Massively Multi-User Online Platform for Large-Scale Applications

by

Allen Yen-Cheng Yu
Bachelor of Applied Science
University of Toronto, June 2004

A thesis
presented to Ryerson University
in partial fulfillment of the
requirements for the degree of
Master of Applied Science
in the Program of
Electrical and Computer Engineering

Toronto, Ontario, Canada 2007

Author's Declaration

I hereby declare that I am the sole author of this thesis or dissertation. I authorize Ryerson University to lend this thesis or dissertation to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis or dissertation by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Massively Multi-User Online Platform for Large-Scale Applications

by

Allen Yen-Cheng Yu
Master of Applied Science
Department of Electrical and Computer Engineering
Ryerson University, 2007

Abstract

Many large-scale online applications enable thousands of users to access their services simultaneously. However, the overall service quality of an online application usually degrades when the number of users increases because, traditionally, centralized server architecture does not scale well. In order to provide better Quality of Service (QoS), service architecture such as Grid computing can be used. This type of architecture offers service scalability by utilizing heterogeneous hardware resources. In this thesis, a novel design of Grid computing middleware, Massively Multi-user Online Platform (MMOP), which integrates the Peer-to-Peer (P2P) structured overlays, is proposed. The objectives of this proposed design are to offer scalability and system design flexibility, simplify development processes of distributed applications, and improve QoS by following specified policy rules. A Massively Multiplayer Online Game (MMOG) has been created to validate the functionality and performance of MMOP. The simulation results have demonstrated that MMOP is a high performance and scalable servicing and computing middleware.

Acknowledgments

I would like to express sincere appreciation to my supervisor, Professor Eddie Law, for the support, assistance, guidance, and encouragement he has shown me during my stay here at the Ryerson University. Working with him has been an educational, challenging and thoroughly enjoyable experience.

I would also like to thank my friends in the Ubiquitous Communications and Security (UCS) Laboratory for all the good times and memories. You really made my life more enjoyable there.

Special thanks to my girlfriend Ming, and my friends Chester Chen, Felica Hsu, Alan Kok, Peter Lee, Tony Lee, and Felica's boyfriend Lawrencè, who reviewed and proofread the thesis despite having no interest in the subject. Without your encouragement and editing assistance, I would not have finished this thesis. So thank you.

Dedication

I would like to dedicate all my work to my parents, my girlfriend and my brother. It could not have been possible without their love and support.

Contents

Acknowledgments iv

Dedication v

List of Figures x

List of Tables xii

1 Introduction 1
- 1.1 Computing Architectures . . . 1
- 1.2 Peer-to-Peer (P2P) Networks . . . 2
- 1.3 Application-Level Quality of Service (QoS) . . . 3
- 1.4 Middleware . . . 5
- 1.5 The Need of Massively Multi-user Online Platform (MMOP) . . . 6
- 1.6 My Contributions . . . 6
  - 1.6.1 Design of MMOP Architecture . . . 6
  - 1.6.2 Protocol and Operation Designs of MMOP . . . 7
  - 1.6.3 Design of Distributed 2D Shooting Game on MMOP . . . 8
- 1.7 Thesis Organization . . . 8

2 Technology Background 10
- 2.1 Grid Computing Fundamentals . . . 10
- 2.2 Peer-to-Peer (P2P) Structured Overlay . . . 12
- 2.3 Quality of Service (QoS) Background . . . 14
- 2.4 Large-Scale Online Application: MMOG . . . 16
  - 2.4.1 The Butterfly Grid . . . 18
  - 2.4.2 BigWorld Technology . . . 19
- 2.5 Summary . . . 19

3 Massively Multi-user Online Platform (MMOP) Architecture 20
- 3.1 Virtual Organizations (VOs) Construction Service . . . 20
  - 3.1.1 Bit Torrent (BT) Protocol . . . 24

3.2 Distributed Data Access Service 26
3.2.1 Paxos 27
3.2.2 Etna 28
3.2.3 RAMBO and RAMBOII 29
3.2.4 Sigma 30
3.2.5 Distributed Semaphore (DISEM) Service 31
3.3 Application Deployment Service 32
3.4 QoS Management Service 34
3.4.1 QoS Monitoring Service 34
3.4.2 QoS Provisioning Service 36
3.5 Distributed Timing Service 37
3.6 Summary 38

4 Massively Multi-user Online Platform (MMOP) prototype implementation 39
4.1 Development Testbed 39
4.1.1 Development Testbed Hardware 40
4.1.2 Development Testbed - Software 40
4.2 Network Layer Implementation 41
4.2.1 Java Remote Method Invocation (RMI) 42
4.2.2 TCP Socket 42
4.2.3 Java New IO (NIO) 43
4.3 Virtual Organization Construction Service 44
4.3.1 Overlay Modules 44
4.3.2 Virtual Organization Construction Algorithm 48
4.4 Distributed Semaphore (DISEM) Implementation 49
4.4.1 Data Refreshing Service 51
4.4.2 Variable Declaration 51
4.4.3 Access Variable 52
4.4.4 Write Variable 53
4.5 Application Deployment Service 56
4.6 QoS Management Service Implementation 58
4.6.1 QoS Monitoring Service Implementation 58
4.6.2 QoS Provisioning Service Implementation 61
4.7 Distributed Timing Service Implementation 62
4.8 Summary 63

5 Distributed 2D Shooting Game Design **64**
5.1 Game Features and Objectives 64
5.1.1 Detailed Game Entity Design 65
5.2 Client Design 66
5.2.1 Graphical User Interface (GUI) Layout 67
5.2.2 Artificial Intelligence (AI) Design 70
5.2.3 Client Side Latency Compensation 72
5.3 Server Design 75
5.3.1 Bandwidth Conservation Strategies 77
5.4 Summary 80

6 Performance Evaluation **81**
6.1 Testbed Setup and Evaluation Framework 81
6.1.1 Testbed - Hardware 82
6.1.2 Testbed - Software 82
6.1.3 Self-similar Traffic Generator 83
6.2 Overlay Protocol Results 85
6.3 Distributed Semaphore (DISEM) Results 86
6.3.1 DISEM: Latency versus Data Size 87
6.3.2 DISEM: Throughput 89
6.3.3 DISEM: Request Rejection Rate 90
6.3.4 DISEM: Number of replicas 91
6.4 Distributed 2D Space Shooting Game Results. 93
6.4.1 MMOP: Packet Drop Rate 94
6.4.2 MMOP: Updates per Second 95
6.4.3 MMOP: Latency 97
6.4.4 MMOP: Total simulation time 97
6.5 Summary 100

7 Conclusion **101**
7.1 Future work 102

Appendices

A Massively Multi-user Online Platform (MMOP) Deployment Script **105**

B Setting up Hierarchical VO **107**

C Script for Space Shooter Simulation on Testbed **110**

References **113**

List of Acronyms **117**

List of Figures

1.1 Possible server cluster model 2
1.2 Simplified P2P network 3
1.3 Network middleware model 5

2.1 P2P key distribution in structured overlay 14
2.2 An in game screen shot of popular MMOG - World of Warcraft 17

3.1 VO hierarchy and encapsulation 21
3.2 Basic Bit Torrent network structure 25

4.1 MMOP middleware structure 40
4.2 Superimposed One Hop Lookups structure used for membership change notification 46
4.3 VO and SubVO construction algorithm 49
4.4 DISEM write request algorithm 54
4.5 A stream-based time synchronization 62

5.1 GUI for SPACE SHOOTER 67
5.2 Title for SPACE SHOOTER 68
5.3 Menu for SPACE SHOOTER 68
5.4 Main gaming area for SPACE SHOOTER 69
5.5 Radar and statistics for SPACE SHOOTER 70
5.6 AI design for player simulation 71
5.7 Timeline for entity interpolation 73
5.8 Quadrant division for SPACE SHOOTER game server 76
5.9 Direct collision detection versus XY entity sorting 79

6.1 The testbed network setup 82
6.2 Single VO setup for evaluating performance of One Hop Lookups and Chord 85
6.3 Latency of three replicas DISEM on One Hop Lookups and Chord versus number of nodes in the overlay 86
6.4 Single VO setup for evaluating performance of DISEM 87
6.5 Latency of different number of replicas versus data size 88

6.6 Semaphores granted per second versus incoming request rate 89
6.7 Request rejection rate with variable data size versus incoming request rate . 91
6.8 Latency with different number of replicas versus incoming request rate . . . 92
6.9 Application VO setup for MMOP simulations 93
6.10 Packet drop rate with different VO setups versus number of simulated players 95
6.11 Update per second versus number of simulated players 96
6.12 Average latency versus number of simulated players 98
6.13 Total simulation time with different VO setup versus number of simulated players 99

B.1 VO hierarchy setup demo 107

List of Tables

3.1 Simplified DISEM API for MMOP 31
3.2 Deployment policy parameters 33
3.3 Application resource parameters 33

4.1 Example finger table of a Chord node 47
4.2 DISEM API for MMOP 51

5.1 Attributes of basic entity in SPACE SHOOTER 65
5.2 Attributes of aircraft in SPACE SHOOTER 66

Chapter 1

Introduction

Supported by the Internet boom in the mid to late 1990s, online services have grown rapidly, both in the types of services and the numbers of users. Aided with widely available broadband technologies of late, users are generally demanding higher level of Quality of Services (QoSs) from the applications. As a result, designs of server architectures are challenged with higher than expected volumes. Commonly used client-server architecture does not scale well with increasing number of users, nor does it provide any flexibility when user demand changes. Furthermore, it is subject to bottlenecks due to its centralized infrastructure. To solve these problems, a fully distributed Peer-to-Peer (P2P) Grid computing middleware for large-scale applications called Massively Multi-user Online Platform (MMOP) is proposed.

1.1 Computing Architectures

The client-server model of network topology is very common in online applications. Typically, there is a group of client machines that want to access the services provided by a server machine. Each client connects directly to the server and is serviced individually. This kind of topology is commonly used in small-scale online applications such as online forums.

A single server works well for small-scale applications, and the numbers of active users do not exceed several hundreds. However, when dealing with thousands or more users simultaneously, this model falls short. Typically, for a larger-scale online application, the service is hosted by a set of machines with dedicated responsibilities as shown in Figure 1.1. This setup

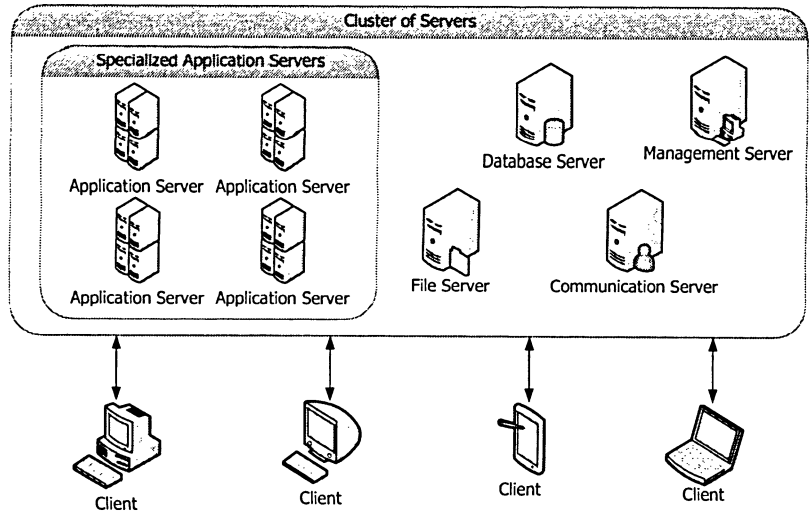


Figure 1.1: Possible server cluster model

is called cluster computing, and is usually deployed to improve performance and/or availability over traditional single server setup. Grid computing is targeted to serve even larger-scale applications whose requirements cannot be met by the cluster computing topology. The Grid computing architecture extends cluster computing and allows the interconnection of heterogeneous clusters. Consequently, resources are shared dynamically, and consistency among clusters can be maintained by the Grid architecture.

1.2 Peer-to-Peer (P2P) Networks

The Internet have evolved much further than just rudimentary communications nowadays. Large amount of computing resources can be gathered through means of Peer-to-Peer (P2P) networks. Figure 1.2 illustrates a basic organization of P2P or overlay network. Each machine in the network is identified as a node. Unlike the client-server model that distinct responsibilities are assigned for clients and servers, all nodes in the overlay may be treated equally and have exactly identical functionality. Although this illustration is overly simplified for modern P2P networks, each node should be capable of carrying out same functions as any other. Modern P2P networks are able to organize themselves into more efficient

structures through means of different overlay algorithms. These algorithms will be discussed in Section 2.2.

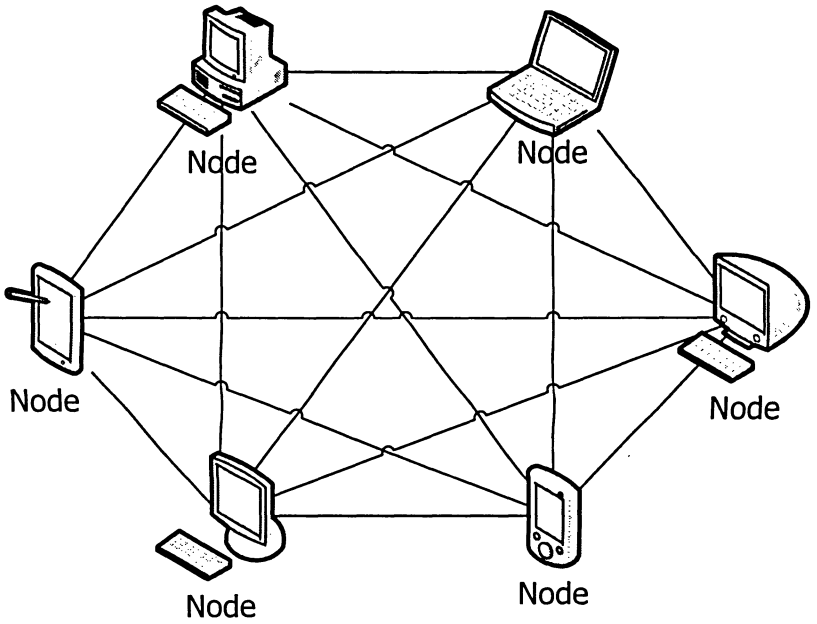


Figure 1.2: Simplified P2P network

P2P networks offer advantages over traditional client-server models with lower cost and higher flexibility. This is our intention in this thesis to take the advantages of P2P networks into a Grid computing architecture. Applications can then be serviced in various locations in a P2P Grid architecture. The bottleneck problem described previously in client-server model is alleviated. Immediate benefits such as better bandwidth utilization and reduced network latency can be expected when switching to the Grid architecture.

1.3 Application-Level Quality of Service (QoS)

As users' demands increase for large-scale online services, maintaining an adequate level of QoS becomes a pressing issue. Moreover, as networks evolve into larger and more complex arrangements, it is extremely difficult to configure and enforce QoS on a per device basis.

To make situation worse, the adaptation of P2P overlay poses challenges on the efficient utilization of heterogeneous hardware equipment. Therefor, it raises the need of an automated QoS management service with the following considerations:

Application Predictability: Computing resources need to be carefully organized when special application requirement is present. For example, computational resources such as CPU cycle and memory must be allocated when executing complex scientific applications. Bandwidth should be managed to meet the needs of real-time voice, video, or multimedia applications. Furthermore, a large amount of traffic on networks may be non-critical. It is important to be able to differentiate such traffic. Therefore, the ability to predict and provision the QoS of an application can improve the overall system performance.

Hardware Utilization: Various computing resources can be connected by P2P overlay in an uniform Grid computing architecture. Without a set of clear guidelines, resources cannot be allocated efficiently. Hardware resources such as CPU cycle, memory utilization and persistent storage spaces should be closely managed when QoS is considered.

Bandwidth Utilization: In general, bandwidth is an expensive resource in today's Internet. With growing demand, networks without managed QoS may face service disruption when traffic congestion or network failure occurs.

Several approaches have been used in the past to manage network QoS. In the traditional approach to network management, an administrator maintains QoS configuration with detailed information about the capabilities of each device in network. In addition, network status is monitored by administrator to manage each device individually. This approach works reasonably well in small networks; however, it is infeasible for large and more complex networking systems. Therefore, policy-based QoS management is preferred on the P2P Grid architecture. Applications can be managed through automated policy reinforcement with a set of predefined policy rules. This type of automated service is ideal for P2P Grid architecture to efficiently utilize its heterogeneous resources.

1.4 Middleware

Companies have been developing large-scale online applications such as online trading systems. The main issues faced by these types of applications are the constraints of networking and computational resources requirements. The common approach is to provide a specialized tool or framework to support such development. These tools or frameworks are designed solely with the specific requirements of original applications in mind. Therefore, very few of these frameworks can be reused for different products. The lack of generality prevents the framework to be reused by other developments.

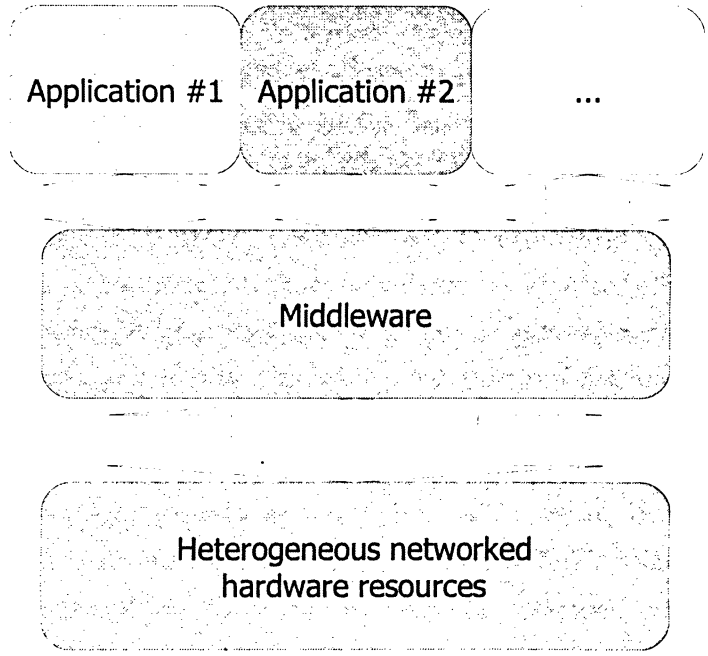


Figure 1.3: Network middleware model

Researchers have devoted much effort to build generalized frameworks to support general distributed application development. A middleware shown in Figure 1.3 can help developers to manage the complexity and heterogeneity of underlying computing environments. Furthermore, a properly designed middleware can improve service quality and shorten the time to market. As a result, companies that use this development approach benefit from the

middleware to produce low-cost, robust, and scalable online applications.

1.5 The Need of Massively Multi-user Online Platform (MMOP)

Currently there are several Grid computing middlewares that try to provide an uniform development environment for large-scale online applications. However, there has yet to exist a middleware to support all possible obstacles faced by large-scale online application developments. Particularly, a complex set of services is usually provided for distributed computing experts, while the majority of online application developers have little or no experience in this type of development. Moreover, some middleware solutions focus on providing scalability but do not efficiently utilize available network resources. This presents a need for a lightweight middleware that solves protocol limitations, scalability, resource utilization, and reliability issues of the Grid architecture.

The proposed Massively Multi-user Online Platform (MMOP) is a lightweight middleware solution based on the P2P structured overlay network designed for large-scale online applications. The MMOP provides a set of simple to use Application Programming Interfaces (APIs) for manage and access networked resources. It inherits the flexibility and robustness of P2P network to provide a scalable Grid computing architecture. Moreover, the MMOP focuses on providing application-level QoS through the means of active monitoring and QoS provisioning in a P2P environment. The design and prototype implementation of MMOP in this thesis clearly indicate that MMOP is a right platform for developing large-scale online applications.

1.6 My Contributions

1.6.1 Design of MMOP Architecture

A new framework for providing large-scale distributed application known as Massively Multi-user Online Platform (MMOP) is presented in this thesis. It provides an extensible, easily

configurable, manageable, highly flexible and scalable middleware for distributed application development. It is designed to alleviate some of the problems imposed by the current distributed systems. Novel mechanisms and features are proposed to support the MMOP, for example, the hierarchical Virtual Organization (VO) resource organization, QoS monitoring, atomic data access, and time synchronization mechanisms.

1.6.2 Protocol and Operation Designs of MMOP

In order to implement a working prototype of MMOP, a multitude of components have been generated to support the proposed functionalities in MMOP. They are:

Virtual Organization (VO) Construction Service supports modularized P2P overlay algorithms, e.g., One Hop Lookups and Chord, for hierarchical organizing and utilizing heterogeneous computing resources.

Distributed Semaphore (DISEM) Service is a high-performance, scalable and configurable atomic data access design for distributed applications.

Application Deployment Service provides on-the-fly module loading and offers application-based QoS policy designs.

QoS Management Services are network monitoring modules which provide simple application-level QoS provisioning.

Distributed Timing Service supplies an accurate and synchronized distributed clock for distributed applications.

Each service is modularized and includes a simple-to-use Application Programming Interface (API) that can be extended easily. Some other minor components are also included to support simulation of the prototype implementation.

1.6.3 Design of Distributed 2D Shooting Game on MMOP

To verify the functionality promised by the MMOP design, a distributed 2D shooting game has been developed. The name of the game is coined SPACE SHOOTER. The game includes a Graphical User Interface (GUI) client, Artificial Intelligence (AI) client and distributed server design utilizing the MMOP services modules. In order to develop a fully functional game, special considerations are put into GUI, AI, client-side latency compensation and server bandwidth conservation design.

Simulations are conducted with the implementation of the 2D game to demonstrate the functionality and performance of the proposed MMOP design.

1.7 Thesis Organization

Chapter 1 is an introduction to current architecture for large-scale online services and the need of MMOP. Background information such as QoS management and functionality of Grid computing has been briefly discussed. This chapter concludes with a summary of motivations and contributions of my work.

Chapter 2 contains background material on QoS, Grid computing, P2P structured overlay, and Massively Multiplayer Online Game (MMOG), and provides a motivation behind the proposed research. Some contemporary utilization of Grid computing in online gaming is briefly mentioned.

In Chapter 3, the architectural details of MMOP, i.e., its layout, organization, and main components, are presented. Various system-level features of MMOP, and functional features of different components are discussed in detail.

Chapter 4 presents the implemented prototype and outlines the work undertaken. Implementation details of the various components and features are discussed. In particular, details of communication protocol and related implementation issues are provided. The limitations of the implementation are also examined.

In Chapter 5, the distributed 2D shooting game, SPACE SHOOTER, has been developed using the prototype implemented in Chapter 4. Special consideration of latency compensation and bandwidth conservation are designed and the design choices are discussed.

Chapter 6 evaluates the novelty proposed Distributed Semaphore (DISEM) protocol and verifies the overall system performance and functionality through simulations with the distributed 2D shooting game.

Chapter 7 summarizes the research presented in this thesis and provides some future research directions in this area.

Chapter 2

Technology Background

In this chapter, technologies mentioned in the previous chapter are discussed in more detail to provide a clearer understanding of the motivation behind this research. In particular, the Grid computing architecture, Peer-to-Peer (P2P) structured overlay, and application-level Quality of Service (QoS) are covered in detail. We also discuss the application of the Massively Multi-user Online Platform (MMOP) for large-scale online applications such as the Massively Multiplayer Online Game (MMOG). Some current MMOG middleware are also introduced. Later in this thesis, a simple MMOG called SPACE SHOOTER is designed using the services provided by MMOP. The game will be used to demonstrate the functionality and performance of the MMOP. The preliminarily simulation results of the game can be found in Section 6.4.

2.1 Grid Computing Fundamentals

As the availability of the Internet and bandwidth outgrows server computational power and storage capacity, the next logical step is to provide a virtualization of a single powerful computer to coordinate these distributed resources. Such technology is referred to as "Grid" computing architecture, in which the computing resources are contributed by distinct individuals or organizations but now under control of a new administration called the Virtual Organization (VO). Such organization reinforces both policies defined by the original contributor as well as the virtual organization administration when utilizing the computing resources.

Grid computing paradigm differs from the traditional client-server model in that it offers efficient and inexpensive utilization of computing resources throughout the Internet. The Grid computing architecture is developed with the following considerations:

- large-scale online applications require large amounts of computing resources, which are not easily supported by current clustered server architectures.
- modern Personal Computers (PCs) offers high performance at low cost when compared to server machines.
- many of these PCs are connected to the Internet, and are underutilized most of the time.

Therefore, by developing a software that manages the interconnection of these computing resources, a virtual computing environment can be constructed. The software that unites all these resources is called the Grid computing middleware. Such middleware provides online applications with access to various resources within a Grid computing environment.

One well known project that utilizes the Grid computing architecture is the Search for Extraterrestrial Intelligence (SETI) [1] project, where narrow-bandwidth radio signals from space are gathered by radio telescopes and processed to detect intelligent lifeforms outside the Earth. Users contribute to the project by allowing the application to process a subset of the gathered data, usually when the computer is idling. By exploiting the parallel processing power of Grid computing architecture, large amounts of gathered data can be processed simultaneously.

The initial platform for SETI is now referred to as "SETI@Home Classic", which was designed specifically for the SETI project. The platform was replaced by the Berkeley Open Infrastructure for Network Computing (BOINC) middleware, and the new project is referred to as "SETI@Home". The middleware enables multiple scientific projects to share the same set of computing resources. It also provides better user control by allowing the users to adjust the time period and computing resources they want to share.

