MIDDLEWARE SERVICES FOR DISTRIBUTED VIRTUAL ENVIRONMENTS

A THESIS
SUBMITTED TO THE SCHOOL OF COMPUTING SCIENCE
OF THE UNIVERSITY OF NEWCASTLE UPON TYNE
IN PARTIAL FULLFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By

Fengyun Lu

February 2006

NEWCASTLE UNIVERSITY LIBRARY

204 26850 6

Thesis L8216

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Doctor of Philosophy.

Dr. Graham Morgan (Supervisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Doctor of Philosophy.

Prof. Gordon Blair (External Examiner)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Doctor of Philosophy.

Dr. Aad van Moorsel (Internal Examiner)

Approved for the School of Computing Science:

Dr. John Lloyd
Head of School of Computing Science

ABSTRACT

MIDDLEWARE SERVICES FOR DISTRIBUTED VIRTUAL ENVIRONMENTS

Fengyun Lu

Ph.D. in Computing Science

Supervisor: Dr. Graham Morgan

February 2006

Distributed Virtual Environments (DVEs) are virtual environments which allow dispersed users to interact with each other and the virtual world through the underlying network.

Scalability is a major challenge in building a successful DVE, which is directly affected by the volume of message exchange. Different techniques have been deployed to reduce the volume of message exchange in order to support large numbers of simultaneous participants in a DVE. Interest management is a popular technique for filtering unnecessary message exchange between users. The rationale behind interest management is to resolve the "interests" of users and decide whether messages should be exchanged between them. There are three basic interest management approaches: region-based, aura-based and hybrid approaches. However, if the time taken for an interest management approach to determine interests is greater than the duration of the interaction, it is not possible to guarantee interactions will occur correctly or at all. This is termed the *Missed Interaction Problem*, which all existing interest management approaches are susceptible to.

This thesis provides a new aura-based interest management approach, termed *Predictive Interest management (PIM)*, to alleviate the missed interaction problem. *PIM* uses an enlarged aura to detect potential aura-intersections and

initiate message exchange. It utilises variable message exchange frequencies, proportional to the intersection degree of the objects' expanded auras, to restrict bandwidth usage. This thesis provides an experimental system, the *PIM* system, which couples predictive interest management with the de-centralised server communication model. It utilises the Common Object Request Broker Architecture (CORBA) middleware standard to provide an interoperable middleware for DVEs. Experimental results are provided to demonstrate that *PIM* provides a scalable interest management approach which alleviates the missed interaction problem.

Acknowledgements

I would like to thank my husband, Kier Storey, who gave me lots of support while writing this thesis. He read through this thesis with me and gave me lots of useful suggestions.

I would like to thank my supervisor, Dr. Graham Morgan, for the instruction he has given me throughout my research.

I would like to thank my parents, GangZhen Lu and WanYi He. They always encourage me when I feel frustrated.

Table of Contents

List of Figures

List of Tables

List of Formulae and Theorem

Chapter 1

Introduction

A Distributed Virtual Environment (DVE) is a virtual environment which allows dispersed users to interact with each other and the virtual world through an underlying network. In a virtual world, each user has it own virtual representation, termed "avatar". Users can control their avatar by an input device, such as keyboard, mouse or HMD (head mounted device). Through avatars, users can immerse themselves into and navigate through DVEs; users can interact with other virtual objects and users by exchanging information through the network.

DVEs have been applied in a wide range of situations. Common applications of DVEs include:

- Military Simulations
- Training
- Teleconference
- Virtual Classrooms
- Entertainment
- E-commerce

Historically, the majority of DVEs were developed and used in military simulations, e.g. SIMNET [Macedonia95] and DIS [Cohen94] [Singhal99]. This

was mainly due to the financial costs of hardware which could support DVEs. However, as hardware became more powerful and affordable, DVE technology has become increasingly popular in other applications. For example, DVEs can be applied to training (e.g. fire rescue excises [CFAINET05]), teleconference [Greenhalgh95], virtual classrooms [IBM05], which provide long-distance learning opportunities, entertainment (e.g. games [Sweeney99]) and e-commerce (e.g. virtual shopping malls [ActiveWorlds05]).

A DVE not only has the properties of a single-user virtual environment, but also the properties of a distributed system. DVEs have the property of distributed participants, in that the physical location (e.g. country) of participants is not important (i.e. participants' access and interaction should not be restricted by their geographical location).

In order to build a successful DVE, the developer not only needs to overcome the challenges faced in developing a single-user virtual environment, such as the rendering and collision detection, but also the challenges of building a distributed system. As the goal of this thesis is to build a middleware to support a scalable DVE, the challenges of a single-user mode virtual environment are not within the scope of this thesis.

A successful DVE should provide scalability, consistency and responsiveness. Scalability requires a DVE to be capable of supporting thousands of dispersed users via heterogeneous networks simultaneously; consistency requires every user participating in a DVE to perceive the same virtual world at the same time; responsiveness requires a DVE to be able to propagate events and respond to them sufficiently quickly that they appear instantaneous to the users. However, due to the limitations of network bandwidth and latency, a technique should be provided to reduce the volume of messages exchanged through an underlying network in order to improve the scalability, consistency and responsiveness of a DVE in real-time. Interest management is such a technique. As a result of

reducing message transmission, the DVE will have more processing resources and bandwidth available to support additional users and provide a more responsive system. This will reduce the likelihood of network congestion, which, in turn, may decrease the message transmission latency, therefore improving the consistency of the DVE. In addition to reducing the message transmission, a technology should be provided to overcome the intrinsic heterogeneity exhibited over the Internet and ease the implementation of the network component of DVEs. Middleware can be utilised to fulfil these requirements.

In addition to the adoption of interest management and middleware, the communication models adopted by DVE developers influences the scalability, consistency and responsiveness of their DVE. Three common communication models are available in distributed systems: peer-to-peer, centralised server and de-centralised server. The details of these communication models and how they affect a DVE will be discussed in more detail in Section 1.3 and in Chapter 2.

1.1 Interest Management

Interest management is a technique designed to filter unnecessary information from being exchanged between participants in the virtual world. The assumption behind interest management is that a virtual world contains a tremendous amount of information, of which each individual participant only needs to know a small proportion at any given time. For example, a participant, who acts as a soldier in a DVE, may not be required to have knowledge of the state of an airplane covering terrain several miles away. However, it is necessary for this participant to receive information regarding the other soldiers nearby. The information which a participant is required to know is described as the information it is "*interested*" in. Therefore, interest management is an approach

for determining what information each participant is interested in and disseminating this information to the relevant participants.

A number of interest management techniques have been proposed. These can be categorised as three different types of interest management: region-based, aura-based and hybrid interest management approaches. Region-based interest management uses spatial subdivision to partition a virtual world into discreet regions. Participants whose objects reside in the same, or neighbouring, regions can exchange state update information with one-another. Aura-based interest management associates a bounding volume, commonly a sphere, with an object to represent the object's area of interest. A participant will receive state update information from all objects which fall inside its object's area of influence. Region-based interest management can be implemented efficiently using tree structures or spatial hashing. However, it is relatively imprecise and can result in a large amount of unnecessary information exchanged between objects in the same region. Conversely, aura-based interest management is a highly-precise interest management approach. Unfortunately, a naive implementation of this approach is very computationally expensive, as aura-based interest management requires every object's aura to be compared for intersection at regular intervals. Hybrid interest management approaches combine both region-based and aura-based interest management to attempt to provide efficient, precise interest management by using aura-based interest management within regions. The detail of these interest management approaches are discussed in more depth in Chapter 2.

1.2 Middleware

Middleware is a class of software that resides between an application and the operating system. A number of different types of middleware exist, but for the purpose of this thesis, the term middleware is used to denote

networking/distributed systems middleware. Middleware shields the application developer from the complexity of networking issues and provides them with services to ease the development of distributed applications. The underlying purpose of middleware is to assist application processes to transparently collaborate regardless of differences in processes and network, such as platform, programming languages, machine data formats and networking protocols. Therefore, middleware is suitable for connecting users residing in heterogeneous networks in a DVE.

Middleware provides two kinds of message models: synchronous messaging and asynchronous messaging. Synchronous messaging requires strict synchronisation between the sender and receiver, such that the sender process is blocked until the receiver process has received the message and provided a response; asynchronous messaging, conversely, does not require the sender and receiver process to participate simultaneously. In addition to this classification, middleware can be categorised into Remote Procedure Calls (RPCs), Message-oriented Middleware (MOM) and Distributed Object Middleware (DOM). RPCs provide distributed procedure calls that mimic the semantics of local method invocation. Depending on the implementation, the synchronous and asynchronous messaging model can be adopted in RPCs, although the former is more common. Unlike RPCs, MOM is specifically designed to implement the asynchronous messaging model. A message queue is utilised to store the messages from clients to a server such that the messages are kept in the message queue when the server is busy or unavailable. Similar to RPCs, DOM can be implemented using both the synchronous and asynchronous messaging models. It incorporates Object-oriented Programming (OOP) concepts, namely data encapsulation and reuse, to provide the abstraction of distributed objects. The details of existing middleware are discussed in Chapter 2.

1.3 Communication Models

The choice of communication model plays an important role in the design and implementation of a distributed system. Three common communication models are available for DVE developers: peer-to-peer, centralised server and de-centralised server. The peer-to-peer communication model exhibits complete connectivity between machines in a network, such that messages are transmitted directly from sender to receiver. The centralised server communication model utilises a central server to route messages from the sender to the desired recipients. As this central server is a single point of failure, the de-centralised server communication model utilises a set of servers to provide additional scalability and fault-tolerance. The pros and cons of these communication models in terms of scalability, consistency and responsiveness are discussed in detail in Chapter 2.

1.4 Contribution of Thesis

This thesis provides a new interest management approach, termed *Predictive Interest Management (PIM) [Lu03]*. As previously mentioned, there are three categories of interest management exist: region-based, aura-based and hybrid approaches. However, existing interest management approaches overlook the *Missed Interaction Problem*. Missed interactions occur when the duration of a pair of objects' interaction is less than the time taken by the interest management approach to resolve the interaction between these objects. In this case, participants may not be aware of the current state or even the existence of objects with which they should be interacting. *PIM* is an aura-based interest management approach designed to alleviate the missed interaction problem by using an expanded aura to predict an object's future interests. It uses three different classes of messages, exchanged at different frequencies, to minimize

the overhead of any additional message exchange as a result of using larger auras.

This thesis describes the implementation of a *PIM* system to support scalable, interoperable DVEs [Lu05]. Middleware that supports the asynchronous messaging model is suitable for handling the large volume of message exchange required by DVEs. In addition, these middleware solutions provide different services to filter unwanted messages (e.g. the notification service in the Common Object Request Broker Architecture (CORBA) [OMG05] and the Java Messaging Service (JMS) in J2EE [Sun05]). However, the message handling speed of these services are often insufficient to maintain real-time performance within DVEs. Therefore, this thesis describes a new interest management middleware, suitable for large-volume message exchange in real-time. The *PIM* system is an experimental system, which utilises existing middleware in its message dissemination layer (see Figure 2.2 in Chapter 2) and implements predictive interest management to alleviate the missed interaction problem. The *PIM* system utilises CORBA, a middleware standard, to provide interoperability. In addition, the de-centralised server communication model is adopted to provide a highly-scalable middleware solution for DVEs.

Finally, this thesis provides a scalability evaluation of the *PIM* system through four sets of performance measurements. It is shown that the additional message exchange utilised in predictive interest management to alleviate the missed interaction does not deteriorate the system performance compared with the traditional aura-based interest management system. Additionally, the adoption of the de-centralised server communication model allows the *PIM* system to support large numbers of users simultaneously; experiments with up to 6000 simultaneous users have been performed.

1.5 Thesis Outline

This thesis is composed of six chapters. Chapter 2 describes background material, including the architecture of DVEs (application layer, message dissemination layer and network layer), interest management, middleware, communication models and related work. Chapter 3 introduces a new theory, termed Predictive Interest Management (*PIM*), to alleviate the missed interaction problem. Chapter 4 describes the implementation of the *PIM* system, which utilises the *PIM* theory as the core technology. Chapter 5 provides the results of experiments to test the scalability of the *PIM* system. Chapter 6 presents the conclusion of this thesis and possible future work.

Chapter 2

Background

This chapter gives an overview of Distributed Virtual Environments (DVEs), the properties of DVEs and the challenges the developers face to build a scalable DVE. A general DVE architecture is introduced, which is constructed from three layers: the application, message dissemination and network layers.

The application layer provides users with a graphical representation of a DVE and input/output devices such that a user can interact with the DVE; the message dissemination layer may provide the developers easy access to the network layers, services to reduce the message exchange through an underlying network, services to regulate the message exchange frequency, and services to overcome the heterogeneities between nodes and networks; the network layer provides a selection of protocols for the developers to suit the requirements of different types of DVEs, such as military simulation or multiplayer computer games.

Middleware and interest management are two technologies which are suitable for integration into the message dissemination layer. Middleware is a class of software which shields the developers from the low-level network implementation and overcome the heterogeneities between nodes and networks. Interest management is a message filtering technology which attempts to reduce message exchange through an underlying network based on given filtering

criteria. Three types of interest management approach are described: region-based, aura-based and hybrid interest management. However, all these interest management approaches are not sufficient to overcome the missed interaction problem, which will be discussed in detail later.

The choice of communication model will influence the scalability, consistency and responsiveness of a DVE. Peer-to-peer, centralised server and de-centralised server communication models are discussed. Finally, related work and the contributions of this thesis are provided.

2.1 Distributed Virtual Environment

Virtual Environments (VEs) are virtual spaces generated by computers to simulate both realistic and imaginary worlds that enable users to navigate and interact with virtual objects. VEs provide an interactive simulation analogous to the way that humans communicate with each other and manipulate objects in the real world. Therefore, spatial embodiments can be generated by the VE application to represent users and objects inside the virtual world. However, traditional single-user VEs do not permit dispersed users to interact with one-another. It is therefore desirable to extend the concept of VEs to permit dispersed users to interact with each-other to provide a richer and more interactive experience. The combination of network and VEs, Distributed Virtual Environments (DVEs), has led to new ways of displaying information and communicating with dispersed users and machines. In DVEs, dispersed users can communicate with each other through graphical representations. This feature of DVEs allows users to share information and cooperate with each other similar to the way people interact in the real world. Therefore, DVEs find applications within training, teleconferencing, long-distance education and entertainment, such as multiplayer gaming.

As the applications of DVEs are varied, different researchers have their own definitions of DVEs, giving rise to different aliases for the systems. The following are a brief review of some definitions of DVEs:

According to the definition of a DVE by Singhal and Zyda in [Singhal99], *a networked virtual environment (net-VE) is a software system in which multiple users interact with each other in real-time, even though those users may be located around the world. Typically, each user accesses his or her own computer workstation or console, using it to provide a user-interface to the content of a virtual environment. These environments aim to provide users with a sense of realism by incorporating realistic 3D graphics and stereo sound, to create an immersive experience.*

Gibson in [Churchill01] described DVEs as: *distributed virtual reality systems that offer graphically realized, potentially infinite, digital landscapes. Within these landscapes, individuals can share information through interaction with each other and through individual and collaborative interaction with data representation.*

Snowdon et al gave a more open-ended description of DVEs in [Churchill01]: *A CVE is a computer-based, distributed, virtual space or set of places. In such places, people can meet and interact with others, with agents or with virtual objects. CVEs might vary in their representational richness from 3D graphical spaces, 2.5D and 2D environments, to text-based environments. Access to CVEs is by no means limited to desktop devices, but might well include mobile or wearable devices, public kiosks, etc.*

According to the definitions, DVEs are VEs which allow dispersed participants to navigate through and interact with a virtual world in a distributed manner. They allow users to interact with virtual objects, independent of the users' physical location, providing users a brand new interactive and dynamic

experience. For the purpose of clarification, the term *object*, or *virtual object*, is used interchangeably to describe an "entity" which has a physical presence in a virtual environment. A node is a machine and may host one or more objects which inhabit a DVE. A node is responsible for providing an interface by which a user can interact with a DVE. Users can immersive themselves into a DVE through *avatars*, which are the users' graphical representation. Users can control their avatars through input devices connected to their respective nodes. The movement of avatars are manifested by other users' nodes through message passing. After receiving a message, a node updates the other user's avatar according to the information (e.g. position) provided inside this message. Therefore, user interaction in a DVE is accomplished by message exchange in the underling network. However, as messages might be lost, damaged or received out of order when transmitted over a network, if no mechanism is provided to guarantee message ordering, consistency and reliability, users will perceive inconsistent views of a DVE.

Due to the distributed nature of DVEs, it is difficult to ensure each user perceives the same, or similar, events simultaneously. It is also difficult to ensure the DVE system responds to the users' input and updates its state in real-time as the number of dispersed users increases. Guaranteeing the consistency and responsiveness of DVEs in real-time is a major challenge which has driven research into DVEs. Due to the widespread adoption of the Internet and the affordability of computer hardware, more people are capable of participating in DVEs. In addition, as processing power has increased, the content of DVEs has become richer, which, in turn, has further increased the user-base of DVEs. Therefore, the scalability of DVEs, in terms of the number of virtual objects and users that can be supported simultaneously, is of utmost importance. As mentioned previously, users' interactions are notified by message exchange over the network. However, large numbers of users participating in a DVE at the same time will result in a huge amount of message exchange. This may cause network congestion, overload the machines participating in the DVE or, in the

worst-case scenario, cause network-induced failure. The scalability of a DVE depends largely on the number of messages exchanged between machines. Therefore, if the number of messages exchanged between machines can be reduced without influencing the users' interaction, the scalability of the DVE can be increased. In general, this is a reasonable assumption as each user is usually only capable of viewing a small proportion of the virtual world at a given time. As such, each user is only required to receive messages concerning objects which are within its field of vision. A message dissemination schema, which is required to determine which messages must be exchanged between each machine depending on users' requirements, should be provided. In addition, the selection of an appropriate network architecture is important as different network architectures provide varying scalability, consistency and responsiveness characteristics.

2.2 General Properties of DVE

2.2.1 Shared Distributed Environment

Different users should perceive the same virtual environment in a distributed manner regardless of the geographical distance between participants and differences in configurations and operating systems between participants' computers. One way to provide consistency in virtual environments is to use a reliable server to store all data in a database. Users can acquire and update objects through the server. Another approach is that every user has a duplicate database, and users can update the objects controlled by themselves and broadcast the update to other users within the same VE.

2.2.2 Virtual Objects

In DVEs, virtual objects include static and dynamic objects. Static objects are objects which cannot be changed by or interact with other objects. For example, the background objects (i.e. the ground, floors, walls, ceilings and doors of which the world is built from) are static and will not change throughout the run time of a DVE. Dynamic objects, conversely, can be manipulated by other objects. There are two fundamental classifications of dynamic objects: avatars and non-avatars. An avatar is a graphical representation of a user. The interactive properties of DVEs, which will be introduced in the next Section, provide all avatars the ability to dynamically change the state of the shared objects, similar to the way that humans manipulate objects in the real world. The application software controls non-avatar objects. Their behaviours may be triggered or changed by events in the DVE, such as avatar interaction, time regulation and target accomplishment.

2.2.3 Interaction and Navigation

The interaction and navigation provided to users by VEs allow avatars to move about, pick up and manipulate shared virtual objects, and communicate with other avatars in the VE. Various input devices are used to accomplish all these tasks. Users can use a mouse to navigate through a VE by changing the viewpoint of the avatars, to monitor the speed of avatars, to interact with virtual objects by picking and moving, and to perform interaction with other avatars, such as shooting. Users can use a keyboard to communicate with other users by typing, to control the direction of avatars etc. Although the mouse and keyboard are the most common input devices used for VEs, the choice of control mechanisms in VEs are application-dependent. Different input devices provide different degrees of immersion for users. For example, a joystick or control pad may be a better choice than mouse and keyboard in some simulations and games. Head-mounted displays [Bungert05][Milgram99] (HMDs) provide the

most immersive experience currently available in VEs. Such devices block out vision of external entities, allowing users to concentrate solely on the VE. With the use of stereoscopic vision, the illusion of true 3D vision can be achieved. The inclusion of analogue feedback devices to determine the orientation of the users head, allows the VE to adjust the displayed image based on the user's physical movement. This technology can be coupled with proprietary interactive devices, such as gloves, to allow users to interact with objects in the VE using only gestures. A field of research, called Haptic Interaction [GIST05], provides a force-feedback technology which gives users the illusion of "feeling" the virtual object through a series of mechanical and electrical sensations through an input device.

2.2.4 Distributed users

The interaction and navigation features provided to users by DVEs allow users to interact with one-another and virtual objects. All interactions occur based on given criteria associated with the virtual world, such as the virtual distance or the specified interests of a given virtual object or virtual space. Therefore, the physical location of participants is irrelevant with respect to the interactions between avatars residing in the same virtual world.

2.3 Challenges of DVE Implementation

Building a scalable DVE requires the developer to have excellent skills and knowledge in constructing the virtual world (the application layer) and to handle the high volume of message exchange in an underlying network. This section only discusses the networking challenges (the message dissemination layer and the network layer) of building a DVE as the application-layer issues are beyond the scope of this thesis. Below are the concise descriptions of the challenges in

bandwidth, network latency, heterogeneity of networks, consistency and responsiveness.

2.3.1 Bandwidth

In a DVE, participating users should be able to share the same context: the same visual, audio and immersive environment. However, it is hard to maintain context-sharing equality between different users with variable bandwidth. One way to guarantee context-sharing equality is to restrict the throughput to that of the lowest user and control the number of users allowed to enter the DVE simultaneously. This reduces the richness and scalability of a DVE; consistency, in terms of users experiencing the same context, will be maintained. Moreover, this wastes the resources available to the users with high bandwidth and fast hardware configurations. In contrast, if a DVE is designed to utilise the available resources of different users, it provides different richness levels to users according to their available resources, although absolute consistency will be sacrificed. In this case, due to the bandwidth differences, low-bandwidth users are not capable of sharing the same audio and visual context as the high-bandwidth users.

2.3.2 Network Latency

Although network latency is an uncertain and unavoidable element in message transmission, it can be influenced by certain factors. Network traffic can contribute to the network latency, which can be affected by the frequency of message transmission and the size of each message. The lower the frequency of message transmission and the smaller the size of each message transmitted, the lower the chance of network congestion occurring, and vice versa. Another factor leading to network latency is the choice of network protocols. For example, to guarantee delivery, the underlying network architecture must use

acknowledgment and error recovery schemes, which can introduce extra transmission latency, e.g. TCP/IP. To reduce the effect of network latency and speed up the message transmission frequency, protocols such as UDP/IP can be used. Such protocols offer higher message transmission speed at the detriment of delivery reliability, i.e. message loss and messages disordering. With regards to the development of a DVE, different network protocols should be implemented to cope with different message transmission requirements. In addition to network traffic and the choice of protocols, application-level techniques for DVEs can be deployed to potentially reduce network latency. For example, extrapolating object's current state from previous messages to predict the trajectory of that object can be used to reduce the message transmission frequency, which can reduce the network traffic and potentially the network latency.

2.3.3 Heterogeneous Network

The Internet connects computers with different hardware configurations running on different operating systems. Due to these variations in computer configuration, operating system and network connection, it is difficult to maintain the same virtual environment for dispersed participants. For example, it is impossible for a participant, who connects using a modem-connection, to share the same audio and visual information with a participant, who uses a cable-connection, as the modem-user is not capable of receiving or sending the same volume of data as the user with a cable-connection. Heterogeneity in computers participating in the DVE means that it is not possible to guarantee that all computers may be capable of maintaining the same speed of message update as other faster machines in the DVE. This difference in speed may be attributed to the speed of the processor, the amount of memory available, the speed of the graphics card installed in the machine, which may not be able to render the DVE fast enough, and the presence of other processor-intensive tasks being run concurrently on the machine. Operating systems often differ in the

programming languages and libraries they provide. Therefore, a DVE should be implemented using middleware to shield the developer from network and platform heterogeneity.

2.3.4 Consistency and Responsiveness

Consistency requires every user participating in a DVE to perceive the same virtual world at the same time. This involves maintaining the shared-state of dynamic objects in a DVE in real-time. Responsiveness requires a DVE to be able to propagate events and respond to them sufficiently quickly that they appear instantaneous to the users. This is primarily observed as the smooth animation of dynamic objects in the participants' machines.

Maintaining the shared-state of dynamic objects in DVEs in real-time can be classified as resolving concurrent access problems in shared resources in a distributed system. Ignoring the timing factor, a number of mechanisms (lock utilisation etc.) have been developed to solve the concurrent access problem successfully. However, maintaining the shared-state of dynamic objects in DVEs, while guaranteeing the smooth animation of dynamic objects in the participants' machines in real-time places a non-trivial challenge on the DVE developers.

One way to maintain the shared-state of dynamic objects is to use a central server to maintain states of all the dynamic shared objects. All dynamic shared objects are updated through the collaboration between the central server and the clients' machines. The advantage of this approach is that it guarantees absolute consistency of every shared object. However the central server is the bottleneck of this approach and, as a result, the scalability of the DVE is diminished. Additionally, due to network latency and the congestion level of the server, participants may perceive jerky animation in their output devices. Consequently, the users' immersive experience may be detrimentally affected.

Frequent state regeneration is based on the assumption that if the frequency of message transmission is high enough, some message loss in transmission will not be detrimental to the smooth animation on the participants' machine, as subsequent messages should arrive soon enough to compensate for any message loss. Compared with the central server technique, frequent state regeneration provides improved scalability and increases the immersive experience at the cost of lower consistency of dynamic shared objects.

Dead reckoning resides on the opposite end of the consistency spectrum, offering low-levels of consistency but potentially reduces the volume of data required to be transmitted, resulting in better scalability and responsiveness. Dead reckoning is an algorithm to predict the future position of objects based on limited information so that the frequency of message transmission can be restricted under a threshold. Each participant's avatar, when displayed in other participants' machines, is called a 'ghost'. Dead reckoning prescribes that each avatar implements a dead-reckoning algorithm to predict its own trajectory. If the deviation of actual movement from the predicted movement of their avatar is greater than some pre-defined threshold, a participant will send a message to update its ghost position on the other participants' stations. When a station receives this message, it uses a convergence algorithm to correct the position or state of the ghost. By using dead reckoning, variable frequency of message transmission can be achieved.

