
27: Remove obj from pub
28: ch_{obj} = `channelList`.get(obj)
29: **for all** Servers/Clients SC_i in $subscribers(S_i, ch_{obj})$ **do**
30: Send INVALIDATE_MESSAGE(objID) to SC_i
31: **end for**
32: Remove channel ch_{obj} from `channelList`
33: Remove all entries for obj in `duplicated`

Algorithm 6 Player Migration

1: **Migration of player P_k of client C_k**

2: **At Server node S_i :**

3: **When P_k from $Cell_i$ hosted by S_i moves to $Cell_j$ hosted by S_j**
4: Send REDIRECTION(S_j) to C_k
5: Disconnect C_k
6: Remove P_k from pub
7: ch_{P_k} = `channelList`.get(P_k)
8: **for all** Servers/Clients SC_i in $subscribers(S_i, ch_{P_k})$ **do**
9: Send INVALIDATE_MESSAGE(objID) to SC_i
10: **end for**
11: Remove channel ch_{P_k} from `channelList`
12: Remove all entries for obj in `duplicated`

13: **At Client node C_k :**

14: **Upon receipt of REDIRECTION(S_j) from S_i**
15: Connect to S_j
16: Send DUPLICATION_MESSAGE(P_k) to S_j

7.10 Applicability of the distributed server architecture

We propose that our distributed server architecture (described in Chapter 4) can easily integrate with this scheme due to the availability of primitives and a distributed infrastructure to support this form of communication. Our architecture provides a cluster of server nodes with complete inter server communication. The Rendezvous server can be reused to redirect a client to its home server based on which cell its player is currently located in. We already have the concept of logical channels and associated subscription lists which can be reused here. Except, while the previous implementation mapped a sub space to a logical channel, we map a publisher object to a logical channel. Multicast and direct message capability can also be reused here to send state updates or duplication/invalidation messages.

7.11 Performance Discussion

Since all the game objects are distributed across a group of server nodes, the computation overhead due to the execution of the match function at each server node will decrease. Each server node can therefore accommodate greater number of players without overloading, thereby increasing the overall scalability of the system. The `CellMatch` function executes in linear time since it simply iterates through all the dynamic objects on a server. Therefore, it should not induce too much overhead at a server node. The latency experienced by clients during game play is not expected to be high for updates which occur on server other than their home server (also known as indirect latency) as the server nodes are located on a Local Area Network with sufficiently large bandwidth. Also, in the previous chapter, we had seen that indirect latency remained low even for a large of number of participating players.

Finally, with the fine grained interest management techniques used in this approach, there will be lesser number of messages transferred between servers, reducing the inter-server communication.

7.12 Comparison with the previous Mammoth Architecture

Comparing the distributed object model with the previous interest management scheme for mammoth, we come across its various advantages and a few shortcomings which are discussed as below :

1. In the distributed object model, subscriptions are made directly to a publisher (player or dynamic object) unlike the previous scheme where subscriptions were made to sub spaces. Combined with effective matching algorithms, this scheme provides a finer grained interest management which substantially limits the number of updates received at the clients to the most pertinent ones. As a result, fewer messages are transferred among server nodes and between server and client nodes leading to reduced bandwidth consumption.
2. In our previous scheme, the responsibility for subscribing/unsubscribing to logical channels was implemented at the client machines inducing on them an additional overhead. It also exposed the subscription mechanism to players in the game causing a security concern. In the distributed object model, the *server nodes* are responsible for subscribing/unsubscribing their clients to the appropriate logical channels. Therefore, the clients are able to operate without the knowledge of the underlying subscription mechanism and at the same time receive only the pertinent state updates from the server nodes.

3. In the earlier scheme, the clients had to retrieve the collective state of all the objects in the sub space they were interested in leading to an exchange of large-sized state update messages causing an increased bandwidth consumption. In the distributed object model, state updates are made at player/dynamic object level. This leads to smaller and fewer state updates thereby lowering the bandwidth consumption.
4. Managing subscriptions at the player/dynamic object level in an MMOG with thousands of participating players and in-game objects can lead to a large number of logical channels increasing the complexity of the task for the servers to manage these subscriptions. Furthur, a game scenario where there are a large number of players moving rapidly around the game space can lead to hefty subscribing and unsubscribing causing an increased overhead at the server nodes.