Another popular Grid computing middleware is the Globus Toolkit 4 (GT4) [2], which provides a set of service implementations for the general Grid applications. Its Grid Resource

Allocation and Management (GRAM) service provides full fledge data management and scheduling control, while the security of the middleware is ensured by incorporating high-level standard based security components with X.509 credentials [3] on the transport-level as default. Moreover, GT4 makes extensive use of document-oriented protocols such as XML for describing, discovering, and invoking network services for flexibility and extensibility. GT4 uses web services to provide loosely coupled interactions that are preferable in robust distributed systems. However, one of the main concerns that has not been addressed by GT4 is the issue of service latency. Particularly, it lacks the ability of handling real-time applications due to the large overhead introduced by using document-oriented protocol.

2.2 Peer-to-Peer (P2P) Structured Overlay

The Peer-to-Peer (P2P) network has gained lots of recognition recently, primarily because it provides a solution to replace the traditional client-server model of file sharing. The users (peers) trade files by direct connection between each other instead of downloading from a particular file server. In the traditional client-server model, the limited server bandwidth is usually the bottleneck when there are many clients requesting files simultaneously. The advantage of a P2P network type file sharing architecture is that files are shared without a dedicated file server. Generally, when more peers access the same file, the available bandwidth for that particular file increases. For all peers accessing the file, they upload the missing pieces to each other, effectively sharing their bandwidth with each other. With such setup, the problem of bandwidth bottleneck in the client-server model is mitigated. However, since the file sharing is based on the goodwill of peers, an insufficient number of peers participating in the file sharing process would result in a lower overall bandwidth and ultimately prevents the file from being downloaded if no peer is sharing it.

To form a P2P network, structured overlay algorithms such as One Hop Lookups [4] and Chord [5, 6] algorithms can be used. In both One Hop Lookups and Chord, hash keys are used heavily for overlay indexing and construction. The algorithms provide a mean of looking up any given key quickly in the structured overlay. The key is also used by the algorithms to determine a node's position in the structured overlay by assigning an identifier string for

each node. Upon carrying out consistent hashing on the identifier string, a unique hash key can be generated to identify the node in a structured overlay. A common hash function used in the structured overlay algorithm is the US Secure Hash Algorithm 1 (SHA1) [7]. A special property of hashing operation is to provide balanced load on the structured overlay, because the hashing function guarantees that each peer receives roughly the same number of keys [8].

In this thesis, hash keys are used extensively to provide high data availability. A distributed redundancy environment can be constructed by associating an identifier string to a data object. Using the hash key of the string, the data object can be virtually mapped onto a node in the overlay. Such mapping is the fundamental element of providing efficient data replications on the structured overlay [9]. The replication is achieved by adding prefixes or suffixes to the identifier string. For example, it is possible to generate multiple hash keys associated with a single data object with an identifier string of `test`. The replica strings can be created by appending suffix to the original string, resulting in `test-1`, `test-2`, and etc., The same data object is then assigned to different nodes according to the replica strings for backup and load balancing purposes. As a result, the availability of the data object is increased. The Distributed Semaphore (DISEM) protocol presented in Section 3.2.5 employs the same technique to increase data availability in a structured overlay.

In an actual P2P structured overlay, nodes are free to join or leave the overlay randomly. Therefore, it is crucial for the P2P algorithm to maintain the key mappings on each node between these changes. When a node joins the overlay, the overlay algorithm determines a set of keys which are under the responsibility of this node. When this node leaves the overlay, the algorithm redistributes its keys onto other nodes. Since the hash keys are evenly distributed on the overlay [10], only a minimum number of key movements is needed when changes occur. The hash key mappings are treated as positive integers in the structured overlay. All the hash keys are then ordered in their numeric order and wrapped around in a circle as illustrated in Figure 2.1. For example, KEY 1 is assigned to node 1 on the structured overlay since they have the same hash number. KEY 2 is assigned to its successor node 3, and KEY 6 is assigned to node 0 because the keys are wrapped around into a circle.

With this cyclic overlay structure, the structured overlay is able to accommodate a maximum

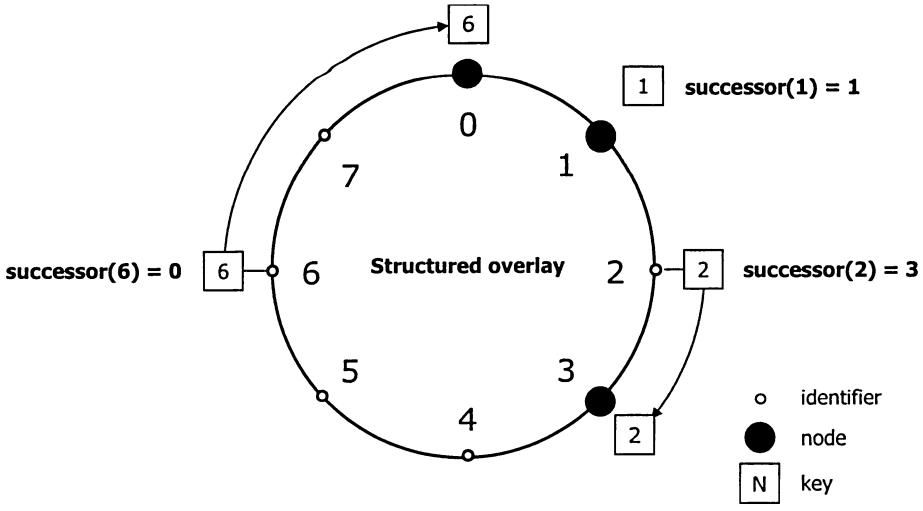


Figure 2.1: P2P key distribution in structured overlay

of N keys. This maximum possible value of N is dependent on the hashing algorithm used. For example, for the SHA1 (160 bits) hash function, the maximum number is $N = 2^{160} - 1$. Furthermore, this restriction also applies to the total number of nodes, since it cannot exceed the maximum number of keys, N.

The P2P structured overlay is the backbone of the MMOP design. In this thesis, it is used to construct Virtual Organizations (VOs), where each VO is an independent structured overlay with possibly different overlay and hashing algorithms. With different algorithms, the resulting VO exhibits different performance and size attributes. The P2P structured overlay algorithms implemented in the MMOP will be discussed and illustrated in Section 4.3.

2.3 Quality of Service (QoS) Background

The basic concept of Quality of Service (QoS) is to ensure an unfair treatment for services. For companies, service charges can be based on the level (quality) of service provided. For customers, they can choose from a pool of available service levels and pay only for the services they desired. An example of QoS being enforced is the Internet Service Providers (ISPs)

providing different levels of Internet services (different bandwidths) for customers. Even though the hardware setup for all the services are identical, the service bandwidth available to the customer is different, hence there are different values for each service level.

When the Internet was deployed many years ago, service was provided as "best effort" service due to the lack of computing power to differentiate the network traffic. Nowadays, it becomes possible to perform traffic characterization with advanced networking equipment without affecting traffic flows in networks. Traffic characterization makes sense since different kinds of traffic have different QoS sensitivities and requirements. For example, real-time video and audio traffic requires low traffic latency, while FTP file transfer does not require any real-time constraints on the network.

The need for enforcing network QoS becomes more apparent with the introduction of the P2P file sharing protocol. In particular, clients using P2P file sharing usually consume an exceptional level of bandwidth when compared to other users who do not use such service. Since the total bandwidth of an ISP is limited, the unbalanced bandwidth usage have pushed some ISPs to introduce special QoS policy rules. There rules are designed specifically to limit P2P file sharing traffic to ensure that bandwidth is shared fairly among all its customers.

In the following, a few fundamental parameters that are used for describing network QoS are presented. They are:

Packet Loss is the number of packets lost or corrupted during transmission. It generally happens when a network malfunctions or when it is overloaded.

Latency is caused by physical delay of the network link such as packet queuing and propagation.

Jitter is the variation in the latency, which can seriously affect streaming type traffic, such as real-time video or audio.

Throughput can also be identified as the available bandwidth, which is independent from other traffic on the same network link.

Traditionally, network QoS is enforced by identifying the service type and providing provisioned network resources to the service. The provisioning process can be done manually by

a network administrator or dynamically through traffic classification, where each packet is marked according to the type of service requested. Although the TOS (type of service) byte in IP header can be used for traffic differentiation in packet level, it has never really been supported by applications and network routers. The Differentiated Services (DiffServ) [11] has been recommended by IETF, and DiffServ Code Point (DSCP) is introduced to replace the TOS byte. The first six bits of the DSCP field classifies the traffic into different service types with different forwarding features and dropping probabilities.

Associating network QoS services to applications running on Grid computing framework will be an extensive research topic: As aforementioned, Grid computing units are a large set of computing resources spread all over the world. These resources include more than just networking resources. CPU cycle, hard disk space, and memory utilizations are also considered as valuable resources in the Grid computing architecture. Moreover, the QoS of such architecture can be affected by several factors such as user demands, service behaviors, and resource changes. If neglected, these factors may lead to inefficient resource allocation and ultimately unsatisfactory QoS. Unlike the networking QoS reinforcement on a per packet basis in the client-server model, Grid computing requires a more complicated application-level QoS that provides reservation of computing and networking resources at the same time.

In this thesis, an application-level QoS architecture is designed where resources in the P2P Grid architecture are actively monitored. The collected statistics are used to provision the application resources according to a set of predefined application policies. By provisioning the computing hardware and networking resources required by each application, MMOP is able to efficiently utilize the available resources within the P2P Grid architecture.

2.4 Large-Scale Online Application: MMOG

Massively Multiplayer Online Games (MMOGs) are large-scale online applications that allow hundreds, sometimes thousands of users to interact in a virtual world, as shown in Figure 2.2. Currently the most popular MMOGs according to the number of active subscriptions are

Blizzard's World of Warcraft [12], NCSoft's Lineage [13] and NCSoft's Lineage II [14]. [15] MMOGs are particularly popular in the Asia/Pacific region, which has the largest worldwide MMOG market. Furthermore, the MMOG industry generates the majority of revenue of online gaming business. According to [16], the market has grown over the past several years, and will almost triple its existing market size by 2010.

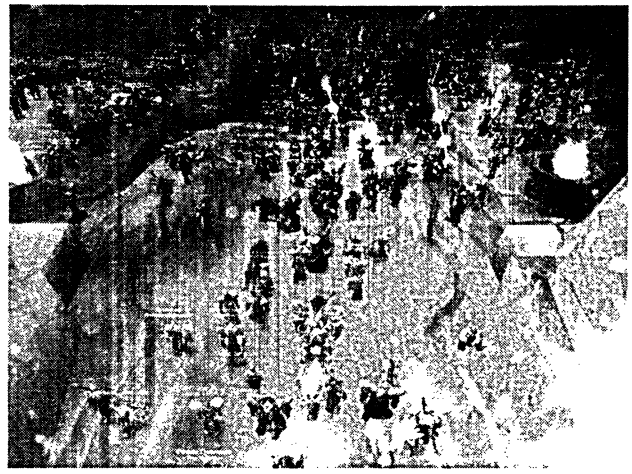


Figure 2.2: An in game screen shot of popular MMOG - World of Warcraft
[Courtesy of Blizzard Entertainment, Inc.]

The effort of creating a successful MMOG title involves several years of careful planning and beta testing prior to actual release. Even with such care, projects may be stopped or dimmed unsuccessful after release. Furthermore, continual support and update of the game is expected by the customer, unlike other online services. In [17], the author clearly outlines the need of a general middleware for MMOG. By concealing the networking details from the MMOG developers, the middleware can provide higher quality games since it allows the developers to concentrate on developing the game content. A proper middleware that provides the organization of the network infrastructure would greatly reduce the time to market and even reduce the maintenance cost of a MMOG.

Currently, many MMOGs continue to use the cluster computing paradigm as their underlying network infrastructure. That is, game companies are hosting the games on dedicated servers with high-speed network connectivity. A bottleneck can be easily identified in this setup

when higher than expected number of players are trying to access the game service. A lot of effort has been devoted towards increasing the QoS of the MMOG, and popular mechanisms implemented by many MMOG companies are:

- utilizing player login queue so the total active players in the game can be controlled
- dividing game world into separate zones that are hosted on different servers
- providing several completely independent game server sets so the users are divided among the servers

However, these methods do not solve the QoS problems completely. With login queue, the player might be spending a long time in the queue since no player is leaving the game. The other two methods introduce additional requirements on server hardware, which can incur a large amount of operation costs. We believe that with P2P Grid computing, it is possible to provide a more flexible, robust, and scalable infrastructure. The bottleneck problem described previously no longer exists in the architecture since the traffic can be spread among the networked peers. In addition, the P2P computing presents a low-cost alternative to its cluster computing counterparts by utilizing less expensive hardware.

The following MMOG Grid middleware solution demonstrates the feasibility of implementing a general, flexible Grid middleware for large-scale online applications such as MMOG.

2.4.1 The Butterfly Grid

The Butterfly Grid [18] was one of the few earliest Grid computing middleware designed specifically for MMOG. It was designed to shift processes to available resources and overcome limitation of cluster computing paradigm with its self-managed and fully meshed network. The Butterfly Grid included two clusters of roughly 50 IBM eServer xSeries servers, which were used to host database and application services. The Globus Toolkit was used to integrate these services together into a computational Grid. Unfortunately, the company that developed Butterfly Grid has moved away from providing open-standard Grid services to developing a proprietary solution called Emergent Platform.

2.4.2 BigWorld Technology

Similar to Butterfly Grid, the BigWorld Technology [19] is a middleware specifically designed for MMOG. Its founding company, Microforte spent several years and invested millions of dollars to develop a general MMOG middleware. The company claims that it is one of the most complete MMOG development packages, which includes tools from client 3D to server back-end development. Tools such as content creation, client engine, server configuration and management are included, to reduce complexity and shorten development period of MMOG design. Furthermore, the BigWorld server provides a highly available and dynamically load-balanced server infrastructure. There are several games being actively developed utilizing this platform.

2.5 Summary

In this chapter, a background of various technologies employed in MMOP are discussed. The necessity and benefits of the Grid computing architecture, P2P structured overlay, and application-level QoS provisioning are reviewed. They form the basis and provide an understanding behind the motivation of the research. Furthermore, a popular large-scale online application would be benefit greatly from the proposed Grid computing middleware.

With these technologies incorporated into the MMOP middleware, a simple MMOG called SPACE SHOOTER is designed in this thesis. The purposes of the MMOG are to (1) verify the functionality and performance of MMOP, and (2) demonstrate the ease of MMOG development with a properly designed middleware. The design and implementation of the game are covered in Chapter 5, and the evaluation of the game and the MMOP are discussed in Section 6.4.

Chapter 3

Massively Multi-user Online Platform (MMOP) Architecture

In this paper, we introduce the Massively Multi-user Online Platform (MMOP) for large-scale applications which is based on the Peer-to-Peer (P2P) structured overlay algorithms. Its goal is to provide an environment where application developers can easily develop distributed applications. Developers will be able to fully utilize the computing resources through a set of Application Programming Interfaces (APIs). The networking and servicing details are encapsulated within the MMOP, so developers can define computational requirements through means of policies. Several distributed services that are aimed for this purpose are designed in the following sections.

3.1 Virtual Organizations (VOs) Construction Service

In order to construct a Grid computing environment using the P2P overlays, nodes are grouped into Virtual Organizations (VOs). However, the definition of VO in our design is slightly different from the one used by general Grid computing. In general Grid computing setup, a VO refers to a virtual administrative domain that controls a shared resource space contributed by different administrative entities. For MMOP, we simplify the definition of VO to be nodes that are connected within a single overlay. A single node can join multiple VOs at the same time, but the VOs should not be aware of the existence of other VOs. Consequently, it is crucial to be able to distinguish different API calls from one VO to

another. All MMOP API service calls require two additional identifiers on top of regular parameters, namely the VO ID and overlay key.

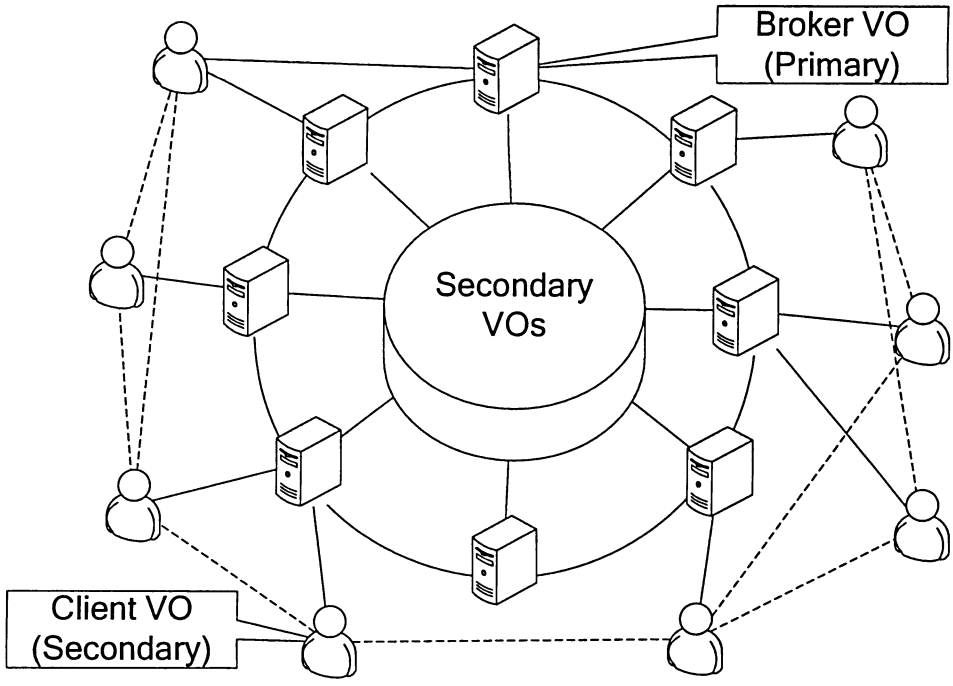


Figure 3.1: VO hierarchy and encapsulation

The VO ID is a universal ID assigned to each VO to identify which VO to relay the API call to. For scalability reasons, a hierarchical ID space is used. There is one primary VO and several secondary VOs as illustrated in Figure 3.1. The primary VO is the *BROKER* VO, which stores all the secondary VOs' contact information using the Distributed Semaphore (DISEM) protocol described in Section 3.2.5. Each VO also stores its first degree subsidiary VOs' (SubVOs) information using DISEM. The VO ID is a multi-level string that has the following format:

$$Primary/Secondary : 1^{st} : 2^{nd} : \ldots : N$$

. For example, a Broker machine which resides in the subsidiary VO of the name *FirstLevel* will have the VO ID of

$$BROKER : FirstLevel$$

and so on.

Since there is no restriction on how many levels of SubVOs are allowed, a VO search might require several queries between different VOs. Therefore, each search request should be cached locally on the requester for future requests to the same VO. Aside from the first degree subsidiary VOs, each VO also stores the information of the primary *BROKER* VO. These records are used to provide a starting ground for all the look up services. Therefore, any given number of VOs can be employed using this hierarchical VO structure. The following are functional descriptions of each VO:

Broker VO (Primary): It is the backbone of the system, relays requests between clients and servers so that the service execution can be masked from the clients. All communication is relayed by the *BROKER* VO, this provides protection as well as scalability to the MMOP platform. The clients are not aware of how their service is provided and the servers do not communicate with the clients directly. The *BROKER* VO tries to cache the VO lookup results with DISEM so that future requests can be served in a timely manner.

Client VO (Secondary): Includes all the clients. The main purpose of this VO is to share non-critical data between clients. Some client to client communication, for example direct chat between clients, file sharing between clients etc., can also be provided through this VO when no specific service within the MMOP need to be involved. Furthermore, the Bit Torrent (BT) concept described in Section 3.1.1 can be incorporated to efficiently utilize the available bandwidth.

Application VO (Secondary): This VO is the execution environment of the MMOP, in which, the application will be deployed and executed. It provides computational services and reserves necessary resources as defined by the applications QoS policy. The developer can further define specific application VOs for specific tasks. For example, a VO can be defined as a database VO by creating a specific VO ID as:

$$APPLICATION : ApplicationName : Database$$

and redirecting the data access to that VO instead of using the current VO.

Free Server VO (Secondary): A free Server VO manages servers that have free computational resources remaining. These servers are ready to be moved into one of the application VOs. Each assignment would assign computing resources according to the QoS policy assigned by the application developer. The machine should remain in this VO while it has extra resources available.

Module VO (Secondary): This particular VO provides storage service for application executables. The storage mechanism is inspired by the popular BT concept discussed in Section 3.1.1, where all the data stored are broken into smaller blocks so that the loads are distributed between the data storage servers. The effect of larger block sizes would have higher bandwidth requirements on the server, while smaller block size, on the other hand, would require more coordination overhead in storing and retrieving the data. Further optimization can be calculated based on the available bandwidth of overall storage servers.

While forming MMOP, a *BROKER* VO is first constructed with bootstrapping service. The function of a bootstrapping service is to provide newly arrived peers with ways of joining or creating a VO. At the beginning of *BROKER* VO formation, a single machine is set as the bootstrap master. This machine provides bootstrap service for the first few *BROKER* machines. When there are more than one node within the *BROKER* VO, the newly arriving nodes are not limited to use only the bootstrap master. Any other machine within the *BROKER* VO can be used to start the bootstrapping service.

The bootstrapping service ensures that when a new VO is being created, a corresponding reference is created for future VO queries. When a new VO is created, the virtual organization construction service creates a total of k references to the new VO and stores the references using DISEM on the parent VO. The parent VO is defined as one level above the current VO, for example: $APPLICATION : Level1 : Level2$ has a parent VO of $APPLICATION : Level1$. As mentioned before, the *BROKER* VO is the parent VO of $BROKER : 1^{st}$ as well as parent VO for all the secondary VOs.

Furthermore, the number of k can be adjusted as the size of referencing VO increases. Each reference is a mapping of the overlay key onto the referencing VO, hence it must be

maintained regularly by the parent VO.

3.1.1 Bit Torrent (BT) Protocol

In the design of Virtual Organizations (VOs) construction services, the Bit Torrent (BT) concept is mentioned to provide data storage on the P2P structured overlay. In BT [20, 21] architecture, when a user wants to publish a file, an associated `.torrent` file needs to be constructed. Such file contains the information of the file to be published, which includes file length, file name, file checksum and location of the trackers. The trackers are peer servers that refer users to one another. It is the heart of the BT protocol, without it, peers will not be able to lookup each other nor share files.

The `.torrent` is usually obtained from the Internet by traditional means such as website or File Transfer Protocol (FTP). Users interested in obtaining the published file need to obtain the `.torrent` file first and connects to the trackers. When a user contacts the tracker, information about the requested file and connection information are sent to the tracker. The tracker then responds with connection information of currently connected users that are downloading the same file. The default number of users included in the returned list is limited to fifty. Therefore, a returned list does not contain all the users that are currently downloading the file. If more than fifty users are connected, random graph algorithm is used to return only the partial list. The algorithm ensures that if there is a sufficient number of nodes in the list that is passed to the user, then there will be a high probability that all users can form a complete network [22]. This partial list of users is then used to establish connections between users. A basic BT network setup is illustrated in Figure 3.2.

To make the file available, the BT protocol requires a user with the complete file to be connected to the tracker, this user is known as a seeder of that particular file. In order to redistribute the bandwidth cost among all downloading peers, the BT protocol breaks the file into smaller file blocks. Each block of files is then associated with a popularity tag which indicates how often the block has been accessed. When a peer is connected to a seeder, it attempts to request the least popular pieces first. It then exchanges the retrieved blocks with other peers who are downloading the same file. By doing this, all peers downloading the file

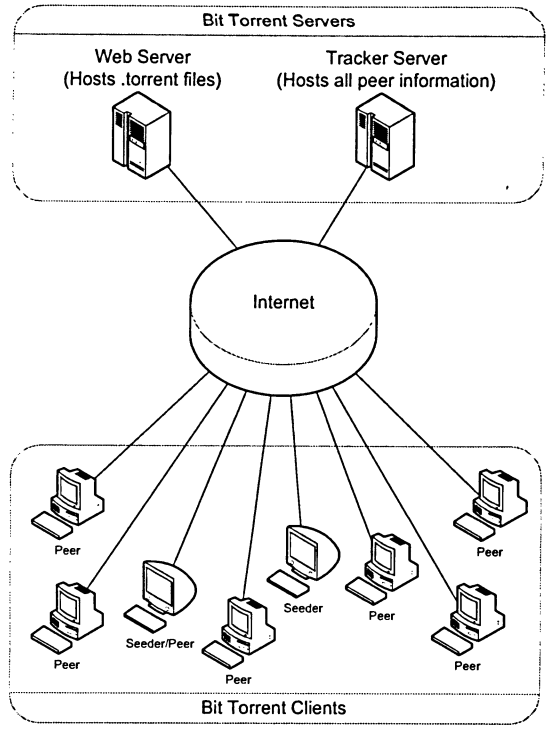


Figure 3.2: Basic Bit Torrent network structure

share the load of a single file server in the traditional client-server model. This approach results in the most efficient bandwidth utilization and allows multiple peers to download the same file at the same time [20]. Furthermore, this redistribution of bandwidth costs makes downloading from a BT network to have the potential of serving unlimited number of clients.

Special caution should be taken into account for the original seeders. The protocol requires that the original seeder of the file sends out at least one complete copy of the file. If the seeder leaves the network without seeding a complete file then any peer downloading the file will not be able to obtain a complete file. This is different from a client-server model where the file received by each client is independent. However, the BT network has the advantage over the traditional client-server model of allowing a downloading peer to become a seeder.

This occurs when a peer has completely reconstructed all the received file blocks for that file. Once the peer has a complete file, it can start the seeding procedure as a seeder. Therefore, as long as there is a seeder in the network then the file can be downloaded correctly.