2.4 Distributed Virtual Environment Architecture

Developing DVEs is not a trivial task. DVEs differ in infrastructure depending on their application and purpose. However, according the message transmission flow, the infrastructure can be generally divided into three different functional layers (Figure 2.1):

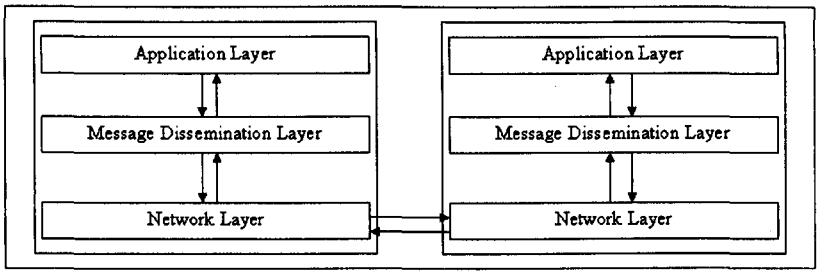


Figure 2.1 Three-Tier DVE Architecture

As can be seen from Figure 2.1, the message dissemination layer is the intermediate layer which provides certain services for the DVE developers to facilitate message passing between the application layer and the network layer. The application layer receives user input and passes it to the network layer through the message dissemination layer. In the network layer, these messages are transmitted to the corresponding nodes' network layer such that the application layer can receive these messages and make appropriate update in the output device. The following three sections will discuss these layers in detail.

2.4.1 Application Layer

This layer contains the graphics/rendering engine and the input/output control and processing units. The graphics/rendering engine collaborates with the Graphics Processing Unit (GPU) to generate the images which are displayed to the end user; the input/output control and processing units take user input, process it and translate it into application-dependent DVE events. These events may cause state updates on one or more objects which will, in turn, be reflected to the user by the graphics/rendering engine. In addition, after the events generated, the application layer passes these events into the message dissemination layer. For example, a user's avatar (a virtual human), participating in a teleconference, may write text on a blackboard corresponding with the user's input. This text is reflected on this user's output device as well as transmitted to the message dissemination layer for further processing.

2.4.2 Message Dissemination Layer

This layer is an intermediate layer between the application and network layers. The purpose of this layer is to provide a platform-independent, protocol-transparent API (Application Programming Interface) for the low-level network implementation. In large-scale DVEs, users may participate in a virtual world from different network architectures. This heterogeneity may manifest itself as, for example, different byte-ordering between machines, i.e. Big-Endian and Little-Endian machines. Differences like this require developers to ensure that the bits are read in the same sequence in the recipient as the sender intended. Hence, this layer can simplify DVE development by providing facilities to ensure interoperability with heterogeneous network architectures and platforms.

The message dissemination layer can provide location and discovery services, which removes the requirement of the application layer to determine the relevant recipients of a state update message. It can employ filtering mechanisms to reduce the number of unnecessary messages which are transmitted over the underlying network.

The application layer may generate a large volume of state updates at potentially high frequencies. The message dissemination layer can be used to ensure that messages are transmitted to the appropriate recipients at suitable frequencies such that events are perceived to occur in real-time, while avoiding overloading the network and therefore ensuring high-levels of consistency. While too high message exchange frequencies may cause network overloading, too low frequencies may result in inconsistencies arising in the state of a DVE between nodes. In addition, this layer may provide services to regulate message transmission frequency according to some filtering criteria. For example, given a distance-based filtering criteria, the frequency of message exchange between two avatars could be proportional to the distance between them within the DVE.

Furthermore, this layer may provide the developers the choice of synchronous and asynchronous messaging models.

Synchronous messaging is a two-way messaging model in which the sender process will be blocked until it receives a reply from the recipient. Asynchronous messaging is a one-way messaging model, which does not require simultaneous coordination between sender and recipient. The use of the synchronous messaging model may inhibit scalability as a DVE may be required to support the transmission of a large volume of messages in real-time. As such, in order to provide high-levels of scalability, the asynchronous messaging model should be employed within the message dissemination layer in a DVE.

2.4.3 Network Layer

The network layer should provide network protocols to enable high-levels of accessibility to a DVE over LANs and public access networks, e.g. the Internet. According to [Tanenbaum96][Javvin05], network protocols are agreements for computers and other network devices to exchange information over a network. The agreements can be summarised as a formal set of rules, conventions and data formats that allow computers and network devices to understand each other. In addition, the network protocols provided in this layer should be able to satisfy the application-dependent transmission requirements. For example, a teleconference DVE may require a reliable network protocol to transmit text messages between users. However, for the transmission of audio or video messages, an unreliable network protocol should be more appropriate. Network protocols are generally built on top of services provided by lower-level protocols and/or hardware.

In general, network protocols can be classified as unicast, broadcast and multicast. Unicast is termed as transmitting data from one point to another point, such as from a sender to a recipient; broadcast is termed as sending data from

one point to all the other points, such as from a sender to all hosts on a network; multicast is termed as sending data from one point to a group of other points, such as from a sender to a subset of hosts on a network.

Five popular network protocols, which are commonly utilised in DVEs, are introduced: The Internet Protocol (IP), Transmission Control Protocol (TCP); User Datagram Protocol (UDP); IP broadcasting; and IP multicasting. TCP/IP and UDP/IP can be used as unicast protocols.

2.4.3.1 The Internet Protocol

The Internet Protocol (IP) is the most popular network layer communication protocol currently used on the Internet. This protocol provides the segmentation and reassembly (SAR) function to satisfy the bandwidth requirement of the Internet. If the network cannot support large packets, IP splits these large packets into small fragments and reassemble the small fragments into the original packets at the destination. The flexibility of IP comes from the fact that IP can connect heterogonous nodes and networks together. To be precise, IP can transmit the packets from the host to the destination ignoring the fact that the transmission path might include phone line, DSL/Cable, wireless radios etc; IP provides a standard communication platform for heterogonous machines regardless of data/memory format differences, e.g. byte ordering.

However, IP is a low-level protocol and is not suitable for applications to use directly. For example, IP does not guarantee message delivery, the ordering of received messages or detect the receipt of damaged packets. Most applications require more facilities than IP can provide, therefore, different protocols are built on top of IP.

2.4.3.2 Transmission Control Protocol

Transmission Control Protocol (TCP) [Cerf74] [Comer91] is a connection-oriented transport layer protocol built on top of IP to provide a best-effort guarantee of reliable and chronological transmission/receipt of a byte stream over a network. It fragments the byte stream into discrete packets and passes each one to the IP layer; the receiver reassembles the packets into the original byte stream. Communication between hosts is established by creating a point-to-point connection. These connections are uniquely identified by the hosts' IP address and the port number (service identifier) which the connection is established on. The connection is maintained until one of the hosts closes the connection, or one of the hosts crashes.

To provide best-effort reliability of message transmission, TCP uses acknowledgement and retransmission schemes. When the destination host receives a packet, it transmits an acknowledgement to the source host to inform of the message arrival. If the communication is unidirectional, this may involve the transmission of a packet containing only the acknowledgement, or in the case of bidirectional transmission, the destination host may piggyback the acknowledgement in the next packet it transmits to the source host. If the source host does not receive the acknowledgement within a threshold time, it will retransmit the unacknowledged packet again.

TCP contains flow control to best utilise the available bandwidth of both the source and destination hosts. This tries to avoid the situation in which a fast sender can swamp a slow receiver with more packets than it can handle. It does this by restricting the volume of data that can be unacknowledged at any time. If the amount of unacknowledged data is at some threshold level, new packets cannot be transmitted until acknowledgements have been received; if the outstanding packets timeout, retransmission occurs.

2.4.3.3 User Datagram Protocol

User datagram protocol (UDP) [Cerf74] [Comer91] [InternetSociety05] is an unreliable, connectionless transport layer protocol. Unlike TCP, there is no connection maintenance between hosts, no acknowledgement transmission obligation and no flow control. The purpose of this protocol is to satisfy situations where delivery speed is considered to be more important than reliability. For example, in a teleconference, the loss of some audio/video messages will not detrimentally influence the quality of the whole meeting; conversely, the overhead of providing best-effort reliable delivery of huge amounts of audio/video messages in real-time would increase network bandwidth consumption due to message retransmission, which would deteriorate the overall performance of the teleconference.

2.4.3.4 IP Broadcasting

Hardware broadcasting allows the delivery of a single packet to all hosts on the network or subnet without requiring the repeated transmission of the packet by the sending host. With most hardware, this is achieved by sending packets to a reserved broadcast address; all hosts residing on the same network/subnet recognize the broadcast address and accept all packets to that address. Hosts on the network benefit from broadcasting by two ways: first, when hosts require information from the network without knowing the exact address of host who can provide the information; second, when a host wants to provide information to a large set of hosts in a timely manner. However, the chief disadvantage of broadcasting is that every broadcast packet consumes resources on all hosts. When a host receives a broadcast packet, the network interface card is not capable of discarding the packet and the Operating System (OS) must process it even if no local application is interested in the broadcast packet; computer resources are wasted.

IP broadcasting is an Internet abstraction of hardware broadcasting. According to [Mogul84], under the IP broadcasting protocol, the datagram is routed by normal mechanisms, just like the normal unicast datagram, until it reaches a router attached to the destination IP network, at which point it is broadcast.

2.4.3.5 IP Multicasting

Unlike hardware broadcasting, where a packet is received by all hosts on a network, multicasting [Comer91] permits a host to transmit a packet to a selected group of hosts on a network. When a host receives a multicast packet, the network interface card can choose to accept or discard the packet based on the multicast address, without needing to pass it to the OS. When a group of hosts wishes to communicate using multicasting, they must configure their network cards to accept packets on a particular multicast address. Once the configuration is complete, all members of the group will receive any packets transmitted to the multicast address.

IP multicasting is the Internet abstraction of hardware multicasting. It allows the transmission of an IP datagram to a group, which consists of zero or more hosts identified by a single IP/multicast address. A multicast datagram is delivered to all members of its destination host group with the same "best-effort" reliability as regular unicast IP datagram, i.e. the datagram is not guaranteed to arrive at all members of the destination group or in the same order relative to other datagram. The membership of a host group is dynamic, i.e. hosts may join and leave groups at any time. There is no restriction on the location or number of members in a host group. A host may join more than one group at a time. Furthermore, a host is not required to be a member of a group to send datagrams to it.

2.4.3.6 Protocol Evaluation

One of the requirements of building a scalable DVE is to ensure the accessibility of the DVE. This places restrictions on the network protocols the developers can use if they are required to enable dispersed users to participate in their DVE over the Internet. In order to allow these users to participate in the same DVE, the network layer should utilise network protocols which are widely supported over the Internet. Therefore, as can be seen from Table 2.1, TCP/IP and UDP/IP are the most appropriate choices for the developers. Additionally, in order to alleviate inconsistency between users, a network protocol, which provides best-effort guarantees for packet delivery and order, should be selected. In summary, TCP/IP is the most suitable network protocol to support large-scale DVEs over the Internet.

	TCP/IP	UDP/IP	IP Multicast	IP Broadcast
Unicast	√	√		
Broadcast				√
Multicast			√	
Message Ordering	√			
Acknowledgement	√			
Internet Support	√	√	PARTIAL	

Table 2.1 Network Protocol Properties

2.4.4 DVE Architecture Summary

In this section, three abstract layers (application, message dissemination and network layers) are described in detail. Different types of DVEs place different requirements on the message dissemination and network layers.

Military simulations generally take place on the same LAN. This provides low-latency and high bandwidth message transmission and supports hardware multicasting and broadcasting. Therefore, this property relieves the service

requirements of the message dissemination layer and removes many restrictions on the choice of network protocols in the network layer.

In Computer Supported Collaborative Work (CSCW), users are dispersed over the world but the number of users who can participate simultaneously is often limited. Therefore, the message dissemination layer should be able to provide services to overcome the heterogeneities between networks and to discover the location of users. In order to allow dispersed users to participate in this type of DVE, the network layer should use a protocol which is commonly supported over the Internet and provides best-effort guarantees for message receipt and ordering to maintain consistency. As TCP/IP meets these requirements, it is the most appropriate protocol to be utilised in CSCW. However, as the number of users is comparatively small, the message dissemination layer is not required to provide a message filtering service as bandwidth usage should be relatively small.

In Massively Multiplayer Online Games (MMOGs), large numbers of dispersed users, in the order of thousands, are able to participate together in a networked game. In order to support this number of participants simultaneously, message filtering technology must be implemented within the message dissemination layer to reduce the bandwidth usage. In practice, many current MMOGs artificially restrict the number of players who can interact with one-another to a manageable amount, e.g. 2-32 players. This inadequacy in current MMOGs could be overcome with the use of sophisticated message filtering technology. As MMOGs are required to operate over the Internet, they share the same requirements of their network layer as CSCW. As such, MMOGs should adopt TCP/IP.

Although different types of DVE have different requirements in their message dissemination and network layers, message filtering technology can be adopted in any type of DVE. Therefore, in order to build a scalable DVE, message

filtering technology should be implemented in the message dissemination layer. Interest Management, a popular message filtering technology, filters unwanted message exchange in the underlying network between nodes according to the users' "interests". In CSCW and MMOGs, network heterogeneity should be of primary concern in the message dissemination layer. In addition, heterogeneity has a large influence on the choice of network protocol in the network layer. Middleware shields the DVE developers from the issues of network heterogeneity. It can be integrated into the message dissemination layer, delegating the low-level network issues, such as protocol interoperability, to the middleware developers. Therefore, the network layer can utilise the protocols supported by the middleware. Figure 2.2 shows an architecture which a developer can use as a template to develop a complete DVE.

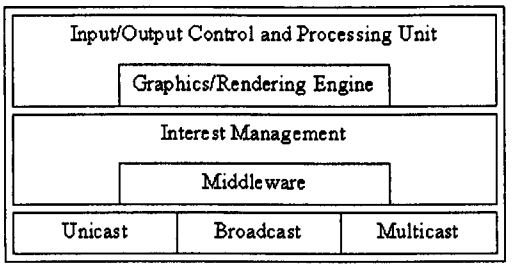


Figure 2.2 DVE Architecture

2.5 Middleware

Middleware is a class of software residing between an application and the operating system. It shields the application developer from the complexity of networking issues and provides them with services to ease the development of distributed applications. The underlying theory of middleware is to assist application processes to transparently collaborate regardless of differences in processes and network, such as platform, programming languages, machine data formats and networking protocols. Middleware provides two types of messaging models: synchronous and asynchronous messaging.

- Synchronous messaging is a two-way middleware messaging protocol that is best suited for tightly coupled client-server applications. The client's process will be blocked until it receives a reply from the corresponding server, which implies the successful delivery of the request message. Client and server simultaneous attendance is essential in the synchronous messaging protocol.

- Asynchronous messaging is a one-way middleware messaging protocol, which does not require simultaneous, coordinated participation between the client and server. Under this messaging protocol, the client process does not require a reply from the server process.

Based on the evolution of middleware in [Ruh99], five types of middleware can be defined: Data Access, Remote Procedure Call (RPC), Transaction Processing Monitors, Message Oriented Middleware (MOM) and Distributed Object Middleware (DOM). Data Access Middleware provides easy-to-use connectivity to database servers. Transaction Processing Monitors provide support for data integrity in mission-critical distributed applications, where the term transaction is used to describe a group of changes which either succeed together or do not occur at all. The following subsection focuses on the development of messaging models. Therefore, it concentrates on the three remaining types of middleware, which are categorised according to the messaging protocols they support in Table 2.2.

Middleware	Synchronous Messaging	Asynchronous Messaging
Remote Procedure Calls (RPCs)	Yes	Limited
Message-Oriented Middleware (MOM)	Limited	Yes
Distributed Object Middleware (DOM)	Yes	Yes

Table 2.2 Middleware Message Model Comparison

2.5.1 Remote Procedure Calls (RPCs)

The major contribution of RPCs is to provide distributed procedure calls that mimic the semantics of local procedure calls, giving users the illusion that the procedure resides in the same computer. In RPCs, an Interface Define Language (IDL) is adopted to define a contract between clients and servers. The IDL compiler generates interface code for multiple programming languages. The application-to-middleware layer interface code is called stub, which is responsible for marshalling (packing) and unmarshalling (unpacking) data to/from the network layer. Three types of message exchange protocols are provided in RPCs:

- Request
- Request-reply
- Request-reply-acknowledge

RPCs usually utilise synchronous messaging protocols. However, asynchronous messaging protocols are achievable using multiple threads that have limited exception-handling facilities. Due to the complexity of implementing the asynchronous mechanisms, RPCs are utilised mostly in synchronous request-reply client-server architectures. In addition, different vendors' RPC implementations often use different data representations, message formats etc., which limits their interoperability.

2.5.2 Message-Oriented Middleware (MOM)

Unlike RPCs, MOM is specifically designed to implement the asynchronous messaging protocol. MOM uses two specialised types of asynchronous communication:

- Message Queuing
- Publish/Subscribe

In the message queuing model, the MOM system provides a message queue between sender and receiver; if the receiver process is not available or is busy, the messages sent by the sender process are kept in a local queue until they are forward to the receiver process. In this case, MOM technology eliminates the dependency on requiring the simultaneous participation of senders and receivers. In this design, messages can be delivered to the destination, even if the receiver process is not available. However, if the message delivery frequency on the sender side is much higher than the message consumption frequency on the receiver side, congestion problems might occur due to the growing size of the message queue.

In the publish/subscribe model, participants act as either publishers or subscribers. Subscribers register to channels to receive messages regarding a subject of interest; publishers send subject-tagged messages to channels. Therefore, channels decouple publishers and subscribers and disseminate messages to registered subscribers without the need for publishers to have knowledge of their messages' recipients.

An issue with MOM is that, as no standard has been defined by any organisation, interoperability among different vendors' MOM products is limited or unavailable. Standards are, however, currently being established in distributed object middleware for MOM, such as the CORBA Notification and Event services and Java Messaging Service (JMS), which will be described briefly later.

2.5.3 Distributed Object Middleware (DOM)

DOM not only provides synchronous and asynchronous messaging protocols, but it also applies concepts from object-oriented software engineering to distributed computing. The essence of the object-oriented paradigm is that objects maintain state and communicate with other objects via message passing,

which can be incorporated into distributed computing paradigms. In addition, due to the encapsulation mechanism and reusability of object-oriented characteristic, distributed objects' location and implementation transparency can be achieved. Like RPC, DOM utilises an Interface Definition Language (IDL) to define a messaging contract between client and server. According to the interface defined for the target object, the IDL compiler generates code for marshalling (packing) data into standard format and unmarshalling (unpacking) data from the message streams exchanged between client and server. Depending on the IDL compiler used, code in different programming language can be generated. Hence, DOM can achieve programming language transparency. For example, a C++ implemented object in the client side can invoke methods of a Java remote object in the server side. On the client side, the Distributed Component Object Model (DCOM) uses the term proxy to represent this code, while the Common Object Request Broker Architecture (CORBA) uses the term stub. On the server side, DCOM uses the term stub; in CORBA, this is called the skeleton. According to this structure, the actual communication is as described in Figure 2.3.

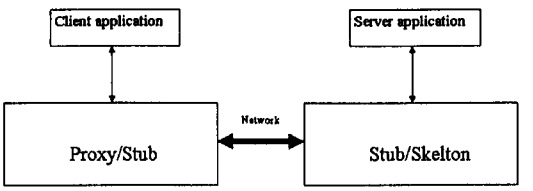


Figure 2.3 The DOM Structure

2.5.4 Middleware Standards

2.5.4.1 ONC and DCE

Open Network Computing (ONC) and Distributed Computing Environment (DCE) are the most broadly supported RPC middleware standards. Both ONC and DCE support point-to-point and broadcast mode. In point-to-point mode, the

request is sent to a specific server; in broadcast mode, the request is sent to a set of servers or all servers available in the same network. Users can select from the supported transport layer protocols to suit their requirements. One of the features of ONC is transport layer protocol-independence while DCE is oriented around the use of TCP and UDP transport-layer protocols.

2.5.4.2 DCOM

DCOM [Microsoft05] is a set of RPC-based extensions to Component Object Model (COM), a dominant component architecture developed by Microsoft to provide a language-independent standard mechanism for packaging program components. COM was originally designed to facilitate Object Linking and Embedding (OLE), which is a framework for assembling and managing compound documents. A compound document is a media-rich document which may contain, for example, a combination of text, images, video, audio and spreadsheet data. Recently, COM has become the core for a number of technologies developed by Microsoft, including, but not restricted to, OLE. DCOM allows COM objects to be distributed by providing a protocol called the Object Remote Procedure Call (ORPC). This protocol is built on top of DCE's RPC and interacts with COM's run-time services. A DCOM server is capable of hosting a number of objects at runtime. Each object utilises different interfaces to represent different functionalities. A DCOM client gains access to one of the server objects' interfaces by creating an interface pointer, which is used to reference the distributed object instances within the context of a client's programming language. According to the server object interface, clients can invoke methods on that object as if the remote object was residing in the same machine as the client.

2.5.4.3 CORBA

CORBA [OMG05] is the vendor-independent architecture designed by the Object Management Group (OMG) to facilitate communication between heterogeneous distributed computing environments. The flexibility of CORBA comes from its standardised internal communication protocols, GIOP/IIOP (General Inter-ORB Protocol/Internet Inter-ORB Protocol) [Ruh99], which are built on top of TCP/IP, allowing different vendors' ORBs to communicate with one-another. Furthermore, CORBA, through the use of Interface Definition Language (IDL), allows developers to define an interface to an object in a programming language-independent manner. An IDL compiler, which is provided by the ORB vendor, takes the IDL representation of an object and automatically generates the interface and additional supporting classes for the object in the desired programming language. Therefore, the implementation details of the object can be separated from the interface and hidden from the client. A client can invoke methods defined in the interface, as if it were a local object, regardless of the physical location of the object, the programming language the object was implemented in and the hardware/software characteristics of the machine the object resides upon. GIOP/IIOP and IDL provide a standard framework that enables interoperability between different vendors' ORBs.

As mentioned previously, although CORBA is a DOM standard, the MOM publish/subscribe model is implemented in the CORBA Notification and Event services [OMG05]. The Event Service is a primitive implementation of the publish/subscribe service, which allows publishers to put events into an event channel which subscribers receive. The subscribers and publishers are not required to have any knowledge of each other. A completely decoupled communication model is established through the utilisation of event channels. The Notification Service can be considered to be a mature extension of the Event Service. The Event Service itself provides no quality of service

monitoring or persistence. The Notification Service retains all the features of the event service, but provides additional support for content-based filtering and quality of service (QoS) monitoring; QoS properties such as reliability and priority can be used to indicate the delivery characteristics of events. Further details can be found on [OMG05].

2.5.4.4 JMS

The Java Message Service (JMS) [Sun05] defines a standard for MOM by combining Java Enterprise Edition (J2EE) technology with MOM. JMS provides a reliable, flexible service for the asynchronous exchange of data and events. The JMS API provides a common API and a framework that enables the development of portable, message-based applications in the Java programming language. Work is being undertaken to provide interoperability between JMS and other MOM, such as the CORBA Notification Service. However, although JMS is platform independent, it is a Java language-specific API. This limits the interoperability and flexibility of JMS.

2.5.5 Middleware Summary

A number of middleware standards are available. The synchronous messaging model results in the client process being blocked. As this will compromise scalability, it is necessary for a middleware solution for DVEs to support the asynchronous messaging model. In addition, as it is desirable for DVEs to be able to support heterogeneous networks and platforms, it is necessary for a middleware solution to be highly-interoperable. Table 2.3 shows a comparison between different middleware standards.

Middleware Standard	Middleware Model	Asynchronous Messaging	Vendor Interoperability
ONC and DCE	RPC	Limited	No
DCOM	DOM	Yes	No
CORBA	DOM	Yes	Yes
CORBA Notification/Event Service	DOM/MOM	Yes	Yes
JMS	MOM	Yes	Language-Specific

Table 2.3 Middleware Standard Comparison

From Table 2.3, it can be seen that DCOM, CORBA, the CORBA Notification/Event Service and JMS provide asynchronous messaging. DCOM is a vendor-specific Microsoft technology, which is not interoperable with other vendors' middleware. JMS is platform-independent, but language-specific. The CORBA Notification/Event Services are high-level concepts, whose performance varies depending on the CORBA vendor's implementation. In addition, the filtering and QoS monitoring in the Notification Service may result in delays in message delivery. However, CORBA offers interoperable asynchronous messaging with low-latency reliable delivery of messages, and therefore it fulfils the middleware requirements of a scalable DVE.

2.6 Interest Management

Recently, considerable research effort has been undertaken to scale up the number of users DVEs can support while maintaining mutually consistent views in real-time. A DVE scales up if thousands of geographically dispersed users can interact with each other simultaneously and the concurrent conflict of manipulating objects can be avoided. Imagine N physically scattered nodes sharing the same DVE, with each node receiving messages from the other $N-1$ nodes participating in the DVE at regular frequencies. Depending on the number of nodes and bandwidth limitation, the volume of messages transmitted

concurrently may overload the underlying network; transmission latency might be increased as a result of network congestion. Furthermore, network congestion may be so severe that the entire DVE system crashes as state update messages can not be delivered. It is worth mentioning that even if the network can survive the large amounts of message exchange, every node needs to cope with the other *N-1* nodes' messages which are transmitted at regular frequencies; a large proportion of which each node may not be interested in.

Past research [Abrams99] has shown that up to 99% of messages transmitted within a broadcast DVE are irrelevant to any participant. Moreover, if the processor allocates a large proportion of resources to deal with the unwanted messages, the other processes contributing to the DVE may be starved of resources, e.g. the process responding to user input, the process for rendering the DVE and the process for sending the host object's state update messages to other nodes. In this case, even though the messages from other participants' state are received on time, due to a lack of processing resources, a mutually consistent view cannot be maintained, which compromises the immersive experience for users. Mechanisms are required to scale up DVEs and maintain mutually consistent views in real-time. Interest management is one of these approaches developed to accomplish this.

It has been observed that although the size of a DVE is limited by the graphical designer's imagination, the interaction between participants and the virtual environment, and the interaction between individual participants, are restricted to certain degrees. For example, the computer screen restricts the visual perception of a participant sharing a DVE, while the audible perception of a participant is bounded by a certain degree of measurement, such as the distance between participants' avatars. Therefore, the interactions of participants can be limited to the boundaries of the avatars' audio-visual perception. Consequently it may not be necessary for participants to be aware of the existence of each other in a DVE if their avatars are not close enough to one. Unwanted messages

between participants can be filtered out without influencing the participants' perception of a DVE.

According to the description above, message filtering is an efficient method to improve the scalability, consistency and responsiveness of a DVE. Interest management is an approach to determine which messages are transmitted to and/or received by nodes based on some specified criteria, which, for the purpose of clarity, will be termed *Interest Expressions* (*IE*s). Different virtual environments have their own application-dependent constructions which can be exploited to define distinct *IE*s for their participants. If a participant's avatar, *A*, fulfils the requirements of the *IE* of another participant's avatar, *B*, *B* is said to be interested in *A* as *A* has fallen into *B*'s area of influence. Interest management is required to resolve the interests between objects based on the up-to-date messages transmitted between nodes.

There are several existing interest management approaches that have been suggested to provide the filtering capability for any DVE. These approaches can be categorised into region-based, aura-based and hybrid interest management. Further message filtering can be accomplished based on the participants particular *IE*s.