CHAPTER 8
Conclusion and Future Work

8.1 Conclusion

We introduce a distributed server architecture to improve the scalability of Massively Multiplayer Online Games. In this architecture, a group of machines called server nodes form a cluster. The game space is divided into smaller *sub spaces* and each server node hosts one or more of these sub spaces. These nodes are responsible to maintain the state of these sub spaces. Clients connect to their home server, i.e., the server node which hosts the sub space where their player currently resides and transfer state updates directly to it. Clients also switch home servers when their player moves across sub spaces in the game. A distributed publish/subscribe system allows clients to receive updates from sub spaces hosted at servers other than its home server.

The scalability of the architecture is evaluated using its implementation in the network engine of the Mammoth Prototype under real network conditions. The CPU utilization, incoming/outgoing bandwidth at each server node along with latency experienced during game play is measured with increasing number of players. The performance is compared against the existing Mammoth network engine which is based on the single server architecture. The results we obtain confirm that the distributed server architecture is able to scale to a greater number of players than the single server architecture without CPU or bandwidth overload at the server nodes

with acceptable latency. We also observer that the scalability of the architecture improves by increasing the number of server nodes in the cluster.

8.2 Future Work

This section outlines the work we propose to perform in the future on the distributed server architecture

8.2.1 Integration of Distributed Server Architecture in the Distributed Object Model

In this previous chapter, we introduced the distributed object model and outlined an scheme to integrate it with our distributed server architecture to improve its scalability. However, due to time limitations, we were not able to implement our approach in the Mammoth MMOG. Therefore, we propose to adapt our existing implementation of Mammoth so that it can be easily integrated with our distributed server architecture and also implement the cell based distribution scheme described in Section 7.4. We also intend to evaluate the scalability of this system and compare it with the existing single server implementation using different definitions of the `CellMatch` function and the matching policy.

8.2.2 Load Balancing and Dynamic Sub Space Management

Currently, the system supports only static partitioning of sub spaces over the server nodes. However, this scheme cannot handle *hot spot* situations where players flock to a specific sub space due to an in-game activity overloading the server hosting this sub space. Therefore, we propose a dynamic sub space migration mechanism which can re-partition sub spaces into smaller sub spaces during overload and migrate them to other lightly loaded server nodes. The sub space migration protocol takes care of the migration of sub spaces to another server node and the redirection of the

clients to this new node. The cluster administration server (CAS), being a central administrative entity can monitor the load at each server node during game play and trigger the sub space migration protocol when the load at a particular node exceeds a given threshold.

Load balancing algorithms determine the efficient way to reallocate sub spaces over the server nodes during migration. This should be done in such a way that sub spaces that are close to each other remain on the same server leading to lower inter-server communication. Also, the maximal load per server node should be minimised. We intend to experiment with different load balancing algorithms using our sub space migration protocol and determine which one perform best, i.e., provide the least maximum server load during game play.

8.2.3 Fault Tolerance and Backup

We propose to make our system fault-tolerant by replicating the game state at server nodes to backup nodes. These backup nodes can either be other server nodes in the cluster or additional nodes kept specifically for this purpose. The server nodes can actively forward the state updates they receive to these replicas to keep them consistent with the actual game state. The cluster administration server (CAS) can keep track of the availability of the server nodes and the location of their replicas. During the event of failure, the CAS delegate the responsibility of the failed server node to its replica and redirect all the clients to it.

8.2.4 Extension to Peer-to-Peer architecture

The distributed server architecture can be extended to a peer-to-peer architecture by removing the concept of server nodes and distributing sub spaces over special

client nodes called *coordinators*. Coordinators of a particular group of sub spaces can form self organizing multicast groups with other clients interested in those sub spaces to disseminate updates. Our distributed publish/subscribe system can be extended to allow clients in a multicast group to subscribe to updates that occur in another multicast group.

8.2.5 Wide Area Network (WAN) Experiments

In order to obtain realistic network delays and bandwidth consumption, we intend to redo the experiments mentioned in Chapter 6 in a WAN setting. In this scenario, the server nodes will form a cluster via Local Area network with clients connecting from geographically distant locations. Although, the Local Area Network (LAN) settings used in the current experimental methodology serves the purpose of comparing the scalability of our system against the single server architecture, having our clients distributed over a WAN can give us a better measure of *true* latency experienced during game play which can help us optimize our system architecture for better performance.