Due to the need of a file server for storing the `.torrent` file and a tracker to refer clients to one another, BT is not suitable to be directly implemented in the MMOP. Furthermore, since the data used in the MMOP simulation test cases are relatively small, DISEM is used to provide general data storage in place of BT protocol. However, since DISEM does not perform well under large data size, a modified version of BT without file server and tracker should be implemented in the future to provide large file sharing in MMOP.

3.2 Distributed Data Access Service

Redundancy is a simple way of providing high data availability on a structured overlay. However, due to the random nature of Peer-to-Peer (P2P) network, node arrival, departure and failures need to be compensated by the replication algorithm. Moreover, replication introduces the challenges of maintaining consistency among all replicas. For providing consistent distributed data, an algorithm should manage dynamic participation as the collection of network locations storing replicas change over time. There are two basic approaches to handle these changes within the structured overlay network [23]:

Optimistic In this approach, the primary objective is to optimize the data access time. The data are not protected when new data are being written. It is usually sufficient when the probability for updating the same variable with different data at the same time is low. Since no synchronization is required, this type of protocol is usually simpler to implement. The read-one/write-all-available (ROWAA) protocol is one of the simplest optimistic protocols.

Pessimistic The main goal of the pessimistic approach is to provide data consistency. Pessimistic approaches are based on locks and lock management. As aforementioned,

network and node failures are unavoidable events on a distributed network. To ensure data consistency and availability, pessimistic protocols provide distributed lock management and data recovery.

Quorum-based replica protocols are examples of pessimistic protocols that have been developed to improve distributed data accessing performance. Some quorum-based replica protocols, such as Etna [24] and Reconfigurable Atomic Memory for Basic Objects (RAMBO and RAMBO II) [25, 26] achieve data consistency through forming read/write quorums. The quorum formation is a result of requesting configuration from the total-ordering algorithm, Paxos [27]. Sigma protocol [28], on the other hand, uses logical clock and ROWAA approach to achieve data consistency. In the next few sections, the Paxos algorithm, the two quorum-based replica algorithms, the Sigma protocol and lastly, design of the Distributed Semaphore (DISEM) service are discussed.

3.2.1 Paxos

Paxos [27] is a distributed consensus algorithm that has three types of agents: proposers, acceptors, and learners. A single process may act as more than one type of agent. The consensus algorithm proposed by Paxos has two parts:

1) choosing a value, and

2) learning the chosen value.

The first part of the algorithm also consists of two distinct phases. The first phase requires a proposer to selects a proposal number, p, and sends prepare request messages to a majority of acceptors. If the message with a number p received by an acceptor is larger than any of the previously responded prepare requests, the acceptor then replies to this request with a promise not to accept other proposals with numbers smaller than p. On the other hand, if the proposed number p is smaller, the acceptor returns the highest-numbered proposal that it has accepted.

The second phase starts when the proposer receives responses to its prepare messages, from a majority of acceptors. The proposer then sends an $accept(p, v)$ request message to each of

those acceptors, where v is the value of the highest-numbered proposal among the responses, or any value if the responses reported no proposals. Upon receiving the $accept(p, v)$, the acceptor accepts the proposal unless it has already responded to another prepare request with a proposal number larger than p.

The second part of the algorithm is to distribute the accepted value to the learner agents. In general, the acceptors can respond with their acceptances to some set of distinguished learners, each of which can then inform all other learners when a value has been chosen. With a larger set of distinguished learners, greater reliability at the cost of greater communication complexity can be provided. The Paxos algorithm guarantees the following:

- Only a value that has been proposed may be chosen.
- Only a single value is chosen.
- A process never learns that a value has been chosen unless it actually has been.

Given that the underlying network should eventually be stabilized, Paxos achieves atomic data access by allowing only one value being proposed at any given time. However, as an extra requirement for the algorithm, an acceptor must remember two numbers: the highest-numbered proposal that it has ever accepted and the number of the highest-numbered prepare request to which it has responded. This criteria, if not met in case of node failure, would lead to network partitioning and result in more than one value being accepted.

3.2.2 Etna

In Etna [24], the Paxos algorithm is used to produce quorum configurations. Each configuration is produced by a single instance of the Paxos protocol. Etna maintains one consistent configuration per variable. Hence, different variables are replicated on different sets of nodes. With each configuration, Etna makes use of the agreement protocol to verify the consistency of its proposed variable and the variable stored on the primary (leader) location. If discrepancy is discovered, Etna uses Paxos to get a consensus on a new configuration in which they are the same.

Each Etna node can create proposed configurations and pass them to Paxos to be accepted or rejected. When consensus is reached, Paxos calls the Etna to see if it will accept the proposed value. At this point, Etna notifies the new configuration of its decision, and the configuration's primary location proceeds to locate the latest version of the variable and serve client requests.

3.2.3 RAMBO and RAMBOII

RAMBO [25] replicates variables at several network locations to achieve fault tolerance and provide availability. To maintain consistency in the presence of small and transient changes, the RAMBO algorithm uses configurations to determine the scope of the update. Such configuration contains a set of members plus sets of read/write quorums. RAMBO's quorum intersection property requires that every read-quorum intersects with every write-quorum for the same variable so that the result of a write can be observed by the readers.

To accommodate larger and more permanent changes, the algorithm reconstructs the existing set of members and the sets of quorums using the Paxos algorithm as reconfiguration service. The reconfiguration service produces a sequence of configurations based on the reconfiguration requests from the environment. This service also informs read/write service about newly-determined configurations, and disseminates information about newly-determined configurations to the members of the configurations. Any configuration may be installed at any time, while obsolete configurations can be removed from the system without interfering with the active configurations.

In RAMBO II [26], it takes a step forward and allows the reconfiguration protocol to upgrade any configuration, even when the configurations with smaller indices are out-of-date. The algorithm allows all configurations with smaller indices to be removed when a configuration is selected. This allows a single configuration upgrade operation in RAMBO II to have the effect of many garbage collection operations in RAMBO.

However, since RAMBO algorithms allows multiple active configurations of replicas at any time, reads and writes can be costly as a result of frequent quorum assembly with every active configuration. According to [29], although quorum algorithms provide performance

improvements in some extreme cases of distributed computing (e.g., write intensive environments), this type of environment does not appear in our everyday application. It is more probable for an application to have workloads such as 70% read and 30% write, or even 80% read and 20% write operations. Under such circumstances, faster read response time would greatly improve the overall performance. The paper [29] suggested that the optimistic ROWAA approach is the best choice for a large range of applications requiring data replication. The Sigma protocol is one of the protocols that was designed with this approach in mind and will be discussed next.

3.2.4 Sigma

The Sigma protocol [28], uses different approaches from the previous two algorithms. As opposed to trying to achieve general consensus with Paxos, it uses a combination of replica queues and Lamport's logical clock [30] to provide a first-come-first-serve service. The protocol treats all failures in a uniform manner and provides fault tolerance by lease and informed back off.

A queue is installed at each replica to identify the order of client requests into a consistent view among all replicas. Replica grants permission to first client in the queue with renewable lease. When the lease expires, replica will grant permission to the next client, if any. If general consensus cannot be achieved during the time of a lease grant, client sends out a YIELD message to each of the acquired replicas. This message has the same effect as releasing the replica and requesting it again. Therefore, the order of the queue is reshuffled and progress is ensured. Typically, this stabilization process can quickly settle.

In the case of replica failures, Sigma requires a replica queue to be rebuilt. Informed back off is a strategy where each replica predicts the expected waiting time, T_w, and advises its requesters to wait before retry. The T_w can be calculated by $T_w = TCS \times (P + 1/2)$, where P is the client's position in the queue and TCS is the average interval between any two consecutive release operations. The $1/2$ in the formula is to take current owner of the replica into consideration. Upon replica failure, the queue may start empty. The T_w is small so the

failed replica can rebuild its own queue relatively fast while the healthy nodes maintain its stabilized queue.

3.2.5 Distributed Semaphore (DISEM) Service

Similar to the Sigma protocol, the DISEM protocol uses the ROWAA approach. This allows Distributed Semaphore (DISEM) to provide MMOP with real-time data access as well as data consistency when required. Unlike the Sigma protocol, the use of logical clock is not required in the DISEM. The replicas collectively maintain progress and consistency of the protocol by notifying the replica status to the requesters.

In DISEM, the variable is replicated n times on the VO. The number n is a constant that can be tuned according to the availability requirement of each particular VO. If there are multiple copies of a variable on the VO, it is crucial to maintain the consistency among all copies. Defective, outdated, or missing replicas are re-created within refreshment rounds. As recommended in [31], a common data access API should contain at least the first three basic methods shown in Table 3.1. In our proposed design, the write operation is further partitioned into two separate stages of operations: *lock* and *unlock*.

Table 3.1: Simplified DISEM API for MMOP

API Interface	Interface description
publish	publishes a variable.
read	retrieves the variable.
write	writes a new value into the variable.
lock	locks the given variable.
unlock	unlocks the variable and update its value.

To access a created variable on DISEM, the API, *read* is used. By exploiting the properties of structured overlay and variable replication, several read operations can be defined. Each operation satisfies certain specific real-time policy requirements of the application policy. Choosing which *read* to use depends on the operating nature of the target application. In the following, the *read* operations in increasing order of data accuracy are defined.

Real-time read: the *read* operation returns immediately if any error is encountered during the data retrieval.

Normal read: retrieves any existing replicas.

Consensus read: this *read* method compares the version tag of all replicas to achieve a general consensus.

Most updated read: variable replica with latest version tag is selected.

Special care should be taken with the DISEM *write* operations. In order to provide distributed lock management, the *write* is constituted of *lock* and *unlock* phases. In the *lock* phase, the Write Requester (WR) first randomly generates a key and finds the corresponding Synchronization Agent (SA) on the structured overlay. The SA then provides lock management for this specific request. The SA then gathers replicas information and notifies the WR of the acceptance or rejection of its request. Upon receiving acceptance, the WR then proceeds with its update and releases the variable with *unlock*. Through the use of the *lock*, *unlock* phases, atomic data access and critical section protection can be achieved. The detailed implementation of DISEM is described in Section 4.4.

3.3 Application Deployment Service

This service provides a simple way of deploying application on to the specific VO for the application developer. The application is identified by the application name and the version number. The name of application should be unique within each VO while the version number is used to identify the version of the application. Combining the version and service name, the application deployment service then generates a unique key for this specific application.

Along with each application, the application developer is required to specify the policy requirement of the application, which will be used when deploying the service and trying to find the most suitable server for such policy. The available parameters are listed in Table 3.2.

Table 3.2: Deployment policy parameters

Policy parameter	**Description of the parameter**
Real time	indicates the real time or non-real time nature of the process.
Priority level	the priority of this process should be executed, where 1 being highest priority and 10 being the lowest.
Execution time	indicates the expected execution time of this process, a value of -1 indicates immediate execution.
Feedback	indicates if feedback is expected when the service finishes executing.
Resources	specifies the resources required when this application is deployed.

In Table 3.2, the resource field can be further broken down into several resource parameters as in Table 3.3. These parameters are monitored by the QoS monitoring service described in Section 3.4.1.

Table 3.3: Application resource parameters

Resource parameter	**Description of the parameter**
Processor utilization	expected processor utilization.
Memory utilization	memory required for executing the process.
Spare storage	free disk space requirement.
QoS of network	bandwidth, latency, packet error rate etc.,
Spawn threshold	it can be any combination of above parameters (CPU, memory utilization, storage and QoS network etc).

With these parameters defined, the application deployment service then queries the QoS provisioning service to determine which free server to deploy the service. The server will then join the specified VO in the *APPLICATION* VO with the new available resources. If there are no more resources available on the application deployed server then it will be

removed from the *FREE_SERVER* VO. Furthermore, if an application sets the spawn threshold parameter, the QoS provisioning service would monitor the executing environment and try to spawn a new process when the threshold is reached. This enables load balancing at runtime, which increases the scalability of the MMOP.

3.4 QoS Management Service

To ensure the quality of the application execution, all the services monitoring and management are done at the application level. The networking QoS is also monitored but it should be reinforced by the network layer instead. This service can be broken down into two specific sub services, namely the QoS monitoring service and the QoS provisioning services. As stated in Section 3.3, each application deployment is allowed to have different service policy requirements. Therefore, the QoS monitoring service is required to gather statistics on the application execution environment on the VOs. With these statistics, the QoS provisioning services can be used to ensure that the application's execution requirements are met. In the next sections we described the design criteria of both services.

3.4.1 QoS Monitoring Service

Network and resource monitoring is an important function required by the Grid system. In the context of load balancing, it is crucial to monitor the performance of the service providing nodes in order to achieve optimal performance. The monitoring utilities provided by the MMOP consist of generic components and supports periodic and event driven updates. Such updates are collected and aggregated by the monitoring services and relayed to the requester. These collected data are processed by the application deployment service and QoS provision service in order to meet the service allocation requirements specified by the application policies.

To gain an accurate picture of the servicing machine's status, the following metrics are monitored in an interval of 300 ms:

Average CPU Load: the CPU Monitoring utility takes the 300 ms average of the user usage over the total system load. It is a fair indicator of the current CPU utilization of the machine.

Average Memory Load: is the ratio of available physical memory versus swap file in use. Memory is an important system metric. A system with more available memory will, in general perform better than one with less. The swap file in use is a fairly accurate measure of memory loading. Specifically in the testbed machines running Linux, the swap file does not come into play unless the free physical memory is exhausted. As a result, an increase in swap file size has a negative impact on system performance. Similar to the average CPU loading, 300 ms average is taken to smooth out the memory utilization spikes.

Network Statistics: can be further broken down into several sub-metrics. The data stored is processed in the same interval (300 ms) as other monitors. All statistics gathered are per network interface controller (NIC) and have two components each, received and sent:

Throughput: the received throughput in conjunction with sent throughput metrics are a good measure of the available bandwidth utilization.

Packet Error Rate: is an indicator of the integrity of the packets.

Packet Drop Rate: is an indicator of missing packets. A high packet drop rate usually indicates that the Network Interface Card (NIC)'s buffer is full and cannot process any more data.

Number of Connections: is a good measure of the server loadings. As the number of connections increases, the system loads increases and overall performance decreases. Furthermore, some Linux system would only allow 1024 simultaneous connections to be opened per process before exceptions will be thrown. Therefore special precaution should be taken for this metric.

Available Storage Size: indicates the total available physical storage space (primarily hard disk space) to MMOP.

Latency: this monitoring utility is the only one that requires special parameters, a target client needs to be specified upon request. The calculated result will be stored locally for future reference.

The above monitoring services should be modularized so that the developer can select their own monitoring services. Furthermore, these basic monitoring services are enabled by default on all machines that are managed by the MMOP. This enables the QoS provisioning service to actively monitor the changes and manage the QoS requirements. The design of QoS provisioning service in the next section will try to take full advantage of these active monitoring services.

3.4.2 QoS Provisioning Service

The basis of QoS provisioning service is to ensure that the target server has sufficient resources according to the application QoS policy defined in Table 3.3. One of the VOs that QoS provisioning service actively monitors is the $FREE\_SERVER$ VO. The service gathers and aggregates the monitored statistics periodically in the VO. When a new application deployment is requested, the gathered data can be checked against the application defined policy to find the most suitable candidate. A server satisfying the requirement would be returned to the application deployment service where deployment can be made. If no such server can be found, the QoS provisioning service will notify the deployment service of such issue. It is then up to the deployment service to retry at a later time if the specified application should be deployed.

An extra step has to be taken by the QoS provisioning service to ensure the QoS requirements are met. The service continuously monitors all active applications deployed on the machine if a particular application has its spawn threshold set in the policy requirement. This feature allows the QoS provisioning service to notify the application deployment service about machine overload and schedule a new application instance deployment. If the predefined threshold is not met at deployment, this monitoring feature will be disabled but otherwise enabled by default. Upon receiving application threshold overload notice, the application

deployment service would then query the QoS provisioning service for another suitable server to deploy the application.

3.5 Distributed Timing Service

To provide a distributed game development environment as well as regular distributed applications, MMOP provides a distributed timing service to fulfill the time synchronization needs for applications. This service is particularly useful in the distributed 2D shooting game design described in Chapter 5. Next we will describe the distributed timing service provided in MMOP.

For gaming purposes, traditional distributed multi-player games use dead reckoning technique to estimate an entity's movement in order to avoid synchronization. The data transmitted from the game server contains the current position and the velocity of each entity. When a participating client receives such data, it puts each entity at the given position specified by the positioning data. It then estimates a path for each entity from their current position while considering the local clock. However, this projection is inaccurate with the network latency between the server and client. As the network latency increases, the inaccuracy increases substantially and renders the game unplayable in the worst case.

In order to minimize the effect of network delays on the player, a distributed synchronized timing service is required to enable the client to render the entities accurately. Experiments conducted in [32] show significant quantitative improvement in accuracy even for minimum delay between the sender-receiver pairs and appreciable qualitative improvement in game playing experience. Currently there are a few timing synchronization services available but they are either too complex (NTP) or too-inaccurate (SNTP) for our purpose. We would like to have a fairly accurate distributed timing service that have relative small clock difference between applications running the service.

The algorithm which is described in [33] is a simple clock approximation of sending packets back and forth between client and server to measure the round trip time (RTT). In order to rule out the effect of bursty network delays, if the measured RTT is one standard deviation

above the median, then it is discarded. By removing these samples we can provide a simple distributed time synchronization service that is accurate in the order of 150 ms or better. A client of the service can easily turn into a server by providing its client with a synchronized clock instead of its own local clock. In this way, a distributed timing service can be achieved where all sub-servers/clients are synchronized to the main server's clock. More than one server can be established to provide a separate time synchronization for a different set of applications.

3.6 Summary

In this chapter, we have introduced the design of Massively Multi-user Online Platform (MMOP) for large-scale applications. The design is based on the Peer-to-Peer (P2P) structured overlay algorithms for its scalability and reliability. The MMOP provides an environment in which distributed application can be developed with ease. Computing resources are exposed to the developers through a set of Application Programming Interfaces (APIs). With all the networking and servicing details encapsulated within the MMOP, the developers simply define a set of application policies to control the behavior of the application. Several distributed services that are aimed for this purpose are designed and described in this chapter. In the next chapter, the actual implementation, issues encountered, workaround strategy of MMOP are discussed.

Chapter 4

Massively Multi-user Online Platform (MMOP) prototype implementation

A prototype Massively Multi-user Online Platform (MMOP) based on the middleware design discussed in the previous chapter has been developed. The aim is to develop a functional middleware which provides simple Application Programming Interfaces (APIs) for accessing its services. In this chapter, the development environment of the MMOP is described and some hardware constraints for the overall MMOP design is discussed. Figure 4.1 shows the structure of implemented MMOP middleware, which is composed of the network layer, P2P structured overlay, virtual organization construction service, Distributed Semaphore (DISEM) service, application deployment service, QoS management service, and distributed timing service.

For the benefit of future developers, some of the problems and issues faced during prototype development have also been mentioned.

4.1 Development Testbed

The MMOP implementation effort was supported by Professor Eddie Law. The prototype was developed at the Ubiquitous Communications and Security (UCS) Laboratory at Ryerson University. The UCS testbed consists of a total of ten machines that act as both routers and end hosts.

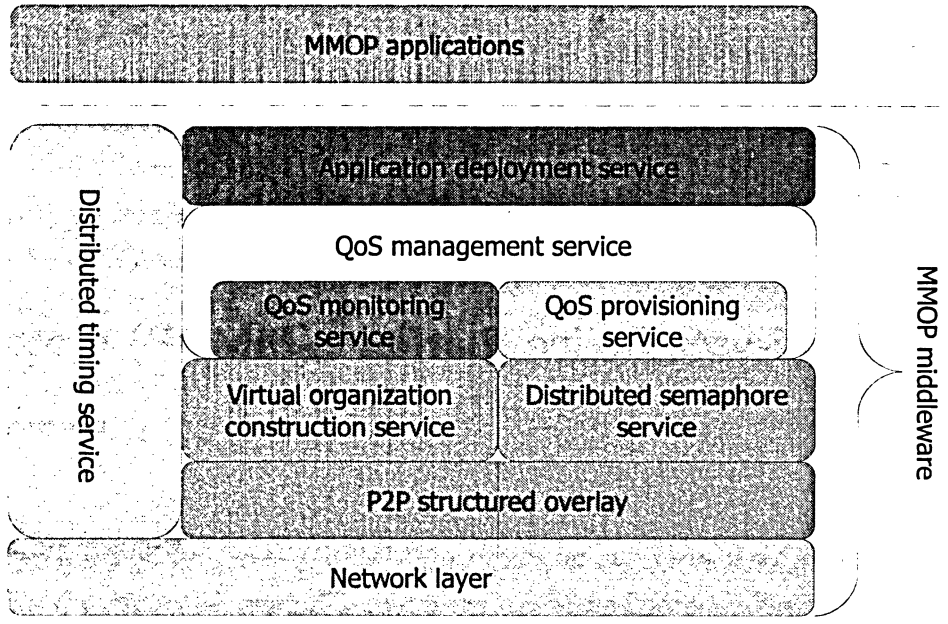


Figure 4.1: MMOP middleware structure

4.1.1 Development Testbed Hardware

The testbed machines consist of ten AMD Sempron 2600+ Personal Computer (PC). These machines each have 1 GB shared Random Access Memory (RAM) with onboard video, and a four port PCI NIC manufactured by Soekris Engineering [34]. Therefore, each machine has total of five independent network interfaces with one onboard network interface. With these components, each machine in the testbed acts as routers and hosts for the MMOP middleware and applications. For consistency purposes, all machines will have a uniform software configuration as illustrated in next section.

4.1.2 Development Testbed - Software

The prototype for MMOP is a middleware based implementation. Several software components used to develop the middleware are outlined in this section. The deployment script for the MMOP prototype is included in Appendix A.

Operation System

All ten testbed machines have Ubuntu 6.0.6 [35] kernel version 2.6.15.6 as their primary operating system. The earlier development and testing of the components were done on Windows XP SP2 [36]. Later in the development cycle, it was moved onto a Ubuntu 6.1.0 machine. The Ubuntu machine has the Linux kernel version $2.6.17 - 11$ and is chosen for developing the QoS related services.

Coding

The Java language from Sun Microsystems, Inc. is selected for developing the MMOP middleware since Java provides great portability between the operating systems used in the testbed. During the development machine transition, no changes were required for porting the code developed in Windows to Linux. The Java version used for code development of MMOP is the Java 2 Platform Standard Edition 5.0 update 10 [37]. The codes are developed and managed in an Integrated Development Environment (IDE) called Eclipse [38]. Furthermore, Eclipse supports Concurrent Versions System (CVS) for backup, modification tracking and bug fixes throughout the development. The CVS also provides the file format conversion required between different operating systems.

4.2 Network Layer Implementation

For the implementation of MMOP, a simple to use network API will simplify the development time and also reduce the debug time of service using such API. There are several ways to implement the network layer in Java and each has its pros and cons. In this thesis, the network layer is first implemented with the Remote Method Invocation (RMI), which is migrated to TCP sockets for performance reasons and finally implemented in the new I/O (NIO) APIs for manageability. In the following sections, the advantages and disadvantages of each implementation are discussed.

4.2.1 Java Remote Method Invocation (RMI)

Java Remote Method Invocation (Java RMI) network layer was the easiest one to develop among all three network layer implementations. It has a well defined structure which can remotely invoke methods from other Java objects that are located on different machines. The parameters for the remote methods are passed through automatic object serialization, which simplifies the service design since Java objects can be used directly.

To use the RMI service, a rmiregistry service must be running and accessible. The server connects to the rmiregistry service, registers the implemented service (method) with it, and waits for clients to request its service. When a client wants to access the service, it first connects to the same rmiregistry and locates the service and server information. Upon obtaining the information, the client can access the service by passing objects as method parameters. After the method completes its executing, the result is returned to the client.

An advantage of the RMI is that it is easy to program, the syntax is simple and Java JDK provides all the necessary tools to construct the RMI service. However, there is a severe problem with this setup; the rmiregistry service is a single point of failure in the system. When the rmiregistry fails, the entire distributed system fails. Furthermore, the performance is greatly limited by the excessive object serialization overhead. Due to these two constraints, we have migrated the network layer to use TCP sockets.

4.2.2 TCP Socket

The TCP sockets network layer implementation resolves the issues with the RMI implementation. Firstly, since this implementation does not require a centralized server, there is no single point of failure. Secondly, there is no restriction on what is transmitted through the TCP socket, hence overhead can be minimized when object serialization is not required.

In order to communicate between two machines using the TCP sockets, a connection must first be established between a pair of sockets. This pair of sockets consists of one listening socket on the server and a connection requesting socket on the client. The connection

is established by connecting these two sockets and data can then be transmitted in both directions once the connection has been established.

The disadvantage to this approach is that as total number of connections increases, it becomes harder to manage the connections since each connection may serve independent services in an application. Furthermore, the Java Socket API prevents the socket channel from being used in a non-blocking mode, which means that multiple services on a server are required to use different connections to communicate with a single client. With these restrictions, a new network layer called Java New IO (NIO) is investigated.

4.2.3 Java New IO (NIO)

Java New IO (NIO) is a multiplexing environment where a selector is used to manage a large number of NIO sockets. Each NIO socket is configured in non-blocking mode and serviced as the data becomes available on that socket. The non-blocking mode enables connection reuse where existing client connections can be shared by multiple services on the same server. Furthermore, this setup increases the manageability of the network layer since connections are controlled by a centralized selector thread.

While migrating the network layer to use NIO, a serious problem was encountered when the selector thread takes 100% CPU usage. After some investigation, profiling and research, the problem was identified as incorrectly enabling write ready flag on sockets. The high CPU usage is caused by the selector going through sockets that are ready for data writing, even though there are no data to be written. To correct this problem, a write ready on socket is only enabled when there is data ready to be written.