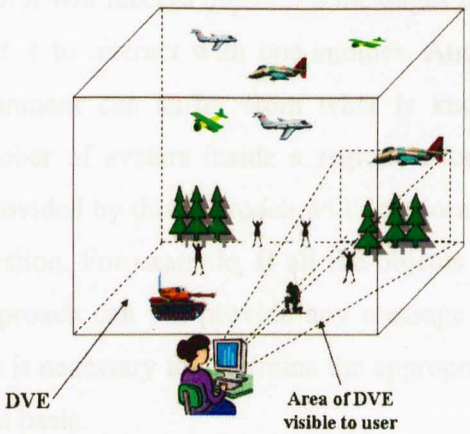


Figure 2.4 Proportion of DVE Visible to an Individual

2.6.1 Region-based Interest Management Approach

The region-based interest management approach divides a DVE into different regions. Region sizes are application-dependent and can vary from region to region. For example, if the virtual world is a library containing four floors: reception and computer cluster in the ground floor, biology books and meeting rooms in the first floor, computing science books in the second floor and other subjects and meeting rooms in the third floor. The designer of this virtual world can divide into four same size regions and each region is corresponding to one floor; or the designer can divide this virtual world into different size region on room basis. Each region is considered to represent the area of influence for all participants it contains. When an avatar enters a region, the avatar's IE will be all objects (other avatars, event-driven objects etc.) inside this region. The advantage of the region-based interest management approach is that it does not require detailed IE management/calculation; hence, the resolution of interests between objects consumes less CPU resources. One of the disadvantages of this approach is that, because it only provides rough message filtering, more messages may be received and processed than are needed. For example, object *A* and object *B* are inside the same region. However, as the size of this region is relatively large, object *A* will receive object *B*'s messages even if object *B* is too far away from object *A* to interact with one-another. Another disadvantage is that a virtual environment can suffer from what is known as crowding or clumping. If the number of avatars inside a region is too large, the message filtering capability provided by this approach will deteriorate to a level that may cause network congestion. For example, if all the objects in a DVE are in the same region, this approach can not provide any message filtering facility. To avoid this problem, it is necessary to determine the appropriate region size on an application-dependent basis.

2.6.2 Aura-based Interest Management Approach

Auras were described with respect to interest management in DVEs in the Spatial Model of Interaction [Benford94]. The aim of the Spatial Model of Interaction was to utilise the properties of space to mediate interaction between objects in a DVE. According to [Benford94], *Aura is defined to be a sub-space which effectively bounds the presence of an object within a given medium and which acts as an enabler of potential interaction.* In addition to aura, focus and nimbus are used to calculate the level of awareness between objects whose auras overlap. To be precise, the more the observed object, *A*, is within the observer object, *B*'s, focus, the more aware *B* is of *A*; the more *A* is within *B*'s nimbus, the more aware *A* is of *B*. When objects' auras overlap, message exchange between the objects occurs. Therefore, there is no need to regionalise a virtual world. However, there is a requirement for all nodes to exchange positional update information relating to the objects they host in order to identify when aura collision occurs. The frequency of message exchange must be sufficient to ensure that aura collision may be determined in a timely fashion to allow nodes to purposely disseminate messages as and when aura collisions occurs.

2.6.3 Hybrid Interest Management Approach

After describing the region-based and aura-based interest management approaches, the size of regions and auras directly affects the filtering capability of an interest management approach. Hybrid interest management approach utilises both regions and auras to divide the virtual world. Hybrid approach offers an optimisation compared with pure aura-based interest management, as the inclusion of regions reduces the computational complexity of performing aura collision detection. This is because it is only necessary to compare the auras of objects which share the same region, or are in neighbouring regions. This approach also offers an improvement in efficiency compared with pure region-based interest management, as it can reduce the amount of data required to be

transmitted within a given region. Depending on the application specification, the designer can choose the most suitable spatial division approach (region, aura or both) to filter the irrelevant information for the specific objects (avatars, static objects, etc.) in a DVE.

2.6.4 Missed Interactions Problem

In existing interest management approaches, messages are exchanged between nodes hosting objects within each-others' influence areas. However, these approaches are not designed for the situation in which objects coexisting in the same DVE have highly variable speed (e.g. foot solders and fighter aircraft). There may be a delay in resolving the membership of nodes participating in the same DVE and informing the relevant nodes of interactions; this delay may be sufficiently high that it is difficult or impossible for the interest management approach to guarantee that nodes will manifest interactions between high speed objects and their own. This is termed the *Missed Interaction Problem.*

In the region-based interest management approach, a message will only be received by nodes hosting objects which reside within the same, or neighbouring, region as the sender. Nodes hosting objects participating in the virtual world identify in which regions their objects belong and send messages to a well-known address (possibly a server, group of servers, or a group multicast address) that supports message dissemination for that particular region. Therefore, a region must be of sufficient size as to ensure objects have the ability to purposely disseminate messages in one region before entering another region. When an object traverses a region boundary a DVE is required to update region membership (identify which regions an object belongs to). If there is a possibility that an object can traverse a region in less time than it takes to realize regional membership changes then a node hosting such an object may be unable to disseminate messages effectively.

When considering an object that represents a fighter aircraft, the size of a region may be appropriately measured in kilometres due to both the speed of the object and the size of its field of vision. However, an appropriate region size for a much slower-moving object, such as a foot soldier, would be measured in perhaps tens or hundreds of meters. If region size was determined using the top speed of a fighter aircraft (Figure 2.5(i)), the presence of foot soldiers may result in unnecessary message exchange within a given region. In Figure 2.5(i), the dash line represents the underlying message dissemination between the node hosting object obj_a and all other nodes hosting objects in the region. The size of the region results in the soldier objects having to transmit messages between one-another, even though they are separated by distances too large for them to exert an influence on another. Additionally, large regions may contain a very large number of objects, potentially resulting in the network being congested or overloaded. Conversely, if region size is more suited to foot soldier objects (Figure 2.5(ii)), then a fighter aircraft may traverse region boundaries with such frequency that region membership may not be resolved in a timely fashion resulting in missed interactions. Therefore, when objects coexist within the same virtual world and can traverse the virtual world at greatly varying speeds, relying on a region-based approach alone may not be appropriate.

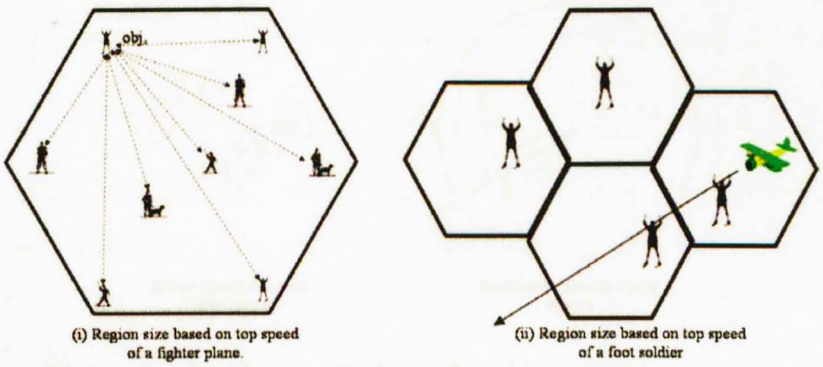


Figure 2.5 Problems with Region-Based Interest Management

In aura-based interest management, there is a requirement for all nodes to exchange positional update information relating to the objects they host in order

to identify when aura collisions occur. The frequency of message exchange must be sufficient to ensure that aura collision may be determined in a timely fashion to allow nodes to purposely disseminate messages as and when aura collisions occur. There is the possibility that aura collisions may occur but objects are unaware of this as such a collision may not exist for a sufficient amount of time to enable a DVE to update the group membership details before the objects move away from each other. Consider again the example of a fighter aircraft object and a foot soldier object. If the fighter aircraft flies over the foot soldier and initiates an attack on the soldier, the DVE must detect when aura collision occurs and enable message exchange between the appropriate objects. The aura of the fighter aircraft object may only collide with the aura of the foot soldier object for such a small period that it may not be possible to resolve the appropriate object group membership in a timely fashion. A missed interaction problem (Figure 2.6 (i)) may rise. A solution to this would be to extend the fighter aircraft's aura to enable such interaction. However, expanding the aura (Figure 2.6 (ii)) may result in the fighter aircraft potentially influencing many more objects than is necessary and may result in scalability problems as the node hosting the fighter aircraft would be required to participate in redundant message exchange with many nodes.

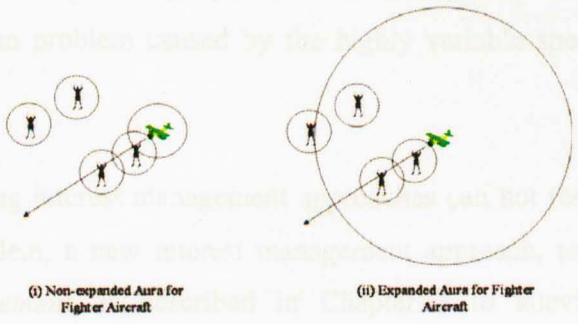


Figure 2.6 Problems with Aura-Based Interest Management

Although hybrid interest management combines the advantage of aura-based and region-based approaches, there is still an issue as to what region sizes are appropriate and the ability to determine aura collisions in a timely fashion. The

hybrid approach still suffers from many of the problems inherent in both region-based and aura-based interest management.

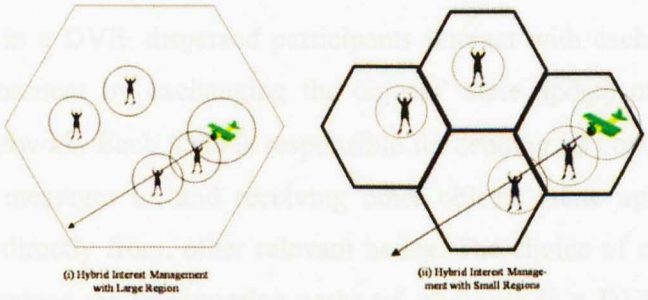


Figure 2.7 Problems with Hybrid Interest Management

The hybrid interest management approach is responsible for determining which regions objects should be in, meaning that high-speed objects may result in either large region sizes or difficulties in updating region memberships on time (Figure 2.7(ii)). Given that region-membership can be established in a timely fashion, this approach is still responsible for determining which objects' auras overlap within given regions. This can still suffer from high-speed objects, as aura collisions may not be determined quickly enough to invoke message exchange between nodes before their hosted objects have become disjoint again (Figure 2.7(i)). Consequentially, hybrid interest management can not address the missed interaction problem caused by the highly variable speed of objects in DVEs.

Since the existing interest management approaches can not resolve the missed interaction problem, a new interest management approach, termed *Predictive Interest Management*, is described in Chapter 3 to alleviate the missed interaction problem.

2.7 Communication Models

To reiterate, in a DVE, dispersed participants interact with each other and the virtual environment by exchanging the objects' state update messages in an underlying network. Each node is responsible for sending its controlled object's state update messages to, and receiving other objects' state update messages directly or indirectly from, other relevant nodes. The choice of communication models determines the transmission paths of messages in a DVE; therefore, it will influence the participants' immersive experience in different ways:

1 Consistency
2 Scalability
3 Responsiveness

The design of the communication model normally relies on the type of DVE being created, e.g. LAN-based simulation or Internet-based MMOG. Communication model issues include determining how the nodes communicate with one another, the overall geographic layout of a network and how it connects to other networks.

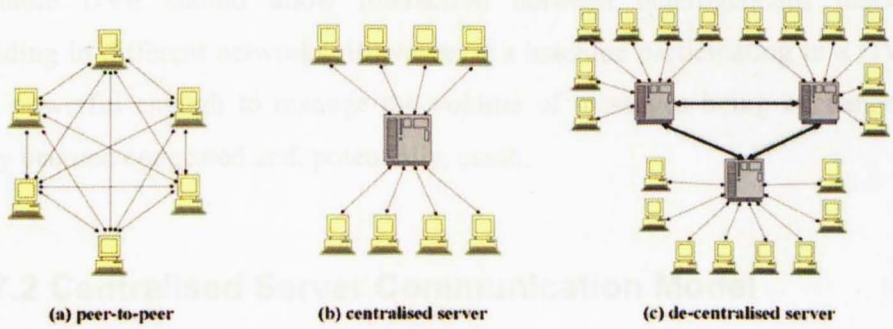


Figure 2.8 Communication Models

2.7.1 Peer-to-Peer Communication Model

The peer-to-peer communication model (Figure 2.8 (a)) involves direct communication between nodes on a network. Each node takes the responsibility of directly receiving messages from and sending messages to all other nodes in a DVE. In this kind of network, the network latency of message passing between nodes is minimal, as messages do not have to pass through any intermediate nodes. When a new node joins a DVE, it must communicate with every node currently participating in the DVE. However, if a DVE consists of N nodes, each node has to send to and receive from *N-1* nodes. As *N* becomes large, the computational expense of dealing with the number of messages being sent and received will greatly affect the performance of the DVE. This may result in a reduction in responsiveness (e.g. user input) as a large proportion of processor cycles must be dedicated to message management. Consistency may also be affected as messages may be held in an inbound message queue for a large amount of time before they are processed, resulting in large delays before object update states are displayed to users. Due to the large number of messages transmitted, the underlying network might be overloaded. Furthermore, a scalable DVE should allow interaction between heterogeneous machines residing in different networks. However, if a machine participating in a DVE is not powerful enough to manage the volume of messages being exchanged, it may become congested and, potentially, crash.

2.7.2 Centralised Server Communication Model

In the centralised server communication model, a dedicated machine, a DVE server, connects all the others client nodes in the network together. The physical distance between nodes in a DVE built on top of a centralised communication model is not important; the logical view can be displayed as Figure 2.8(b). In this kind of network, communication only occurs between the DVE server and

clients, there is no direct message passing between clients. Client nodes send their own object's state update messages to the server as well as receive other objects state updates information from the server. The server receives objects' state messages from clients, updates its internal representation of the objects states (e.g. object state table), and sends the required objects' state to the relevant clients. In the centralised server communication model, the concurrent issues of updating the shared objects state inside the virtual world can be handled easily using the lock mechanism. For example, the server issues a lock permitting the manipulation of the shared object state to one client and may either reject other clients' requests, or force other clients to wait, based on the first in first out (FIFO) order queue. After the client sends back a message to indicate the new state of the shared object with the lock, the server updates the object states and sends messages to all other clients to declare the new state of that object. If there are other clients waiting for the lock, the server will allocate the lock to the client with the earliest outstanding request. Consistency of the objects' state can be maintained satisfactorily.

However, compared with the peer-to-peer communication model, the centralised server communication model will introduce extra latency in message passing. In addition, when the number of clients increases, the server will receive a huge number of messages, which it must transmit to the relevant clients. Moreover, the server may have additional computational overheads involved in managing and maintaining a DVE, which a single machine may not be capable of performing in real-time as the number of participants increases. The single server is a bottleneck on the performance of the DVE system. The scalability of a DVE, in terms of the number of clients that can be supported, and complexity of the virtual world, in terms of the number of objects that populate the virtual world, will be limited by the centralised server communication model. Moreover, due to the deteriorated performance of the server, the frequency of the messages clients receive from the server will decrease. Therefore, when a client node receives an object state update message, the object's movement may

appear erratic to the user. The responsiveness of the virtual world will be damaged by the delayed message arrival, compromising the users' immersion.

2.7.3 De-centralised Server Communication Model

Unlike the centralised server communication model, a de-centralised server communication model utilises multiple DVE servers to bring all geographically dispersed clients together. The logical view of this architecture is displayed in Figure 2.8(c). Each server is responsible for a group of clients and the physical locations of clients are not important. Inter-communication between servers is required to transmit messages among relevant clients. Indeed, the de-centralised network can be viewed as a network that links different centralised networks together. Each server not only needs to manage receiving messages from and sending messages to the corresponding clients, but also needs to cope with the message passing between servers. Compared with the centralised server communication model, due to the utilisation of multiple servers, the number of clients and objects that can be supported by a DVE will increase, which, sequentially, improves the scalability of a DVE.

The de-centralised server communication model may introduce further message transmission latency when compared to the centralised server communication model, as messages may be routed through additional intermediate server before reaching their destination. However, as the number of messages which must be delivered increases, the message delivery speed in the de-centralised server architecture will be faster than the delivery speed exhibited by the centralised server architecture as the computational overhead of message dissemination will be distributed between the servers participating in a DVE. This will result in the faster update of objects' state in the client side, which will enhance the consistency and responsiveness of the DVE.

2.8 Related Work

The focus of research into DVEs has been divided into three distinct categories: military, academic and commercial DVEs. Although all three categories still receive a large amount of interest, the majority of investment received has shifted from military applications to commercial DVEs [Smed02]. Below is a brief description of some DVE systems from IEEE standards to commercial DVEs.

2.8.1 IEEE Standards

Distributed Interactive Simulation (DIS)

In 1983, the Defense Advanced Research Projects Agency (DARPA) sponsored the SIMNET (SIMulation NETworking) project to develop a "low-cost" distributed military endeavour for training small units to fight as a team. According to [Singhal99], there were three basic components in the SIMNET networking architecture: an object-event architecture, the notion of autonomous simulation nodes (simulator) and an embedded set of predictive modelling algorithms called "dead-reckoning". The object-event architecture modelled the virtual world as a collection of objects that interact with each other in the underlying network by message (event) passing; the notion of autonomous simulation nodes implies a node may control one or more objects in the virtual world and is responsible for sending messages related to its controlled objects and receiving messages from other objects; dead reckoning is a predictive algorithm to extrapolate objects' positions, based on previously received positional and motion data, designed to allow the reduction of message exchange frequency with minimal detrimental effect on the consistency of the simulation. A message describing the motion of an object is transmitted by an object's controlling node when the deviation between its actual position and the extrapolated position using the dead reckoning algorithm is greater than some

threshold. The receiving nodes will update the object's position and follow a convergence path for the object from its current position to the new position indicated by the update message.

Building on the successes of SIMNET, the need to connect heterogeneous distributed simulations together led to the development of a consistent framework. DIS, IEEE 1287 standard, was designed to link various types of distributed simulations to create highly interactive, realistic and complex virtual worlds. It was a platform-independent structure allowing heterogeneous machines to interact with one-another. In addition to the three basic components inherited from the SIMNET network architecture, DIS introduced the Protocol Data Unit (PDU) to provide a standard message structure for networked simulations and define the rule of issuing PDUs. There were 27 types PDUs defined by the IEEE 1287 standard, however, for most DIS-compliant simulations, only 4 PDUs were used by nodes to interact with each others.

High Level Architecture (HLA)

As DIS matured, the Department of Defence (DoD) was seeking an approach to reuse and interoperate the existing simulations in order to reduce the tremendous amount of resources expended on simulations. New simulations would only be built if no existing simulation model could satisfy the new requirements. The concept of HLA was born. According to [Kuhl99], HLA was built on the assumptions:

1. No single simulation can satisfy the requirements of all users.
2. No simulation developers can comprehend all simulated domains.
3. No one can predict the utilisation and combination of simulations.
4. Future technology and tools must be incorporated.

The HLA was a software architecture rather than a particular implementation or set of tools. In order to understand HLA, some basic concepts and the relationship between them are required to be described:

- Federate: a single simulation which is combined to form a simulation system.
- Federation: a simulation system which is created from a number of constituent federates. In addition to federates, each federation contained the Runtime Infrastructure (RTI) and Federation Object Model (FOM).
  - The RTI was software to support federates to execute together.
  - The FOM described the objects and interactions involved in the federation execution.

In a federation, federates can not communicate with each other without agreeing with a specified FOM. Before interactions occur between federates, each federate must convert its internal simulated entities to HLA objects as specified in the agreed FOM. After the simulated entities have been translated to HLA objects, federates interact with the RTI with the FOM format data; the RTI sends this data to federates which have the same FOM. In this case, under the HLA architecture, federates can subscribe to receive data from and publish data to certain federates through the RTI. Federates do not need to know of the existence of each other. All data transmission between federates are delivered through an underlying network in such a way that a federate sends FOM data to the RTI and it is the RTI's responsibility to distribute the data to the other federates with the identical FOM. The volume of data received by each federate will be reduced.

As mentioned previously, HLA was a software architecture designed to promote interoperability between different federates. Therefore, the simulations' designers are obliged to obey the HLA rules which govern how federates interact with one-another during a federation execution and describe the responsibilities of federates and federation designers. In addition, the FOM is application-dependent. If the original federates are used in a new federation, the original FOM must be extended to add new attributes. For example, a federate simulating a car is subscribed to and publishes to the car position data. Consider

that this federate is going to be used in a new federation, consisting of other federates, simulating different kinds of cars: race motorcycles, trucks and buses. In addition to the position attribute of those federates, race motorcycles may have speed attributes, trucks may have attributes to represent the number and type of goods being transported, buses may have an attribute for the number of passengers. If a federate updates the position and number of goods attributes in a truck, the original car simulator will be notified of the update of the positional attribute of the truck, but not the update of numbers of goods as the car simulator is not 'interested' in this additional information. Therefore the HLA architecture provides a meta-model for all FOMs, called Object Model Template (OMT) which prescribes the allowed structure of every FOM. Furthermore, the HLA design provides an interface specification for the interaction between federates and the RTI in order to avoid interference of the implementation of federates or the RTI change.

2.8.2 Military Research

SIMNET

In addition to the system design described in Section 2.8.1, SIMNET [Macedonia95] applied other network technologies to ameliorate scalability. The peer-to-peer communication model was exploited to allow simulators to exchange state update messages. Each simulator had its own copy of the world database and was responsible for maintaining its state. Ethernet multicasting is used to assign different multicast addresses to different exercises. No interest management approach was used in SIMNET; the main technology used to reduce the network traffic was dead-reckoning.

NPSNET (1, 2, 3, Stealth)

The NPSNET series were virtual environments, developed by Naval Postgraduate School, Monterey, CA to support military simulations in the US

Department of Defense (DoD). NPSNET-1 [Singhal99] was demonstrated at SIGGRAPH 1991 as part of the Tomorrows Realities Gallery on three workstations. Communication was implemented using an NPS-invented ASCII protocol for exchanging information between workstations. NPSNET-1 used no dead reckoning and transmitted update messages at frame rate. The simulation itself lacked any form of collision detection, drastically reducing its realism [Zyda93]. The lack of realism in the simulation of NPSNET-1 was one of the goals outlined in NPSNET-2. An additional goal in the development of NPSNET-2 was to provide compatibility with SIMNET, as the cost of SIMNET was prohibitively expensive ($350,000 per copy). NPSNET-3 [Zyda93_2] introduced separate threads for rendering and networking. This was intended to maintain the graphics display rate, regardless of any network communication. It used dead-reckoning to reduce the network traffic. NPSNET-Stealth [Singhal99] was a side-project built on NPSNET-1 which was intended to read SIMNET terrain databases and network protocols. It was operational in March 1993 and was the only workstation-based VE compatible with SIMNET. All the previously described NPSNET systems were built on the peer-to-peer communication model with no interest management.

NPSNET-IV

NPSNET-IV [Macedonia95] partitioned the virtual world into different fixed-sized two-dimensional (2D) hexagons. It was built on top of the peer-to-peer communication model and exploited multicasting to reduce the overhead of duplicate message transmission. Each hexagon had its own multicast group and each avatar's area of influence corresponded with the hexagon in which it resided. However, an avatar, in addition to subscribing to the hexagon in which it resided, was required to subscribe to the six neighbouring hexagons. When an avatar moved from one hexagon to another, according to the direction of movement, it un-subscribed from three hexagons but subscribes to a further three (Fig. 2.9). To clarify, the red hexagon is the hexagon in which an avatar (black dot) resides and must therefore be subscribed to. In addition, the yellow

hexagons are the hexagons which the avatar must unsubscribe from, and the purple hexagons are the hexagons the avatar is required to subscribe to. The avatar could send data to and receive data from the hexagon in which it resides and its six neighbouring hexagons. The subscription/un-subscription mechanism had the effect of decreasing latency caused by joining and leaving the multicast group associated with a hexagon. However, the disadvantage of this approach is that it was unsuitable for large number of objects residing in the same hexagon.

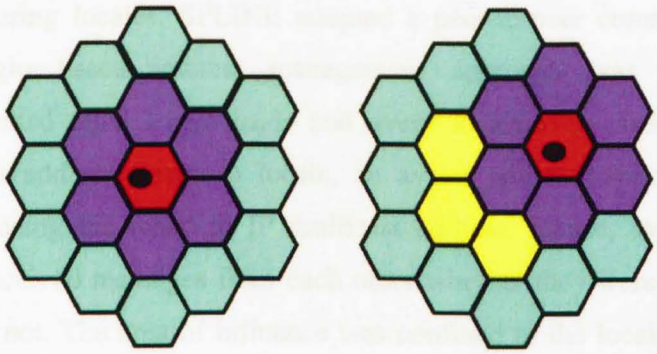


Figure 2.9 The Subscription/Un-subscription Mechanism

Avatar = black dot, red = residence hexagon,

yellow = unsubscribed hexagon, purple = subscribed hexagons.

2.8.3 Academic Research

RING

In RING [Funkhouser95], the virtual environment was partitioned into different size cells whose boundaries were comprised of the static, axis-aligned polygons of the virtual world. A visibility-based algorithm was implemented as an interest management approach to determine the cell-to-cell visibility. The de-centralised server communication model was adopted as the network structure. Every node was in charged of one entity, which had a geometric description and a behaviour, in the virtual environment. Each server acted as the message

arbitrator and redirector, based on the cell-to-cell visibility algorithm. Therefore, each node merely received messages from other nodes whose entities were inside cells which are visible from the cell its entity resided in.

SPLINE

SPLINE [Barrus96] used the concept of locales, which are essentially regions, to divide the virtual world. Different locales were linked together to construct the virtual environment. Every locale contained the network ID (IP address) of its neighbouring locales. SPLINE adopted a peer-to-peer communication model. The region-based interest management approach was adopted. It was implemented on a locale basis and every locale was associated with an IP multicast address. Inside a locale, an avatar could communicate with other avatars using the reserved IP multicast address. Hence, avatars in the same locale received messages from each other whether they were interested in each other or not. The area of influence was confined to the locale in which avatars were situated. When an avatar moved to a neighbouring locale, the avatar's information was sent to the other nodes occupying the locale. Although message passing was reduced, the locale-based approach could not cope with the situation of increased avatar density in the same locale. To illustrate, if thousands of avatars resided in the same locale, thousands of messages would be exchanged through an underlying network at the same time.

DIVE

In the Distributed Interactive Virtual Environment (DIVE) network software architecture [Frecon98], the virtual world was partitioned into smaller regions (worlds) with each node containing a duplicated database of the regions its avatar resided in. A world represented a separate virtual space, which was disjoint from other worlds, with its own set of objects, actors (avatars), and views. A gateway object, serving as a portal to another world, was used to connect the disjoint virtual worlds. Group communication, based on peer-to-peer multicasting, was implemented to exchange state updates inside the same world.