References

[1] Quazal inc. http://www.quazal.com/.

[2] Y. Amir and J. Stanton. The Spread Wide Area Group Communication System. Technical Report CNDS 98-4, The John Hopkins University, 1998.

[3] G. Armitage. An experimental estimation of latency sensitivity in multiplayer Quake 3. *11th IEEE International Conference on Networks*, pages 137–141, September 2003.

[4] Electronic Arts. Ultima online. http://www.uo.com/.

[5] R.K. Balan, M. Ebling, P. Castro, and A. Misra. Matrix: Adaptive Middleware for Distributed Multiplayer Games. *6th International Middleware Conference*, pages 390–400, November-December 2005.

[6] A.R. Bharambe, S. Rao, and S. Seshan. Mercury: A scalable publish-subscribe system for internet games. *NetGames*, April 2002.

[7] Jean-Sebastien Boulanger. Interest Management for Massively Multiplayer Games. Master's thesis, McGill University, August 2006. To be submitted.

[8] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, 2002.

[9] J. Chen, B. Wu, M. Delap, B. Knutsson, H. Lu, and C. Amza. Locality aware dynamic load management for Massively Multiplayer Games. *Principles and Practice of Parallel Programming*, June 2005.

[10] E. Cronin, B. Filstrup, and A. Kurc. A distributed multiplayer game server system. Technical report, University of Michigan, Ann Arbor, 2001. UM EECS589 Course Project Report.

[11] Blizzard Entertainment. Starcraft. http://www.blizzard.com/starcraft/.

[12] Blizzard Entertainment. World of warcraft achieves a new milestone with two million paying subscribers worldwide. http://www.blizzard.co.uk/press/050614.shtml, June 2005.

[13] Sony Entertainment. Everquest. http://eqplayers.station.sony.com/index.vm.

[14] Sony Entertainment. Everquest 2. http://everquest2.station.sony.com/.

[15] J. Faerber. Traffic modelling for fast action network games. *Multimedia Tools Appl.*, 23(1):31–46, 2004.

[16] T. Fritsch, H. Ritter, and J. Schiller. The effect of latency and network limitations on MMORPGs (A field study of EverQuest2). *NetGames*, October 2005.

[17] Epic Games. Unreal. http://www.unrealtournament.com/.

[18] S. Hu and G. Liao. Scalable Peer-to-peer Networked Virtual Environment. *SIGCOMM*, August-September 2004.

[19] id Software. Doom. http://www.idsoftware.com/games/doom/doom3/.

[20] id Software. Quake 1. http://www.idsoftware.com/games/quake/quake/.

[21] T. Iimura, H. Hazeyama, and Y. Kadobayashi. Zoned Federation of Game Servers: a Peer-to-peer Approach to Scalable Multiplayer Online Games. *SIGCOMM*, August-September 2004.

[22] DFC Intelligence. Challenges and opportunities in the online game market - executive summary. http://www.dfcint.com/game_article/june03article.html, June 2003.

[23] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for Massively Multiplayer Games. *IEEE Infocomm*, March 2004.

[24] Michael R. Macedonia, Michael J. Zyda, David R. Pratt, Donald P. Brutzman, and Paul T. Barham. Exploiting reality with multicast groups. *IEEE Comput. Graph. Appl.*, 15(5):38–45, 1995.

[25] Quazal. *Duplication Spaces^{TM} Quazal Multiplayer Connectivity White Paper*, January 2002. http://www.quazal.com.

[26] P. Rosedale and C. Ondrejka. Enabling player-created online worlds with grid computing and streaming.

www.gamasutra.com/resource_guide/20030916/rosedale_01.shtml, September 2003.

[27] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale Peer-to-peer systems. *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, November 2001.

[28] McGill University School of Computer Science. The Minueto framework. http://minueto.cs.mcgill.ca/, 2005.

[29] NC Soft. lineage. http://www.lineage.com/.

[30] B.D. Vleeschauwer, B.V.D. Bossche, T. Verdickt, F.D. Turck, B. Dhoedt, and P. Demeester. Dynamic microcell assignment for Massively Multiplayer Online Gaming. *NetGames*, October 2005.

[31] S. Yamamoto, Y. Murata, K. Yasumoto, and M. Ito. A distributed event delivery method with load balancing for MMORPGs. *NetGames*, October 2005.