With the NIO selector structure, the network layer is redesigned to provide several features that simplifies the design of other MMOP services. First, an abstract protocol interface is provided for the service that wants to send and receive data through the NIO network layer. A unique hash is computed for each abstract protocol implementation for multiplexing data between multiple protocols. Multiple protocols can be added to one communication channel by using the *attachAbstractProtocol* method, which provides multiple services on one single channel.

The abstract protocol interface also provides high performance communication by separating raw data from the object automatically. For applications requiring high performance, raw data can be sent and received directly to eliminate unnecessary serialization overhead. Furthermore, similar to the RMI, the object serialization is automatically done by the network layer so that objects can be sent directly through the *SendMessage* method of the interface.

A distinct feature of the NIO network layer is its ability to provide thread independent blocking communication on the non-blocking channels. That is, a single communication channel can be used by multiple threads that are required to send and receive data synchronously. This feature greatly simplifies the design of many MMOP services since synchronization communication is taken care of by the network layer. Furthermore, such service allows the TCP channels to be reused efficiently by sharing a single channel with several services. This effectively reduces the total number of active connections on each server from one per service to one per client (multiple services).

4.3 Virtual Organization Construction Service

The Virtual Organizations (VOs) in MMOP are constructed by P2P structured overlay algorithm. The algorithm is implemented as overlay modules in the middleware; therefore, as new overlay algorithms are developed, they can be added to the middleware. In this prototype design, two overlay algorithms are implemented with the virtual organization construction service. Based on these overlay algorithms the construction of the hierarchical VO in MMOP is illustrated in Section 4.3.2.

4.3.1 Overlay Modules

Two different structured overlay algorithms are implemented in the virtual organization construction service. They are One Hop Lookups [4] and Chord [5, 6]. Although these two algorithms both construct a circular overlay with hash keys, different algorithms are used in

each to maintain the overlay and lookup a given key in the overlay. In the next section, the implementation of both algorithms is discussed in detail.

Bootstrap Operations

In order to make the two overlay algorithms compatible with each other in the middleware, a generalized bootstrap operation is developed. The function of a bootstrapping service is to provide newly arrived nodes with a way of joining an existing overlay. At the beginning of the formation of an overlay, a single node is set as the bootstrap master node. This node provides bootstrap service for a specific overlay. When there are many nodes in an overlay, the newly arriving nodes are not limited to use just the bootstrap master node. Any other nodes within the structured overlay can be used to start the bootstrapping service.

When a node on an overlay receives a request for the bootstrapping service, it returns the information about the requesting client's successor. The requesting client should then contact its successor to obtain the latest overlay topology, and receives data objects that it is supposed to host based on the currently deployed overlay algorithm. If, in case, the given nodal information is invalid, the client should request the bootstrap service again from another node until it is able to join the overlay.

One Hop Lookups Algorithm

In One Hop Lookups [4], each node has a complete view of a structured overlay. Virtually, it seems that all networked nodes can be accessed with one single hop. Also, there is a hierarchical network structure which is superimposed on top of the overlay for propagating changes among all nodes quickly. An example of the superimposed One Hop Lookups propagating structure is shown in Figure 4.2.

The key space of One Hop Lookups is being divided into several slices, while each slice is further divided into multiple units [39]. For example, the configuration in Figure 4.2 contains four slices and each slice has two units. Slices are denoted by S1, S2, S3, and S4, and the units are denoted by U1 and U2. The size of the units and slices are predetermined and

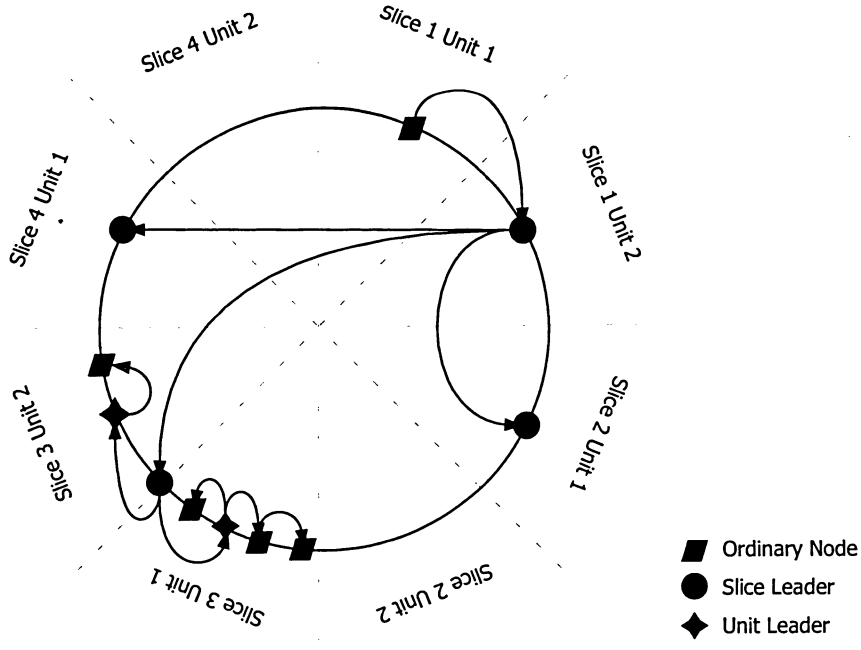


Figure 4.2: Superimposed One Hop Lookups structure used for membership change notification

must be able to divide the key space completely. Each node in the middle of a slice or a unit will become the leader. They are indicated as a circle or a star respectively in the figure. These nodes have the responsibility of notifying other nodes within its slice or unit regarding the membership changes of the overlay. Inter-slice messages are aggregated and exchanged between slice leaders periodically.

Using the bootstrap algorithm discussed in Section 4.3.1, an arriving node receives a list of existing peers currently on the overlay from its successor. The successor notifies the slice leader about the newly joined node and the notification is propagated to the other nodes, as shown in Figure 4.2. When a node fails or leaves the overlay, a similar notification strategy is deployed. This structure ensures timely response to each node but also requires higher bandwidth consumption of the leader nodes. However, since the changes are aggregated within each slice and unit periodically, the bandwidth consumption is minimal.

Chord Algorithm

The Chord [5, 6] algorithm provides each node with a finger (index) table, as showed in Table 4.1. In Table 4.1, k is the current node's key, and $MaxKeyBits - 1$ depends on the current hashing function used. The node recorded in the finger table does not necessarily pinpoint the exact location of the key. It is a rough mapping of a node that is closer to the key than the current node. The query of a key can be executed sequentially or recursively. In our implementation, a sequential key lookup is implemented in order to minimize the resource consumed by lookup queries.

Table 4.1: Example finger table of a Chord node

Index	Node
$k + 2^0$	A node succeeds $k + 2^0$
$k + 2^1$	A node succeeds $k + 2^1$
$k + 2^2$	A node succeeds $k + 2^2$
...	...
$k + 2^{MaxKeyBits-1}$	A node succeeds $k + 2^{MaxKeyBits-1}$

Similar to One Hop Lookups, the same bootstrap operations detailed in Section 4.3.1 is used. A newly joined node retrieves the finger table from its successor as its own finger table with updated index. Unlike One Hop Lookups, a membership change within the network is only noticeable by the two nodes that are directly neighboring the node that caused it. All other nodes need to learn of changes by periodically refreshing their own finger table entries.

When looking up a specific key, a node first goes through its finger table and finds the closest successor for the key. It then contacts that node to determine if such a key is located on it. If the key is not found, the node contacted returns a closer successor from its own finger table. The originating node then restarts the query with the returned nodal location until it reaches the node that contains the key. As a result of using the finger tables, this algorithm is able to locate any key within $O(\log_2 N)$ time [5, 6].

4.3.2 Virtual Organization Construction Algorithm

In this section, a VO construction algorithm is developed and implemented to create a hierarchical VO structure as described in Section 3.1. The hierarchy is created by parent VO storing subsidiary VO (SubVO) contacts. Such record is stored by DISEM with the SubVO's ID as the variable's name. In the implementation, four contact records are stored for each SubVO. These four contacts refer to the successor of 0, $N/4$, $N/2$, and $3N/4$ respectively. These records are updated every minute by looking up their corresponding successor mappings, which ensure correct references are maintained.

With these records, a VO can be easily looked up by contacting any node in the $BROKER$ VO to trigger the VO lookup service. The service refers the client to the parent VOs of the requested VO sequentially as shown in Figure 4.3. The VO construction service also makes use of the VO lookup service to determine if a new VO should be created or not.

With the above described hierarchical VO construction, unlimited layer of VO can be created. For example, a two level $FREE\_SERVER : L1 : L2$ VO can be created by creating $FREE\_SERVER$ VO, $FREE\_SERVER : L1$ VO and $FREE\_SERVER : L1 : L2$ VO in order. We should note that $FREE\_SERVER$ is a SubVO of $BROKER$ VO so that its references can be looked up by contacting the nodes in the $BROKER$ VO. The steps to create layered VO hierarchy are included in Appendix B.

To traverse the complete VO hierarchy, all the lookups and VO creation require contacting a node in the $BROKER$ VO. Therefore, each node stores four recently accessed $BROKER$ nodes as references. At least one $BROKER$ contact should be supplied when starting a MMOP node in order for that node to join an existing VO hierarchy. Without such contact, the node would assume that a new VO hierarchy should be formed and creates a new $BROKER$ VO. Even though the newly created $BROKER$ VO possesses the same VO ID as the intended $BROKER$ VO, these are two independent VOs and nodes in each will not be aware of the existence of the other VO.

From the above example, we demonstrated that it is possible to create infinite levels of VO by creating SubVOs one by one. However, since each level of VO requires a separate query

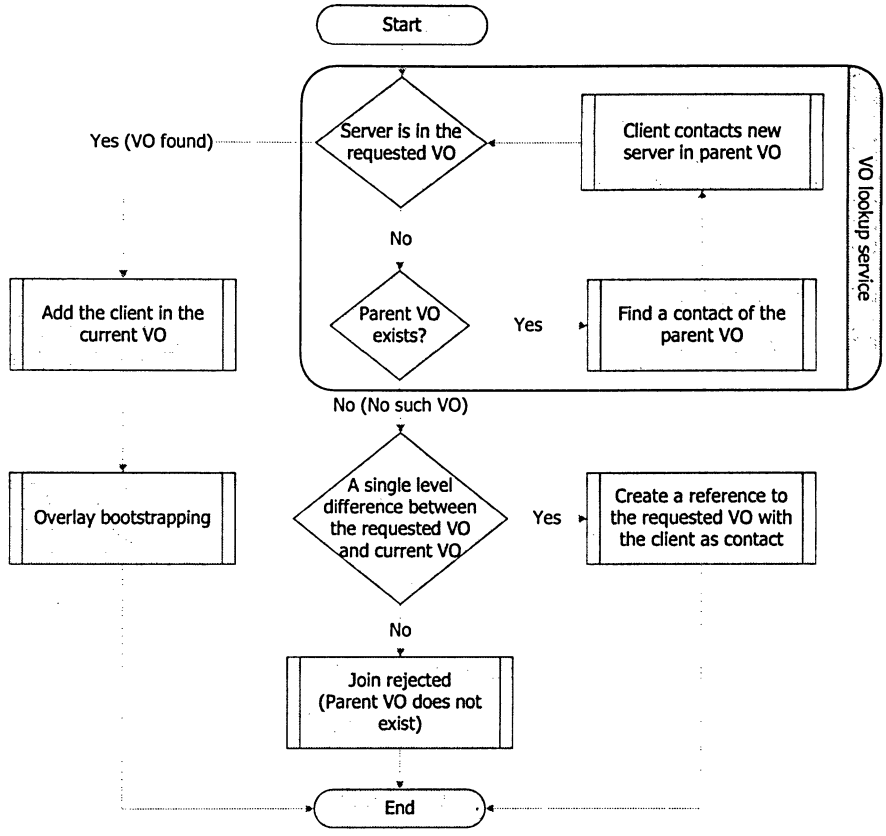


Figure 4.3: VO and SubVO construction algorithm

to its parent VO, it is not advisable to create high level count VOs. A flatter VO hierarchy with minimal levels would provide faster VO lookup time since only queries to a limited number of VOs is needed. It is more than sufficient to create at most one single SubVO level to represent all the VOs since the VO ID is constructed by concatenating ID strings.

4.4 Distributed Semaphore (DISEM) Implementation

Following the design in Section 3.2.5, the main focus of Distributed Semaphore (DISEM) is to provide real-time data access and data consistency with a single set of API. Before the actual API is defined, the structure of the variable should be considered since a properly

designed variable structure allows a much cleaner API design. The data structure of a DISEM variable is defined in Listing 4.1.

Listing 4.1: VO Variable structure

```
class VOVariable {
        String variableName;
        Object value;
        long version;
        Key updating;
}
```

Each variable has an unique name, `variableName`, for identifying the variable. A `value` portion for saving the content is associated with the variable. The `updating` field is to be used to identify the current Synchronization Agent (SA) of the given variable. The SA is a node in the overlay that handles the synchronization request for the client. The SA is constructed dynamically by the requesting client. When client wants to access a variable, it locates the SA by generating a random hash key. The successor node of the key will then be assigned to be the SA of this particular variable access. The `version` field is used to determine the latest version of the variable and prevents old data from writing to the variable.

Expanding the API design in Table 3.1, the DISEM API for MMOP has these basic methods and parameters shown in Table 4.2.

Using this set of APIs, we can access the networked data objects on the structured overlay as local variables. The *CreateVariable* is equivalent to the declaration of the variable, and *ReadVariable* is for accessing the data content of a variable. The tuple *GetVariableForUpdate* and *WriteUpdatedVariable* operations act as the distributed semaphore to protect the critical code section. Moreover, they can also be used for writing content to the variable. In the next section, we will describe the function of the data refreshing system service and then outline the operations of each DISEM APIs using the `VOVariable` structure and DISEM API designed.

Table 4.2: DISEM API for MMOP

API Interface	**Interface description**
CreateVariable (*VOKey*, *VOVariable*)	publishes a variable according to the given *VariableName*.
ReadVariable (*VOKey*, *VariableName*, *ReadType*)	returns the variable associated with the *VariableName* using read method identified by *ReadType*.
GetVariableForUpdate (*VOKey*, *VariableName*)	locks the given variable.
WriteUpdatedVariable (*VOKey*, *VOVariable*)	unlocks the variable and update its value.

4.4.1 Data Refreshing Service

To ensure the integrity of the data, an important data refreshing service (DRS) that operates periodically for data maintenance is required. During each refreshment round, a node iterates over all locally stored variables, and checks for missing replicas. If it detects a discrepancy between replica copies, it tries to recreate them by making a copy of local variable. Since there are no centralized records of created variables, we require every node to run DRS independently. Referring back to the data structure aforementioned, the DRS also checks for outdated data by comparing the version flags. The missing data will eventually be re-established if there exists at least one working replica of that variable.

4.4.2 Variable Declaration

The *CreateVariable* API is used to declare a variable on the overlay. The *VOKey* and the `variableName` pair uniquely identifies the existence of a value under such name in the specified VO. The second parameter, `variableName`, is used with the total replica number n to generate a key that will be used to store such mapping on the structured overlay node. Keys for the given variable replicas are generated by the *variableName* : r pair, where r

indicates a replication number and an integer of the value $0 \leq r < n$. The generated keys and variable pairs are sent to the desired location according to the overlay algorithm used. The value of n can be tuned to achieve certain level of data availability. Simulations have been conducted to determine the best value of n under different networking scenarios, and they will be discussed in Section 6.3.4.

4.4.3 Access Variable

To access an existing variable, the API *ReadVariable* is used. Such API requires a *ReadType* parameter, which is used to indicate which type of read will be used. Several types of read operations are discussed in Section 3.2.5 and the operation of each read operation will be discussed in detail next. The number in brackets is the *ReadType* number to use when a particular operation is desired.

Real time read (0): If the total number of replicas is known, a key can easily be generated for *ReadVariable* access. In order to meet the real time requirement, *ReadVariable* operation returns immediately if any error is encountered during the data retrieval. A `NoSuchVOVariableException` is raised and the variable retrieval terminated. This allows the application to quickly query a variable without any consistency guarantee. However, this method would provide inaccurate result when the node failure rate is high or when the network is not stable.

Normal read (1): This method tries to provide an extra level of security to the real time read while having short access latency. As opposed to the fail-fast behavior of the real time read, the data retrieval should be repeated on all possible replicas. This *ReadVariable* operation simultaneously requests the variables retrieval and returns the first received data. This operation and its two following *ReadVariable* operations guarantee that as long as one of the replica exists, the content of a variable can always be received.

Consensus read (2): Similar to the normal read operation, the consensus operation simultaneously requests the variable retrieval. Unlike the normal read operation, it waits

until all responses are received. It then compares the version tags of all received data to achieve a general consensus. Such variable would represent a consistent view of all the existing replicas.

Most updated read (3): Parallel requests are also used for this *ReadVariable* method. Although achieving consensus would provide the most consistent data across all replicas, this read method provides the most updated replica instead. As its name suggests, the replica with latest version tag is selected and relayed to the requester.

By providing four different types of read operations, the application designer can select the best read operation to fulfill the special need of each read operation. If not specified, the most updated read ($ReadType = 3$) would be used since it works best with DISEM write algorithm.

4.4.4 Write Variable

The most important aspect of the DISEM algorithm is the write algorithm. It is used to ensure data consistency and provides distributed semaphore in the MMOP. In order to provide distributed lock management, the write operation of DISEM is constituted of two phases, $GetVariableForUpdate$ and $WriteUpdatedVariable$. In the $GetVariableForUpdate$ phase, the Write Requester (WR) first randomly generate a key and finds a Synchronization Agent (SA) on the overlay. The SA provides lock management for this specific write access by sending its address and storing it in the replica's `updatingKey`. This key uniquely identifies this specific lock request and is used to resolve discrimination when consensus cannot be established among replicas. A request is broadcasted to all replicas simultaneously from the SA as shown in Figure 4.4.

There are three possible results for the nodes receiving a lock request:

- general failures (which include connection failure, node failure or the variable does not exist)
- the variable is being updated by another node

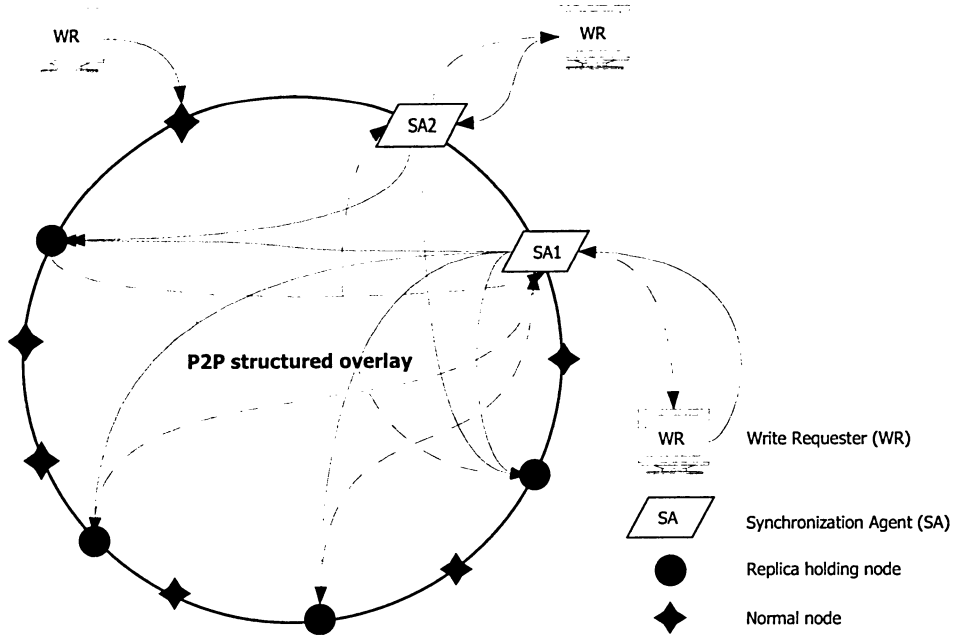


Figure 4.4: DISEM write request algorithm

- and the variable is available for update

A replica only accepts a SA's request when it is not currently being updated by another SA. The request is accepted on a first-come-first-serve basis, where the first node that requested a lock should be granted and the rest should be ignored unless overwritten. The nodes will record the SAs' requests between updates and notify them when it finishes updating. In this way, the SAs will be able to collect a consistent and up-to-date view of all replicas. Accumulating the responses from all the replicas, the SA maintains the following:

- total number of failed requests ($|R_{failed}|$)
- the set of all successful requests ($R_{successful}$)
- the set of successful requests by current SA ($s \subset R_{successful}$)
- and the set of all successful requests by other SAs ($o \subset R_{successful}$), where o_i represents all the replica secured by SA_i.

Moreover, $m(n) = \lfloor \frac{n}{2} \rfloor + 1$ is defined for the given overlay where n is the total number of replicas on the overlay and $m(n)$ is the majority of the replicas. By using these defined values, a single SA can be selected to provide atomic writing. Furthermore, $m(n)$ can be adjusted for each VO according to different values of n.

Case I: $|R_{failed}| = n$

This particular case indicates that there is no such variable or all replica retrievals have failed. Therefore, a lock for the variable cannot be granted.

Case II: $m(n) \leq |R_{failed}| < n$

This case signifies that the lock cannot be obtained because majority of the requests have failed due to node or network error. Since the lost replicas will eventually be recovered by DRS, the WR will be notified to retry later.

Case III: $|s| \geq m(n)$

The SA is able to secure the majority of replicas for this write request and relay data to WR for writing.

Case IV: In this case, multiple SAs are created simultaneously for writing a single variable. Being an atomic data access algorithm, only a single SA should be allowed to obtain the lock at any given time. Therefore, in order to achieve $|s| \geq m(n)$, some SAs need to give up their lock requests so that one single SA obtains the lock. The following two cases are considered to determine which SA should quit gracefully and which SA should try again to obtain the majority of locks:

Case A: $|s| > \max\{u\}$

where $\max\{u\}$ is the maximum number of secured nodes being held by any other SAs, this case indicates that the current SA has the highest probability to achieve $|s| \geq m(n)$.

Case B: $|s| = \max\{u\}$

in this case, we have more than one SA that have the maximum number of secured nodes. Therefore, we need to resort to the key of the SAs to determine the proper candidate. The keys are compared among the SAs, the SA that has the highest

key will be identified. A single SA is then selected to proceed to overwrite other SAs' locks.

With only one SA satisfying one of the above two conditions, atomicity can be achieved. SAs that do not satisfy the requirements simply notify the WR about the unavailability of the variable.

Upon securely obtaining the variable lock, the data is relayed to WR for writing. WR can manipulate or update the given data multiple times. The version number is updated every time a new update is being made to keep track of the changes. When WR finishes the update sequence, the variable lock is released by entering the *WriteUpdatedVariable* phase. WR sends the updated variable to the SA identified by the `updatingKey` of the variable. If any step fails before a variable update is completely relayed to the SA, the SA is responsible for releasing all the replica locks. If the SA fails during the update, the replicas will be able to detect it since TCP connections are established between SA and replica servers and the variable lock will be released.

With the updated variable completely received by the SA, the SA then writes the updated variable onto each locked replica on behalf of the WR. However, the replica only accepts a variable version number that is larger than the current version number to ensure the delayed data updates will not be written on top of new data. When every replica is updated, the SA unlocks all replicas for the next write request.

4.5 Application Deployment Service

A simple application deployment service is provided for deploying a Java application class file to any *APPLICATION* VO. The basic interface of the service is as follows:

```
public interface MMOPService {
        final String name;
        final String version;
        final ApplicationPolicy policy;
        final Object[] initParameters;
```

```
        public void run(Object[] runParameters);
}
```

To deploy a MMOP service that implements such interface, the application deployment service requires two additional parameters: the target *APPLICATION* VO, which the application will be deployed in, and target *FREE_SERVER* VO, where the available resources will be taken from. In addition, these VOs should already exist prior to the service deployment.

The application deployment service relies on the QoS provisioning service to locate a free server that satisfies the requirement of the application policy. When such server is found, an independent thread with the specified priority is created on the server. The loaded application is first initialized with the `initParameters` and then the *run* method will be executed with `runParameters` in the thread created. After an application is successfully deployed, the resource requirement of the application will be subtracted from the original available resources. If there are no resources left on the machine, the machine will be removed from the *ResourceList* and the *FREE_SERVER* VO.

As mentioned in Section 3.3, the name and version of the service are used to generate a unique key for this application. A DISEM variable *name* : *version* is created on the *APPLICATION* VO to store the total number of service replica number ($Total_{replica}$). This enables the MMOP to balance the load of a particular service by randomly selecting between service replicas. Each replica service's identifier string is generated as *name* : *version* : *replica#*, where *replica#* is the integer replica number of that service. The first service is always deployed as *name* : *version* : 0.

If the application has a set spawn threshold, the application service will register the application with the QoS provisioning service to actively monitor the set threshold. If such threshold has been exceeded, the deployment service creates a replica of original service, where the identifier string of the new service replica will be *name* : *version* : $Total_{replica}$ and the *name* : *version* DISEM variable will be increased by one after a successful service deployment. However, if no server is available for service deployment, the deployment service will disable the spawn threshold after five attempts.

4.6 QoS Management Service Implementation

There are two main services for the QoS management service as described in Section 3.4, QoS monitoring service and QoS provisioning service. The implementation of each service will be discussed in this section in detail.