A multicast address was assigned to a world so that a user could send messages to or receive messages from other users inside the same world. However, a group based on a world experiences large granularity and density problems. One multicast address might not be enough to support the message exchange of all users in the same world. In order to reduce this granularity, a light-weight group was implemented as a sub-group of the world to multicast messages to users in the same light-weight group and its sub-groups. Users could join and leave a light-weight group in an application-dependent manner, rather than the DIVE system specifying the membership criteria. Therefore, the DIVE system provides more flexibility to DVE developers. To illustrate, an aura-based interest management schema can be deployed on top of the DIVE network architecture; an Aura Manager (server) can be used to detect the aura intersection of avatars in a certain region, or the aura collisions can be determined on a per-node basis. Additionally, the DVE developers can apply application-dependent mechanisms, based on region-based interest management, to divide a world into sub-regions.

MASSIVE-1

In MASSIVE-1 [Greenhalgh95], auras were the unit of interest management. The network architecture of MASSIVE-1 utilised a mixture of client-server and peer-to-peer communication models. The collision between objects' auras indicated message exchange between the relevant nodes might occur. The size of every aura could be altered to simulate situations in the real world. Like DIVE, MASSIVE-1 used gateways as portals to connect different worlds or different locations within the same world. Upon entering the virtual world, a user (client) contacted the relevant local aura manger (server) and declared its avatar's information (e.g. different auras sizes and shapes). When a collision between objects' auras was detected, the responsible local aura manager sent the relevant interface reference (IP address) to inform the involved users that their avatars had collided; therefore the users built a peer-to-peer communication channel independently inside the same world. Message exchange was controlled

by the calculation of the mutual levels of awareness through the focus and nimbus.

MASSIVE-2

MASSIVE-2 [Greenhalgh96] inherited the awareness calculation of MASSIVE-1. However, scalability was enhanced through the introduction of third party objects, which were defined as an extension to the spatial model. In addition, MASSIVE-2 implemented dynamic hierarchies of multicast groups as the network architecture, instead of peer-to-peer connection as in MASSIVE-1. A third party object was an independent object (which may be a spatially-defined region) that affected the awareness of other objects. A third party object was represented by a group in MASSIVE-2. Each group was associated with one or more multicast address for distributing state and update messages for different media. A virtual world was mapped onto a nested hierarchy of group objects so that multicast groups could be organised in a flexible and dynamic manner. The receiving member of a group could determine the awareness level of the transmitting member of the same group through the awareness calculations, but the parent group could only receive the aggregate view (an abstract view) of the sub-group. For example, in a virtual world, ten passengers inside a train can hear, see and talk to each other. The train acts as the third party object. People external to the train merely perceive shadows inside the train as it passes by. The communication between the passengers and their exact number are hidden from the non-passengers. MASSIVE-2 supported full interaction with the contents of the virtual world through a third party object.

MASSIVE-3

MASSIVE-3 [Greenhalgh00] extended the concept of locales from the SPLINE system to subdivide a virtual world. In addition to locales, MASSIVE-3 introduced another concept, aspects, as the sub-division of a local. Every locale contained a base aspect, which specified all links to neighbouring locales and all

links to the other aspects of that locale. Aspects were the fundamental units of interest management in MASSIVE-3. Each aspect corresponded with an environment database. Each environment database was fully replicated on each application which was interested in it (i.e. each node maintained a local copy of its environment's database). The decision as to which aspects/locales should be replicated could be selected using a number of techniques, for example, replicate all aspects/locales within a given distance of the client's current locale.

Each aspect could be assigned a cost value. This could be used to restrict either rendering or replication to being below a certain cost. The client-server network architecture was adopted in MASSIVE-3 using unicast TCP/IP, as TCP/IP provides flow and congestion control at the network level, which allowed support for heterogeneous computers and networks; this was one of the design purposes of MASSIVE-3. MASSIVE-3 used a separate thread for sending messages. Messages/events which were to be sent were placed in an outbound queue. As such, as TCP/IP manages flow control, the sender could determine it had a slow receiver as its outbound queue would grow. At this point, the sender could apply some application-level adaptation to reduce the amount of communication required. Considerable effort was put into MASSIVE-3's consistency management, which attempted to ensure that shared object updates were perceived in the same order on all nodes participating in the DVE. Three different kinds of consistency schema (ownership transfer, centralised update and CIAO-style updates) were available in MASSIVE-3. These were user-selectable so that the appropriate schema could be chosen for the available network connection.

PARADISE

PARADISE [Singhal99], the Performance ARchitecture for Advanced Distributed Interactive Simulation Environments, was initiated in 1993 in the Distributed Systems group of Stamford University. The purpose of the system was to reduce the bandwidth used within DVEs at the time. It used IP multicast,

assigning a different multicast address to each active object. However, as the early graphics-capable workstations available to the group did not support hardware multicast, an application-level multicast simulator was used on a LAN. Hosts transmitted updates for local objects in a similar manner to SIMNET and DIS. Additionally, a hierarchy of area of interest (AOI) servers collected information subscriptions from each host. The servers monitored the positions of objects and notified hosts about which objects' multicast groups they should subscribe to.

PARADISE, differing from SIMNET, treated all objects, including the terrain, as first-class entities capable of transmitting state updates. PARADISE also attempted to optimise its message exchange schema to recognise that objects need to transmit state updates at different frequencies: rapidly-changing objects should transmit state updates more frequently than slowly-changing objects. To support rapidly-changing objects, an improved Dead Reckoning algorithm, called Position History-Based Dead Reckoning (PHBDR) was developed, which transmitted smaller update packets and was more accurate when objects move wildly: the situation which caused DIS to transmit state updates at near frame-rate. PARADISE supported multiple independent communication flows per object, with each flow enabling differing level of accuracy remote dead-reckoning. Additionally, PARADISE also provided a form of message aggregation, combining information about groups of objects based on both their virtual position and type.

In order to better deal with slowly changing objects, PARADISE attempted to use reliable multicast protocols to eliminate the frequent "heartbeat" messages present in DIS. Log-Based Received-Reliable Multicast provided a lightweight reliable multicast service that included a persistence mechanism. This allowed nodes joining an existing DVE to retrieve the current state of slowly changing objects directly from a system of logging servers.

BrickNet

BrickNet [Singh95] [Singhal99] was a software toolkit to support graphical, behavioural and network modelling of virtual worlds. It intended to accomplish the virtual collaboration required in design or learning applications. Each client subscribed to a server in a DVE, which could not be dynamically changed during the lifetime of the DVE. Servers were implemented to mediate the communication and requests for shared objects between clients. If a client expressed an interest in a shared object in its own virtual world, it requested the subscribed server for the ownership of the shared object. If the shared object was owned by another client, the server would put the request into a queue until all the previous requests have been performed. The shared object update information was sent to other relevant clients through servers.

NetEffect

NetEffect [Das97] was intended to provide highly scalable media-rich distributed virtual worlds which could support several hundreds or thousands of participants. A DVE in NetEffect was hosted on a number of servers. NetEffect subdivided the virtual world into communities, with each community representing some physical space in the virtual world, such as a room. A server may host one or more communities. A client connected to the corresponding server depending on which community it wishes to be part of. The client maintained its connection to this server until it wished to leave its current community and join another, at which point it may be required to disconnect from its current server and connect to the server hosting its new community. Communication was based on the "need-to-know" principal, in that communication occurs only within communities. As such, there was no need for inter-server communication as a community was hosted on a single server. As a user moved within the DVE, it transfers between different communities, which resulted in the client machine disconnecting from and connecting to servers dynamically. NetEffect allowed point-to-point audio communication, which

involved two clients directly connecting to one-another and transmitting an audio-stream to avoid forming bottlenecks at the servers. To further reduce bottlenecks at servers, NetEffect periodically checked the density of clients in each community. The system dynamically transferred some communities from heavily loaded servers to those with lower loads in an attempt at load-balancing.

VNet

VNet [Singhal99] was a client/server Java-based DVE which used VRML. It was one of the first networked VRML-based worlds. It used rudimentary spatial interest management and provided both graphical representation and textual chat. Textual chat was displayed between all objects located within 30 meters of each-other. Communication occured using an application-level protocol called VRML Interchange Protocol (VIP) built on top of TCP/IP, which was used to send VRML field change information across the Internet.

Mercury

Mercury [Bharambe02] [Bharambe04] was a completely distributed content-based publish-subscribe protocol proposed for use in DVEs. It used a high-level subscription language to express the content of the publications and subscriptions a node makes. This language was essentially a subset of SQL. Mercury attempted to avoid the bottlenecks associated with the centralised server communication model, while avoiding the scalability restrictions of broadcast-based DVEs. It endeavoured to yield good scalability by distributing the responsibility of matching game events to player interests. Mercury divided this responsibility among nodes by partitioning them into groups, called attribute hubs. Each attribute hub was in charge of a special attribute in the overall schema. For example, the virtual world can be divided by its dimensions (e.g. x, y); two attribute hubs can be created, in charge of the x-coordinates and y-coordinates respectively. Inside each attribute hub, the coordinate can be subdivided into ranges based on the number of nodes in that attribute hub. Nodes inside an attribute hub were logically arranged in a circle and connected

to each other using successor and predecessor pointers. Subscriptions were routed through one of the attribute hubs and stored in one or more nodes in the hub, termed rendezvous points. Publications were routed through every attribute hub. These publications were then routed through all rendezvous points within this hub until all the subscriptions to this publication were fulfilled. If subscriptions and publications matched, publications were delivered to nodes which issued the subscriptions.

ATLAS

ATLAS [Lee02] provided the DVE developers with a network framework. It supported region-based, aura-based, specific user interests and inter-region-based interest management to filter the unnecessary messages in various DVE applications. Each region and shared object was assigned multicast addresses. A server subscribed to each region's multicast address in order to maintain the membership of participants in each region and the state of the virtual world. ATLAS supported two network architectures: client/server and peer/server. The peer/server architecture allowed consistency management to be done by the server and communication among users to be performed directly by themselves using multicast. However, as multicasting is not fully supported over the Internet, the client/server structure was provided for real-world deployment.

MOVE

MOVE [Lopez02], Multi-user Oriented Virtual Environments, was a client/server publish/subscribe 3D collaborative environment. It was mainly deemed for educational purposes, such as virtual classrooms. In MOVE, each region of the virtual world, for example a classroom, was called a "Place". A portal object was used to link different places together. MOVE was built on top of a component groupware framework, called ANTS, which used JMS/Elvin notification service to handle message transmission through the underlying network. Aura-based interest management was adopted in the server as a filtering approach to reduce the message receipt from other relevant clients.

OpenPING

OpenPING [Okanda05], Open Platform for Interactive Networked Games, was a reflective middleware supporting the development of adaptive networked games. Rather than focusing on the scalability, persistency and responsiveness of DVEs, this middleware platform addressed flexibility, maintainability and extensibility. OpenPING used the notion of computational reflection, whereby a system has a self-representation, or meta-representation, to enable itself to adapt to a changing platform environment. OpenPING provided a plug-in framework which could support interest management. It also provided three event channels: reliable, Application Level Framing (ALF) and unreliable. This allowed the appropriate event channel to be selected according to the changing states of the application and platform environment.

2.8.4 Commercial

2.8.4.1 First-Person Shooter

First-Person shooter (FPS) games are an extremely popular computer games genre. These games are played from a first-person perspective, in which the player's view of the virtual world is perceived through the eyes of the game's character. These games have evolved from single-player games to real-time world-wide multiplayer experiences. FPS games usually have a short duration and allow a relatively small number of players (usually not more than 32) to play against one-another in a single game.

Doom

The earliest quintessential networked game, the first-person-shooter (FPS), to become widely popular was Doom [Id05] [Sweeney99]. Doom was released in two forms: a shareware version containing the first chapter of the game and the

network game was released on the Internet, whereas the full version of the game, containing two more chapters and additional multiplayer maps, was available to paying customers. The networked component of the game was very restricted. A fixed number of players could enter a dungeon and compete to see who could kill the others the most: a death-match game. The game would only allow players to join before the dungeon was initiated. Once the game was underway, no new players could join, and no player could leave without destroying the game. Doom performed its networking using either TCP/IP or UDP/IP – the choice was up to the network administrator who installed the game. Both networking versions operated using the peer-to-peer communication model. They synchronized their inputs and timings with one-another, and performed the exact same calculations on the exact same inputs. Due to this synchronization property, the chance of network-induced failure increased linearly with the number of players. Additionally, because the timings and frames were synchronized, it was difficult to support a wide variety of machine speeds.

Quake

Due to the restrictions of peer-to-peer communications, the next generation of networked computer games replicated the client/server model. In Quake [Id05] [Sweeney99], developed by Id Software, the server made all game-play decisions. The machines connected to the server were regarded as dumb rendering terminals. Their responsibility was to forward keystrokes to the server and wait for a response to instruct them as to how to update their view of the virtual world. This architecture enabled widespread Internet gaming to become a reality, as game servers sprang up all across the Internet. However, the responsiveness of the game was directly related to the distance, and therefore the network delay, between the client and the server, offering unfair advantages to players who were situated close to the server.

Quake II

Following from the success of Quake, the client/server model was extended with Quakeworld and Quake II [Id05] [Sweeney99]. Predictive logic was introduced into the client to lessen the burden on the servers hosting games. Technologies like dead reckoning, a position prediction, extrapolation and convergence algorithm, were introduced to reduce the perceived lag to users. Although the dead reckoning was rudimentary, using only linear interpolation, it had a tremendous effect on the smoothness of the animation for users. Also, due to the shift of some calculation from the server to the client, the bandwidth requirements of the game were reduced, allowing the game and servers to handle more clients simultaneously.

Unreal Tournament

Further developments were achieved with later FPS games, such as Unreal Tournament [Epic05] [Sweeney99] by Epic Games. Unlike the FPS games which came before it, Unreal Tournament provided very little single-player content; instead it provides a game which was designed from the ground-up as an Internet FPS. Unreal Tournament was based on the client/server model. The Unreal Tournament engine introduced an object-oriented scripting language to express object and avatar (termed actor) behaviour, and networking requirements, which decoupled application code from networking code. The scripting language was quite sophisticated, allowing the creation of user-defined object types. It provided the ability to transmit both reliable and unreliable messages, using TCP/IP and UDP/IP respectively. The scripting language allowed the calling of remote procedure calls, termed function call replication, to allow programmer-defined communication to be achieved without hard-wiring the code to a specific network protocol.

In Unreal Tournament, the server was viewed as being the authoritative game state. The problem of ensuring that the clients view a consistent world with the

server is termed state replication. Due to bandwidth limitations, it was not possible to replicate the state in the server on the clients in real-time. As such, an approximation to the server's game state was provided by the clients. Essentially, the clients executed the same operations on the game objects as the server, albeit on variables which may deviate from those stored on the server. The different aspects of the game state were prioritised, with a larger proportion of bandwidth dedicated to A/I-controlled objects (termed bots) and actors than superficial animations in the game. The developers of the Unreal engine recognised that not all actors need to be replicated on all clients at any given time. Interest management is adopted. Therefore, Unreal Tournament could support heterogeneous platform over a network. Additionally, not all state variables within the server need to be replicated. For example, the state variables used by the server to control bots need not be replicated on the client machines. Essentially, the server maintained a huge set of variables and function calls, of which only a fraction needed to be replicated on the clients to provide a decent approximation to the server's game state.

2.8.4.2 Massive Multiplayer Online Role Play Game

Massively Multiplayer Online Role Play Games (MMORPGs) are a highly popular games genre. These games usually take place in a fantasy world in which the player creates an online representation of themselves, an avatar. The players interact and join forces with one-another to fight hordes of monsters. Through fighting monsters, the player's avatar receives experience which causes their character's statistics to improve. The player usually collects virtual money, which can be used to purchase equipment such as weapons and armor to improve their character's abilities. These games encourage social groups to be formed, termed guilds, between players with common interests. As the purpose of MMORPGs is to improve the player's character, a game's duration is unbounded, providing the illusion of a continuous pervasive virtual environment.

Diablo

Diablo [Money97] [Kuo01] [Blizzard05] was an online Role Play Game (RPG) published by Blizzard Entertainment in 1997. It was immensely successful, selling over 2.1 million copies. In Diablo, each player is required to build his/her own character, which they strengthen and develop through countless hours of play.

In order to increase the message transmission speed and the character update speed, the peer-to-peer communication model was adopted in Diablo, storing character information on each player's computer. There are two disadvantage of this approach. Firstly, although the network latency associated with the peer-to-peer architecture is comparatively low, this approach is not scalable, which is shown by the fact that the game could only support parties of up to 4 players. Secondly, players could alter the characters stored on their computer using tools freely available on the Internet, making them much more powerful. Additionally, as messages were sent between players' computers directly, it was possible for dishonest players to modify the game themselves to alter the messages which were sent to the other players. For example, players were able to send a message which would result in every other player's characters instantaneously dying. The rampant cheating which occurred in Diablo damaged the confidence and enjoyment of the honest players. As such, in the inevitable sequel, Blizzard had to address the problem of cheating in order to win back the players' confidence. No interest management is implemented in Diablo.

Diablo II

Diablo II [Money97] [Kuo01] [Blizzard05] was designed to address the cheating problem and released by Blizzard entertainment at June 2000. Blizzard provided two modes in the Diablo II to satisfy different players: the peer-to-peer model as in Diablo and the secure client-server model in which players' characters were stored on secure servers, termed "The Realms". In the client-server model

[Ng02], each server maintains a unique game state, which is not shared among other servers. No inter-server communication is implemented in Diablo II.

Players could either play with other players via the Realms, in what was intended to be a cheat-free environment, or they could play via the peer-to-peer architecture, which was just as susceptible to cheating as the original Diablo. However, characters hosted on the Realms could not be used for peer-to-peer gaming and, similarly, locally-hosted characters could not be used to play with people via the Realms. In addition to the increased security, Blizzard increased the maximum party size from 4 to 8 players. In order to maintain the cheat-free property of the Realms, Blizzard was forced to release update patches for Diablo II every 4-6 weeks, which players were required to download and install before they could continue to play. These patches not only fixed any known bugs or security holes, but often contained additional content or improvements to the game. However, the Realms became a high-profile target for hackers. In December 2000, just 6 months after its release, the Realms were hacked. Players' items were stolen and many of the top players' characters were deleted. As with the original game, no interest management was implemented in Diablo II.

Ultima Online

Ultima Online [Ultima97] [Origin05] was released by Origin Systems in September 1997. The client/server communication model was adapted to support thousands of players simultaneously on various game servers, known as "shards". Each shard acts as a central repository for the state of the game and all game logic is executed on the shards. Each player is required to send "commands" relating to their character (walk, take objects, fight) to the server. Failing to send a command in time may cause the death of the character, its injury or other types of damages. Therefore, players with high-latency or low-bandwidth Internet connections may be more disadvantaged than their faster opponents. To reduce the bandwidth usage and improve the performance of the

server, Ultima Online divides the game world into multiple continuous zones [Ng02], whereby information is only provided to a player about objects within the player's zone. Region-based interest management is implemented in Ultima Online. Different zones may be hosted on different shards. To avoid player perceiving delays when a player traverses across neighbouring zones, and therefore connects to a new shard, neighbouring shards mirror boundary content.

EverQuest

EverQuest [EverQuest99][Kushner05] was a MMORPG released by Sony Online Entertainment in November 1999. The client/server model was implemented as the ground of the networking architecture. EverQuest divided the entire virtual world into distinct zones. Each individual zone was maintained by a game server. A player may connect to any zone (a game server) to join the game and may travel from one zone to another freely.

However, this structure became too inefficient to handle the new content the game developers were releasing. Therefore, Sony utilised a new technique to manage the computing resource, termed just-in-time computing. In the just-in-time system, the computer resources were allocated dynamically based on player demand. For example, a player is running through a corridor when they come to a door. By opening that door, the player triggers actions on several machines. If, say, a dungeon lies behind the door, Sony's system looks up the data and software that describe that dungeon on one computer, finds some idle processing resources within its cluster of dedicated servers, probably on a different machine, and runs the software on it. Sony's servers download the dungeon's data to the player's computer "just in time" to meet the user's requirements.

2.9 Summary

A Distributed Virtual Environment (DVE) is a virtual environment which allows dispersed participants to navigate through and interact with the virtual world in a distributed manner. The general properties of a DVE (shared virtual environment, virtual objects, interaction and navigation, and distributed users) and the challenges (bandwidth, network latency, heterogeneity, and consistency and responsiveness) to build a scalable DVE were briefly discussed in Section 2.1 and Section 2.2 respectively.

In general, a DVE is constructed from three different layers: the application layer, the message dissemination layer and the network layer. The application layer presents users with a graphical representation of the DVE allowing users to interact with the virtual world and other users through input/output devices; the message dissemination layer may provide services to ease access to the network layer for developers, to overcome heterogeneity between nodes and networks, and to regulate message passing to better utilise available bandwidth; the network layer provides appropriate network protocols for the DVE developers. Due to the different requirement of different types of DVEs, such as military, CSCW and MMOG, the message dissemination layer and the network layer should provide the appropriate services and network protocols. Five popular network protocols are described in Section 2.4.3. As for the message dissemination layer, middleware and interest management are suitable for integration into this layer regardless of the types of DVEs.

Middleware shields the developers from the complication of low-level networking, providing platform and language independence. Three types of middleware (RPC, MOM and DOM) and five standards (ONE, DCE, DCOM, CORBA and JMS) of middleware were discussed. It was found that, although the CORBA Notification and Event Services, which are the MOM

implementation in CORBA, provides interoperable, reliable, asynchronous messaging, these services are not suitable for use in scalable DVEs as current implementations are not able to support the volume of message exchange required. Furthermore, additional message delivery delays may be introduced due to the advanced features of the Notification Service, such as QoS monitoring. Therefore, CORBA is most suitable as it provides low-latency, interoperable, reliable, asynchronous messaging.

Interest management is a message filtering technology for reducing message exchange through an underlying network without compromising the users' immersive experience in DVEs. Region-based interest management divides the virtual world into different regions; users inside the same or neighbouring region can exchange information through an underlying network. Aura-based interest management provides each object an area of influence; objects can exchange information when their areas of influence overlap with each other. Hybrid interest management is the combination of the region-based and aura-based interest management. However, existing interest management approaches are not guaranteed to resolve objects' interests sufficiently quickly to initiate message exchange before the objects' interaction has ended. If the duration of a pair of objects' interaction is short, it is possible that the objects' interaction will have ceased before the interest management scheme detected it; this situation is termed a *Missed Interaction*.

The choice of communication models will influence consistency, scalability and responsiveness of DVEs. It was found that the peer-to-peer model offered weak scalability, but was capable of delivering messages fastest. However, it was likely to lead to the largest bandwidth usage; increasing the number of participants may lead to network overloading, which consequentially affects the consistency and responsiveness of a DVE. The centralised server model was found to offer better scalability and reduce the network bandwidth consumption. However, a centralised server could become a bottleneck if the number of

participants increased, which in turn affects the consistency and responsiveness of a DVE. In addition, a central server is a single point of failure. The de-centralised server model was found to offer best scalability. Each server is potentially less likely to become a bottleneck than in the centralised server model. However, this model provides these features at the cost of the highest message transmission latency. Compared with the peer-to-peer and centralised server models, this model offers better scalability, consistency and responsiveness as the number of participants increase.

Subsequently, a broad selection of related work was discussed in Section 2.8, including two IEEE standards, six military simulation systems, fourteen academic research systems and eight online computer games. Some of these systems (e.g. NPSNET-1,2,3, Bricknet and Doom) did not implements interest management at all; other systems (NPSNET-IV, DIVE and EverQuest) have implemented various interest management approaches. However, all of these existed systems do not provide interoperability and are susceptible to missed interactions.

Chapter 3

Predictive Interest Management

This chapter provides a description of Predictive Interest Management (*PIM*), an aura-based interest management approach for alleviating the missed interaction problem in DVEs.

3.1 Overview of Technique

The missed interaction problem, which was described in Chapter 2, occurs in DVEs if objects have highly variable speed (e.g. foot solders and fighter aircraft). The delay in resolving the required message exchange of nodes participating in the DVE, and informing the relevant nodes an interaction has occurred, may be sufficiently high that it is difficult or impossible for existing interest management systems to guarantee that nodes will manifest interactions before the interactions have ceased.

If an interest management approach can inform the relevant nodes that an interaction between their hosted objects may potentially occur, this will, theoretically, solve the missed interaction problem. Predictive Interest Management (*PIM*) is such an aura-based interest management approach. The reason for utilising auras instead of regions is that aura-based interest

management provides a much more accurate expression of the interests and interactions within a DVE than region-based interest management. *PIM* uses Predicted Areas of Influence (*PAIs*) to enlarge objects' auras such that future interactions between auras can be detected so that the relevant nodes can be informed on time. However, as discussed in Chapter 2, enlarged auras may lead to unnecessary message exchange between nodes. Therefore, a Collision Window (*CW*) and its associated values (see Section 3.2.3) are used in *PIM* to regulate the message exchange frequency between nodes such that the message exchange frequency is proportional to the distance between the objects, i.e. message exchange frequency will increase as objects are more closer to each other.

PIM utilises three kinds of messages in its message exchange schema which are transmitted at different frequencies: Position Update Message (*PUM*), Admin Position Update Message ($APUM_{admin}$) and Local Admin Position Update Message ($APUM_{local}$). The *PAI* and *CW* are used to determine which types of messages should be exchanged between nodes and, in the case of $APUM_{local}$ messages, the frequency at which the messages are exchanged.

3.2 Calculations

The definition of *PAI* and *CW* is based on three assumptions. In this section, these three assumptions will be discussed in details before describing the *PAI* and *CW*.

3.2.1 Assumptions

Assumption 1: An aura of an object describes a virtual space enclosed by a sphere. The radius of an aura is specified on a per-object basis and is defined at object creation time, with each object having a single aura and the position

vector of an object identifying its aura's centre. Objects have the ability to influence each other when their auras collide. This influence is exerted via the exchange of messages between the nodes hosting these objects.

Assumption 2: Each object has a constant highest speed. The reason for defining a constant highest speed is that a *PAI* can be defined at object-creation time such that an object can exert its influence over other objects that fall into this area of influence.