4.6.1 QoS Monitoring Service Implementation

Several monitoring modules are implemented in this service, particularly the CPU, memory, network and latency monitoring modules. Two modules are excluded in this set of modules. The active connection monitoring is excluded because several simulated clients will be active on the same machine and contribute to a constant high connection count, which renders the monitored results futile. The storage size monitor is also not implemented because the storage size parameter is not used for our particular MMOP implementation.

The earlier versions of Java (before 5.0) do not support direct monitoring of system statistics because the Java code is executed on top of Java Virtual Machine and has no direct access to that information. Fortunately, the Linux machine has `/proc` a file system that records such information. The file system contains statistics of computing resource utilization and is a common feature on all Linux systems. Therefore, the QoS monitoring services are implemented by reading different files in the Linux `/proc` file system.

Detailed CPU and memory usage status of each service can be gathered by exploring the `/proc/[number]/stat` directory where `[number]` is the service (process) ID. In MMOP, we are more interested in gathering system- wide statistics as it indicates the overall system performance of such machine. Therefore, the monitoring modules will focus on gathering these data by reading the corresponding file every 300 ms.

CPU Monitor Module

This module is used to monitor the overall CPU usage of the host system, the file `/proc/stat` is read periodically to extract this information. The data stored in the file looks like the

following:

```
cpu  70318 17200 16003 215775100 12748 523 1044 0
cpu0 70318 17200 16003 215775100 12748 523 1044 0
intr 542283892 539681460 [... lots more numbers ...]
ctxt 30096832
btime 1172090912
processes 62517
procs_running 1
procs_blocked 0
```

The data that is important to the CPU monitoring module comes from the first line, where the first four values are common to all Linux systems. The module only makes use of these values so it can be used on any machine running Linux. The four values following "cpu" in the first line indicate user, low priority user (nice), system and idle CPU usage respectively. With these values the current CPU usage is calculated as

$$\frac{user + nice}{user + nice + system + idle}$$

Memory Monitor Module

The overall memory usage is monitored by reading the `/proc/meminfo` file. The content of the file looks like the following:

```
MemTotal:       1002168 kB
MemFree:         472476 kB
[... more data ...]
SwapTotal:      2939852 kB
SwapFree:       2939852 kB
[... more data ...]
```

In the above listing, only the relevant data to the module is shown. Particularly the values indicating Random Access Memory (RAM) usage, the `MemTotal` and `MemFree` are used to cal-

culate the current memory utilization. The utilization can be identified as $\frac{MemTotal-MemFree}{MemTotal}$. From the listing we can also observe that `SwapTotal` is equal to `SwapFree`, which means that no swap memory is used. In most Linux systems, the swap memory will only be used once the `MemFree` goes to zero because the swap memory is usually on a much slower memory storage, such as a hard disk.

Network Monitor Module

This particular module provides network usage and both inbound and outbound traffic information are made available to the MMOP. The monitored values include throughput, packet error rate, and packet drop rate. On each testbed machine, there are a total of five network interfaces available. They are identified as *ethN* in the `/proc/net/dev` file, where N is the interface number. The interface *lo* is the loop back interface for network traffic directed from this machine to itself.

```
Inter-|   Receive                                                |  Transmit
 face |bytes    packets errs drop fifo frame compressed multicast|bytes    packets errs drop fifo colls carrier compressed
    lo: 3481712     681    0    0    0     0          0         0  3481712     681    0    0    0     0       0          0
  eth6:20541222  245279    0    0    0     0          0         0 20979892  249116    0    0    0     0       0          0
  eth7:20122134  241552    0    0    0     0          0         0 21064652  249878    0    0    0     0       0          0
  eth8:19991112  242157    0    0    0     0          0         0 20798014  245693    0    0    0     0       0          0
  eth9:20134888  241707    0    0    0     0          0         0 21531466  254675    0    0    0     0       0          0
  eth5:45162072  265593    0    0    0     0          0     17298 54618706  172782    0    0    0  4585       0          0
```

To query the statistics on a particular interface, an IP associated with that interface can be used. The module then finds the corresponding network usage information by mapping the IP address to an interface in the `/proc/net/dev` file.

Latency Monitor Module

The latency monitor module consists of two parts: the latency monitor server and the latency monitor client. A UDP probing packet is exchanged between the client and server to measure the latency between them. The client sends out a probing packet to the server, while noting the packet send time T_{send}. The received server then stamps the packet with its own time T_{server} and sends it back to the client. The client then records the receive time $T_{received}$ upon receiving the server's reply. The round trip time (RTT) is calculated as

$RTT = T_{received} - T_{send}$. The one way latency in MMOP is defined as $RTT/2$ since static routes are setup in the testbed and the paths for packet to travel to and from the server is symmetric. When dealing with an asymmetric network, the client to server latency can be calculated by $T_{server} - T_{send}$, while the server to client latency is $T_{received} - T_{server}$. However, the latency calculation is only valid when the clocks on both machines are synchronized.

4.6.2 QoS Provisioning Service Implementation

The QoS provisioning service is primarily used by the application deployment service for service deployment purposes. In order to efficiently utilize computing resources, all the servers joining the $FREE\_SERVER$ VO will also put its service information (particularly their IP and port number) into the *ResourceList* DISEM variable. The application deployment service can request a server lookup by giving the application's policy requirement to the provisioning service. The provisioning service takes a snapshot of current *ResourceList* and contacts these resources in several batches. Each resource lookup batch includes total of K entries, the value K can be adjusted to obtain faster server lookup at the expenses of higher network and CPU usage. In the current implementation $K = 5$ is used.

The policy requirement is sent to each of the contacted machines so that it can be checked against the available resources. The machines that satisfy the policy will respond to the provisioning service with an available indication, while the others respond with an unavailable indication. The provisioning service then randomly picks one available machine from the batch request, which will be used by the application deployment service for installing the service. If there are no machines satisfying the policy requirement after going through the whole *ResourceList*, the application deployment service will be advised to retry at a later time.

Upon application deployment, the policy requirement of the application will be passed to the provisioning service for continual service monitoring. More specifically, the provisioning service will be actively monitored each second for the applications that have specified the Spawn Threshold parameter. When such threshold is reached, the provisioning service will notify the application deployment service. The application deployment service, upon

receiving such notice, will try to acquire a new server to balance the load of the overloaded application.

4.7 Distributed Timing Service Implementation

The distributed timing algorithm presented in [33] is implemented to fulfill the synchronization timing needs of distributed applications. The algorithm implemented is illustrated in the following:

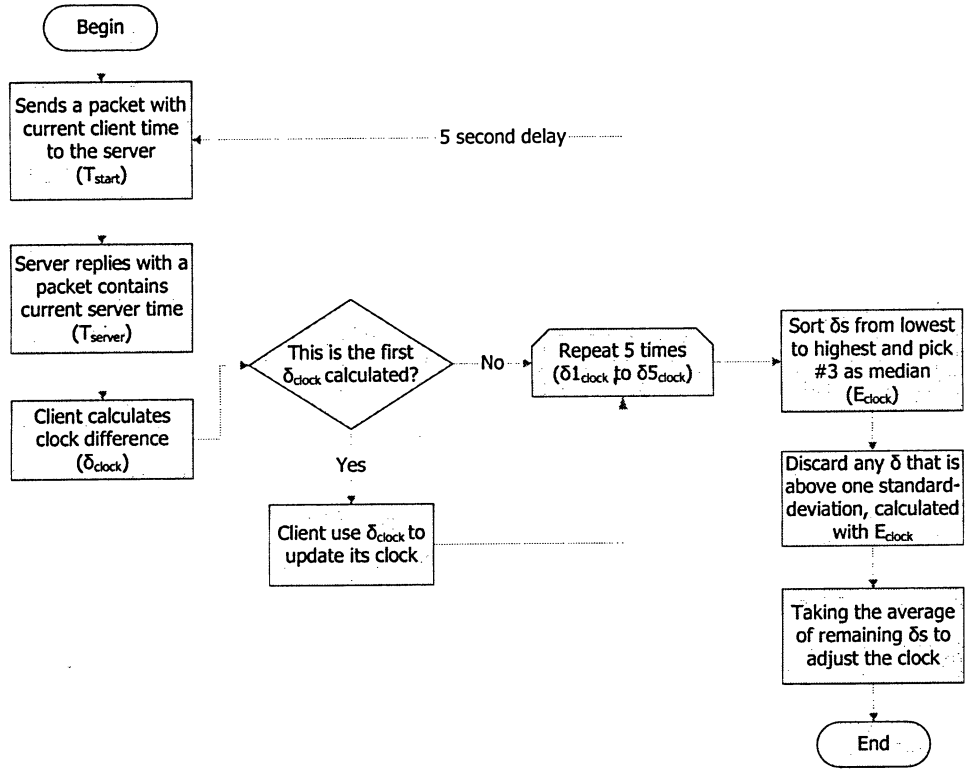


Figure 4.5: A stream-based time synchronization

In Figure 4.5, δ_{clock} is calculated by $\delta_{clock} = RTT/2 - (T_{server} - T_{send})$, where $RTT = T_{received} - T_{send}$ and $T_{received}$ is the time that the server's reply packet is being received on the client. In the second last step of the algorithm, the δ_{clock}s above one standard-deviation

are discarded to eliminate extra latency introduced by packet retransmission. By removing these anomalies from the rest, the algorithm is able to determine an accurate clock difference between the client and server.

In MMOP, the implemented service allows any client to become a secondary time server. With this option, the service can construct several levels of secondary time servers. A distributed timing system that synchronizes to one central clock can be formed using these secondary time servers, which allows direct use of the received time stamps for other MMOP services.

4.8 Summary

In this chapter, we elaborate the MMOP design shown in Chapter 3 and implemented a prototype Massively Multi-user Online Platform (MMOP) middleware. The network layer of the middleware is constructed by Java NIO, the network layer provides a simple set of methods for the other MMOP services to use. Prototype MMOP services developed are: the virtual organization construction, Distributed Semaphore (DISEM) service, application deployment service, QoS management service, and distributed timing service. In the next chapter, a simple MMOG called SPACE SHOOTER is designed using the prototype MMOP middleware. The prototype MMOP verification and evaluation can be found in Section 6.4.

Chapter 5

Distributed 2D Shooting Game Design

In this chapter, we will discuss design and implementation of a simple game called SPACE SHOOTER, it is a distributed 2D shooting game that uses services provided by the Massively Multi-user Online Platform (MMOP) middleware will be discussed. Features and objectives of the game will be introduced in the first section, followed by discussions on the technical design of the game and the game server, as well as the clients that utilize services provided by MMOP. The game functionalities, such as Graphical User Interface (GUI), Artificial Intelligence (AI), latency compensation, and server bandwidth saving techniques, are will be discussed and implemented as well.

5.1 Game Features and Objectives

SPACE SHOOTER is based on the Space Invaders 101 source code and graphics provided by [40] with extensive modifications. The designed game is played on a two-dimensional space. Each player controls an aircraft that can move freely in this space. The objective of this game is to obtain the highest score by using missiles to shoot down as many of their opponents' aircraft as possible. Each missile hit will deduct an opponent's number of shields by one and increase a shooter's score. When the number of shields an aircraft has reaches zero, the player will be removed from the game and the score is reset to zero.

A list of top scoring players is shown in the game in order to create a sense of competition. In addition, the names of all the opponents whose aircrafts are approaching a player will appear on the screen. Furthermore, a player can create additional simulated players from

the user interface directly to make the game more interesting. In order to support these in-game features, detailed designs of the in-game entity and controls are presented next.

5.1.1 Detailed Game Entity Design

There are two basic entities in the game: one is the missile and the other is the aircraft. Both of these entities require the basic property of position and the ability to move. In order to identify such entities on a players screen, the server sends updated messages containing information on position and movement of the entities. Furthermore, a unique identifier of the entity is included in the message in order to match an update message to the entity replica on the client. Therefore, a universally unique identifier (UUID) is required. The UUID should be generated randomly for a missile entity or from the name string of a player entity. The UUID will generate the same identifier if the same string is used. A 128 bits long identifier defined in [41] is used in the game, which guarantees uniqueness across space and time. With the UUID, the server can send out update messages targeted for each entity. Basic setup of an in-game entity is illustrated in Table 5.1.

Table 5.1: Attributes of basic entity in SPACE SHOOTER

Parameters	Parameter description
UUID	a unique identifier that is used to identify objects throughout the game.
Position (X, Y)	location of an entity in 2D space; consists of X and Y coordinates.
Angle (R)	the direction which an entity faces.
Velocity	speed of an entity, which is used to calculate the position of an entity as the game progresses.

The missile entity is a simple extension of a basic entity with a constant speed of 450 pixels per second. The aircraft entity, controlled by a player, is a more complex extension of the basic entity and has a maximum speed of 250 pixels per second. More attributes of an aircraft are listed in Table 5.2.

Table 5.2: Attributes of aircraft in SPACE SHOOTER

Parameters	Parameter description
Player Name	the name of player who controls the aircraft.
Shield	number of shields an aircraft has; in the beginning, each aircraft starts with five shields. Each missile hit reduces the shield value by one. When the shield value reaches zero, the player is removed from the current game.
Score	score obtained by the player. When the player launches a missile that hits other aircrafts, the players score is increased by 100. The score is reset to zero if the player leaves the game, is removed from the game, or is disconnected from the game.

The player controls the aircraft by issuing commands through the directional keys on the keyboard to move it either forward or make it rotate to the left or right. When the forward key is not pressed, the aircraft will come slowly to a halt. The player can fire missiles using the space key. The player's movement is aggregated and sent to the server every 100 ms. This time frame is defined as one action interval. While the player must press and hold the "Forward" key for the aircraft to continue flying forward, the client only sends an update message to the server once a change in command occurs to reduce network congestion. This is achieved by having the server assume that the same keys are pressed as long as the client does not send any update message. Unlike the movement updates, the missiles can only be fired every second. This is to avoid excessive missiles being fired when a player holds down the space key. In the next section, technical details about the client and server designs are discussed.

5.2 Client Design

There are three main components of the client design which will be discussed in details in the next few sections. First, a graphic user interface (GUI) is developed on the client side for visualizing and experimenting the actual game play. The game GUI is shown in

Figure 5.1 and the explanation of each component is given in Section 5.2.1. Second, the artificial intelligence (AI) design provides ability to simulate the game without the need of actual players to play the game. Its design is discussed in this section. Third, in order to provide a smooth gaming experience, client side interpolation and dead reckoning techniques are used to reduce visible network jitters and delays.

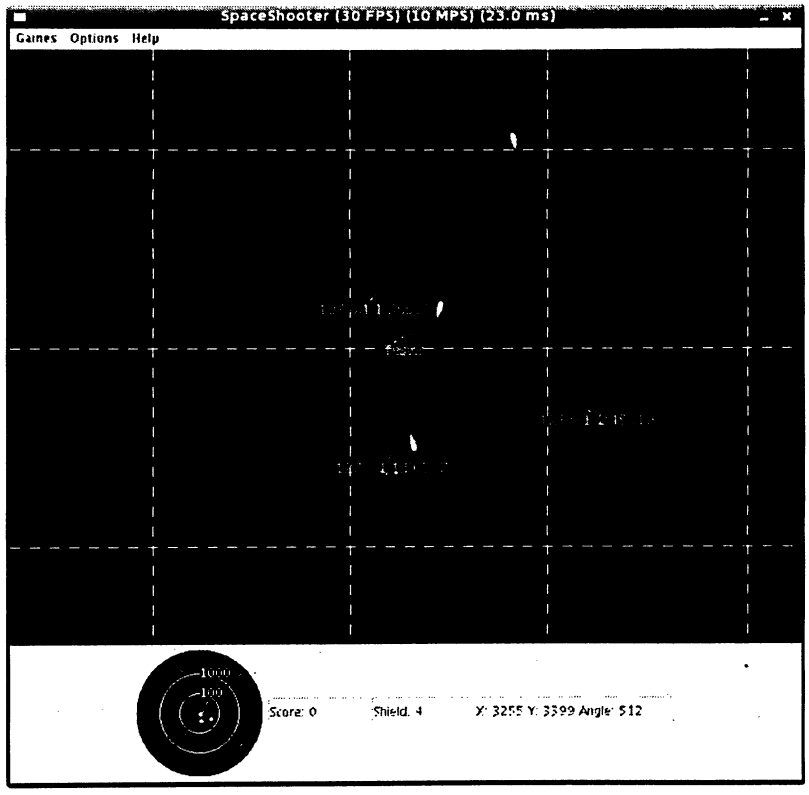


Figure 5.1: GUI for SPACE SHOOTER

5.2.1 Graphical User Interface (GUI) Layout

The basic layout of the GUI is shown in Figure 5.1. Each component of the GUI will be discussed in details in this section as they appear in a top-bottom and left-to-right order. The first component is the title area of the GUI. As shown in Figure 5.2, this component displays several useful statistics about the game. They are:

Title the name of the game is displayed here.

FPS indicates the number of frames displayed per second. Each frame is a snapshot of the entities within a player's visible range, which is limited by the main game window.

MPS is the number of received update messages per second. A larger number of updates received per second indicates a higher accuracy of entities' rendered position on the screen.

Latency is the average of latencies for the update messages received over one second period. A lower latency indicates a faster response rate for player's control.

Figure 5.2: Title for SPACE SHOOTER

In a fast paced action game like this one, players must be able to visually observe their surroundings and react quickly. Through trial-and-error, frame rates between 25 to 30 FPS are determined to provide a pleasant gaming experience. The game will try to generate as many as 30 frames each second. However, as there are other processes being executed on the same machine, the FPS value may reduce.

Figure 5.3: Menu for SPACE SHOOTER

To provide game features that are accessible and user-friendly, Figure 5.3 shows three main game menus: Games, Options, and Help.

Games is the main gaming option, there are three sub-menus accessible through this option. They are:

New Game upon selecting this option, player is required to provide a unique name to join the game. If another player is currently using the name provided, then the join request will be denied.

Simulation this option allows player to specify a number larger than zero in order to create simulated clients. Player should note that the simulated clients are created locally and would greatly reduce the performance of the game if many clients are created.

Exit closes the GUI and disconnects the player from the server.

Options this menu has two selections:

Toggle Name Display toggles the visibility of other players' name plates.

Highest Scores contacts the server and retrieves the top ten scores along with the name and the ranking of the record owner.

Help displays a simple help menu for game control.

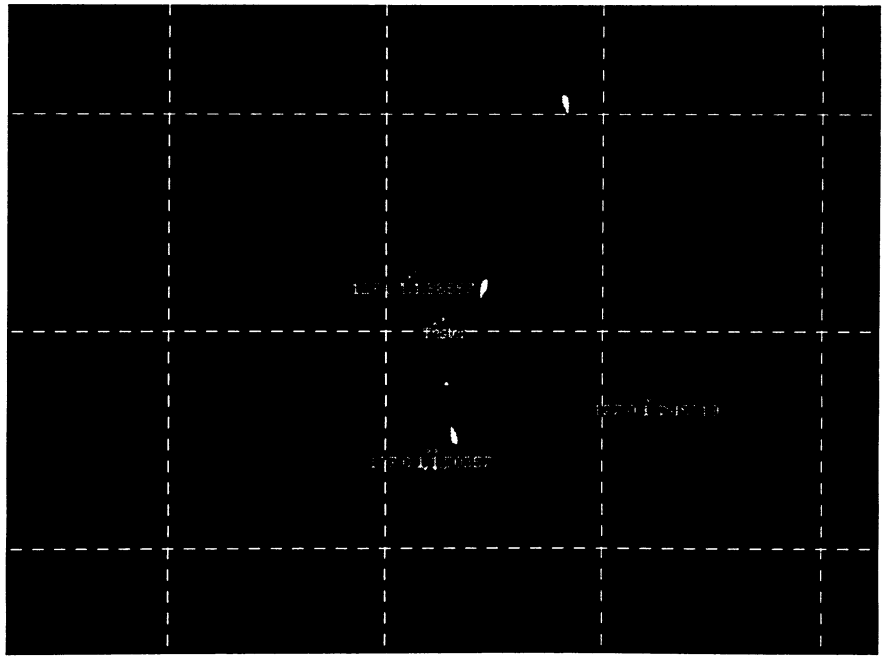


Figure 5.4: Main gaming area for SPACE SHOOTER

In the main gaming window shown in Figure 5.4, the aircraft controlled by the current player will always be on the center of the display window. The display area is 800×600 pixels^{2}with

dotted lines spaced at 200 pixels. The dotted lines will move as the aircraft moves through the space. This is used to create a sense of movement since the player controlled aircraft appears stationary and centered on the display window.

Figure 5.5: Radar and statistics for SPACE SHOOTER

At the bottom of the GUI, a radar is included along with basic statistics of the aircraft. Other statistics displayed include the score, shield, current location and angle of the aircraft. The radar is a useful reference when players are looking for opponents or avoiding missiles. The radar has a radius of $Radius_{radar} = 64$ in pixels and a scale factor of $Scale = 20$, and is updated every 100 ms. The actual visible radar radius can be identified as $Radius_{actual}$ and is calculated by $Radius_{radar} \times Scale = Radius_{actual}$. It covers an actual area of $\pi Radius^2_{actual}$ pixels^2, which is significantly larger than the main display area of 800×600 pixels^2. In order to properly distinguish the distance and the type of entities on the radar, the entities are represented as colored dots on the radar:

Blue indicates the aircraft controlled by the player, which is always located in the middle of the radar.

Yellow indicates opponents' aircrafts that are within the range of $ActualRadius$.

Red represents all the missiles within the range of $ActualRadius$.

Green is used to represent opponents' aircrafts that are outside the range of $ActualRadius$.

5.2.2 Artificial Intelligence (AI) Design

To simulate a player's action, a simple AI has been designed. At each action interval, the AI controlled client follows the decision logic flow shown in the Figure 5.6.

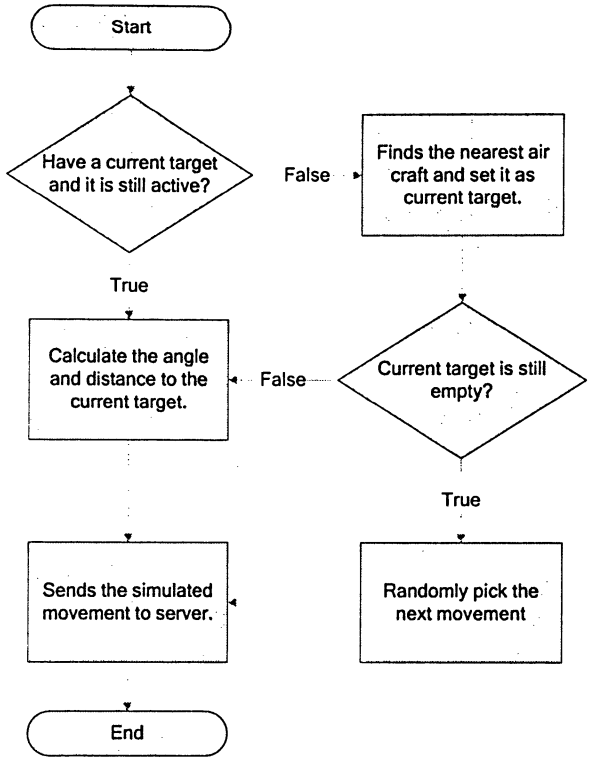


Figure 5.6: AI design for player simulation

The distances between the AI controlled aircraft and other aircrafts are calculated using $(DiffX)^2 + (DiffY)^2 = Distance$, where $DiffX$ is the difference in x-coordinates and can be represented as $DiffX = X - Other.X$. $DiffY$ is similar to $DiffX$ but represents the difference in y-coordinates instead.

In order to move the AI controlled aircraft closer to the target, the aircraft needs to turn first towards the target. The turning angle is calculated by converting $DiffX$ and $DiffY$ into polar coordinates. To properly align the aircraft towards the target, the AI computes the Θ component of the resulting polar coordinates. Then the Θ component is converted into degrees and subtracted from the current aircraft's angle to determine if the aircraft should rotate left or right.

The *Distance* between the current aircraft and the target is also used to determine if forward

movement and fire missile command should be issued. If the *Distance* value is outside the range of $[900, 90000]$ then the forward command should be issued to move the aircraft closer to the target. When the simulated aircraft is inside the range, it should slow down and adjust its aim towards the target. On the other hand, if $Distance \leq 1000000$, then the missile fire routine is called and a missile fired if one second has passed.

5.2.3 Client Side Latency Compensation

In this game design, two types of delay compensating techniques are used to make latency issues less visible on the client side. They are client side interpolation and dead reckoning techniques. Each technique's effect on the game play and the game mechanics are discussed.

Interpolation

When interpolation is not used, the entities can only be rendered at the position received from the server's update message. However, since there is no position information available between each updates, the movement of entities would appear choppy. Moreover, the moving entities and animation would appear jittery, since the network latency between the server and client is not always constant. Noticeable glitches would also result as burst background network traffic is introduced.

The basic idea of client side interpolation technique is to always render past game state instead of the most current game state. By rendering past position, the GUI can create continuous animated position snapshots between two most recently received server updates. With ten snapshots per second, a new update from server arrives about every 100 ms. If the client render time is shifted back by 100 ms as well, then entities can always be interpolated between the last received update and the update before that. Since TCP is used for our game design, a tighter bound can be selected since no server updates will be dropped. For games utilizing the UDP protocol, a higher degree of interpolation is required to compensate the possibility of updates being dropped. For example, taking twice as much interpolation

time (200 ms) would ensure that there are at least two valid updates to interpolate between. Following figure shows the interpolation timeline with ten incoming updates per second as used in the game:

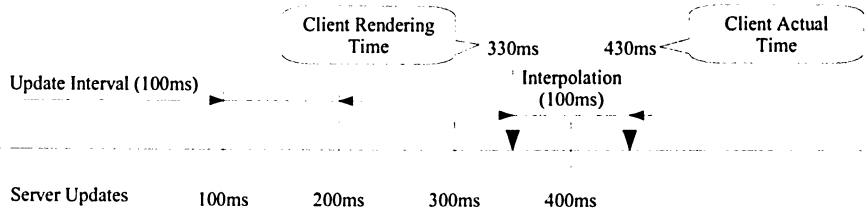


Figure 5.7: Timeline for entity interpolation

From Figure 5.7, we can see that the last update received on the client was at 400 ms. The actual time on the client continues to advance and when a frame is being rendered, the current rendering time is the actual time minus the view interpolation delay of 100 ms. This would be at 330 ms in Figure 5.7 and all entities' positions are interpolated between 300 ms and 400 ms updates. By doing this, the player is effectively seeing a constant delay of 100 ms.