Assumption 3: The future time, used to calculate a *PAI*, is measured under the hypothesis that objects travel toward each other in a straight line. This assumption is based on the Triangle Inequality Theorem [Lengyel02] (Figure 3.1 & Theorem 3.1), which proves that the sum of any two sides of a triangle is longer than the third side using the vector property shown in Theorem 3.1; and therefore that the shortest distance between two points is a straight line. It follows that, therefore, the earliest possible collision between two objects occurs when the two objects move in a straight line towards one-another at maximum speed. Although it is likely that the objects will not have collided at this time (e.g. one or both objects change their velocity), it offers a practical estimated time that ensures that potential collisions are not missed. The velocity of an object is a vector $(v_1, v_2, ..., v_n)$ which can represent both the direction and the speed the object is travelling in. In three-dimensional space, this would be represented by a three-dimensional vector **v** (v_x, v_y, v_z).

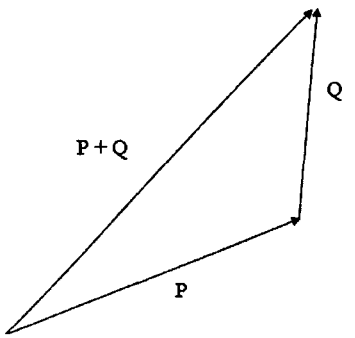


Figure 3.1 The Triangle Inequality Theorem

Theorem 3.1. Given any two vectors **P** and **Q:**

$$\|\mathbf{P} + \mathbf{Q}\| \leq \|\mathbf{P}\| + \|\mathbf{Q}\|$$

Lemma 3.1 *Cauchy-Schwartz Inequality*

$$|\mathbf{P}\cdot\mathbf{Q}| \leq \|\mathbf{P}\| \, \|\mathbf{Q}\|$$

Note:

(1)$P\cdot Q$ is dot product of vector $P(p_x, p_y, p_z)$ and $Q(q_x, q_y, q_z)$, which results in the scalar:

$P\cdot Q = p_x q_x + p_y q_y + p_z q_z$

(2) $||P||$ represent the magnitude of vector $P(p_x, p_y, p_z)$, which is the scalar:

$$\| P \| = \sqrt{p_x^2 + p_y^2 + p_z^2}$$

Proof of Theorem 3.1 (Triangle Inequality)

Given that $\|\mathbf{P}+\mathbf{Q}\|$ represents the magnitude of the vector P+Q:

$$\|\mathbf{P} + \mathbf{Q}\|^2 = (\mathbf{P} + \mathbf{Q})\cdot(\mathbf{P} + \mathbf{Q})$$

$$= \mathbf{P}^2 + \mathbf{Q}^2 + 2\,\mathbf{P}\cdot\mathbf{Q}$$

Using Lemma 3.1 to attain an inequality

$$\leq \mathbf{P}^2 + \mathbf{Q}^2 + 2\,\|\mathbf{P}\|\,\|\mathbf{Q}\|$$

$$= (\|\mathbf{P}\| + \|\mathbf{Q}\|)^2$$

Taking Square Roots arrives at the desired result

3.2.2 Predicted Area of Influence

A Predicted Area of Influence (*PAI*) identifies the extent of an object's aura, given the maximum distance the object may travel in a straight line in any direction (Figure 3.2) over a given period of time.

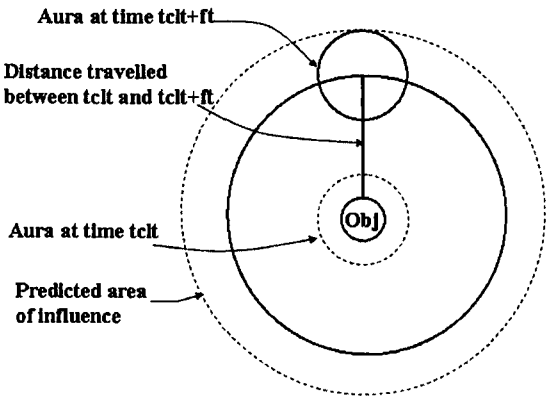


Figure 3.2 Defining Predicted Area of Influence (*PAI*)

The period used to identify a *PAI* is bounded by the current local node time, say *tclt*, and some future time (*tclt+ft*, where *ft* is a constant, defined system-wide). By this method, the distance that an object, say obj_1, travels in a straight line identifies the radius of a sphere that encloses all the areas of virtual space reachable by obj_1 between *tclt* and *tclt+ft*, with the position vector of obj_1 at time *tclt* defining the centre of this sphere. Extending this radius by the radius of obj_1's aura defines a sphere that describes the *PAI* for obj_1. When determining a *PAI*, an object is modelled travelling at its highest speed, V_m, in a straight line at time *tclt* and continues at this speed and direction until *tclt+ft*. This presents a *PAI* that is guaranteed to contain the aura of an object for all possible movements this object can make between *tclt* and *tclt+ft*. Assuming the highest speed remains constant for an object throughout its lifetime, a *PAI* can be calculated and fixed at object-creation time. The radius of *PAI* can be calculated using the formula below:

$$Radius(PAI) = Radius(Aura) + ft * V_m$$

Formula 3.1 PAI Calculation

However, the use of *PAI*s may cause scalability problems, as the node hosting an object may be required to participate in redundant high fidelity message

exchange with many other nodes. As a result, the node may be overloaded by the superfluous messages thereby detrimentally affecting the users' immersive experience due to slow input/output response of the DVE. Therefore, it is necessary that the redundant messages received by each node be limited to some level to reduce the impact of potential scalability problems. In the next section, *Collision Windows (CWs)* are introduced to calculate the message exchange frequencies between objects to provide a scalable solution to the missed interaction problem.

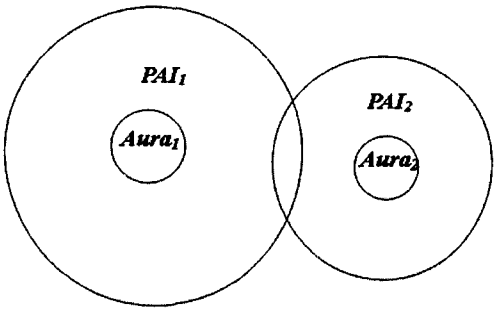
3.2.3 Collision Window

When the *PAIs* of two objects collide, but their auras do not, there is a possibility that such objects may influence each other and subsequently exchange messages at some point in the near future. A *Collision Window (CW)* is defined as a period of time within which the auras of two objects may collide. However, the establishment of a *CW* does not guarantee that an aura collision will occur. Once a *CW* has been established, there are three values which must be considered: *Upper Bound Value (UBV)*, *Optimistic Upper Bound Value (OUBV)* and *Approximate Upper Bound Value (AUBV).*

Upper Bound Value (UBV) Determination

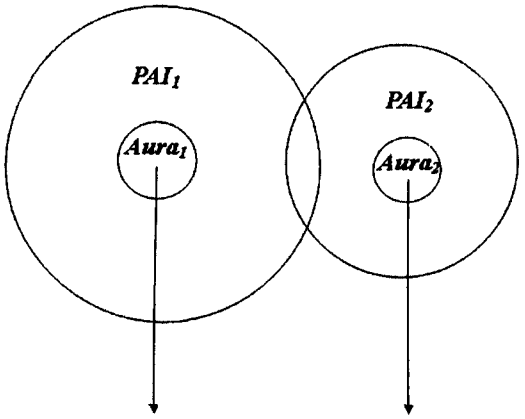
The *UBV* of a collision window is infinity (Figure 3.3), as both objects may:

- Remain stationary



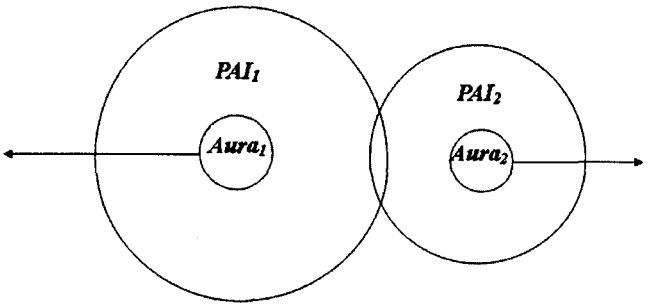
Stationary Objects

- Move in parallel to one-another



Parallel Movement Objects

- Move in opposite to one-another



Opposite Movement Objects

- Cross each other's path without intersecting each other

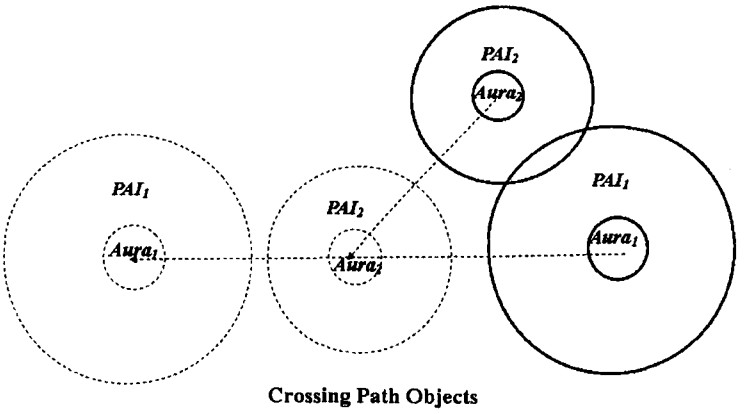


Figure 3.3 Infinity *UBV* Collision Windows

Optimistic Upper Bound Value (OUBV) Determination

Assuming two objects are travelling towards each other in a straight line at their respective top speeds provides an *OUBV*, which is the shortest time for two objects' auras to collide. According to the definition of *PAI*, at the instant a *CW* is established, *OUBV* is the time *ft* (Figure 3.4). Additionally, because *ft* is defined globally, the *OUBV* of each object pair is the same.

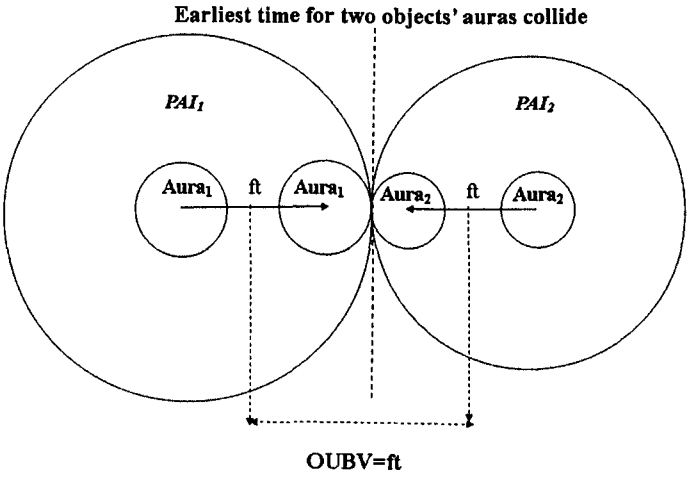


Figure 3.4 Defining OUBV

Using collision detection techniques based on the intersection of spheres, it is possible to identify if a *CW* exists between two objects. This technique is computationally cheap compared to collision detection between polygons, as we only need to determine if the distance (*sd*) that separates the objects is less than the sum of the radii of the *PAI*s associated with the objects. The formula below can be applied to determine whether a *CW* exists between obj_a and obj_b.

$$Radius(PAI_a) + Radius(PAI_b) \leq sd_{ab}$$

Formula 3.2 CW Determination

Once a *CW* has been established between two objects, further calculations can be performed to determine whether these two objects' auras collide with each other. Collision detection techniques can be applied to identify whether the auras overlap, i.e. the distance that separates two objects is less than the sum of the radii of each object's aura.

$$Radius(Aura_a) + Radius(Aura_b) \leq sd_{ab}$$

Formula 3.3 Aura Overlap Determination

Approximate Upper Bound Value (AUBV) Determination

If a *CW* exists but the objects' auras do not overlap (Figure 3.5), an *Approximate Upper Bound Value (AUBV)* is required to be calculated in order to predict the minimum time taken for the auras of these objects to collide. The assumption that these objects are modelled travelling towards each other in a straight line at their maximum speeds is applied.

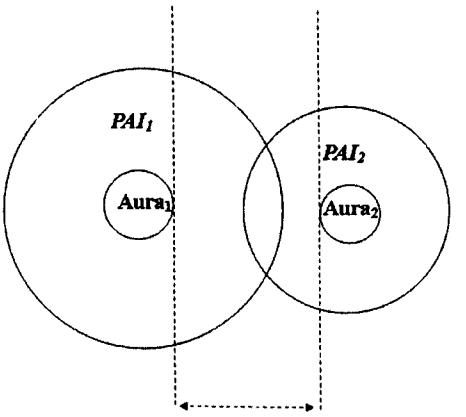


Figure 3.5 CW Exists but Auras Do Not Overlap

AUBV may be derived by first calculating the distance between the edges of the two objects' auras and dividing it by the sum of the two objects' maximum speeds.

$$AUBV = \frac{sd_{ab} - (Radius(Aura_a) + Radius(Aura_b))}{V_{ma} + V_{mb}}$$

Formula 3.4 (a) AUBV Calculation

Formula 3.4 (a) is the most straightforward way to calculate the *AUBV*. However, According to Figure 3.4, when two objects' *PAIs* touch, $V_{ma}+V_{mb}$ can be re-written as:

$$V_{ma} + V_{mb} = \frac{(Radius(PAI_a) + Radius(PAI_b)) - (Radius(Aura_a) + Radius(Aura_b))}{ft}$$

Therefore, *AUBV* can be represented using the formula:

$$AUBV = \frac{sd_{ab} - (Radius(Aura_a) + Radius(Aura_b))}{(Radius(PAI_a) + Radius(PAI_b)) - (Radius(Aura_a) + Radius(Aura_b))} * ft$$

Formula 3.4 (b) AUBV Calculation

Also, according to the Formula 3.4 (b), a relationship formula can be constructed as below:

$$sd_{ab} \leq Radius(PAI_a) + Radius(PAI_b) \Rightarrow AUBV \leq ft \Rightarrow AUBV \leq OUBV$$

Formula 3.5 Relationship between AUBV and OUBV

AUBV provides a basis for predicting the appropriate frequency for message exchange between two objects within a *CW* before their auras overlap. If two objects move towards one-another, sd_{ab} will decrease causing *AUBV* to become smaller; conversely, if the objects move away from one-another, sd_{ab} will increase causing *AUBV* to become larger. This variable frequency of message exchange between nodes reduces the impact of redundant messages overloading the underlying network and compromising the users' immersive experience in DVEs.

3.3 Message Exchange Scheme

With respect to a pair of *PAIs* and a *CW*, there are three different situations that may occur between a pair of objects in a DVE which should result in different message exchange frequencies:

(1) *CW* does not exist ⇒ *PAIs* do not overlap ⇒ lowest message exchange frequency should be applied to the respective nodes or the mediator.

(2) *CW* exists but auras do not overlap ⇒ *PAIs* overlap but auras do not ⇒ higher message exchange frequency should be applied to the respective nodes or the mediator.

(3) CW exists and auras overlap ⇒ *PAIs* and auras overlap ⇒ highest message exchange frequency should be applied to the respective nodes or the mediator.

According to the descriptions of the situations above, three types of messages with different exchange frequencies are defined in Section 3.3.1. Two message channels are introduced in Section 3.3.2 as the transmission media for nodes or the mediator to exchange different frequency messages. In Section 3.3.3, according to the objects' intersection degrees, these objects may be required to subscribe or unsubscribe from each other's corresponding message channels and receive the appropriate frequency of messages.

3.3.1 Message Types

Three different kinds of messages are introduced as the basis of the *PIM* message exchange scheme:

1. Positional Update Messages (*PUMs*)
   PUMs are messages transmitted at a regular frequency, which construct the basic mechanism for high fidelity message exchange in a DVE. Each *PUM* identifies the position vector of the objects that a node hosts and carries the unique identifier of the node that sent it. This message may be extended to include additional information, such as orientation, velocity, acceleration etc.

2. Local Admin Positional Update Messages ($APUM_{local}$)
   $APUM_{local}$ carry the unique identifier of the node and the position vector of the residing objects. It is exchanged between the relevant nodes at a variable frequency. This is the core message designed to alleviate the missed interaction problem in DVEs.

3. Admin Positional Update Messages ($APUM_{admin}$)
   Compared with *PUM* and $APUM_{local}$, $APUM_{admin}$ are exchanged at the lowest frequency. $APUM_{admin}$ contain the aura radius, *PAI* radius and vector position information for all the objects a node hosts and the unique identifier of that node. Essentially, $APUM_{admin}$ contains the

information required by *PIM* to determine the recipients and frequencies of *PUM* and $APUM_{local}$.

3.3.2 Message Channels

In the message exchange scheme, the notion of message channels is applied for disseminating messages. A message channel is a medium for delivering certain types of messages to the appropriate recipients. A message channel may be registered with the *PIM* message exchange scheme, allowing publishers/senders to place messages on message channels. Subscribers/receivers register to one or more message channels and receive messages placed on these message channels. Two message channels are defined in the message exchange schema:

- **Admin Channel (*AC*)** – Used to disseminate $APUM_{admin}$ to all nodes. All nodes subscribe and publish to this message channel.
- **Local Channel (*LC*)** – Created on a per-node basis to provide a mechanism for passing $APUM_{local}$ and *PUM* between nodes without the need to publish such messages to all nodes.

Each node must register with the *AC* to send/receive $APUM_{admin}$ to/from other nodes at a consistent low frequency. This ensures that the interactions between uninterested objects can be re-estimated at a given, low frequency to initiate the appropriate message exchange between nodes as and when their objects become interested in one-another. For example, the objects hosted on nodes n_a and n_b respectively are not interested in each other at time t_0. Under the *PIM* message exchange schema, this will be represented as the *PAIs* of these objects not overlapping at time t_0. However, because the objects in a DVE are dynamic, it is difficult to guarantee after a certain time, say t_1, that the objects controlled by n_a and n_b are not interested in each other (i.e. *PAIs* overlap or auras overlap). The time interval used to define the $APUM_{admin}$ frequency may be appropriately measured as a percentage of ft. The reason for this is that it insures $APUM_{admin}$

are exchanged during the predicted period (*ft*) so that the required message exchange between nodes can be determined on time.

Each node is associated with two different lists in the *LC*: the *PUM* list (L_{pum}) and the $APUM_{local}$ list (L_{apum}). Each list contains the set of nodes currently subscribed to a node's *PUM* or $APUM_{local}$ messages and the frequency to exchange these messages respectively. To illustrate, given that n is the set of all nodes subscribed to the DVE, and $n_i \in n$. $L_{pum}^{\ i}$ and $L_{apum}^{\ i}$ represent the *PUM* list and the $APUM_{local}$ list for node n_i. $L_{pum}^{\ 0} = \{n_1^{0.5}, n_2^{0.5}\}$ indicates that nodes n_1 and n_2 are registered to receive *PUM* messages from node n_0 with the message exchange frequency at 0.5 second; $L_{apum}^{\ 0} = \{n_3^{1}, n_8^{1.5}, n_{10}^{2.2}\}$ specifies that nodes n_3, n_8 and n_{10} are registered to $APUM_{local}$ messages from node n_0 with the message exchange frequency at 1, 1.5 and 2.2 second respectively.

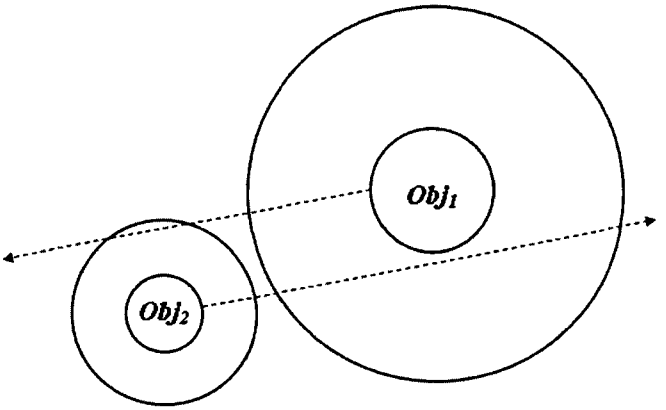
3.3.3 Message Channel Subscription Policy

Nodes are not compulsorily subscribed to other nodes' L_{pum} and L_{apum}. However, if nodes do subscribe to each other in the *LC*, they subscribe to either each other's L_{pum} or L_{apum} exclusively. The rationale for this is that when nodes subscribe to each other's *LC*, one of the following situations will occur:

- The *PAIs* of objects belonging to the relevant nodes overlap, but the auras do not.
- The auras of objects belonging to the relevant nodes overlap.

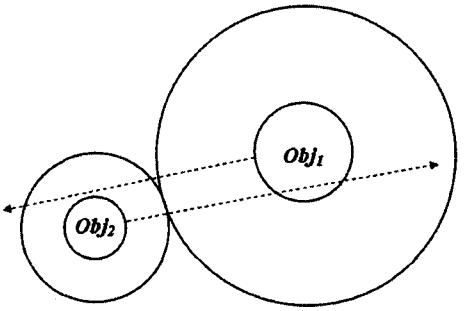
The following example demonstrates how the different message types are utilised in *PIM*. As previously mentioned, every node must be constantly subscribed to the *AC*, regardless of their subscriptions to any *LC*. However, for the clarity of this example, only the highest-frequency message type subscribed to is indicated in each individual diagram in Figure 3.6.

1. CW does not exist

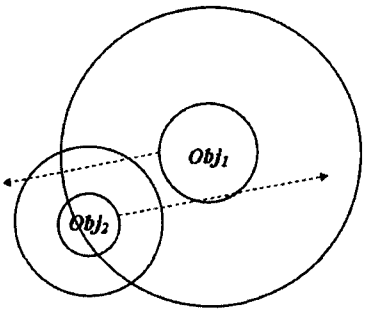


(1) $APUM_{admin}$ Exchange

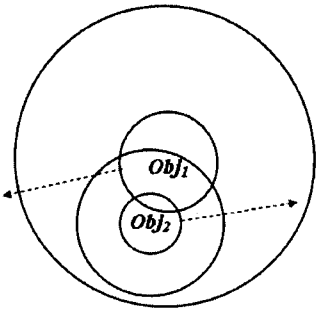
2. CW exists



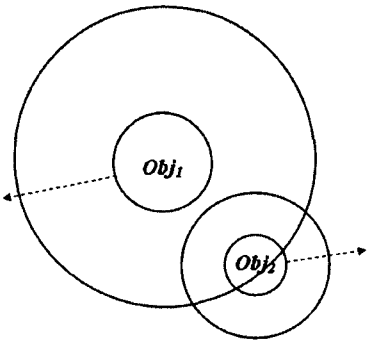
(2) $APUM_{local}$ Exchange Starts



(3) $APUM_{local}$ Exchange

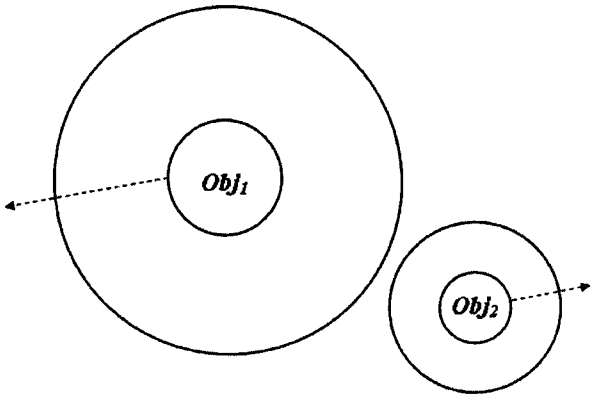


(4) PUM Exchange



(5) $APUM_{local}$ Exchange

3. CW ceases to exist



(6) $APUM_{admin}$ Exchange

Figure 3.6 The *PIM* Message Exchange Schema

To clarify, obj_1 and obj_2 are two objects controlled by nodes n_a and n_b respectively. As can be seen from (1) in Figure 3.6, before the *PAIs* of obj_1 and obj_2 overlap (and therefore the *CW* does not exist), the messages exchanged between nodes n_a and n_b are $APUM_{admin}$ in the *AC* only. In (2), when *PIM* detects these two objects' *PAIs* overlap (*CW* exists), $APUM_{local}$ exchange starts between nodes n_a and n_b. Nodes n_a and n_b should subscribe to each other's L_{apum}. In (3), as the distance between these two objects reduces, the frequency of exchanging $APUM_{local}$ is higher than the exchange frequency in (2). In (4), obj_1 and obj_2 move close enough so that their auras overlap; n_a and n_b should unsubscribe from each other's L_{apum} and subscribe to each other's L_{pum}. Therefore, the message type exchanged between these nodes changes from $APUM_{local}$ to *PUM*. The frequency of message exchange will change from a variable lower frequency ($APUM_{local}$) to a constant high frequency (*PUM*). In (5), both objects move far away from each other. Their auras no longer overlap, but *PAIs* are still intersecting; the membership of L_{pum} and L_{apum} should be exchanged. Nodes n_a and n_b should unsubscribe from each other's L_{pum} and instead subscribe to L_{apum}. The $APUM_{local}$ exchange frequency decreases as both objects' distance increase. In (6), both objects' *PAIs* are no longer overlap (*CW*

does not exist), nodes n_a and n_b unsubscribe from each others' L_{apum} and the messages exchanged between them are only $APUM_{admin}$.

In the previous example, according to Formula 3.4 & 3.5, two considerations must be made. As previously discussed, the transmission frequency of $APUM_{admin}$ is a proportion of ft. As such, it is possible for $AUBV$ to be greater than the frequency of transmitting $APUM_{admin}$. In this case, it is not necessary to transmit $APUM_{local}$ messages as $APUM_{admin}$ will be sufficient for the PIM message exchange schema to determine the required messages exchange between nodes. In addition, it is possible for $AUBV$ to be less than the frequency of PUM transmission. In this case, nodes will unsubscribe from each others' $APUM_{local}$ list and subscribe to each others' PUM list.

The frequency of exchanging $APUM_{local}$ for a pair of nodes is based on the calculation of the highest exchange frequency among their respective controlled objects. To demonstrate, given obj_{a1} and obj_{a2} are controlled by node n_a, whereas obj_{b1}, obj_{b2} and obj_{b3} are controlled by node n_b. The $PAIs$ of object pairs (obj_{a1}, obj_{b2}) and (obj_{a2}, obj_{b1}) overlap but their auras do not. Using $AUBV_m^n$ to represent the AUBV of obj_m and obj_n, $AUBV_{a1}^{b3}$=2 and $AUBV_{a2}^{b1}$=3, therefore, the $APUM_{local}$ exchange frequency between nodes n_a and n_b is 2 second per $APUM_{local}$.

The previous frequency definition example is based on one-to-one node situation. However, in DVEs, it is necessary to consider the one-to-many nodes case.