In order to calculate the accurate interpolated positions, each update message is stamped with the server clock. However, if the the clocks on server and client are not synchronized, the interpolated position would be meaningless as the client time may not produce a valid interpolation render time that falls between the updates. For example, if the server clock is one second behind the client's clock, the client rendering time would be interpolated to be at 1330 ms using the example above. Such time does not fall between any of the server updates and therefore cannot be correctly displayed. Therefore, the distributed timing service described in Section 3.5 is used to ensure that the server and client clocks are synchronized.

By comparing the gaming quality before and after implementing client side interpolation, the animation jitters are reduced significantly. Furthermore, the selected value of 100 ms view interpolation delay is not noticeable when playing the game.

Dead Reckoning

The dead reckoning technique used in the game design is a stripped down version of the ones used in the Distributed Interactive Simulation (DIS) protocol [42]. This technique has been widely implemented and was once a popular form of navigation used to determine the position of a ship or airplane back in the old days. A new position can be estimated by advancing a previously known position to a new one based on the direction of a vehicle's motion and velocity. Although this method of navigation has been largely replaced by electronic navigation systems nowadays, it is still often used as a backup in case of equipment failure.

In the game, missiles are generated by the dead reckoning technique. When a server receives the fire command from a client, it creates a missile entity and sends its position to all players within range. Since the speed of the missile and its heading are constant, the client can calculate given the missile's position at any rendering time by using the following formula:

$$Position_{current} = Position_{beginning} + Speed_{missile} \times Time_{current}$$

. To utilize the dead reckoning technique, the server only needs to send out two update messages to each client for each missile. The first message is used for notifying the creation of the missile and the second for notifying the removal of the missile. To further reduce the bandwidth requirement, all missiles have a constant maximum flying time ($TFlying_{max}$) of five seconds. When the $TFlying_{max}$ is reached, the missile will be automatically removed and hence, no update message is required for such a removal to take place. The only time a removal message is required is when a missile collides with an aircraft.

In comparison with the ten position updates per second for each missile, the dead reckoning technique effectively reduces the total number of updates from $10 \times TFlying_{max}$ to merely one or two. Significant improvements on gaming quality have been observed when employing the dead reckoning technique. Without dead reckoning, playing a game with as few as then clients on a single server can become quite an insurmountable task. However, a single server can easily handle more than one hundred clients if the dead reckoning technique is made available.

5.3 Server Design

Many popular Massively Multiplayer Online Games (MMOGs) provide competitive environments for players to interact with each other. In these games, player states are persistent and cheating would have long term effects on the balance of the game. In most MMOGs, clients are only authorized to perform tasks that do not result in cheating such as movement prediction. Current MMOGs rely on the server architecture to manage most of the game state computations, resulting in higher computational resource requirements. Hence a high performance server architecture is needed to provide services to a large number of clients.

The SPACE SHOOTER game design utilizes the most common client-server architecture for MMOG design. All of the aircraft movements, missile movements, and collision detections are done on the server side. The client only receives the location and status update messages from the server. Although simple, this basic structure ensures a consistent view of the game world to all the players. The downside is that when many players are connected to a single server, the server becomes overloaded and results in a poor gaming experience for the players.

However, many MMOGs suffer from the same server overload problem. One approach to solve the server overload problem is to divide the gaming space into several disjointed areas (zones), with each of them hosted on a different server. These servers are usually called the zone servers. Each client is connected to a specific zone server that is in charge of maintaining the game state of that particular client. The update information is aggregated by the zone server and then relayed to the client. In the distributed 2D shooting game, the idea of zone-based approaches is adapted to create the quadrant servers as shown in Figure 5.8. Similar to the zone server, each quadrant server controls all the entities within its service area. In Figure 5.8, the red and green colored numbers indicate four and sixteen quadrants VO setup respectively. By creating sub quadrants from a larger quadrant, the load on a larger quadrant can be reduced by distributing the load on four servers with smaller service area.

In contrast to the traditional client-server design, the Broker VO in the MMOP acts as bridges between the clients and servers. The virtual organization construction service cre-

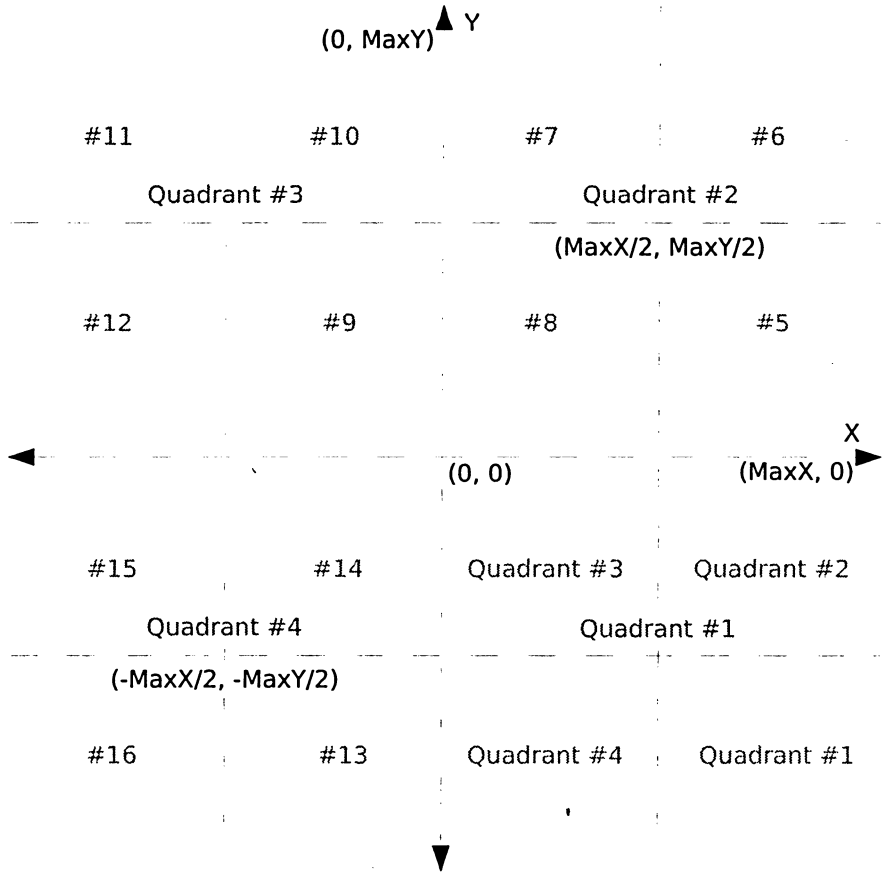


Figure 5.8: Quadrant division for SPACE SHOOTER game server

ates the VO structure as requested by the deployment script. Aside from the zone server construction, the game uses DISEM described in Section 4.4 to store persistent player data, which is listed in Table 5.2 every second. The highest score player will also be recorded using DISEM on the *APPLICATION* : *SpaceShooter* VO. There is a total of ten slots in the highest score record. The player data between quadrant servers are not shared using the DISEM protocol. This is to avoid extra communication overhead of storage through the DISEM protocol. The quadrant servers exchange the entity replica directly between each other. As a result, the player's aircraft is able to travel between quadrant servers seamlessly.

Moreover, each server application is assigned a policy profile to monitor the QoS of the server. A simple threshold of 80% of CPU usage is set for each server. When a server is overloaded, the QoS management service will be trigger and a new server will be located for load balancing purposes. However, since the actual available machines in the testbed are limited, no new server will be allocated. This is because all the machines are usually at their maximum load while simulation is being done. In other words, either the machine is loaded with simulated clients or the server application is already running on it.

With the help of MMOP services, we are able to construct the 2D space shooter game fairly easily. However, the result is not always satisfactory at first due to the amount of traffic generated from the large number of active simulated clients. With the server bandwidth identified as the main bottleneck of our prototype design, several strategies will be discussed in the next section in an attempt to resolve this issue.

5.3.1 Bandwidth Conservation Strategies

There are three main strategies commonly used in online games to reduce the bandwidth requirements of a MMOG. These strategies consist of interest filtering, message aggregation and data compression. Only the first two strategies are implemented in the design of the SPACE SHOOTER game given the fact that they significantly reduce the bandwidth requirement for the server. The compression technique is not implemented primarily because it requires more computational resources compared to the other two strategies. However, compression can be easily added as an extra module to the game protocol encoder and decoder.

Areas of Interest Management (AOIM) Filtering

The interest filtering strategy used on the server is called the Areas of Interest Management (AOIM) filtering [43]. It uses different strategies to decide which host is interested in which particular portion of the game state. This effectively eliminates the update messages that will not be noticed immediately by the client.

For example, in the SPACE SHOOTER, the most important information to the players are the entities within their visible range. These are entities that will be displayed on the GUI main window as shown in Figure 5.4. Therefore, as far as the player is concerned, any entity movements outside of this area are not relevant. However, since the GUI also provides a radar as in Figure 5.5, entities updates are also necessary within the radar range. With these considerations, two tiers of areas of interest filtering are setup for each player. The first tier filtering includes the entities within the radar range of $Radius_{actual}$. The updates satisfying this criteria are issued ten times per second. The second tier filtering includes all the entities outside of the $Radius_{actual}$, and hence requires less frequent updates. The entities within the second tier filtering range are updated every second.

There are two approaches to implement the actual filtering. First, a strategy similar to collision detection is used in the SPACE SHOOTER game server. This approach involves checking all the other entities when constructing update message for a particular client. In our experiments, this strategy works fairly well when there are less than a thousand active players online.

Second, a more advanced filtering strategy is to sort the entities by their X and Y coordinates. The entity information needs to be included in the update so it can be quickly identified by looking up the range for each client on the sorted list. This strategy can also be applied for collision detection. However, since the entities are moving rapidly, the list needs to be sorted every time an update is required.

Simulation result in Figure 5.9 is conducted with increasing number of total players. In the simulation, $10 \times \#players = \#missiles$ are produced, hence each player needs to be checked against at most $\#missiles$ in the direct collision detection case. We can observe that when there is less than a thousand players on a single server. Direct collision detection provides better performance than XY entity sorting. This is mainly due to the excessive data sorting and object creation overhead required for XY entity sorting strategy.

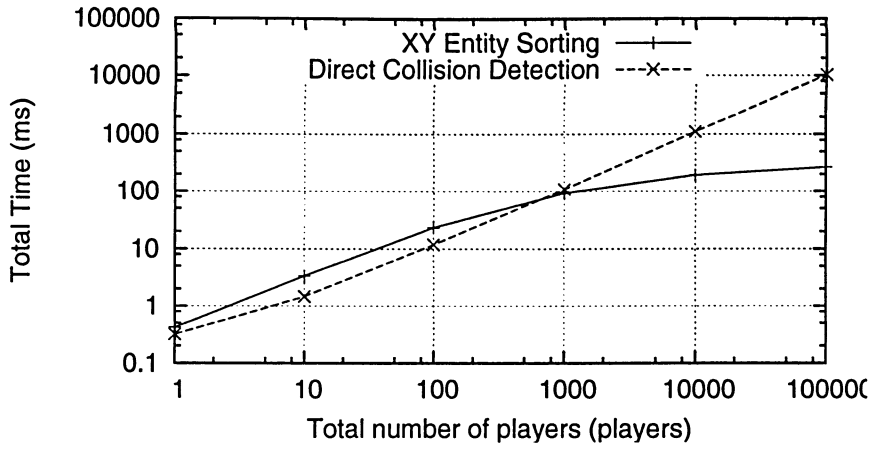


Figure 5.9: Direct collision detection versus XY entity sorting

Data Aggregation

The notion of data aggregation is to combine all the update messages using aggregation strategies to reduce the total number of messages to send. This strategy works well when there are many small entity updates. For example, a typical entity position update consists of UUID, position and angle. The size of such update is $UUID+X+Y+R = 128+64\times3 = 320$ bits (or 40 bytes), where UUID is 128 bits long value and X, Y, R are 64 bits double value respectively.

Each typical TCP packet contains a header of at least 20 bytes. It is possible to have 40 bytes extra as options. Therefore, a maximum of 60 bytes of TCP header is possible. Without data aggregation, each update message is sent independently and hence, requires a separate TCP header for each update. The bandwidth efficiency of the update can be calculated by:

$$Size_{data}/(Size_{data} + Size_{header}) \times 100\%$$

Using this formula, the efficiency for single message can be calculated as $40/(20+40)\times100\% = 66.67\%$ at best or $40/(60 + 40) \times 100\% = 40\%$ at worst. On the other hand, with data aggregation of ten update messages, the efficiency would be $(40\times10)/(20+40\times10)\times100\% = 95.24\%$ at best or $40 \times 10/(60 + 40 \times 10) \times 100\% = 86.96\%$ at worst.

Based on the above calculations, we can clearly see that by aggregating many small update messages, a higher bandwidth efficiency can be achieved. This is due to the fact that a predefined header size is required when sending a message to the network using TCP.

5.4 Summary

In this chapter, a simple distributed 2D space shooting game named SPACE SHOOTER is designed and implemented on the MMOP platform. It covers the basic objectives and detailed design of the game. In particular, client GUI, AI, latency compensation, server utilizing MMOP services and bandwidth saving techniques are implemented. Several issues and problems faced during the development and implementation phase, and related design choices are also discussed. In the next chapter, empirical results of game simulation will be reviewed.

Chapter 6

Performance Evaluation

A prototype based on the Massively Multi-user Online Platform (MMOP) middleware design has been developed. This chapter evaluates the performance of the MMOP's innovative Distributed Semaphore (DISEM) services, and the SPACE SHOOTER game implemented in Chapter 5. The aim is to demonstrate the capabilities of MMOP, in an actual networking environment. Furthermore, a traffic generator is implemented in order to generate self-similar background network traffic. The implementation follows the Pareto distribution is incorporated in Section 6.1.3. To better understand the simulation results, the hardware and software used to produce the simulation results are mentioned briefly.

6.1 Testbed Setup and Evaluation Framework

Although all ten machines available in the testbed have identical hardware and software setup, one particular machine is very unstable at high system load. In some cases it shuts down randomly and affects the overall performance result. Therefore, that machine (EPH06) is not included in the actual simulation testbed. A topology of nine machines is formed with the remaining stable machines. During the following performance evaluations, each machine functions as clients, servers as well as routers since the computing resources are limited.

6.1.1 Testbed - Hardware

Each of the testbed machine is equipped with AMD Sempron 2600+, 1 GB RAM and four port 100 Mbps PCI Network Interface Card (NIC) along with one port 1 Gbps onboard network controller. The 1 Gbps link is not included in the simulation to ensure that all simulated traffics are treated equally. Furthermore, to simulate actual Internet environment, the machines are used as routers and each is connected with up to three other machines. Figure 6.1 shows the network setup of these machines.

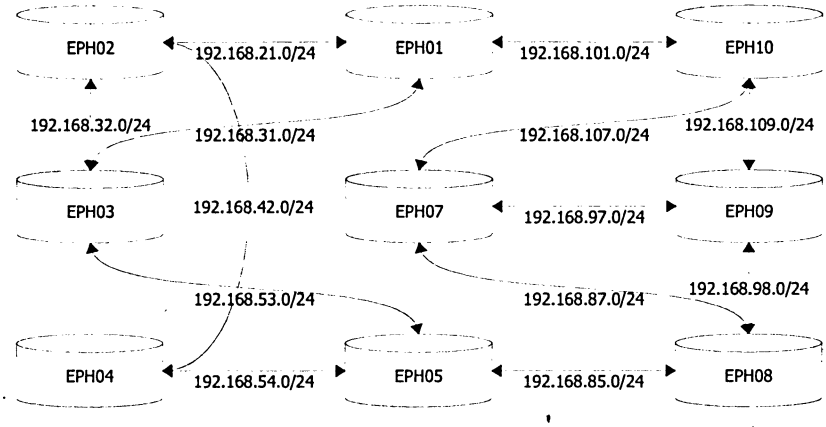


Figure 6.1: The testbed network setup

6.1.2 Testbed - Software

Ubuntu 6.0.6 (Kernel 2.6.15.6) with Java 2 Platform Standard Edition 5.0 update 10 [37] was used on each of the testbed machines; they are the primary software components of our simulation environment. Since our hardware resources are limited, we need to be able to execute multiple processes on a single machine. However, Linux limits the number of file descriptors that any one process may open to 1024 per process. To make the situation worse, both log files and TCP connections are counted towards this value. In order to properly simulate the implementations, the default max number of open files (nofile) is increased to 205454 to prevent undesired execution problems. Additionally, Internet Protocol version 6 (IPv6) has been disabled on all machines to restrict only Internet Protocol version 4 (IPv4)

addressing. This reduces conversion overhead between IPv6 and IPv4 since IPv6 is enabled in Ubuntu by default.

In Figure 6.1 there are nine Linux machines used as routers. This is achieved by enabling IPv4 forwarding and routing on all machines. Originally, the RIP [44] protocol was used, but it failed to generate the correct routing table. Therefore, the OSPFv2 [45] routing protocol is used as a replacement which gives the correct routing table. Both protocols are part of the Quagga [46] routing software suite. All links are preconfigured and have a subnet mask of 24 bits as shown in Figure 6.1. A link between any two machines is in the subnet of 192.168.HL.0/24, where HL indicates a specific link between a machine with higher number (H) and a machine with a lower number (L). The interface of a lower numbered machine will take the address of 192.168.HL.1, whereas a higher numbered machine will take the address of 192.168.HL.2. For example, a link between machine EPH07 (L) and EPH08 (H) has an HL value of 87 and the connected port on EPH07 will have the IP address of 192.168.87.1.

All the prototype components developed for the MMOP framework were installed on all machines, so that they could perform any functionality required. It is important to note that all components are debug builds, and have logging enabled. Although this may degrade performance, it is necessary for evaluation purposes.

6.1.3 Self-similar Traffic Generator

In order to simulate real network traffic a network traffic generator following the Pareto distribution is developed. The traffic generator has basically two parts, client and server. The traffic is directed from client to server so every machine requires both a client and a server. The traffic following the Pareto distribution is calculated as follows:

$$f(x) = \frac{\alpha k^{\alpha}}{x^{\alpha+1}},\ x \geq k, \alpha, k > 0 \tag{6.1}$$

$$E(x) = \frac{k\alpha}{\alpha - 1},\ \alpha > 1 \tag{6.2}$$

$$F(x) = \begin{cases} 1 - \left(\frac{k}{x}\right)^{\alpha}, & x \geq k \\ 0, & x < k \end{cases} \tag{6.3}$$

$$Var(x) = \frac{k^2\alpha}{(\alpha-1)^2(\alpha-2)}, \quad \alpha > 2 \tag{6.4}$$

where α is the shape factor, k is the scale factor, and x is a random number generated between 0 and 1.

Each instance of Pareto distribution uses start time as their seed to ensure the traffic generated does not have repeated pattern. In the traffic generator, Pareto distribution is used for both the file size and thinking time of the traffic generating client. The following equation is derived from the mean values of file size and wait time so the desired traffic loading can be achieved.

$$L_D \times B_{min} = \frac{E_f(x)}{E_w(x)} = \frac{k_f \times \alpha_f/(\alpha_f - 1)}{k_w \times \alpha_w/(\alpha_w - 1)} \tag{6.5}$$

$$k_f = \frac{L_D \times B_{min} \times k_w \times \alpha_w \times (\alpha_f - 1)}{\alpha_f \times (\alpha_w - 1)} \tag{6.6}$$

The parameters for waiting time are $\alpha_w = 1.5, k_w = 0.5$, and for the file size we have $\alpha_f = 1.2$ to represent the normal file downloading and web browsing usages [47]. The user can specify the desired loading L_D, and the minimum link bandwidth B_{min} of the traffic in order to calculate the file size using Eqn. (6.5).

$$k_f = 0.25 L_D \times B_{min} \tag{6.7}$$

Substituting the variables with predefined values, we can then generate the traffic that provides desired traffic loading on a specific link.

6.2 Overlay Protocol Results

The first simulation concentrates on exploring the impact of the overlay protocols on the implemented design. The Distributed Semaphore (DISEM) algorithm is selected since it requires large amount of lookup requests, which puts extra strain on the underlying overlay protocol. More specifically, the DISEM protocol with three replicas is compared on the One Hop Lookups and Chord in this experiment. Figure 6.2 shows the layout of a single application VO configuration with no other secondary VOs.

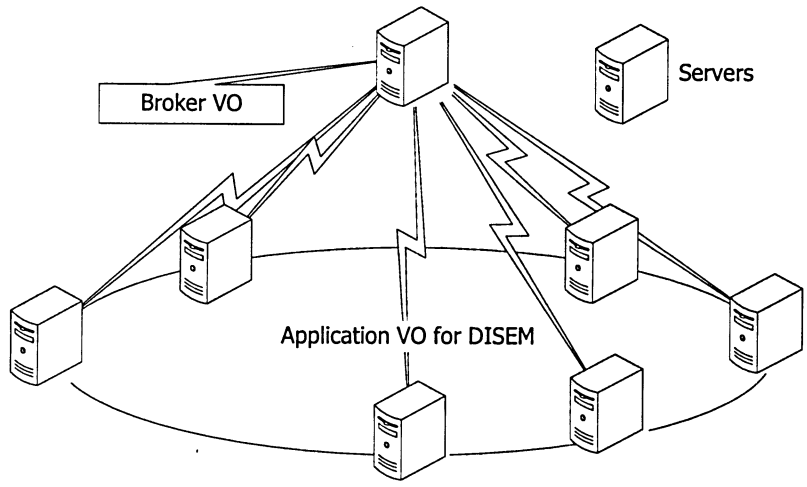


Figure 6.2: Single VO setup for evaluating performance of One Hop Lookups and Chord

This experiment takes a look at the overhead introduced by the overlay network after the overlay is stabilized. In the case of One Hop Lookups, since the total number of simulated nodes is known, the stabilization means that the total node entry size on each node equals to the total number of simulated nodes. On the other hand, Chord's stabilization process takes approaches of periodically refreshing each index within the finger table until no updates need to be made. These criteria are also used for the rest of simulations.

In this particular experiment, the single Broker node requests to update a variable from the DISEM application VO one thousand times every 500 ms. The average latency value is then recorded in Figure 6.3.

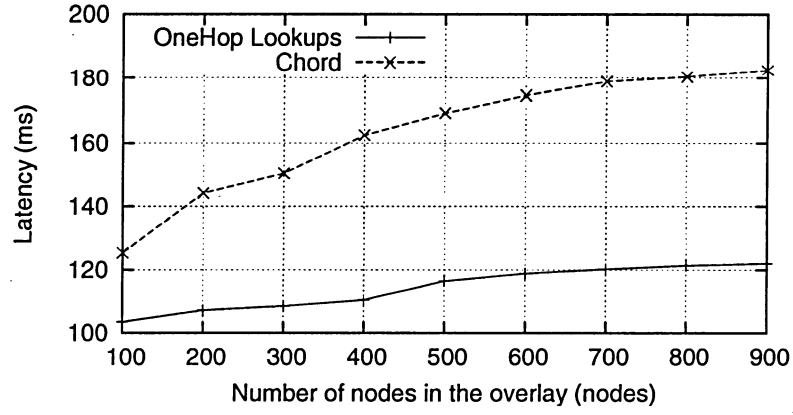


Figure 6.3: Latency of three replicas DISEM on One Hop Lookups and Chord versus number of nodes in the overlay

From the graph, we can observe that One Hop Lookups clearly has an advantage over Chord. This is due to the fact that Chord uses fingers to index the storage nodes, where in the worst case, it may take $\log_2 N$ requests to access the data content. In the case of One Hop Lookups, all the indices are stored locally so actual indexing has shorter latency.

However, One Hop Lookups usually takes longer to stabilize the indices. In some cases it might take up to several minutes for all the updates to be received by every node within the network. On the other hand, Chord overlay only requires around one minute maximum to be able to correctly index a given key. Furthermore, One Hop Lookups has larger memory footprint that increases linearly with the total number of nodes, whereas Chord has a constant memory footprint. Therefore, selecting one over the other is depends on the situation. In the following simulations, One Hop Lookups is selected since we are focusing on achieving the best performance of the implementation.

6.3 Distributed Semaphore (DISEM) Results

First, the Distributed Semaphore (DISEM) protocol has been fully implemented and deployed in a networking testbed described in the previous section. Similar to the single VO

configuration, a single application VO setup is used with no other secondary VOs. Figure 6.4 illustrates the layout. However, unlike previous simulations that only Broker node requests for variable update, all servers within the application VO can request for the variable update. By allowing simultaneous requests, we are able to verify the integrity of the proposed DISEM protocol. More specifically, we verify that the *GetVariableForUpdate* and *WriteUpdatedVariable* provide atomic access for MMOP by allowing only one variable write at any given time.

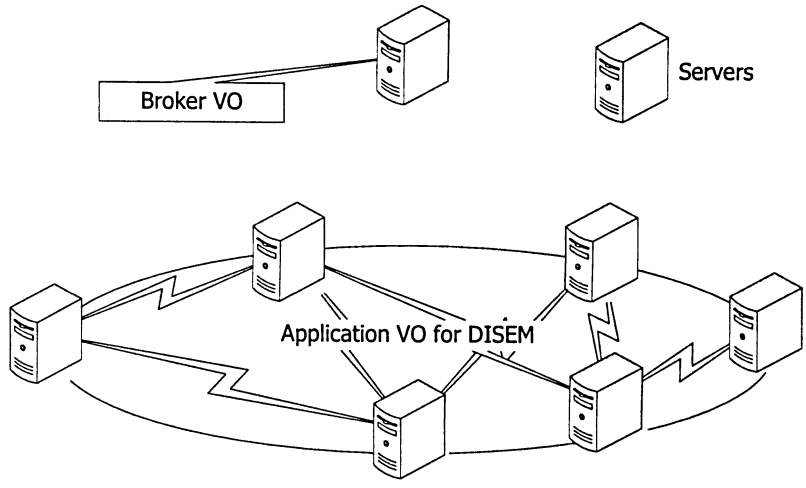


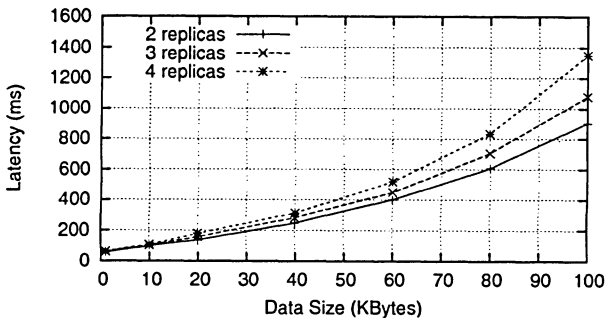
Figure 6.4: Single VO setup for evaluating performance of DISEM

The experiments of the DISEM design are tested under different sizes of the One Hop Lookups overlay algorithm. To focus on the performance aspects of the protocol, the Write Requester (WR) updates the obtained variable, and then exits the critical section immediately. The measured results will be reported in the next section with overlays containing 100 and 900 nodes respectively.

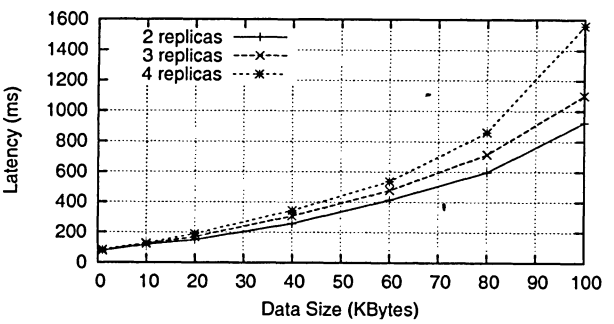
6.3.1 DISEM: Latency versus Data Size

In the first set of experiments, the latency of data retrievals are recorded with the number of total replicas while the data size vary. To see how well the DISEM performs under multiple requests, all nodes within the application VO requests to update the variable at 500 ms

intervals. The latency is calculated by noting the start time and finish time of the update. If the request is rejected, the timer is restarted. Each node performs 100 successful requests and the average latency is recorded in Figure 6.5.



(a) 100 nodes



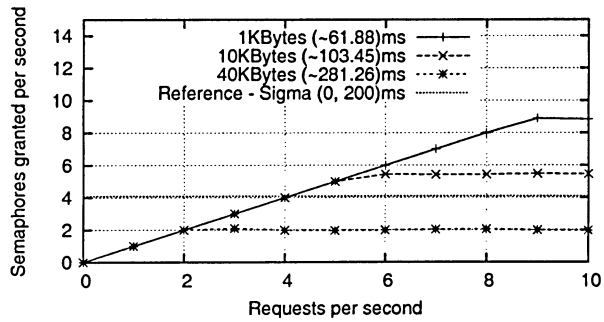
(b) 900 nodes

Figure 6.5: Latency of different number of replicas versus data size

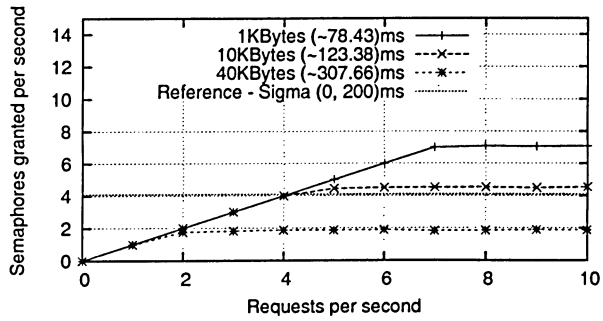
We can observe from Figure 6.5(a) that with small data sizes, the number of replicas does not have a strong impact on the latency. However, as the data size increases, the effect is more observable. This result is also verified in Figure 6.5(b). Moreover, the similarity in both graphs suggest that the overhead added by the number of nodes is constant and does not affect the trend of increasing latency. In conclusion, to provide the highest level of availability for DISEM, the data sizes of variables and the number of replicas should be minimized and maximized, respectively.

6.3.2 DISEM: Throughput

In this section the maximum throughput of the DISEM protocol with three replicas is measured. The throughput is calculated by measuring the average number of semaphores granted per second. Each node requests the same variable at 500 ms intervals 100 times, and again, only the successful requests are recorded. The number of successful requests per second per client is calculated and accumulated to compute the total number of semaphores granted per second shown in Figure 6.6. For comparison purpose, the results from the Sigma protocol [28] are included. It is measured with a total of 32 replicas (24 as majority), where the latency is uniformly distributed between $(0, 200)$ ms. However, the data size of the variable was not specified in the paper.



(a) 100 nodes



(b) 900 nodes

Figure 6.6: Semaphores granted per second versus incoming request rate

We can verify the observation from the previous section that the variable size has significant impact on the DISEM protocol's performance using Figure 6.6. The overall semaphore granted per second increases linearly with the request rate until it reaches a saturation level. Given a fixed size of the variable, the saturation rate $r_{sat}(s)$ can be approximated as

$$r_{sat}(s) \approx \frac{1}{1.8 \times \bar{l}_{var}(s)}, \tag{6.8}$$

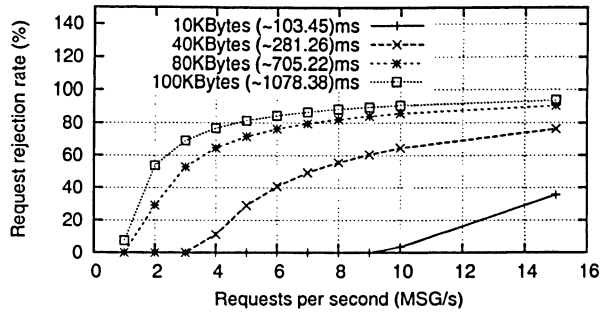
where $\bar{l}_{var}(s)$ is the mean latency to retrieve the variable, and s is the size of the variable. The Eqn. (6.8) is valid for all the performed simulations with variable sizes between 1 KBytes and 100 KBytes.

From Figure 6.6(a), the effect of overlay query overhead on lowering the semaphore granting rate is clearly visible if the data size is 1 KBytes. However, as the size of a data object increased to 40 KBytes, the query overhead can be neglected. Moreover, comparing Figure 6.6(b) with Figure 6.6(a) suggests that DISEM protocol's performance depends on the size of the VO. Since the lookup latency of One Hop Lookups protocol increases as the size of the VO increases, it is more efficient to have a smaller VO to obtain a faster response time.

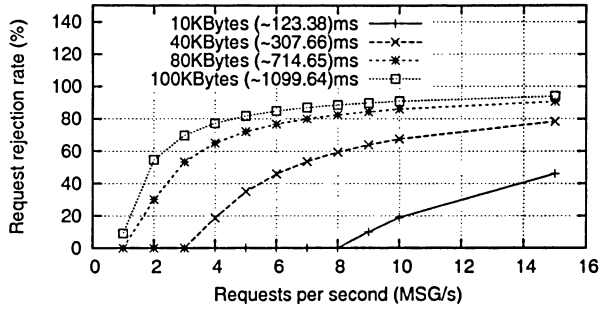
6.3.3 DISEM: Request Rejection Rate

In this set of simulations, the request rejection rate of three replicas DISEM is evaluated. The Broker machine requests for the same variable at a given rate of request per second (MSG/s) and records the $R_{rejected}/T_{requests}$ ratio, where $R_{rejected}$ is number of rejected requests and $T_{requests}$ is the total number of requests. In the following experiment, $T_{requests} = 1000$ is used and the rejection rate is plotted in the Figure 6.7.

From Figure 6.7, we can observe that larger data sizes result in higher request rejection rates. This result is expected due to the longer data transmission times for larger variable data. This also implies that the semaphore requires a longer data updating period before it is released. Comparing the figures for 100 nodes and 900 nodes reveals that the request rejection ratios among different data sizes have similar impact on both settings. Therefore



(a) 100 nodes



(b) 900 nodes

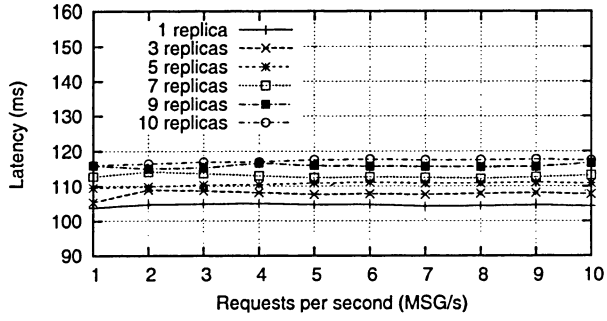
Figure 6.7: Request rejection rate with variable data size versus incoming request rate

we can conclude that the size of the VO does not have strong impact on the rejection rate of the requests. From the above observations, we recommend to use only small variable data sizes in actual practice to minimize the variable accessing time.

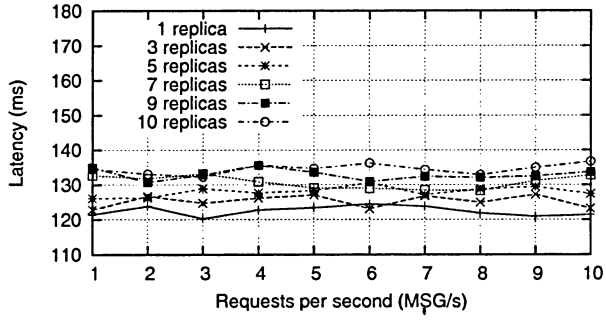
6.3.4 DISEM: Number of replicas

In the next set of experiments, the number of replicas are adjusted from 1 to 10 in steps of two. This demonstrates how the number of replicas affects the variable retrieval latency of DISEM protocol. A constant data size 10 KBytes is used for this particular simulation. Each machine in the overlay generates the required number of requests per second using the Poisson distribution. A total of 1000 successful requests per node were conducted to obtain

the average variable retrieval latency as shown in Figure 6.8.



(a) 100 nodes



(b) 900 nodes

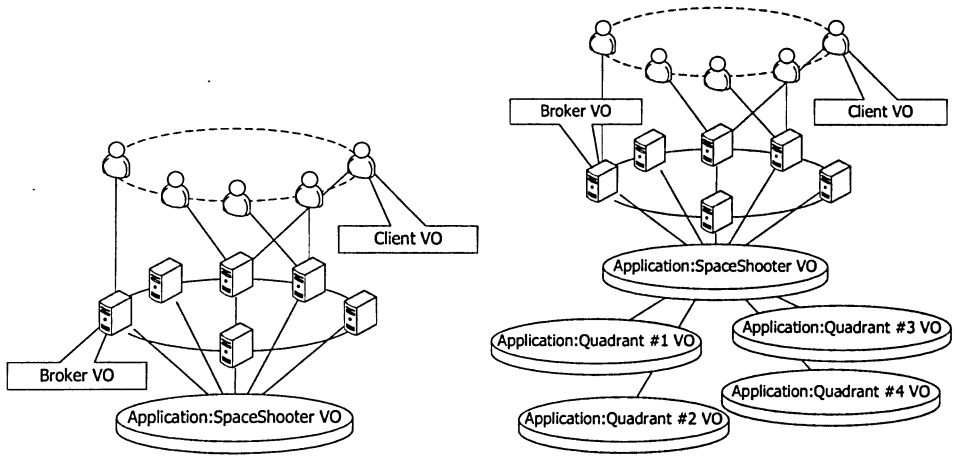
Figure 6.8: Latency with different number of replicas versus incoming request rate

From the results plotted in Figure 6.8, we notice that the cost of adding extra replicas does not have noticeable impact on the access latency. This is mainly due to the fact that DISEM utilizes parallel requests to minimize delays. This simulation also demonstrates DISEM's ability to increase variable availability by increasing the total number of replicas. Additional replicas can be introduced to reduce the effect of VO membership changes as well as hardware failures. In the worst case, if the majority of replicas are missing, then no one should be allowed to obtain the variable lock. The data refreshing service needs to repair enough replicas before the next semaphore can be granted. Study has shown that nodes with a P2P network are usually online for about $2.\dot{7}$ hours [48]. Therefore, rapid membership can be ignored for the time being although carefully selecting the servers should improve the

availability of the data.

6.4 Distributed 2D Space Shooting Game Results

In this section, the correctness of the implemented Massively Multi-user Online Platform (MMOP) service is verified with the implementation of the distributed 2D shooting game designed in Chapter 5. The developed game GUI is used to monitor the progress of each simulated game. In the following simulations, three types of VO configurations are evaluated. Two of the three VO configurations are illustrated in the following figures:



(a) Single VO Setup (b) Four VO Setup

Figure 6.9: Application VO setup for MMOP simulations

The first VO setup places all four game quadrant servers within a single application VO. This setup is close to the cluster computing setup, where all the servers are located within a single administrative domain as in Figure 6.9(a). The second VO setup requires two tiers of application VOs to be created and places quadrant servers on four different subsidiary application VOs as shown in Figure 6.9(b). Similar to the second VO setup, the third VO configuration further divides each of the four quadrant VOs into four smaller VOs, resulting in a sixteen VO configuration.

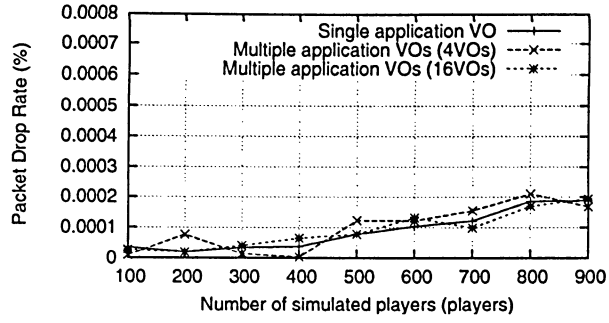
Due to the lack of an automatic application scheduler, the server applications are deployed directly using a script file which is included in Appendix C. The script also assigns a total of nine Broker machines on each of the testbed machines. The traffic between simulated clients and game servers are then be distributed between these Broker machines. The total number of simulated clients varies from 100 to 900 and the results are compared for the three different VO setups described above.

The traffic generator is also used and deployed along with the MMOP, where a traffic generator server and a client is installed on each machine. For evaluating the performance under different traffic loadings, two traffic loadings are measured. We have selected 10% traffic loading as our lightly loaded network model and 60% traffic loading as heavily loaded network model. The reason that no more than than 60% bandwidth utilization is used is because the network interfaces on the testbed machines stop working randomly. A restart of the machine is usually required when this happens. The cause of this is yet to be determined since the packet drop rate, packet error rate, total number of connections and memory usage all appear to be normal.

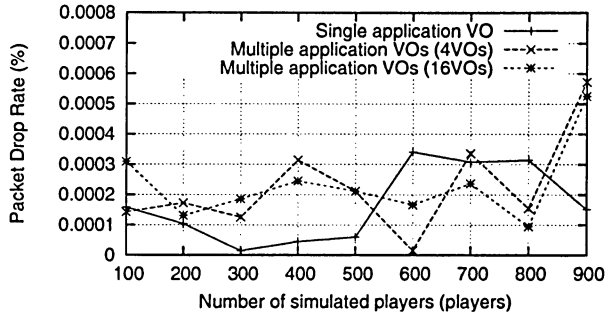
6.4.1 MMOP: Packet Drop Rate

In the first set of simulations, the packet drop rate of clients is measured. A single machine in the testbed hosts several simulated clients, Broker nodes as well as application servers. The packet drop rate, calculated as $Packet_{dropped}/Packet_{total} \times 100\%$ represents how well the MMOP utilizes the available bandwidth. Since each machine has three interfaces that are connected to other machines, the data is gathered at the end of each simulation session, and then the average drop rate is calculated. Simulation results for different VO setups versus the total number of simulated players are plotted in Figure 6.10.

From Figure 6.10 we can observe a general trend of increasing packet drop rate as more simulated players are created. Throughout the simulations, the packet drop rate is in the order of 10^6 for both traffic loadings. This indicates an acceptable level of bandwidth utilization for the current MMOP and SPACE SHOOTER game implementation. In Figure 6.10(b), we can see a more irregular packet drop pattern throughout the simulation; this irregularity is



(a) 10% traffic loading



(b) 60% traffic loading

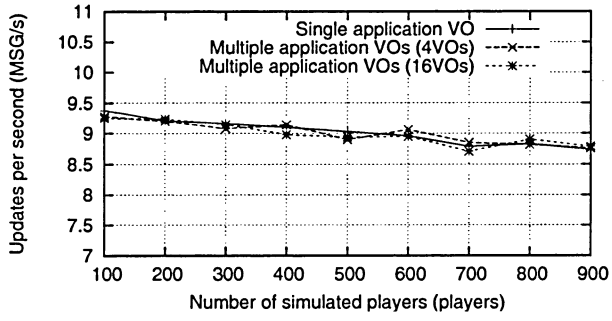
Figure 6.10: Packet drop rate with different VO setups versus number of simulated players

not observed in Figure 6.10(a). Further investigations reveal that this irregularity is a result of higher level of traffic loading. Since 10% traffic loading is a much lower setting compared to 60%, the bursty traffic generated by the traffic generator has a more prominent effect on the packet drop rate. However, no significant visual differences from the simulations can be observed even though these two sets of simulation have different packet drop rate patterns.

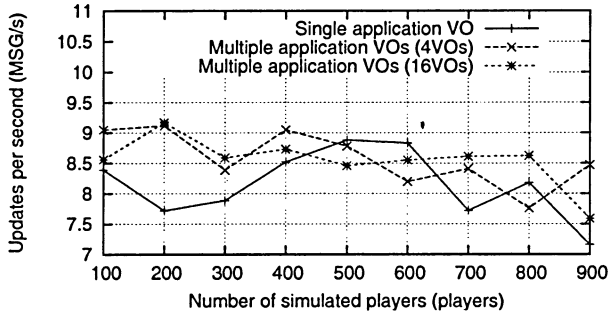
6.4.2 MMOP: Updates per Second

In this section, the number of updates per second for each client is recorded. In general, a higher number of updates per second results in a higher accuracy in entity rendering. However, a higher number of updates would also require more computational resources.

Therefore, with our restricted resources, we limit the server update message generation time to be 100 ms, which results in at most 10 updates per second. The reason that less than 10 updates are expected is because it takes some time to generate the update messages on the server. The average updates per second received by clients versus the number of simulated players is plotted in Figure 6.11.



(a) 10% traffic loading



(b) 60% traffic loading

Figure 6.11: Update per second versus number of simulated players

As the total number of simulated clients increases, less computational resources are available to the quadrant server. Figure 6.11(a) shows the reduction in the number of update messages received per second as expected. However, in Figure 6.11(b), we can see the background traffic becoming the dominating factor in the simulation results. A visible difference in the graph indicates that the received updates per second are lower when the traffic loading is higher. Furthermore, due to the bursty nature of the produced traffic, the plotted graph in

Figure 6.11(b) indicates a more random set of received updates per second when compared to Figure 6.11(a).

Similar to the previous simulation, no visible glitch was noticed when monitoring the game using the game GUI. A steady 30 FPS was observed; hence we conclude that the variance between measured values does not effect the gaming experience.

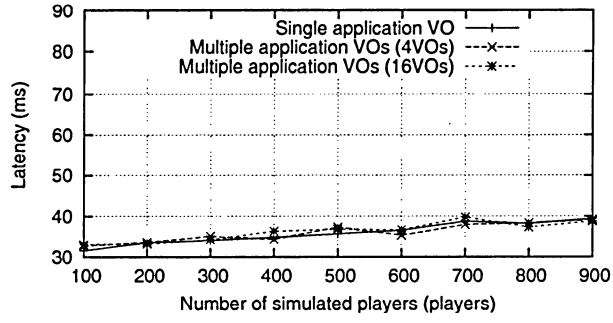
6.4.3 MMOP: Latency

Latency is a measure used in most online games to indicate the gaming quality. Higher latency indicates that it takes longer for the server updates to travel to the client, which means that the game appears less responsive to player's commands. In this section we measure the effect of VO configurations and total number of players on the latency of the game. Since the client times are synchronized by the MMOP distributed timing service, the latency can be calculated by subtracting the current client time from the server's time stamp included in each update message. In Figure 6.12, the average latency for the clients are plotted.

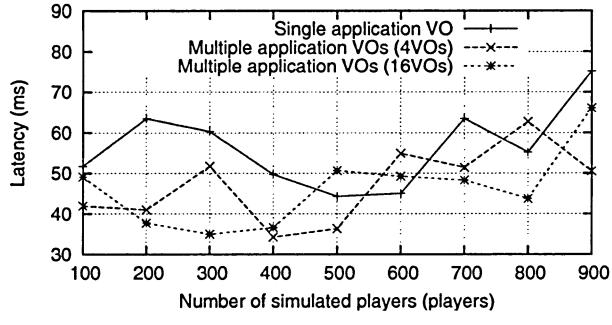
Figure 6.12(a) indicates that the latency increases as the number of simulated players increases. This is expected as more computational resources are required for each game cycle. However, it represents a very slow increasing trend compared to the total number of players. This signifies the ability for MMOP to handle a large number of simulated players even though the simulation resources are limited. Moreover, the latency is masked by the interpolation technique implemented. The interpolation time of 100 ms ensures there are no visual differences between each simulation scenario, even when the latency is around 70 ms in the worst case as shown in Figure 6.12(b).

6.4.4 MMOP: Total simulation time

Last, the total simulation time of the game is measured. This simulation is used to determine if the design of the game AI functions correctly. Furthermore, the simulation challenges the prototype MMOP design to see if it is able to sustain a large number of membership changes



(a) 10% traffic loading

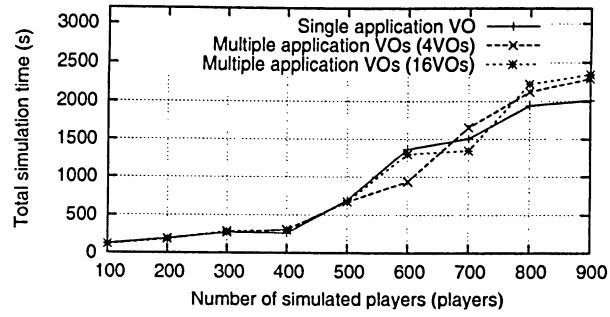


(b) 60% traffic loading

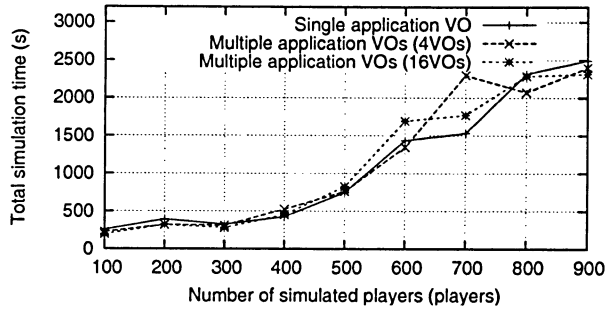
Figure 6.12: Average latency versus number of simulated players

(simulated clients joining/leaving the client VO). The timer starts when the quadrant servers are setup and ready for simulation. When the simulation begins, the simulated players start joining the client VO and compete with each other in the game. If the simulated aircraft is destroyed, the client will disconnect from the server hence leaving the client VO. The timer stops when one or no more players remain in the game.

The result indicates that all simulations complete normally without errors. Therefore, it is safe to say that the AI designed is more or less correct. However, when observing from the game GUI, we found that the simulated players appear to turn left and right excessively instead of aiming directly at the target. This might be due to a combination of several factors, such as predefined angle, latency, and value rounding etc., Unfortunately, AI design is not really part of our research focus. However, further investigation into this area will



(a) 10% traffic loading



(b) 60% traffic loading

Figure 6.13: Total simulation time with different VO setup versus number of simulated players

greatly improve the gaming experience for the players.

As we can observe from Figure 6.13(a), the difference between total execution time of each VO setup as well as traffic loading is minimal. This is an indication that the network traffic and VO configuration do not have a huge impact in terms of actual game performance. Although, there are minor differences at 700 players for Figure 6.13(b), it still follows the general trend of the graph.

6.5 Summary

The Massively Multi-user Online Platform (MMOP) framework's prototype has been reviewed and evaluated. In particular, the integrity of Distributed Semaphore (DISEM) services is verified to provide atomic data access through the means of parallel data access. We also observed that in DISEM, data variable with smaller sizes can offer better performance. The protocol has significant performance improvement when comparing to the Sigma protocol even though the data size of Sigma protocol is not defined. Furthermore, we demonstrated that increasing the number of replicas does not have major impact on the variable retrieval latency with DISEM protocol. As a result, the overall data access availability for the DISEM protocol is increased.