L_{apum} Node	Node Subscribed	Transmission Frequency
n_1	n_2	2
	n_3	3
	n_6	5
	n_{19}	7
n_2	n_1	2
	n_3	4
	n_8	5
n_3	n_1	3
	n_2	4
	n_{16}	8

Table 3.1 L_{apum} Subscriptions

As can be seen from Table 3.1, nodes n_2, n_3, n_6 and n_{19} subscribed to node n_1's $APUM_{local}$ list; this is represented as $L_{apum}{}^{1} = \{n_2{}^{2}, n_3{}^{3}, n_6{}^{5}, n_{19}{}^{7}\}$. Accordingly, the $APUM_{local}$ lists related to nodes n_2 and n_3 correspond to $L_{apum}{}^{2} = \{n_1{}^{2}, n_3{}^{4}, n_8{}^{5}\}$ and $L_{apum}{}^{3} = \{n_1{}^{3}, n_2{}^{4}, n_{16}{}^{8}\}$ respectively. Therefore, the frequency for node n_1 to publish $APUM_{local}$ is the lowest $AUBV$ (2 seconds per $APUM_{local}$). The reason for this is that if another $AUBV$ was used (e.g. 5 instead of 2 seconds per $APUM_{local}$), nodes n_2 and n_3 would not receive $APUM_{local}$ from n_1 at a high-enough frequency to guarantee accurate aura prediction. Therefore, in the case of multiple nodes subscribed to a node's L_{apum}, the lowest $AUBV$ should be selected as the node's $APUM_{local}$ publish frequency in the LC.

3.4 Summary

This chapter described Predictive Interest Management (*PIM*), which is aura-based interest management approach to alleviate the missed interaction problem. In *PIM*, a Predictive Area of Influence (*PAI*), which is an enlarged aura, is used to include all the possible movements of an aura over a predefined future time. A Collision Window (*CW*), which is defined as a period of time within which the auras of a pair of objects may collide, and its associate values *UBV*, *OUBV*

and *AUBV* were introduced to regulate the message exchange type and frequency between nodes.

PIM's message exchange scheme was discussed. Three types of messages (Positional Update Message (*PUM*), Local Admin Position Update Message ($APUM_{local}$) and Admin Position Update Message ($APUM_{admin}$)) and two message channels (the Admin Channel (*AC*) and the Local Channel (*LC*)) were defined in the message exchange scheme. To elaborate, given two objects hosted on different nodes:

(1) CW does not exist ⇒ *PAIs* do not overlap ⇒ the nodes are subscribed to the *AC* only.

(2) CW exists

   a. *PAIs* overlap but auras do not ⇒ the nodes should subscribe to each other's $APUM_{local}$ list (L_{apum}) in the *LC*; Approximate Upper Bound Value (*AUBV*) between the hosted pair of objects should be calculated to provide the appropriate exchange frequency for $APUM_{local}$ messages in the *LC;*

   b. Auras overlap ⇒ the nodes should unsubscribe from each other's L_{apum} in the *LC* and subscribe to each other's *PUM* lists (L_{pum});

   c. Auras no longer overlap but *PAIs* still do ⇒ the nodes should unsubscribe from each other's L_{pum} in the *LC* and subscribe to each other's L_{apum}.

(3) *CW* ceases to exist ⇒ the hosted objects have passed by each other ⇒ *PAIs* no longer overlap ⇒ the nodes should unsubscribe from each other's *LC*.

To clarify, *PIM* can be proved to solve the missed interaction problem provided network latency is bounded below some threshold value, i.e. all messages will be received by their respective recipients within this threshold time. This property can be leveraged by *PIM* by ensuring that the value of *future time* (*ft*) used to calculate an object's Predicted Area of Influence (*PAI*) is greater than or

equal to this threshold value. Given that all messages will be received within *ft*, *PIM* can guarantee to avoid any missed interactions. However, this property does not exist in real-world networks, e.g. the Internet. Network transmission delays are often relatively uniform but can, as a result of network congestion or failure, become infinitely large. As such, *PIM* can not be proven to solve the missed interaction problem over the Internet, but can be said to alleviate it.

Chapter 4

System Implementation

This chapter describes the design and development of the *PIM* system. Prior to the description of the *PIM* system implementation, some related development issues are introduced. The interaction models, which include the server/server and the server/node models, are introduced to describe the message exchange in the *PIM* system; the development technologies, which include descriptions of Java and CORBA are discussed. Following, the design and implementation of a *PIM* server, which is constructed from five components (Message Service Servant, Message Buffer Unit, Thread Pool Processing Unit, PIM Processing Unit, Message Supplier), and the system exceptions are described in detail.

4.1 Development Issues

The *PIM* system, which is an experimental system, adopts the de-centralised server communication model, middleware to satisfy the networking requirements and predictive interest management to improve bandwidth utilisation and alleviate the missed interaction problem. In this section, the de-centralised server communication model is further decomposed into two interaction models: the server/server and server/node interaction models. The development technologies applied in the *PIM* system (Java as the system

implementation language and CORBA as the middleware to handle the networking issues) are discussed.

4.1.1 Interaction Models

In the de-centralised server communication model, which was discussed in Chapter 2, the server that a node is subscribed to, and the location of nodes and servers, is not important. Nodes could subscribe to the logically or geographically closest server.

- **Logical Server**

  A server, implemented as a logical server, controls a certain region in a virtual world but may be located physically far away from the nodes sharing its region of the virtual world. When an object is detected inside a server's region, the node, to which the object belongs, should subscribe to that server regardless of the physical distance between the server and the node.

- **Physical server**

  A server, implemented as the physical server, should be subscribed to by nodes for which it is the geographically closest server. For example, nodes locate in the UK should subscribed to a UK server; nodes locate in America should subscribed to an American server. This kind of server is responsible for the whole virtual world, rather than a specific region in a DVE. It updates the status of objects which belong to nodes subscribed to it, and disseminate these objects' information to the other DVE servers.

The choice of server implementation depends on the system design and purpose. This decision may involve a number of application-dependent criteria. However, in both case, servers are responsible for receiving messages from and sending the relevant messages to their subscribed nodes.

The *PIM* system adopts the physical server model. The reason for this is that dynamic subscription (grouping) is time consuming and, therefore, it might be impractical to implement logical server for distributed virtual environments with variable speed objects. To clarify, if a server is implemented as a logical server, the virtual world is divided into different regions or the virtual world is constructed from a set of disjoint smaller virtual worlds, termed sub-worlds. Each server is responsible for the interaction between objects inside the same region/sub-world. In this case, nodes which control objects in a DVE can be considered as a member of a group which share the same environment. Because objects can leave and join a region/sub-world, the membership of that region/sub-world may change frequently. The delay associated with leaving, joining and informing other nodes in the same region/sub-world makes the logical server implementation unsuitable to solve the missed interaction problem. Conversely, in the physical server implementation, the membership of nodes in a server remains consistent unless existing nodes unsubscribe from, or new nodes subscribe to the server. This property makes the physical server implementation an appropriate approach for the *PIM* system.

Based on the physical server implementation, two interaction models are provided in the *PIM* system: the server/node and server/server interaction models. In the server/node interaction model, messages from nodes are sent to their server at a constant time interval; the required messages are transmitted from a server to its subscribed nodes at an appropriate frequency. In the server/server interaction model, servers are required to subscribe to each other when they join the *PIM* system via a web server, which acts as a naming service server providing the other registered servers' network information, e.g. IP address, upon request. Servers can subscribe to and unsubscribe from each other during the lifetime of a DVE. Each server forwards its subscribed nodes' messages to the other servers whose subscribed nodes are interested in these

messages. For clarification, the server/node interaction model and the server/server interaction model are presented in Figure 4.1.

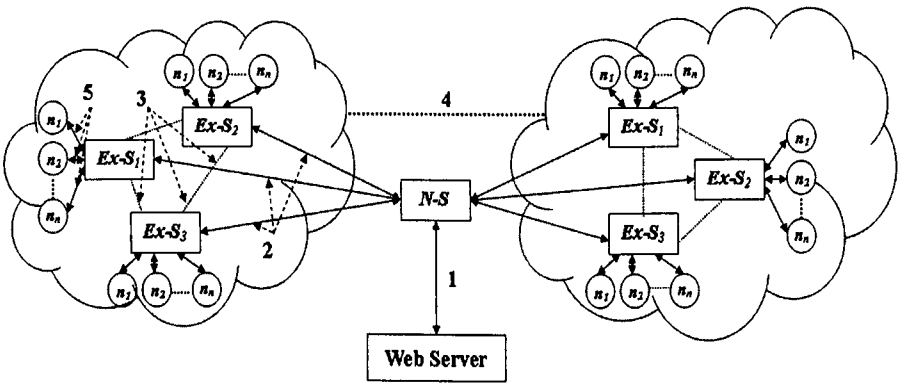


Figure 4.1 Server/Server and Server/Node Interaction Models

Ex-S_i= an existing server i, N-S=a joining server

The server/server interaction model:

- A new server joins the existing *PIM* system:
  1. N-S server registers itself to the web server and requests the existing servers' network information.
  2. N-S server subscribes to the other existing servers. In response, the existing servers subscribe to N-S server.

- Internal servers message exchange:
  3. Messages with fixed/variable frequency exchanged between existing servers residing in the same network.
  4. Messages with fixed/variable frequency exchanged between existing servers residing in different networks.

The server/node interaction model:

  5. Message exchange between nodes and their subscribed server.

4.1.2 Development Technologies

This section justifies the choice of Java as the implementation language and CORBA as the middleware to handle networking issues.

Java

Java was chosen as the implementation language of the *PIM* system. According to [Reilly99], Java is a revolutionary language which combines object-oriented programming, portability, garbage collection and is specially designed for networking and the Internet.

- Object-oriented language

  Programming languages can be divided into two types: procedural languages and object-oriented languages. In procedural languages, like C and Pascal, each procedure is a block of code which can have data passed to it and can return results. In this case, data and code are separated. This feature of the procedural languages makes it hard for the debugger to track down which procedure has gone wrong. Java is an object-oriented programming language, in which each object contains both data (variables) and code (methods). Each object is an instance of a class, which defines the variables and methods the object contains. Compared with procedural languages, putting data and code in the same class facilitates abstraction making it easier to implement and debug programs.

- Portability

  Many programming language, like C++ and C, are compiled into platform-specific machine language. Therefore, the source code of the same program is required to be re-compiled for each individual platform, often requiring platform-specific alterations to the code. Although the

execution speed of such programs is fast, this property restricts the portability of these programming languages. This also raises questions of the future use of such applications, as many modern operating systems struggle to execute legacy applications without the use of emulators. Conversely, interpreted programming languages are not compiled at all. Instead, they are evaluated by a run-time environment in real-time during execution. Programs in such languages are generally slower than compiled programs, but they are truly platform independent. In Java, the source code of a program is compiled into platform-neutral virtual machine code, called Java bytecode. Running on an interpreter, called the Java Virtual Machine (JVM), Java can convert the bytecode into machine code directly. Unlike purely interpreted languages, Java bytecode can be efficiently converted into machine instructions in real-time, providing performance faster than purely interpreted languages. Therefore, provided the target platform supports the JVM, the source code can be compiled once and run anywhere. Java is an evolving language, in which a number of new features are added in each new release. However, Java is not arbitrarily extended and backwards compatibility is guaranteed, as part of the JVM specification, such that Java programs written to previous Java specifications must be able to execute correctly on later JVM release. This property makes Java suitable for heterogeneous distributed network environments.

- Garbage collection

  In some programming languages, like C++ and C, the programmer is required to manage the memory usage of the program. Programmers are required to allocate and de-allocate memory for data and objects manually. When programmers forget to de-allocate memory, the amount of free memory available will decrease. This problem is called a memory leak. In Java, the JVM handles the memory allocation and de-allocation for the programmer, called automatic garbage collection. This frees the

programmer from the concern of avoiding memory leaks, thus allowing the developers to concentrate more on high-level issues.

- Networking and Internet support

  Java was designed from the ground-up to support networking. The Java API provides extensive network support, from sockets and IP addresses, to URLs and HTTP. Compared with C++ and C, it's extremely easy to write network applications in Java. Java also includes support for more high-level network programming, such as remote-method invocation (RMI), CORBA and Jini. These technologies make Java an attractive choice for large-scale distributed systems.

CORBA

In Chapter 2, different technologies and DVE systems have been briefly discussed. A number of existing DVEs have utilised IP multicasting as a basis of their message exchange protocols. The reason for this is that multicasting can potentially reduce the number of messages transmitted through an underlying network. In a DVE, audio and video messages consume a massive amount of bandwidth. Multicast can lower the bandwidth consumption, as well as the latency of message transmission, by reducing the number of duplicate messages transmitted. Therefore, it is suitable for large-volume message exchange in a rich DVE. However, unlike unicast, multicast is not completely supported in Wide Area Networks (WAN) or over the Internet; routers which do not support multicast may drop multicast packets. In order to solve this problem, multicast packets can be wrapped inside unicast packets. When a router, which does not support multicast, receives one of these wrapped multicast packets, it can forward these packets to a router which supports multicast or understands this packet format, e.g. MBONE [Savetz96]. In addition, multicasting only offers a limited number of unique multicast addresses, the number of addresses available to a particular DVE application may not be sufficient to fulfil the networking

requirements of an interest management schema. When multicasting is implemented within a LAN or LAN segment, the decision as to whether a machine should receive a message is made by the recipient only; no knowledge of group membership is required for the sender or any routing hardware. However, when multicasting is implemented over a more complex network, such as a WAN or the Internet, a message may need to travel through a number of intermediate networks and routers in order to reach its recipients. In this case, although the sender still does not require any knowledge of its message's recipients, any routing hardware responsible for delivering the message must know which networks to forward the message to. With this in mind, there may be a substantial delay between a node deciding to join or leave a multicast group and this change in group membership filtering through all the routing hardware responsible for the delivery of messages. As such, multicasting may not be suitable for highly-dynamic communication groups over the Internet.

The Common Object Request Broker Architecture (CORBA) shields the application developer from the complexity of networking issues so that developers can concentrate on application issues. The underlying CORBA protocol, the Internet Inter-ORB Protocol (IIOP), which is built on top of TCP/IP, provides the unicast network communication to guarantee full support over WANs or the Internet. Furthermore, the *PIM* system is designed to solve the missed interaction problem in DVEs with highly variable speed objects. The *PIM* system introduces extra messages ($APUM_{local}$) to notify the relevant nodes when their objects' auras will intersect so that the missed interaction problem can be alleviated. However, due to the overhead of additional messages, the scalability in the *PIM* system may be influenced. Therefore, the *PIM* system is purely an experimental system to test the scalability and message drop rate in the system. No audio or video messages are transmitted in this system. Hence, unicast is suitable for the implementation of the *PIM* system. In addition, to simplify the *PIM* system, each node hosts only one object. As Java is the most appropriate programming language for distributed systems, JacORB, a Java-

based open source CORBA ORB, was adopted to fulfil the *PIM* system networking requirement.

4.2 System Design and Implementation

The *PIM* system implements predictive interest management as its core algorithm to filter the irrelevant messages exchanged between nodes and alleviate the missed interaction. Each server in the *PIM* system implements the message dissemination layer and the network layer in the DVE architecture; each node simply sends and receives message from the server it is subscribed to. Therefore, there is no description of the *PIM* nodes implementation in this chapter.

Before further describing the *PIM* system, some terminologies must be defined:

- **Local Server**: The physically closest *PIM* server to a given node;
- **Local nodes**: Nodes subscribed to a given server;
- **Local Object**: An object participating in the DVE and hosted on a given node;
- **Remote Server**: Another *PIM* server, with respect to a given server;
- **Remote nodes**: Nodes subscribed to a remote server;
- **Remote Object**: An object participating in the DVE and hosted on a remote node;
- **Object reference**: A reference to an instance of a class in an object-oriented programming language, such as Java;
- ***PUM* subscriptions**: The *PUM* subscriptions of a node are the set of nodes whose object's auras intersect the aura of this node's object;
- ***APUM*$_{local}$ subscriptions:** The *APUM*$_{local}$ subscriptions of a node are the set of nodes whose object's *PAIs*, but not auras, intersect the *PAI* of this node's object.

Servers and nodes are required to understand the format of the messages they send to each other. A node must register itself with, and obtain a node ID from, its local server. After obtaining the node ID, this node is able to view the world information in the local server and decide which world it wants to join. If the node has created a new virtual world, it can register this virtual world with the local server. If other nodes are interested in this virtual world, they can register themselves to this virtual world. After deciding the virtual world, a node registers its object to its local server and specifies the world in which the object should be registered within. If the registration is successful, the server sends the objects' information to the other remote servers and a confirmation message to that node. When a node tries to un-register its object from a world, it is required to send an object removal indication to the server. If the node wants to leave the DVE completely, it must send a node un-subscription message to its local server.

In order to describe the design and implementation of the *PIM* system, an IDL file is discussed to introduce the messages types exchanged between servers and between a server and a node. Following, the implementation of a *PIM* server and the exceptions thrown in the system are discussed in detail.

4.2.1 IDL File

As mentioned in Chapter 2, CORBA, through the use of the Interface Definition Language (IDL), allows the developer to define an interface to an object in a programming language-independent manner. Modules and interface are naming scopes, which allow the developer to define the structure and operations of a distributed application. A module in IDL maps to a package in Java and an interface in IDL maps to a Java class. The set of operations offered by an interface can be extended by declaring a new interface that inherits from the existing one.

An IDL file, called *messageservice.idl* (see Appendix A), is defined to specify the "contract" between the servers and the "contract" between nodes and their local server. When referring to module names, the Java naming convention for packages is utilised, wherein a module hierarchy is defined such that "ms.idl" means that the module idl exists within the module ms. Inside the module "ms.idl", an interface and three modules are specified in the *messageservice.idl*. The interface "util" defines the message types exchanged within the *PIM* system. The module "exceptions" defines the possible exceptions which can be thrown by the server and caught by the node. The module "clients" defines an interface called *MessageServiceUser*, which inherits from the interface "util" and declares the operations which can be invoked by the server on a node. The module "servers" defines an interface called *MessageService*, which inherits from the interface "util" and declares the operations which can be invoked by the nodes and remote servers on a server.

4.2.2 PIM Server Structure and Implementation

A *PIM* server is constructed from five components: the Message Service Servant, the Message Buffer Unit, the Thread Pool Processing Unit, the *PIM* Processing Unit and the Message Supplier. The arrows in Figure 4.2 represent the flow of data from one component to another. The Message Service Servant provides the CORBA interface for the *PIM* system so that the DVE participants (local nodes/remote servers) can invoke the operations specified in the IDL. A local node and a remote server are represented as N and S respectively in Figure 4.2; the node N_i and remote server S_k represent a local node with ID i and a remote server with ID k respectively. In addition, the Message Service Servant receives messages from either the local nodes or remote servers and directs them into the relevant inward message buffers in the Message Buffer Unit. The inward message buffers of the Message Buffer Unit store not only *PUM/APUM* messages directed by the Message Service Servant, but also the $APUM_{local}$ subscriptions determined by the *PIM* Processing Unit. After the Message Buffer

Unit receives *PUM/APUM* messages, it invokes the Thread Pool Processing Unit to construct a Request (message) for further processing. Once the Request from the Thread Pool Processing Unit is received by the *PIM* Processing Unit, it processes the Request based on the criteria specified in predictive interest management in Chapter 3. According to the result, the *PIM* Processing Unit updates both the $APUM_{local}$ subscription of the corresponding outward message buffers in the Message Buffer Unit and the message delivery frequency in the Message Supplier. Finally, the Message Supplier will deliver the related messages in the outward buffers to the local nodes and remote servers. In Figure 4.2, the Subscribe Channel and the Publish Channel are message channels which allow internal servers to exchange messages. The admin channel allows servers to exchange $APUM_{admin}$ at the lowest frequency; the local channel allows servers to exchange $APUM_{local}$ and relevant *PUM* messages.

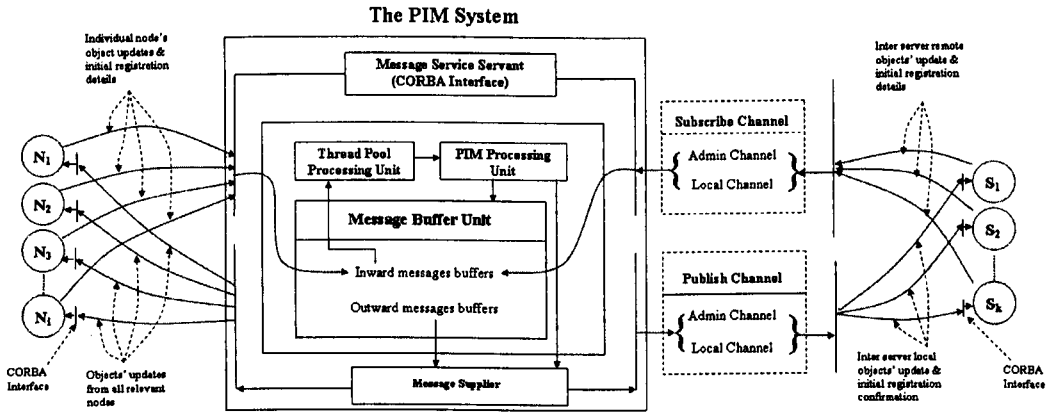


Figure 4.2 *PIM* System Server Structure

4.2.2.1 Message Service Servant

Each node must register its objects to a server before it joins a DVE. This process is represented by attaining the server's reference and invoking the subscription methods in the IDL file. When the Message Service Servant, which handles remote method invocation, receives a subscription message from a node's object, it will parse the information into the corresponding message

buffers and instructs the message buffers to construct an enter request, termed R_{enter}. The server will then instruct the Message Supplier to send a new object notification to the other servers. This is a reliable, high-priority message, which is sent immediately, rather than being buffered and sent at a fixed time interval. This ensures that the entrance of a new object into the virtual world is recognised by all servers in the DVE in a timely fashion. If an $APUM_{admin}$ is received containing a message about a new object before the corresponding new object notification message is received, then any messages regarding this new object will be discarded by the receiving server. As such, it is necessary to ensure that new object notification is sent as soon as possible to avoid a missed interaction between the new and existing objects.

In the *PIM* system, eight message types are defined in the *messageservice.idl* for the purpose of assisting message exchange. Table 4.1 outlines the name and contents of each message types. Among these message types, the asterisk message types are assistant message types, which are not transmitted in the underlying network directly. Instead, these message types are parts of other message types. Furthermore, as can be seen from the contents in the table, the aggregation message includes an array of other message types; the number of data packets being sent in the underlying network is reduced. This is beneficial because each packet must contain a header storing, for example in TCP/IP, the sender and receiver's network address and port number, CRCs (for error-checking), acknowledgements and the length of data being transmitted. Additionally, if a packet is smaller than the minimum size required for a valid frame (a packet wrapped inside a header, including start and stop bits added for network hardware transmission), it may be necessary to add data to the frame to ensure it is the minimum size, termed "padding". With these considerations, the utilization of larger packet-sizes uses the available network bandwidth more efficiently than sending multiple small packets.

Message Type	Contents
(1)SinglePumMessage(SPM)	Node_id, object_id, world_id and position
(2)SingleApumMessage(SAM)	Ms_id, world_id, node_id, object_id, pia, aura and position
*(3)AggregatedPumMessage(APM)**	An array of SinglePumMessage
*(4)AggregatedApumMessage(AAM)**	An array of SingleApumMessage
*(5)MserviceToUserPumMessage(MTUPM)**	Ms_id, world_id and SinglePumMessage
(6)AggregatedMTUPumMessage(AMTUPM)	An array of MserviceToUserPumMessage
(7)MSExchangeAdminApumMessage(MSEAAM)	Ms_id, world_id, maxSupplierId and an array of AggregatedApumMessage
(8)MSExchangePumMessage(MSEPM)	Ms_id, world_id, maxSupplierId and AggregatedPumMessage

Table 4.1 Message Type

Messages Sent from Node to Server

Each node must send its hosted object's messages, called *SPM (PUM)*, to its registered server. When the server receives a *SPM* from a node, it will construct a *SAM* ($APUM_{local}$) on behalf of that node. The frequency of delivering the *SAM* on the local channel depends on the intersection degree of the related objects. Each *PIM* server is responsible for transmitting *PUM* messages from its local objects to the other *PIM* servers, provided the remote server hosts an object interested in the *PUM* messages.

Messages Sent from Server to Server

The *MSEPM* is designed for exchanging *PUM* messages of the registered objects to the other servers in the *PIM* system. The main component of *MSEPM* is the *APM*, which combines a number of *SPMs* into one. The length of the *APM* is not fixed to the number of objects registered to the server, therefore further reducing bandwidth usage. If the server has not received a new *SPM* from a node before it constructs a new *MSEPM*, the server will continue constructing the new *MSEPM*, but excluding this object's information, rather than waiting for the arrival of an up-to-date *SPM* from that node. This not only helps to reduce the time taken to construct the *MSEPM*, but also helps to save the network bandwidth. In addition to the *MSEPM*, servers exchange *MSEAAMs* at a lower frequency. The *MSEAAM* is mainly composed of *AAMs*, which are an array of

SAMs. However, contrary to the *APM*, the length of an *AAM* is fixed at the number of local objects registered on the corresponding *PIM* server.

Messages sent from Server to Node

When a server receives *SPMs* from its local nodes or *MSEPMs* from the remote servers, it constructs an *AMTUPM* and sends it to a local node. An *AMTUPM* is composed of an array of *MTUPMs*, which are constructed using *SPMs*. The size of an *AMTUPM* is flexible and depends on the *PUM* subscriptions of a node.

4.2.2.2 Message Buffer Unit

In order to uniquely identify an object, the *PIM* system uses an object ID which is composed of three parts: an object ID issued by hosted node, a node ID issued by the local server and a server ID, which was allocated by the web server.

Conceptually, the Message Buffer Unit contains two components: inward buffers and outward buffers. However, in the actual implementation, the inward and outward buffers are combined. Each server has buffers to store *PUMs* related to its local objects. N_i in Figure 4.3 (a) represents the *PUM* message buffers of a local node with ID *i*. In addition to the *PUM* message buffers, each local node is capable of converting the information contained in *PUMs* to construct an $APUM_{local}$.

Each server has message buffers for both itself and the remote servers. A diagram of a server's message buffers is represented in Figure 4.3 (b). MS_i represents the message buffers of a server with ID *i*. Each MS_i has a set of *PUM* buffers, *APUM* buffers and *NodeRecord* buffers. MS_i can represent either the local server's message buffers or a remote server's message buffers:

- If MS_i represents the message buffers of the local server, the *PUM* buffer is an empty set. The reason for this is that the *PUM* messages are already stored in the Nodes' message buffers; it is unnecessary to waste memory to repeatedly store *PUM* messages. *APUM* buffers are used to store $APUM_{local}$ messages constructed from the *PUM* messages, which are stored in the nodes message buffers.
- If MS_i represents the message buffers of a remote server. Both *PUM* and *APUM* buffers are intended to store the latest *PUM* and $APUM_{local}$ sent by the remote server with ID *i*, respectively.

Figure 4.3 (c) depicts all the message buffers in a server.

$APUM_{admin}$ are constructed from an array of $APUM_{local}$. Therefore, it is unnecessary to have separate message buffers to store $APUM_{admin}$. An $APUM_{admin}$ can be constructed from the available $APUM_{local}$ messages in the *NodesRecord* buffers. Similarly, $APUM_{local}$ messages can be extracted from an $APUM_{admin}$ message.

Regardless of the representation of MS_i, the *NodesRecord* buffers are used to store the $APUM_{local}$ subscriptions of each node subscribed to MS_i. The purpose of the *NodesRecord* message buffers is to discard $APUM_{local}$ messages from being processed in the *PIM* Processing Unit when:

- An $APUM_{admin}$ containing information about a new object is received before the new object's entrance notification. In this case, a consistency issue may occur. To clarify, an object, O_n, belongs to a node which subscribed to a server with ID 0, say S_0. When O_n registers with S_0, a new object entrance notification will be issued to all servers in the DVE, includes itself. The reason for issuing the new object entrance notification to S_0 is to introduce a delay in processing this entrance notification to compensate for the network delay of in transmitting this entrance notification to the other servers. In S_0, subsequent $APUM_{admin}$

messages disseminated to other servers might contain this new object's positional update information. However, due to network and processing delays, some remote servers may not have received and processed the new object's entrance notification before receiving an $APUM_{admin}$ message including the state of this new object. If no mechanism to prevent the new object's information from being processed, an inconsistency will occur.

- An $APUM_{admin}$ containing information about an object which no local objects are interested in is received.

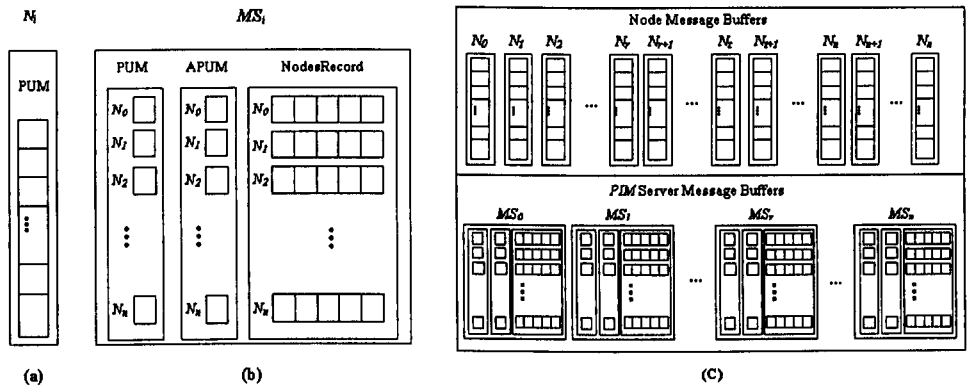


Figure 4.3 Message Buffers

4.2.2.3 Thread Pool Processing Unit

The previous section introduced two types of message buffers (N_i/MS_i) in the *PIM* server and explained the source of these messages. The Thread Pool Processing Unit acts as the first stage in consuming these messages. The reason for implementing thread pools is to avoid the server blocking, while restricting the number of threads competing for processing resources. To clarify, the *PIM* server receives $APUM_{admin}$ messages from other servers in first in first out (FIFO) order; if there was no thread pool implemented, the server would be required to receive and process the first $APUM_{admin}$ before it could consume further $APUM_{admin}$ messages; the server would be blocked until the processing

has finished. This would have a detrimental effect on system responsiveness and therefore on the throughput rate of messages. To avoid the server blocking until message processing has completed, multiple threads can be dispatched to deal with message processing. This allows the main server thread to continue processing without having to wait for message processing to finish. However, if a new thread is created to deal with each received message, this could cause a large number of threads being created and deleted during run-time. However, there is a cost associated with creating and deleting a thread, which may slow down the server if a large number of threads are created and deleted in rapid succession. Additionally, if there are a very large number of threads competing for a processing resource, each thread will spend a very large proportion of time waiting to access the processor(s). A solution to this is to use a thread pool: a fixed number of threads to perform a set of computations.

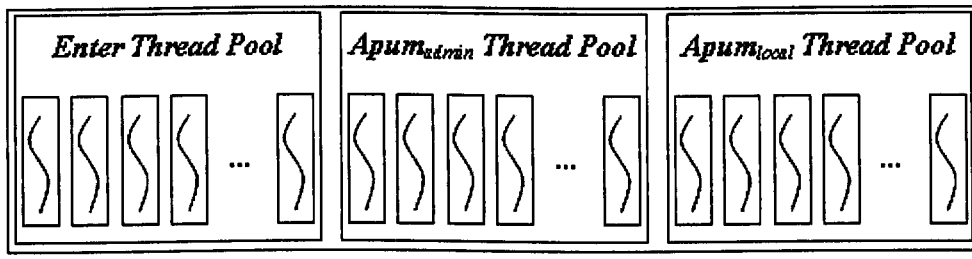


Figure 4.4 Thread Pool Processing Unit

In the *PIM* system, three different thread pools are implemented to construct three types of messages to avoid the blocking problem:

- **The Enter Thread Pool**
- **The $APUM_{admin}$ Thread Pool**
- **The $APUM_{local}$ Thread Pool**

The Enter Thread Pool is designed to construct an Enter Request (R_{enter}) when a local/remote object's information is received by the server for the first time. The Enter Thread Pool can be triggered to construct a R_{enter} message by two entities: N_i and $MS_i(0 \leq i \leq n)$. When a server receives a notification of local/remote object

joining a DVE, signals are sent from N_i to the the Enter Thread Pool to indicate the arrival of joining local objects; otherwise, signals are sent from MS_i to the Enter Thread Pool to trigger the creation of R_{enter}. The $APUM_{admin}$ Thread Pool is designed to handle $APUM_{admin}$ messages. When MS_i receives an $APUM_{admin}$, it will be extracted to an array of $APUM_{local}$. These $APUM_{local}$ messages will overwrite the existing *APUM* message buffer values and trigger the $APUM_{admin}$ Thread Pool to construct an $APUM_{admin}$ Request (R_{admin}). The $APUM_{local}$ Thread Pool is required to create an $APUM_{local}$ Request (R_{local}) when MS_i receives an $APUM_{local}$. The corresponding *APUM* message buffer value is overwritten.

Although these three thread pools have their own attributes and differing functionalities, due to the similarity of the thread pools, e.g. fixed numbers of threads, waking up by events and putting back to the pool after accomplishing a task, it is good programming practice to use the same structure to implement all three thread pools. An interface, called Request, is defined and passed as a parameter to the constructor of each thread pool. R_{enter}, R_{admin} and R_{local} inherit from Request but provide their own properties. In this case, passing an argument to indicate the types (*Enter*, $Apum_{admin}$ and $Apum_{local}$) allows three thread pools to be created using the same source code.

In addition to thread pool implementation, it is necessary to consider the situation where there are no threads available when a new task arrives. Under these circumstances, the thread pool may choose to discard or accept the new task. If the task is not considered important by the system and it is received at a relatively high frequency, the discard policy can be used to speed up processing; otherwise, an accept policy should be adopted to guarantee the completion of task. There are two solutions that can be utilised to implement the accept policy:

- Block the calling process, wait for one of the threads to become available and perform the task, then release the calling process.
- Put the task into a waiting queue, allow the calling process to continue, and then perform the task when a thread becomes available.

The second solution is more efficient than the first, as it does not result in the message receiving thread blocking until a processing thread becomes available. Therefore, in the thread pool processing unit, a waiting queue is utilised to accept tasks and place them into a buffer when all threads are busy.

4.2.2.4 PIM Processing Unit

As can be seen from the Figure 4.5, Request is designed to invoke the related operations in the *PIM* processing unit depending on the Request type delivered by the thread pool processing unit. The enter table is used to store the basic information of joining nodes (e.g. whether they are either local nodes or remote nodes). The *LocalNode* list, used to store a set of *LocalNode* objects, is designed to assist the subscription engine to perform the subscription/un-subscription operations. Each *LocalNode* stores three different lists (*PUM*, $APUM_{local}$ and $APUM_{admin}$) to identify its subscription status with respect to other nodes, which will be discussed in detail later. The arbiter is a mechanism designed to determine the subscription status of two nodes depending on their object's intersection degree. Finally, the subscription engine performs the subscription and un-subscription operations between nodes according to the result produced by the arbiter.

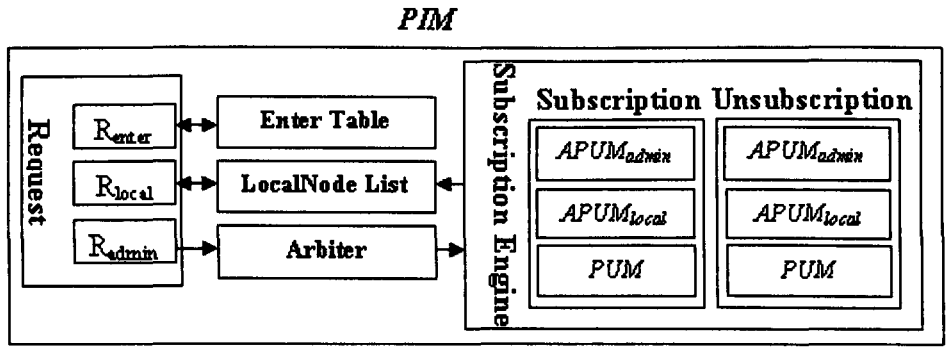


Figure 4.5 PIM Processing Unit

Enter Table

In the *PIM* Processing Unit, an enter table is utilised to record the number of nodes subscribed to each server. The enter table is implemented using a hashing function, with each server's message service ID acting as a key to uniquely identify a list. Each list simply includes the IDs of the nodes subscribed to that server. When the *PIM* processing unit receives a R_{enter}, it is required to determine the subscription type between the joining node and the relevant existing nodes. This is achieved by iterating through the corresponding lists in the enter table. Each relevant node is passed, along with the joining node, to the arbiter, which will return the subscription type of this pair of nodes. If the R_{enter} is issued by a local node, the iteration is required to go through all lists in the enter table. However, if the R_{enter} is issued by a remote node, only the list with the local server message service ID is iterated through. In both cases, a *NodesRecord* message buffer is created in the corresponding MS_i message buffers to record the *APUM* subscription of that node.

LocalNode List

The *LocalNode* list stores a set of Java objects, called *LocalNode*. When the PIM processing unit receives a R_{enter}, if the R_{enter} is issued by a local node, a *LocalNode* object corresponding with this local node is created. Three lists (*PUM*, $APUM_{local}$ and $APUM_{admin}$) are initialised in a *LocalNode* to assist the subscription engine in handling the subscription/un-subscription requirements. Each list stores a set of Java objects called *InterestNode*, which are created by the subscription engine. Each *InterestNode* corresponds to a node in a DVE and has a flag to indicate the subscription type between the joining local node and its related node. If the flag indicates the subscription type in the *InterestNode* is *PUM*, this *InterestNode* will be inserted into the *PUM* List; if the flag is $APUM_{local}$, this *InterestNode* will be inserted into $APUM_{local}$ list; otherwise this *InterestNode* will be inserted into the $APUM_{admin}$ list. Unlike the *PUM* list and $APUM_{admin}$ list, the $APUM_{local}$ list is a minimum-oriented priority queue. The

reason for using a minimum-oriented priority queue is to provide the $APUM_{local}$ message supplier fast access to the highest $APUM_{local}$ frequency.

Arbiter

When the arbiter receives a Request, it utilises predictive interest management to determine the subscription of objects. This can be expressed by the following pseudo-code:

```
SubscriptionType typeDetermine(Obj1, Obj2, Time(Pum))
{
    SUM(R_Pai) = Radius(PIA(obj1)) + Radius(PIA(obj2));
    SUM(R_Aura) = Radius(AURA(obj1)) + Radius(AURA(obj2));
    SUM(dis) = distance(POS(obj1), POS(obj2));
    Time(AUBV) = ((SUM(dis) - SUM(R_Aura))/(SUM(R_Pai) - SUM(R_Aura))) * FUTURE_TIME;
    (Formula 3.1)
    if (SUM(dis) > SUM(R_Pai))
    {
        return Apum_admin;
    }
    else if (SUM(dis) ≤ SUM(R_Pai) && SUM(dis) > SUM(R_Aura))
    {
        if (Time(AUBV) ≥ Time(Apum_admin))
        {
            return Apum_admin;
        }
        else if (Time(AUBV) < Time(Apum_admin) && Time(AUBV)> Time(Pum))
        {
            return APUM_local;
        }
        else
        {
            return PUM;
        }
    }
    else
    {
        return PUM;
    }
}
```

There are three parameters passed to the arbiter: the reference of the registrar object, the reference of the registrant object and the *PUM* delivery frequency of the registrar object. The registrar object must be a local object; the registrant object may be either local or remote. The reason for that is that the *PIM* server is only responsible for delivering its local objects' up-to-date information to

objects interested in it. The responsibility for delivering any remote objects' messages lies with its respective server.

The *LocalNode*, corresponding to a local node, can be used to record the subscriptions of different message types. The purpose of the arbiter is to determine the *Subscription Type (ST)*, which can be obtained by calculating the following four values related to the registrar and registrant objects: the sum of their *PAIs*' radii; the sum of their auras' radii; the distance between them; and their *Approximate Upper Bound Value (AUBV)*. According to *PIM* , three cases are required to be considered:

- The distance between the objects is larger than the sum of their *PAIs'* radii; the *ST* is $Apum_{admin}$.
- The distance between the objects is greater than the sum of their auras' radii but less than or equal to the sum of their *PAIs'* radii. In this case, the *ST* depends on the *AUBV*. If the *AUBV* is larger than or equal to the $Apum_{admin}$ exchange frequency, the *ST* is $Apum_{admin}$; if the *AUBV* is greater than the registrar object's *PUM* delivery rate but less than the $Apum_{admin}$ exchange frequency, the *ST* is $Apum_{local}$; otherwise, the *ST* is *PUM.*
- The distance between the objects is less than the sum of their auras' radii; the *ST* is *PUM.*

Subscription Engine

The subscription engine is composed of two parts: subscription component and un-subscription component. When the arbiter obtains the *ST*, it will pass the *ST* along with the registrar object and registrant object to the subscription engine. Based on this information, the subscription engine performs the subscription and un-subscription of different message types between these objects. The subscription/un-subscription components co-operate with the message suppliers. The details of this co-operation are discussed in the message suppliers section.

The subscription/un-subscription of message type is based on the co-operation between the subscription engine and the registrar object's related *LocalNode* object. To clarify, if this is the first time the arbiter has passed the *ST* of the registrar and registrant objects, the subscription component in the subscription engine will add the registrant object into the corresponding list in the registrar object's *LocalNode* depending on the *ST* between them. However, if these objects already have a subscription between one-another, the subscription engine may be required to instruct the subscription and un-subscription components to adjust the subscription between the registrar and registrant objects depending on the value of *ST*. A new object, termed *InterestNode*, is defined to represent the registrant object. Each *InterestNode* object contains a flag to indicate the message type the registrant object should receive from the registrar object. The purpose of this is to ensure the required subscriptions and un-subscriptions occur in all related message suppliers. For example, the arbiter passes a pair of objects, obj_a and obj_b, for the first time to the subscription engine and the *ST* is *PUM*. The subscription engine will create an *InterestNode* corresponding with obj_b and set its message type flag to be *PUM*. Subsequently, it will put the *InterestNode* into the *PUM list* of the *LocalNode* corresponding to obj_a. Later, the arbiter passes the same registrar and registrant objects to the subscription engine. The subscription engine will compare the previous message type with the current message type. If both of the message types are the same, no subscription/un-subscription will be performed. However, if the current message type differs from the previous one, which is now, for example, $APUM_{local}$, the subscription engine will instruct the un-subscription component to remove the corresponding *InterestNode* from the *PUM list* and un-register it from its related message suppliers. Then, the subscription component will insert the *InterestNode* into the $APUM_{local}$ *list* and register it with the corresponding message supplier.

In addition to the co-operation with *LocalNode* object, the subscription/un-subscription components change the $APUM_{local}$ subscriptions in the

NodesRecord buffers in the message buffers. This ensures that unnecessary messages are filtered by the servers thereby avoiding unnecessary processing.

4.2.2.5 Message Supplier

The final component of the *PIM* system is the message supplier, which is composed of the *PUM* message supplier, $APUM_{local}$ message supplier and $APUM_{admin}$ message supplier. Each message supplier delivers different types of messages at different frequencies. A detailed description of each message supplier is provided below:

PUM message supplier

The main function of the *PUM* message supplier is to deliver the relevant *PUM* messages to the local nodes and the remote servers in the *PIM* system. A list, called *MS* list, containing elements corresponding with all *PIM* servers in a DVE is defined in the *PUM* message supplier. The elements of this list are Java objects, called *PumSupplier*. *PumSupplier* stores a server's object reference and a list of *LocalNode* references, which were previously created within the *PIM* processing unit. Using a remote server's object reference, a local server can transmit the *PUM* messages to the remote servers. A thread is used in the *PUM* message supplier to iterate through the *MS* list to deliver the *PUM* messages to the corresponding server in turn. If the *PumSupplier*'s object reference is the local server, rather than sending the *PUM* messages to itself, the *PUM* messages are sent to the local nodes. The *PUM* messages are delivered to the local node in an aggregation message, which combines the *PUM*-interested nodes' message into a single *AggregatedMTUPumMessage*, to optimise the network bandwidth usage. If a local node is not interested in any other nodes' *PUM* messages, no messages are sent to it. To achieve this, the *PUM* list stored in the *LocalNode* assists in filtering unnecessary message transmission. If the *PumSupplier*'s object reference corresponds to a remote server, the list of *LocalNode* object

contains the local nodes which are interested in the remote server's local nodes. An *AggregatedPUMMessage* is created to transmit these *PUM* messages to the remote server.

The frequency of delivering *PUM* messages to either local nodes or remote servers depends on the processing speed of the server. During the lifetime of the local server, the *PUM* message supplier runs in the background, waiting to deliver *PUM* messages, as they become available, to their corresponding nodes/remote servers. When the first *PUM* messages are received from the local nodes, the message service servant wakes the *PUM* message supplier, causing the *PUM* message delivery process to begin. *PUM* messages are delivered at high frequency, e.g. 3 messages per second. Therefore, if such a *PIM* server has one thousand local nodes, it must be capable of processing 3000 *PUM* messages per second. If these messages are delivered using a single thread, a substantial delay may exist between the time a message is received by a server and the time the *PUM* messages are delivered to their desired local nodes/remote servers. Therefore, a thread pool is used in the *PUM* message supplier to assist the *PUM* message delivery and reduce the delay imposed by the system processing.

$APUM_{local}$ message supplier

As described in Chapter 3, different local nodes have different $APUM_{local}$ exchange frequencies. One possible way to implement this is to create a thread for each local node and transmit the $APUM_{local}$ to the interested nodes at the corresponding highest *AUBV* frequency. However, from an engineering point of view, it is impractical to design a DVE system which has thousands of threads competing for processing resource concurrently. Another possible implementation is to use a thread pool to check the highest *AUBV* frequency of each local node. If the time to transmit a local node's $APUM_{local}$ is due, the next available thread will deliver this $APUM_{local}$ to all its required recipients. However, as mentioned previously, there are already four thread pools in the *PIM* server: three thread pools in the thread pool processing unit and one in the

PUM message supplier. As $APUM_{local}$ message delivery frequency is lower than *PUM* frequency, to reduce the multi-threading resource competition overhead, rather than having multiple threads, a single thread is used in the *PIM* server to deliver the $APUM_{local}$ for all the local nodes.

In order to enable a single thread to deliver the messages in a timely fashion, a minimum-oriented priority queue prioritised by time is implemented using an ordered list to store the highest *AUBV* frequency values for each node. Let $S = \{N_a^5, N_b^8, N_c^{10}, N_d^8\}$, where the superscript and subscript represent the highest *AUBV* frequency and the node ID respectively. The highest *AUBV* frequency within *S* is 5 seconds. Therefore the highest-priority value in the minimum-oriented priority queue corresponds to the highest *AUBV* frequency from *S*. The priority queue, *P*, for *S* would be: $P= \{N_a^5, N_b^3, N_d^0, N_c^2\}$, where the superscript represents the difference between the previous element's *AUBV* and the current element's *AUBV*, termed the displacement time value. The first element in the priority queue's displacement time value is an absolute value, rather than a displaced value. In this example, the $APUM_{local}$ message supplier thread will check the first element in the list. If the value is 0, the $APUM_{local}$ message supplier thread will deliver the $APUM_{local}$ message to its corresponding nodes. The thread will then remove this element from the queue and re-insert it into the queue based on its current highest *AUBV* value. The thread will then check the first element, and repeat the process previously described. If the first element's displacement time value is greater than 0, the thread sleeps for the duration of the displacement time and then delivers the $APUM_{local}$ to the relevant nodes. This avoids the delivery thread busy waiting and helps to ensure that the *PIM* system's performance is acceptable.

The *PIM* system will re-evaluate the *AUBV* values of a node when messages are received. If the *AUBV* value of a node becomes more frequent, it is necessary to re-insert the node into the priority queue in the $APUM_{local}$ message supplier to ensure that messages are transmitted frequently enough to alleviate the missed

interaction problem. If, however, the *PIM* system re-evaluates a node's *AUBV* value and determines that it has become less frequent, the node is not re-inserted into the priority queue straight away. This is because inserting into an ordered list priority queue is time consuming. Therefore, in order to enhance performance, rather than frequently modifying the priority queue, messages are sometimes transmitted sooner than is absolutely necessary. Once the message has been delivered, the most recent *AUBV* value is used to schedule the next delivery of the node's $APUM_{local}$.

$APUM_{admin}$ message supplier

As described in Chapter 3, compared with *PUM* and $APUM_{local}$ message transmission frequency, $APUM_{admin}$ message transmission is the lowest frequency. This allows a single thread to be used in the $APUM_{admin}$ message supplier to deliver the $APUM_{admin}$ messages to all servers. The transmission frequency of $APUM_{admin}$ messages is sufficiently low that a single delivery thread should not exacerbate the message delivery delay imposed by the *PIM* server. The $APUM_{admin}$ message supplier thread iterates through a list of *PIM* servers' object references and delivers an *MSExchangeAdminApumMessage* to each server in a DVE at an application-defined constant rate.

4.2.3 System Exceptions

There are four network exceptions defined in the *messageservice.idl* file. Following is a list of these exceptions and the conditions in which they are thrown:

- WorldNotExist
  - i) A node tries to register its object to a non-existent world;
  - ii) A node tries to remove a non existent world.
- ObjectNotExist
  - i) A node tries to remove an object which does not exist.

- PermissionDenied
  i) A node tries to remove a world which it did not create;
  ii) A node tries to remove an object which does not belonged to it.
- SubscriptionExceeded
  In the *PIM* system, in order to guarantee the performance of each server, a maximum subscription limitation is set. When a node tries to register itself to a local server and the maximum node subscription limit has been reached in the local server:
  i) If there are no remote servers in a DVE, or the registration request is received before the remote servers have completed subscribing to the local server, a SubscriptionExceeded exception is thrown by the server.
  ii) If there are remote servers already in a DVE, it will redirect the registration requirement to one of the remote servers. If all other remote servers have reached the registration limit, a SubscriptionExceeded exception is thrown by the server.

4.3 Summary

This chapter described the *PIM* system's design and implementation. Prior to describing these issues, the interaction models and development technologies were discussed. The interaction models described the message exchange between servers (server/server) and between servers and nodes (server/node); the development technologies include Java as the programming language and CORBA as the middleware to handle networking issues.

In the *PIM* system, each server is constructed from five components: the Message Service Servant, the Message Buffer Unit, the Thread Pool Processing Unit, the *PIM* Processing Unit and the Message Supplier. The Message Service

Servant provides a CORBA interface to enable the local nodes and remote servers to invoke the operation it defines. In addition, it receives requests from the local nodes and remote servers and passes them to the corresponding component. The Message Buffer Unit contains the node message buffer and *PIM* server message buffer, which act as inward and outward buffers. The Thread Pool Processing Unit constructs different types of Requests from the Message Buffer Unit to the *PIM* Processing Unit for further processing. The *PIM* Processing Unit determines the objects' intersection degrees and, therefore, the message types exchanged between local nodes and other servers. The Message Supplier delivers the appropriate messages to the local nodes and servers.

Chapter 5

Experimentation

This chapter describes the *PIM* system experiments, which were conducted on a shared resource of approximately 20 machines on the same LAN segment. Each server is hosted on one of the machines and the nodes are distributed evenly between a set of non-server machines. Each node controls one object. Following, the world and object simulators are introduced, which simulate a virtual world and the movement of an object respectively. Finally, four different experiments are described, concentrating on four particular aspects: purpose, methods, results and analysis.

5.1 Experimentation Environments

The experiments are based on the GIGA cluster in the School of Computing Science at Newcastle University. The cluster has approximately 20 machines on the same LAN segment. Each machine has a 2GHz Intel Xeon processor (equivalent of 2x2GHz Pentium 4 processors with Hyper Threading) with 1GB RAM running Red Hat Linux 7.2.

The *PIM* servers are located on different machines, termed server simulators, on the same LAN segment. Nodes may be co-located on different machines, called

client machines, which reside in the same LAN segment as the servers. By using the client machines, synthetic networking traffic for nodes is created. Each node hosts one object. Node numbers are increased by increments of 500 from 500 up to 6000, with measurements taken at each increment. The reason for that is that the numbers of nodes simultaneously handled by a server is limited by the processing power of the server. This will be discussed further in the later experiments.

Nodes are distributed as evenly as possible between server machines and client machines. Each experiment's duration was one hour to ensure the initialisation overhead of each node did not skew the results. However, the machines used for this experiment are a shared resource. As such, the performance of the machines and the available network bandwidth can vary considerably depending on the number and nature of the processes running on each machine at the time each experiment was performed.

5.2 Simulators

Each node has two simulators: world simulator and object simulator. The world simulator is used to generate an appropriately-sized cubic world based on the predicted number of objects in the DVE. The object simulator is used to simulate the movement of each object.

5.2.1 World Simulator

As mentioned in Chapter 4, the *PIM* system is an experimental system to determine the scalability of the *PIM* approach, there are no audio or video messages transmitted. In order to simulate a skeleton DVE, in which the only information objects observe is the dimension of the world, a

WorldSizeGenerator is used to generate the size of a cubic world. The world size is generated based on the formulas below:

$$V_{auras} = N_{objects} * 4/3 * \pi * Radius(aura)^3$$

Formula 5.1 Calculating the Total Volume of Auras

$$W_{size} = \sqrt[3]{V_{aura}/C}$$

Formula 5.2 Calculating the World Size

V_{auras} represents the total volume of all objects auras; $N_{objects}$ is the number of objects in the world; *Radius(aura)* is the radius of an aura, which is fixed at the world creation time and is set as 200 meters in all experiments, however, the radius of an aura can be altered depending on the requirements of the simulation; C is the coverage rate of the objects' auras in the world and it is set to 5%; W_{size} is one of the dimension of a cubic world. In Figure 5.1, a three dimension cubic world is represented and each dimension is identical (W_{xdim}= W_{ydim}= W_{zdim} = W_{size}).