Moreover, the SPACE SHOOTER game described in Chapter 5 is used to demonstrate the capabilities of MMOP. In order to simulate real world traffic, a traffic generator is implemented following the Pareto distribution. Traffic loads are varied during the game simulation to discover the effect of high background network traffic. The simulations show promising results such as flexible VO configuration and low latency variance when a large number of clients are considered. These results also demonstrate that the MMOP is suitable for developing large-scale distributed applications such as MMOGs.

Chapter 7

Conclusion

A lightweight middleware based on the Peer-to-Peer (P2P) structured overlay network designed for large-scale online application, the Massively Multi-user Online Platform (MMOP), has been proposed and implemented in this thesis. The Grid architecture of the MMOP maintains the heterogeneous resources dynamically through the use of P2P overlay algorithms, which inherently provides flexibility and robustness to the middleware. Moreover, application-level QoS policy is enforced through means of QoS management services. A simple set of Application Programming Interfaces (APIs) are designed for managing and utilizing these resources. Main features supported by MMOP are hierarchical Virtual Organization (VO) resource organization, atomic data access, QoS monitoring, and distributed time synchronization. These features have been designed in the thesis and several components were implemented to produce a funtional MMOP prototype. Implementation work carried out includes:

1) the design and implementation of the virtual organization (VO) construction service which includes two fully modularized overlay algorithms has been presented. A highly scalable hierarchical VO structure can be constructed with a simple API. Moreover, the Broker VO provides communication relay between the Client VO and other Secondary VOs.

2) highly scalable and configurable Distributed Semaphore (DISEM) atomic data access service has been developed and fully implemented. The service provides critical code section protection as well as several read operations for satisfying different real-time requirements when developing distributed applications.

3) the application deployment service allows the modules to be deployed on-the-fly without service interruptions. A simple API has been developed to allow direct service deployment. Application based policy has also been designed and partially implemented.

4) network monitoring modules and application level QoS provisioning are provided by the design of QoS management service. Such a service provides monitoring of data aggregation and provisioning. It works in conjunction with the application deployment services to ensure the QoS requirements of the application are met.

5) an accurate and synchronized distributed timing service has been implemented.

A simple distributed 2D shooting game called SPACE SHOOTER is designed and implemented using the prototype MMOP services. The game client includes a Graphical User Interface (GUI) front-end, which is optimized with several client side latency compensation techniques such as interpolation and dead reckoning. The client also comes with an Artificial Intelligence (AI) component which is used for simulation purpose. The game server utilizes every aspect of the prototype MMOP, as well as bandwidth conservation techniques, such as areas of interest management (AOIM) filtering and data aggregation.

The MMOP prototype has been subjected to performance evaluation tests, and the results for the DISEM and the distributed 2D shooting game verify the performance and functionality of the MMOP. This clearly illustrates the potential of using MMOP to develop large-scale online applications.

7.1 Future work

The current prototype implementation of MMOP provides the basic functionality such as resource organization and application deployment for large-scale online applications. However, when developing commercial grade products, there remains many areas in need of improvement. Here we list a few directions that can be explored to improve the functionality of MMOP.

Dynamic Application Scheduler Currently, only a simple command based deployment script is provided with MMOP. It works fine when the application only needs to be deployed once. However, many complex scientific applications require unattended application deployment and even redeployments for different sets of data. Therefore, an application scheduler can alleviate this problem by automatically scheduling tasks based on the available resources. Furthermore, optimization can also be implemented to provide better resource utilization. A scheduler similar to the Grid Resource Management service (GRAM) from Globus Toolkit 4 should be implemented to improve the usability of MMOP.

Security Security is a critical issue for P2P Grid computing because large-scale online applications are just as vulnerable to hackers as any other Internet services. Communications between machines must be able to support the highest standard encryption for commercial products. For an online revenue generating service such as a trading system, this is necessary for protecting both client and the company through a set of authentication rules. Furthermore, security in terms of service access must be reinforced to prevent illegal service access. For example, it must be impossible for an user to access a service if they do not have the correct permission. Security measures such as X.509 credentials can be incorporated into the middleware to natively support the public key infrastructure.

NAT Traversal Due to availability of broadband connections, routers have become very popular because of their ability to share one single Internet connection with several machines within a house. Clients connected to the Internet through routers require Network Address Translation (NAT) service which enables multiple hosts on a private network to access the Internet using a single public IP address. However, NAT service also prevents connection to be initiated from outside of the private network, or causing disruption in stateless protocols such as UDP. To resolve this problem, the router has to be manually configured to allow outside traffic to pass through. However, it is time consuming and difficult for an inexperienced user to perform such task. To increase the usability of MMOP, automatic NAT configuration should be included. Universal Plug and Play (UPnP) comes with a solution for network address translation traversal

which can be implemented to resolve this problem.

Multi-Language Support The MMOP middleware is written in Java since it offers advantages in terms of defect count and development time. In particular, the automatic garbage collection eliminates the memory management errors in C/C++ development. Moreover, Java can be ported easily from one platform to another with no additional modification, given that the target platform has a compatible Java Runtime Environment. Java also provides a Java Native Interface (JNI) framework to interact with native applications and libraries that are written in other languages, which should be explored when multi-language support is considered.

Dynamic QoS Support Current MMOP's application deployment service only provides simple application QoS provisioning. To provide consistent QoS for each service, we should support real-time QoS policy modifications to meet rapidly changing real-time traffic and process execution conditions. Furthermore, new metrics should be added to QoS policies to meet the needs of dynamic load-balancing. Notable metrics that could potentially improve QoS includes the geographic location of the machine and available time frame for each resources.

Appendix A

Massively Multi-user Online Platform (MMOP) Deployment Script

A sample script (deploymmop) for deploying MMOP middleware is included below. The script checks out the latest MMOP implementation from the CVS and compiles the class file specified by the `CLASS` variable. The sample below compiles the `VOManagerImpl.java` file and distributed the compiled codes to all the machines specified in the `host` file.

Listing A.1: Sample `deploymmop` script

```
#!/bin/sh
clear
FOLDER=~/workspace/
SOURCE=MMOP
CLASS=ca/ryerson/mmop/services/vomanaging/VOManagerImpl.java
CLASSPATH=mmop
if [ -z $1 ]; then
        echo "Please provide a host file"
elif [ ! -e $1 ]; then
        echo "$1 file does not exist"
else
        #set echo off
        cd $FOLDER
        rm -rf $SOURCE
        cvs -Q checkout $SOURCE

        cd $SOURCE
        javac $CLASSPATH $ $CLASS

        echo "... Updating $SOURCE on all machines ..."
        declare -a array1
        array1=(`cat ~/$1`)
        for EPHS in ${array1[@]}
        do
                ssh $EPHS "rm -rf $FOLDER/$SOURCE;exit;"
                scp -q -r $FOLDER/$SOURCE $EPHS:$FOLDER/.
                echo "... Updated $SOURCE on $EPHS ..."
        done
        echo "... Done ..."
        #set echo on
fi
```

The script requires a single parameter, a file name which contains a list of machines that the compiled code will be deployed, a sample host file used for the simulation is included in Listing A.2.

Listing A.2: Content of the sample `host` file

```
192.168.21.1
192.168.32.1
192.168.32.2
192.168.54.1
192.168.85.1
192.168.97.1
192.168.98.2
192.168.109.2
```

The execution result of the `deploymmop` script with the above `host` file is listed below.

Listing A.3: Execution result of the installmmop script

```
allen@EPH08:~$ ./deploymmop hosts
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
... Updating MMOP on all machines ...
... Updated MMOP on 192.168.21.1 ...
... Updated MMOP on 192.168.32.1 ...
... Updated MMOP on 192.168.32.2 ...
... Updated MMOP on 192.168.54.1 ...
... Updated MMOP on 192.168.85.1 ...
... Updated MMOP on 192.168.97.1 ...
... Updated MMOP on 192.168.98.2 ...
... Updated MMOP on 192.168.109.2 ...
... Done ...
```

The `deploymmop` script can also be modified to deploy other application(s) with the MMOP middleware. This is done by adding check out and compile commands for the application(s) at Line 16 and Line 19 of the `deploymmop` script respectively. By adding these commands, a complete simulation package can be deployed on each machine specified by the `host` file.

Appendix B

Setting up Hierarchical VO

To better demonstrate the construction of a hierarchical VO, a simple program is written to accept commands from keyboard and invoke the VO construction service. Such program takes two commands, `connect` and `list`. The `connect` command calls the *JoinVO* command from MMOP API to construct the corresponding VO. It requires two parameters, a VO ID string and a number, separated by a space. The number is used to indicate the type of VO algorithm used for constructing the VO, valid values are 1 and 2 which refer to One Hop Lookups and Chord respectively. The `list` command is used to view the constructed VOs in current system. In the following, a VO hierarchy as illustrated in Figure B.1 is constructed on three machines running the program. These machines are referred to as S1, S2, and S3 respectively in the demo.

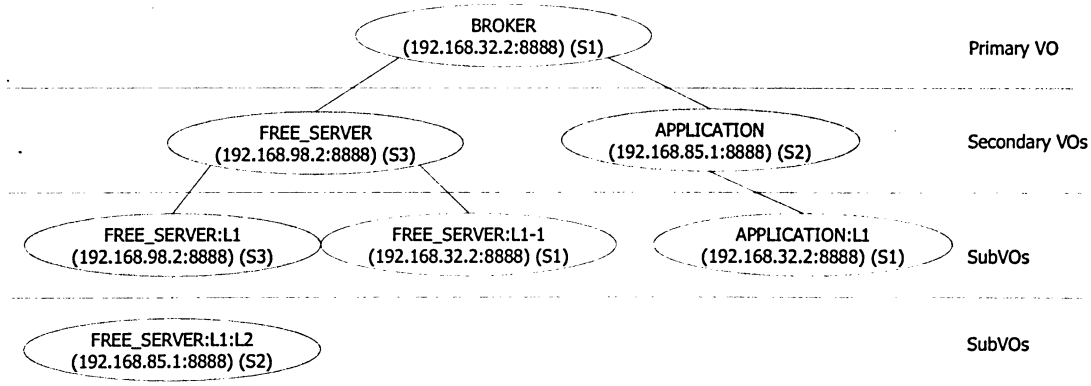


Figure B.1: VO hierarchy setup demo

The commands used on each machine to create the VO hierarchy are executed in order in Listing B.1.

Listing B.1: Commands executed on each machine to create VO hierarchy

```
S1:     connect BROKER 1
```

```
S2:     connect APPLICATION 1

S3:     connect FREE_SERVER 2

S1:     connect APPLICATION:L1 1
        connect FREE_SERVER:L1-1 2

S3:     connect FREE_SERVER:L1 2

S2:     connect FREE_SERVER:L1:L2 2
```

After the `connect` commands are executed successfully on each machine, the `list` command is issued and the result on each machine are recorded. To reduce the size of the `list` outputs, the finger table output of the Chord algorithm is omitted.

Listing B.2: list command on 192.168.32.2:8888 (S1)

```
20 Local Address: 192.168.32.2:8888
20 VO: BROKER
20 Predecessor: 192.168.32.2:8888 Successor: 192.168.32.2:8888
20 SliceLeader: 192.168.32.2:8888 UnitLeader: 192.168.32.2:8888
20 KEY: 26350676379183373389226178831449644399 8 Client @ 192.168.32.2:8888
20 VO: APPLICATION:L1
20 Predecessor: 192.168.32.2:8888 Successor: 192.168.32.2:8888
20 SliceLeader: 192.168.32.2:8888 UnitLeader: 192.168.32.2:8888
20 KEY: 26350676379183373389226178831449644399 8 Client @ 192.168.32.2:8888
20 VO: FREE_SERVER:L1-1
20 Predecessor: 192.168.32.2:8888 Successor: 192.168.32.2:8888
[.... 160 finger table entries ....]
20 KEY: 26350676379183373389226178831449644399 8 Client @ 192.168.32.2:8888
20 SubVO Mappings:
20 KEY: APPLICATION Contacts: 192.168.85.1:8888 192.168.85.1:8888 \\
 192.168.85.1:8888 192.168.85.1:8888
20 KEY: FREE_SERVER Contacts: 192.168.98.2:8888 192.168.98.2:8888 \\
 192.168.98.2:8888 192.168.98.2:8888
```

Listing B.3: list command on 192.168.85.1:8888 (S2)

```
18 Local Address: 192.168.85.1:8888
18 VO: APPLICATION
18 Predecessor: 192.168.85.1:8888 Successor: 192.168.85.1:8888
18 SliceLeader: 192.168.85.1:8888 UnitLeader: 192.168.85.1:8888
18 KEY: 140506169650137150276475491608065180404 Client @ 192.168.85.1:8888
18 VO: FREE_SERVER:L1:L2
18 Predecessor: 192.168.85.1:8888 Successor: 192.168.85.1:8888
[.... 160 finger table entries ....]
18 KEY: 140506169650137150276475491608065180404 Client @ 192.168.85.1:8888
18 SubVO Mappings:
18 KEY: APPLICATION:L1 Contacts: 192.168.32.2:8888 192.168.32.2:8888 \\
 192.168.32.2:8888 192.168.32.2:8888
```

Listing B.4: list command on 192.168.98.2:8888 (S3)

```
24 Local Address: 192.168.98.2:8888
24 VO: FREE_SERVER
24 Predecessor: 192.168.98.2:8888 Successor: 192.168.98.2:8888
```

```
[.... 160 finger table entries ....]
24 KEY: 103844790955283766714130393609180068046 Client @ 192.168.98.2:8888
24 VO: FREE_SERVER:L1
24 Predecessor: 192.168.98.2:8888 Successor: 192.168.98.2:8888
[.... 160 finger table entries ....]
24 KEY: 103844790955283766714130393609180068046 Client @ 192.168.98.2:8888
24 SubVO Mappings:
24 KEY: FREE_SERVER:L1 Contacts: 192.168.98.2:8888 192.168.98.2:8888 \\
 192.168.98.2:8888 192.168.98.2:8888
24 KEY: FREE_SERVER:L1-1 Contacts: 192.168.32.2:8888 192.168.32.2:8888 \\
 192.168.32.2:8888 192.168.32.2:8888
24 KEY: FREE_SERVER:L1:L2 Contacts: 192.168.85.1:8888 192.168.85.1:8888 \\
 192.168.85.1:8888 192.168.85.1:8888
```

Appendix C

Script for Space Shooter Simulation on Testbed

To perform game simulation on the testbed, the `deploymmop` script needs to be modified to include the check out and compilation of the game simulation code. When the simulation codes are in place, the following script is used to start the simulation servers. Two host files are required for this particular script: The first host file is the `brokerhosts`, it contains the machines that should be in the *BROKER* VO. The second host file is the `serverhosts`, it contains the machines that should be in the *FREE_SERVER* VO, where the simulation server can be deployed.

Listing C.1: Simulation server script

```
1  #!/bin/sh
2  FOLDER=~/workspace/MMOP
3  VONAME="APPLICATION:Quadrant #"
4  SERVER="192.168.85.2"
5  CLASSPATH=../mmop:../spaceshooterV2
6  if [[ -z $1 || -z $2 ]]; then
7          echo "Please provide two host files"
8  elif [[ ! -e $1 || ! -e $2 ]]; then
9          echo "$1 or $2 file does not exist"
10 else
11         #set echo off
12         echo "... Starting BROKER servers ..."
13
14         APP="ca.ryerson.mmop.spaceshooter.v2.server.Broker"
15         java -classpath $CLASSPATH $APP $SERVER
16         echo "... BROKER Server started on $SERVER ..."
17
18         array1=(`cat ~/$1`)
19         for EPHS in ${array1[@]}
20         do
21                 ssh -n 'cd $FOLDER;java StartApp java -classpath \\
22                 $CLASSPATH $APP $SERVER $EPHS;exit;'
23                 echo "... BROKER Server started on $EPHS ..."
24         done
25
26         echo "... Deploying Application ..."
27         APP="ca.ryerson.mmop.spaceshooter.v2.server.Server"
28         VONumber=1
29         DEFAULT="APPLICATION:SpaceShooter"
30         VOS="$((${#array1[@]} + 1))"
31
```

```
32          echo "... Server in $DEFAULT VO started on $SERVER ..."
33          java -classpath $CLASSPATH $APP $SERVER $DEFAULT $VOS
34
35          declare -a array1
36          array1=(`cat ~/$2`)
37          for EPHS in ${array1[@]}
38          do
39                  ssh -n 'cd $FOLDER;java StartApp java -classpath \\
40                  $CLASSPATH $APP $SERVER $EPHS "$VONAME$VONumber" $VOS;exit;'
41                  echo "... Server on $VONAME$VONumber started on $EPHS ..."
42                  VONumber=`expr $VONumber + 1`
43          done
44          echo "... Done ..."
45          #set echo on
46  fi
```

Two critical problems were encountered while developing the simulation script. First, the SSH connection terminates all processes associated with it when the connection terminates. Therefore, it is impossible to start several processes on different machines using a single script since only one active SSH session can be managed by the script. To resolve this issue, a simple program is written to start the target process on behalf of the SSH connection. Since the process is started by the program, it will not be terminated when the SSH connection is closed. In Listing C.1, the `StartApp` in Line 39 provides such process invocation of the actual simulation process.

The second problem encountered involves a bug in SSH such that it sometimes wait for the channel terminating signal even when an `exit` command is included. Which renders the script useless since user input is required after each exit command. Adding the -n switch prevents SSH from waiting for the standard input and terminates quietly after each `exit` command so several service can be started one after another with one script.

Using the script in Listing C.1, a *APPLICATION* : *SpaceShooter* VO is always started before all the other subsidiary VOs. This VO may or may not be used for actual simulation, depending on the number of the subsidiary VO created. For example, if a `serverhosts` file such as the one in Listing C.1 is used, then four VOs will be created and *APPLICATION* : *SpaceShooter* will not be used for simulation.

Listing C.2: Content of the `serverhosts` file

```
192.168.21.1
192.168.32.2
192.168.85.1
192.168.98.2
```

With the servers started, the following script is used to start the clients. Similar to the other scripts, a host file is required to specify which machine will be used in the simulation. To adjust the total number of simulated players in the game, the `PARAM` value can be adjusted

accordingly. The total number of simulated clients on the MMOP can be calculated by $TotalHosts \times PARAM$, where $TotalHosts$ is the total number of machines in the host file.

Listing C.3: Simulation client script

```
#!/bin/sh
FOLDER="/workspace/MMOP/testing
APP="ca.ryerson.mmop.spaceshooter.v2.simclient.ClientSimulator "
PARAM=100
CLASSPATH=../mmop:../spaceshooterV2
if [ -z $1 ]; then
        echo "Please provide a host file"
elif [ ! -e $1 ]; then
        echo "$1 file does not exist"
else
        #set echo off
        echo "... Starting clients with $PARAM ..."
        declare -a array1
        array1=( 'cat ~/$1')
        for EPHS in ${array1[@]}
        do
                ssh -n 'cd $FOLDER;java StartApp java -classpath \\
                $CLASSPATH $APP $EPHS $PARAM;exit;'
                echo "... Client started on $EPHS ..."
        done
        echo "... Done ..."
        #set echo on
fi
```

References

[1] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, November 2002.

[2] Ian T. Foster. Globus toolkit version 4: Software for service-oriented systems. In *NPC*, volume 3779 of *Lecture Notes in Computer Science*, pages 2–13. Springer, 2005.

[3] W. Polk D. Solo R. Housley, W. Ford. Us secure hash algorithm 1 (sha1). RFC 2459, IETF, January 1999.

[4] Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. One hop lookups for peer-to-peer overlays. In *Ninth Workshop on Hot Topics in Operating Systems (HotOS-IX)*, pages 7–12, Lihue, Hawaii, May 2003.

[5] Emma Brunskill. Building peer-to-peer systems with chord, a distributed lookup service. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 81, Washington, DC, USA, 2001. IEEE Computer Society.

[6] Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM 2001*, San Diego, CA, September 2001.

[7] 3rd D. Eastlake and P. Jones. Us secure hash algorithm 1 (sha1). RFC 3174, IETF, September 2001.

[8] David Karger, Eric Lehman, Tom Leighton, Mathhew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[9] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, February 2003.

[10] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: High-bandwidth content distribution in a cooperative environment. In *IPTPS'03*, February 2003.

[11] M. Carlson E. Davies Z. Wang W. Weiss S. Blake, D. Black. An architecture for differentiated services. RFC 2475, IETF, December 1998.

[12] Inc. Blizzard Entertainment. World of warcraft. worldofwarcraft.com.

[13] NCosft Corporation. Lineage. http://www.ncsoft.com.

[14] NCosft Corporation. Lineage ii. http://www.ncsoft.com.

[15] Bruce Sterling Woodcock. An analysis of mmog subscription growth - version 21.0. http://www.mmogchart.com.

[16] IDC. Asia/pacific online gaming report. http://www.idc.com, 2006.

[17] Michi Henning. Massively multiplayer middleware. *Queue*, 1(10):38–45, 2004.

[18] Butterfly.net Inc. The butterfly grid: Powering next-generation gaming with on-demand computing, 2003.

[19] BigWorld Pty Ltd. Bigworld technology.

[20] Bram Cohen. Incentives build robustness in bittorrent, 2003. [Online; accessed 10-Dec-2006].

[21] Bram Cohen. Bit torrent. http://www.bittorrent.com.

[22] Alberto-Laszlo Barabasi. *Linked: The New Science of Networks.* Perseus Books Group, 2002.

[23] Tamer M. Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems (2nd Edition).* Prentice Hall, January 1999.

[24] Yatin Chawathe, Sriram Ramabhadran, Sylvia Ratnasamy, Anthony LaMarca, Scott Shenker, and Joseph Hellerstein. A case study in building layered dht applications. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 97–108, New York, NY, USA, 2005. ACM Press.

[25] N. Lynch and A. Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of the 16th International Symposium on Distributed Computing*, pages 173–190, 2002.

[26] Seth Gilbert, Nancy Lynch, and Alex Shvartsman. Rambo ii: Rapidly reconfigurable atomic memory for dynamic networks. *dsn*, 00:259, 2003.

[27] Leslie Lamport. Paxos made simple, fast, and byzantine. In *OPODIS*, pages 7–9, 2002.

[28] Shi-Ding Lin, Qiao Lian, Ming Chen, and Zheng Zhang. A practical distributed mutual exclusion protocol in dynamic peer-to-peer systems. In *IPTPS*, pages 11–21, 2004.

[29] Ricardo Jiménez-Peris, Marta Patiño-Martínez, Gustavo Alonso, and Bettina Kemme. Are quorums an alternative for data replication? *ACM Trans. Database Syst.*, 28(3):257–294, 2003.

[30] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[31] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a common api for structured peer-to-peer overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, February 2003.

[32] Sudhir Aggarwal, Hemant Banavar, Amit Khandelwal, Sarit Mukherjee, and Sampath Rangarajan. Accuracy in dead-reckoning based distributed multi-player games. In *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 161–165, New York, NY, USA, 2004. ACM Press.

[33] Zachary Booth Simpson. A stream-based time synchronization technique for networked computer games, 2000. [Online; accessed 22-Dec-2006].

[34] Soekris Engineering. Four port 10/100 mbit pci ethernet · board. http://www.soekris.com/lan16x1.htm.

[35] Canonical Ltd. Ubuntu 6.0.6. http://www.ubuntu.com.

[36] Microsoft Corporation. Windowsxp sp2. http://www.microsoft.com.

[37] Inc. Sun Microsystems. Java platform, standard edition (java se) 5.0 update 10 jdk. http://java.sun.com/j2se/1.5.0/index.jsp.

[38] The Eclipse Foundation. Eclipse sdk 3.2.1. http://www.eclipse.org.

[39] Sameh El-Ansary, Luc Onana Alima, Per Brand, and Seif Haridi. Efficient broadcast in structured p2p networks. In *IPTPS*, pages 304–314, 2003.

[40] Kevin Glass and Ari Feldman. Space invaders - 2d rendering in java. http://www.cokeandcode.com/tutorials.

[41] R. Salz P. Leach, M. Mealling. A universally unique identifier (uuid) urn namespace. RFC 4122, IETF, July 2005.

[42] Wentong Cai, Francis B. S. Lee, and L. Chen. An auto-adaptive dead reckoning algorithm for distributed interactive simulation. In *PADS '99: Proceedings of the thirteenth workshop on Parallel and distributed simulation*, pages 82–89, Washington, DC, USA, 1999. IEEE Computer Society.

[43] Katherine L. Morse. Interest management in large-scale distributed simulations. Technical Report ICS-TR-96-27, University of California, Irvine, 1996.

[44] G. Malkin. Rip version 2. RFC 2453, IETF, November 1998.

[45] J. Moy. Ospf version 2. RFC 2328, IETF, April 1998.

[46] Quagga routing software suite. http://www.quagga.net.

[47] Anja Feldmann, Anna C. Gilbert, Polly Huang, and Walter Willinger. Dynamics of ip traffic: A study of the role of variability and the impact of control. In *SIGCOMM*, pages 301–313, 1999.

[48] Stefan Saroiu, P. Krishna Gummadi, and Steven Gribble. A measurement study of peer-to-peer file sharing systems. In *SPIE Multimedia Computing and Networking (MMCN2002)*, 2002.

List of Acronyms

AI Artificial Intelligence

API Application Programming Interface

BT Bit Torrent

DISEM Distributed Semaphore

FTP File Transfer Protocol

GUI Graphical User Interface

IDE Integrated Development Environment

IPv4 Internet Protocol version 4

IPv6 Internet Protocol version 6

MMOG Massively Multiplayer Online Game

MMOP Massively Multi-user Online Platform

NIC Network Interface Card

P2P Peer-to-Peer

PC Personal Computer

QoS Quality of Service

RAM Random Access Memory

RMI Remote Method Invocation

VO Virtual Organization