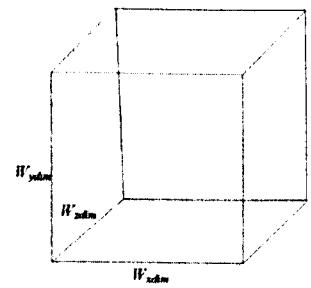


Figure 5.1 Cubic World

Through passing the same variables to the WorldSizeGenerator, each node's object's movements are simulated in the same three-dimensional (3D) world.

5.2.2 Object Simulator

Each node has a program, called RandomWayPointWorld, to simulate the movement of its object. Five positions, called markers, are generated at world creation time for each node. Each object chooses a random marker and moves towards the marker for a random time, termed marker selection time (*MST*). During the *MST*, the object's position is updated at the same frequency as the PUM messages sent from the node to its local server. Once the *MST* has been exceeded, the object selects another random marker, and continues the process. Each marker remains at a position for a random amount of time, called marker relocation time (*MRT*), and then relocates to a new position in the world. In order to determine the *MST* and the *MRT*, four values are used to calculate the minimum and maximum range of *MST* and *MRT*. As the x-, y- and z-dimensions are identical in a cubic world, the diagonal size of this world can be calculated as below:

$$Size_{dia} = \sqrt{3W_{size}^2}$$

Formula 5.3 Calculating the Length of Diagonal

MRT_{lower} is the lower bound of the *MRT* and it is defined as the time taken for an object travelling at its maximum speed to cover a distance equal to half the diagonal size of the world. MRT_{upper} is the upper bound of the *MRT*. Compared with the MRT_{lower}, MRT_{upper} is the time taken for an object to travel a distance equal to the full diagonal size of the world, at top speed. These two variables are represented as the formulas below:

$$MRT_{lower} = (\frac{1}{2} * Size_{dia}) \Big/ Speed(top)$$

$$MRT_{upper} = Size_{dia} / Speed(top)$$

Formula 5.4 Calculating the Upper Bound and Lower Bound of the *MRT*

MRT is a random time selected within the range [MRT_{lower}, MRT_{upper}] and can be decided based on the formula below:

$$MRT = CurrentTime() + (MRT_{lower} + Random() * (MRT_{upper} - MRT_{lower}))$$

Formula 5.5 Calculating MRT

In the Formula 5.5, CurrentTime() is a function to get the current time of the system; Random() returns a decimal number uniformly distributed between 0 and 1. After the previous selected MRT has passed, the MRT is recalculated. The process will repeatedly occur during the lifetime of the object in the DVE. This selection ensures that the time a marker remains in a given position is a sufficient time, with respect to the size of the world, to avoid markers repositioning too frequently. If markers reposition very frequently, the object's movement towards the markers exhibits strange behaviour: when the objects are initialised, they are uniformly distributed within the virtual world but, as time passes, the majority of objects clump together in the centre of the world. This is because, once an objects reaches the centre of the world, the direction they travel in changes sufficiently rapidly that it is unlikely they will be able to move to the extremities of the world before they change direction. If an object is currently located at one of the extremities of the world, for example at the minimum x-coordinate of the world, it is far more likely that the marker will position itself at an x-coordinate greater than the object's current position, causing the object to move towards the middle of the world. However, once the object reaches the centre of the world, it is equally probable that the marker will be at either a smaller or larger x-coordinate than the object. In this case, the object will, on average, remain near the centre of the world; the object will move back and forth around the centre of the world.

MST is chosen within the range of [MST_{lower}, MST_{upper}]. MST_{lower} and MST_{upper} should be less than MRT_{lower} and MRT_{upper} respectively. Therefore, an object can

trace one marker and change to different marker before the marker relocation happen. MST_{lower} and MST_{upper} can be defined as below:

$$MST_{lower} = (MRT_{lower} + MRT_{upper})/4$$

$$MST_{upper} = (MRT_{lower} + MRT_{upper})/2$$

Formula 5.6 Calculating the Upper Bound and Lower Bound of the *MST*

Based on the calculated MST_{lower} and MST_{upper}, *MST* can be determined:

$$MST = CurrentTime() + (MST_{lower} + Random() * (MST_{upper} - MST_{lower}))$$

Formula 5.7 Calculating the *MST*

Similar to *MRT*, *MST* will dynamically change during the lifetime of the object in the DVE.

In a 3D world, each object's position, velocity and acceleration are represented by 3D vectors. The pseudo-code below represents the position and velocity generation process.

```
Vector(diff)= Position(marker)-Position(obj);
Number dis_maker_obj = Magnitude(Vector(diff));
Normalise(Vector(diff));
Number dis_travelled = Magnitude(Velocity(obj)) * dt;
Vector(vec_travelled) = Vector(diff) * dis_travelled;
if(dis_travelled >= dis_marker_obj)
{
  Position(obj) = Position(marker);
  Velocity(obj) = Veclocity(0,0,0);
}
else
{
  Position(obj) = Position(obj) + Vector(vec_travelled);
  Number accel = Magnitude(Acceleration(obj));
  Vector(newSpeed) = Vector(diff)  * (Magnitude(Velocity(obj)) + accel);
  Vector(newSpeed) = Vector(newSpeed) + Velocity(obj);
  Number newSpeed = Magnitude(Vector(newSpeed));
  if(newSpeed > MAX_SPEED(obj))
  {
   Normalise(Vector(newSpeed));
   Vector(newSpeed) = Vector(newSpeed) * MAX_SPEED(obj);
  }
  Velocity(obj) = Vector(newSpeed);
```

To clarify the pseudo-code, it is necessary to explain some of the basic vector operations which are used:

- A vector (Vector(obj)) is a mathematical entity which has both a magnitude and direction. It is represented as a n-dimensional tuple (obj_1, obj_2, ..., obj_n). A vector can be used to represent an object's position (Postion(obj)) in space, such as the location of an object in a DVE. Additionally, vectors can be used to represent the spatial direction of an object such as the velocity (Velocity(obj)) and acceleration (Acceleration(obj)). In 3D space, vector(obj) corresponds to (obj_x, obj_y, obj_z).
- Magnitude(v) is the scalar magnitude of the vector $\sqrt{\sum_{i=1}^{n} v_i^2}$. Magnitude is used to calculate the distance between two points in 3D space, where $\text{Magnitude}(v) = \sqrt{(v_x)^2 + (v_y)^2 + (v_z)^2}$
- Normalise(v) is a vector with a magnitude of 1 representing the same direction as $v = \left(\frac{obj_x}{Magnitude(v)}, \frac{obj_y}{Magnitude(v)}, \frac{obj_z}{Magnitude(v)} \right)$. Normalise is commonly used to move an object a scalar distance along a given vector, where the normalised vector is scaled by the distance required to be moved.

As mentioned previously, after the markers' positions have been generated and the object has chosen a maker randomly, in order to simulate the movement of an object, the trajectory can be predicted based on a set of formulas, which are used to calculate the current position, velocity and acceleration of the object according to the previous relevant information. Given a fraction time (dt), if the distance (dis_marker_obj) between the object and the marker is less than the distance (dis_travelled) an object can travel based on the previous velocity, the position of the object is set as the marker position and the velocity is set as 0. As

can be seen from the pseudo-code, the object will stay at the marker once its position is set as the marker and it remains still until next marker is selected. This will give an object variable speed before its speed reaches its maximum speed. If dis_marker_obj is larger than dis_travelled, the object is still moving toward the selected marker, the new position and velocity of this object is required to be calculated. To simplify the calculation process, the acceleration is set as a fixed value, 10 meters per second in each dimension.

5.3 The PIM System Experiments

In this section, four experiments have been conducted to test four different aspects of the system:

- The maximum number of objects which can be supported by one server;
- The upper bound of message frequency for a node to send *PUMs* to its local server;
- The overhead of $APUM_{local}$ in the *PIM* system compared with a traditional aura-based interest management system;
- The scalability of the system in term of the number of nodes (objects) the system can support simultaneously as the number of servers increases.

The first two experiments' results can be used to assist DVE developers to estimate appropriate system variables (*PUM* frequency, number of servers, maximum number of objects supported) to provide acceptable performance. For example, given a threshold maximum drop rate and a *PUM* transmission frequency, assuming that the target machines are of similar specification to the test machines, the results of the first two experiments can be used to estimate the number of servers required to achieve acceptable performance for a given number of users in certain simulations.

The last two experiments focus on determining the scalability of the system. The purpose of *PIM* is to alleviate the missed interaction problem, which occurs when the time taken by the interest management approach to resolve the objects' interests is longer than the duration of the pair of objects' interaction. The rationale behind *PIM* is to use an enlarged aura, the *PAI*, to exchange $APUM_{local}$ (pairs of objects' message transmitted at variable frequencies) to notify the existence of objects before their auras overlap. However, due to the additional message exchange, the performance of the system may degrade due to the increased message transmission. This can limit the scalability of the system.

The third set of experiments is intended to investigate the effect of $APUM_{local}$ messages on the performance of the system. The final set of experiments concentrates on demonstrating the scalability offered by the decentralized server architecture employed in the *PIM* system. These experiments are intended to investigate the effect of increasing the number of servers on the overall performance of the system. The desired effect, which would imply that the system is scalable, is for the system to be able to support larger numbers of objects as the number of servers is increased, while maintaining acceptable performance. It is also expected that an increased number of servers with the same number of objects supported will result in equal or better performance, provided the number of objects being supported is not small.

Although the different experiments have their own purposes, all the experiments are based on the same conditions:

- There is no application-imposed restriction on the frequency a server will transmit an aggregated *PUM* message to the other servers; it depends on the processing speed of the server and the volume and frequency of *PUM* messages the server receives from the nodes.
- The frequency a server sends an aggregation $APUM_{admin}$ message, which is the message exchanged between different servers, is set to one message every 10 seconds.

- The drop percentage is calculated by comparing the number of messages sent by the nodes with the number of messages received by the servers. The percentage drop rate, *PD*, is calculated using the following formula (Total(Sent)= the number of messages sent by all nodes in the DVE, Total(Received)= the number of messages received by the servers):

$$PD = \frac{(Total(Sent) - Total(\text{Re}ceived))}{Total(Sent)}$$

Formula 5.8 Calculating the Drop Rate

Experiment 1: Maximum Number of Objects Supported by One Server

- **Purpose**

  This experiment is intended to determine the number of objects a server can support simultaneously. Due to physical restrictions, such as CPU speed and the amount of free memory, the number of objects a server can support simultaneously is limited. As the number of objects increases, servers can be added to improve the scalability of the system. Although objects are evenly distributed between servers in this system, each server still hosts a copy of every remote object as *PIM* checks the intersection between remote objects and local objects. Therefore, the number of local objects, after distributing them evenly, should be less than the maximum numbers a server can support.

- **Methods**

  In these experiments, the number of objects is increased from 500 to 1500 by increments of 500. The number of servers is fixed at 1. Each test lasts one hour and is repeated three times.

- **Results**

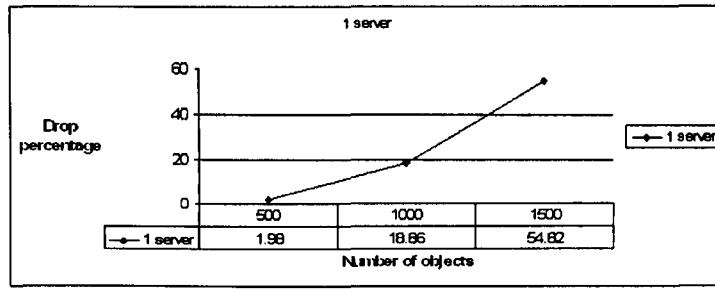


Figure 5.2 Average Single Server Results

- **Analysis**

According to Figure 5.2, the message drop rate increases steeply as the number of objects increases. As can be seen from the graph, with 500 objects, the drop rate was just 1.96%; with 1000 objects, the drop rate increases 16.9% compared with 500 objects; with 1500 objects, the drop rate reaches 54.82%. The performance of the system sharply degrades. The reason for this is that the server received more messages than it can handle per second. Therefore, under the current test conditions, the maximum number of objects a server can support is 1500 objects.

Experiment 2: *PUM* Frequencies

- **Purpose**

As mentioned previously, due to limited processing and memory resources, the number of objects a server can support is restricted. Additionally, the frequency a node sends *PUM* messages to the local server must be limited to some extent to avoid intolerable drop rates. Therefore, the purpose of this experiment is to determine the maximum acceptable frequency a node can send *PUM*s to its local server.

- **Methods**

  In these experiments, the number of objects is increased from 500 to 1500 by increments of 500. The number of servers is increased from 1 to 3 servers. Each test lasts one hour and is repeated two times. The frequency each node sends *PUM*s to its local server ranges from 2 messages per second to 5 messages per second. Following this, a further set of experiments was conducted in which the nodes transmitted *PUM*s at 10 messages per second.

- **Results**

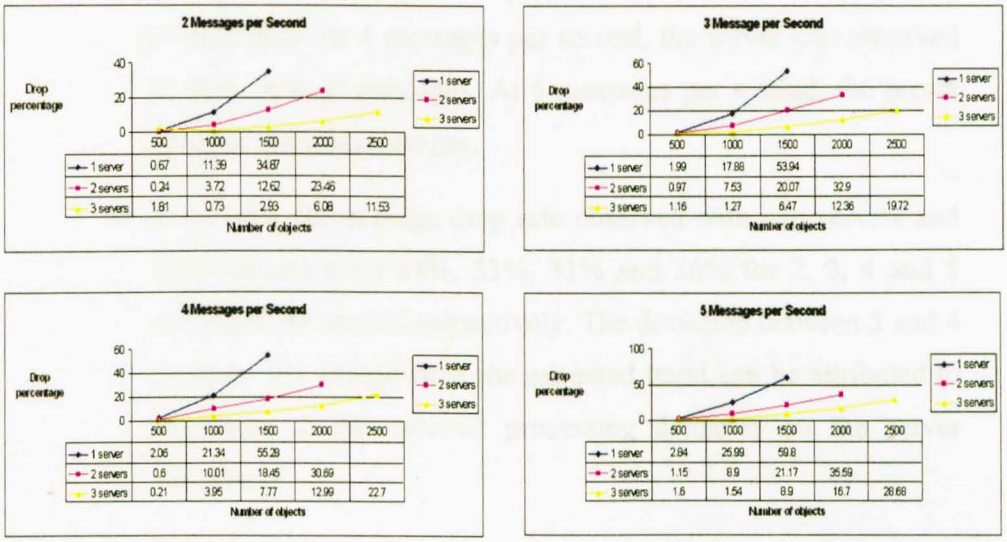


Figure 5.3 Varying the Number of PUM Messages per Second

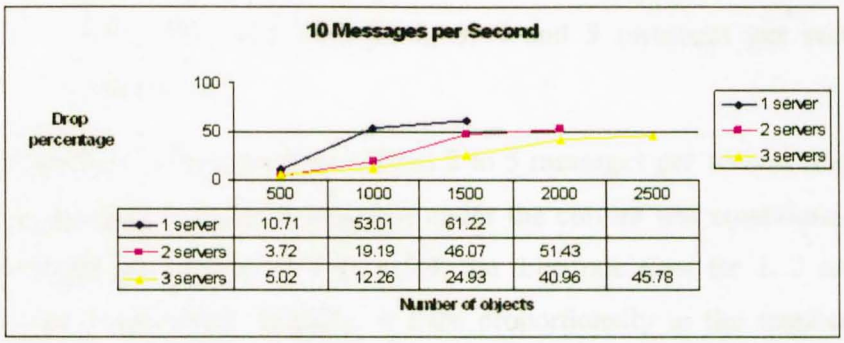


Figure 5.4 10 Messages per Second

- **Analysis**

  As can be seen from Figure 5.3, the *PIM* system's drop rate increases when the *PUM* transmission frequency is increased. In order to analyse the results, only the experiments with the largest number of objects for each number of servers will be discussed (1500, 2000 and 2500 objects for 1, 2 and 3 servers respectively):

  - **1 Server**: at 2 messages per second transmission rate, a single server with 1500 objects drops less than 40% of the messages. At 3 messages per second, a single server was observed to drop 53% of messages. At 4 messages per second, the server was observed to drop 56% of messages. At 5 messages per second, the server dropped 60% of messages.

  - **2 Servers**: the average drop rate observed with two servers and 2000 objects were 24%, 33%, 31% and 36% for 2, 3, 4 and 5 messages per second respectively. The deviation between 3 and 4 message per second from the expected trend can be attributed to variations in the external processing demands on the server machines.

  - **3 Servers**: the trend in drop rate is very obvious in the 3 server experiments. The message drop rates grow proportionally to the frequency of message transmission. The drop rates were 12%, 20%, 23% and 29% for 2, 3, 4 and 5 messages per second respectively.

  In addition to the experiments from 2 to 5 messages per second, Figure 5.4 provides an extreme situation under the current test conditions: 10 messages per second. In Figure 5.4, the drop rate rises for 1, 2 and 3 servers respectively. Initially, it rises proportionally to the number of objects, as was observed with lower frequency message transmission.

However, the drop rate increase appears to reach some plateau in which the rate of increase reduces. This may indicate that this specific experiment is reaching the computational and message-handling limits the servers are capable of. If the number of objects was further increased, it is assumed that the drop rate increase would eventually tend towards zero, as the drop rate approached a limit, 100%. This phenomenon can be explained using a fairly simple model. Given M is the maximum number of messages can be processed by the system in one second; O is the number of objects participating in the DVE simultaneously; R is the message transmission frequency an object sends messages to the system; N is the number of messages transmitted per second, which is also equivalent to $O * R$; D is the message drop rate, which includes both the messages dropped in transmission and in the system buffers. D can be represented using the formula below:

$$D = \frac{N - M}{N} = 1 - \frac{M}{N} = 1 - \frac{M}{O * R} = 1 - \frac{M}{R}\left(\frac{1}{O}\right)$$

Formula 5.9 Approximate Drop Rate

Given that M and R are constants, D is the function with respect to O.

$$D = f(O)$$

Therefore, the derivative of D with respect to O obtains the rate of change of D. According to [MathWorld05], the derivative of function x^n, where x is a variable, is:

$$\frac{d}{dx}x^n = nx^{n-1}$$

Therefore, the derivative of D with respect to O is:

$$D' = f'(O) = \frac{d}{dO}\left[1 - \frac{M}{R}\left(\frac{1}{O}\right)\right]$$
$$= -\left[-\frac{M}{R}(O^{-2})\right]$$
$$= \frac{M}{R}\left(\frac{1}{O^2}\right)$$

Formula 5.10 Approximate Drop Rate Deviation

As $O \Rightarrow \infty$, $\frac{1}{O^2} \Rightarrow 0$, therefore, as O (the number of objects) becomes large, the drop rate deviation tends towards 0 as the drop rate tends towards 100%.

Experiment 3: $APUM_{local}$ Overhead

- **Purpose**

  This experiment is designed to determine the overhead of $APUM_{local}$ message exchange. As mentioned previously, the difference between the *PIM* system and a traditional aura-based interest management system is *PIM*'s utilisation of an additional message, $APUM_{local}$, to pre-empt potential aura intersections. However, this additional message exchange may degrade the system's performance. Therefore, it is necessary to determine to what extent the additional message, $APUM_{local}$, affects performance.

- **Methods**

  A simple aura-based system was implemented to determine the overhead of the additional message, $APUM_{local}$, in the *PIM* system compared to existing aura-based systems. In the aura-based system, there are only two types of messages exchanged between the servers: *PUM* and $APUM_{admin}$. Apart from the absence of $APUM_{local}$ messages, the two systems are identical. The frequency each node transmits *PUM*s to its local server is

set to 3 messages per second in these experiments. Each experiment lasts for one hour. Results are taken as the average of three runs. The number of servers ranges from one to three servers.

- **Results**

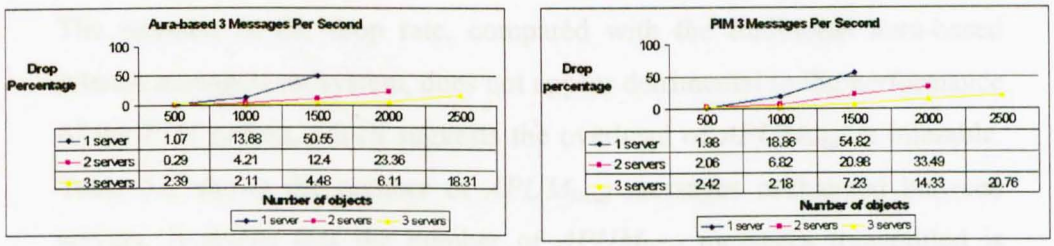


Figure 5.5 System Comparison

	500	1000	1500	2000	2500
1 server	0.91	5.18	4.24	-	-
2 servers	1.77	2.61	8.58	10.13	-
3 servers	0.03	0.07	2.84	8.22	2.45

Table 5.1 Drop Rate Deviation

	500	1000	1500	2000	2500
1 server	449766	527431	502570	-	-
2 servers	829483	1095555	1131482	1154284	-
3 servers	1087085	1588617	1688556	1689022	1656775

Table 5.2 Number of $APUM_{local}$ Handled by *PIM* Servers

- **Analysis**

In Figure 5.5, two sets of results are presented to show the average percentage drop rate of the two systems. In both system, the drop rate decreases as the number of servers increases. However, compared with the aura-based system, the message drop rate in the *PIM* system is higher in general. The reason for this is that an extra type of message, $APUM_{local}$, is

exchanged between *PIM* servers; therefore, the servers are responsible for generating, sending, receiving and processing $APUM_{local}$. As can be seen from Table 5.1, the drop rate deviation between the two systems ranges between [0.9%, 5.18%] in the 1 server experiments, [1.77%, 10.13%] in the 2 servers experiments and [0.03%, 8.22%] with 3 servers.

The increase in the drop rate, compared with the traditional aura-based interest management system, does not appear detrimental to the performance of the *PIM* system, which suggests the overhead of $APUM_{local}$ is tolerable. Table 5.2 shows the number of $APUM_{local}$ messages exchanged between servers. It shows that the number of $APUM_{local}$ messages transmitted is directly proportional to the number of servers. The table also shows that the number of $APUM_{local}$ messages transmitted increases as the number of objects increases, although this increase does not appear to be directly proportional. Although the $APUM_{local}$ messages cause an increase in the drop rate, without the $APUM_{local}$ messages *PIM* would not be able to effectively alleviate the missed interaction problem which is exhibited in current interest management systems.

Experiment 4: Scalability Tests

- **Purpose**

  After analysing the overhead of the $APUM_{local}$, these experiments intend to determine the scalability of the system. According to the results displayed in the first experiment in this chapter, the maximum number of objects a server can support simultaneously is 1500 objects under the current test conditions. However, in the following experiments, the number of objects the *PIM* system must support is increased as more servers are added to the system. The experiments in this section are intended to determine the scalability of the system by increasing the

number of objects simultaneously participating in the DVE while the number of servers is increased from one to ten.

- **Methods**

  The number of servers is increased from 1 to 10 and the number of objects increased from 500 to 6000 in increments of 500. The numbers of objects tested on 1 server ranges between 500 and 1500. The maximum number of objects in each test will increase by 500 for each server added to the system. The frequency that each node sends *PUMs* to its local server is fixed at 3 messages per second in these experiments. The duration of each run is one hour and each experiment is performed 3 times.

- **Results**

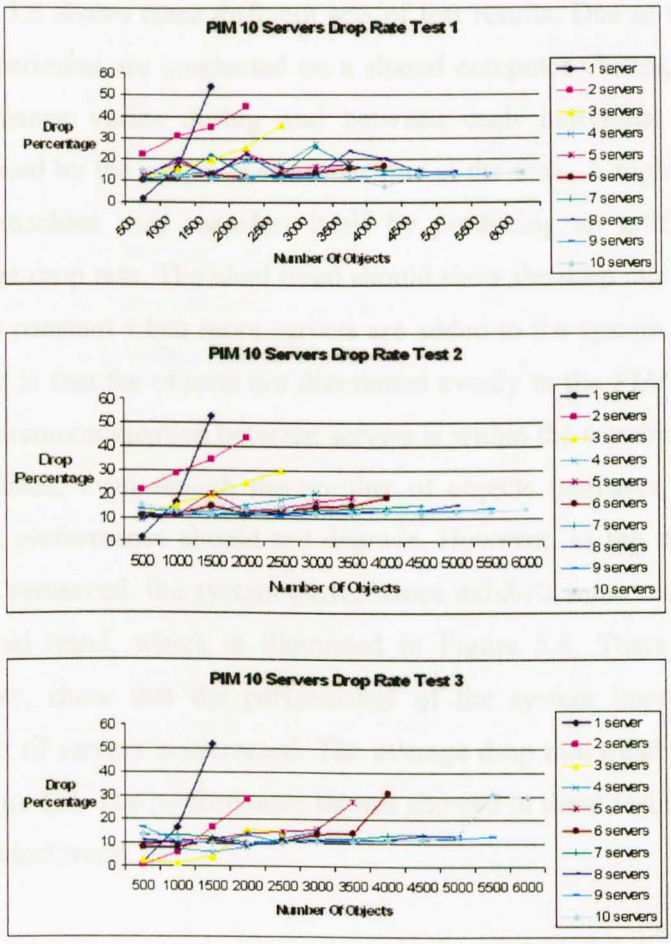


Figure 5.6 Scalability Results

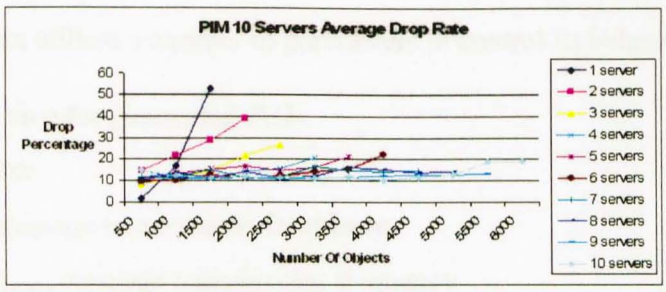


Figure 5.7 Average Scalability Results

- **Analysis**

  Figure 5.6 shows three different sets of test results. Due to the fact that the experiment are conducted on a shared computer cluster, the system performance varies during and between each individual run, being influenced by the usage level of machine at the time. A high usage level for a machine may manifest itself by producing an artificially high message drop rate. The ideal trend should show the drop rate decrease or remain constant when more servers are added to the system. The reason for that is that the objects are distributed evenly in the *PIM* servers and the intercommunication between servers is within the servers' processing capabilities; even though the number of objects increases, the overall system performance should not degrade. However, as the machines are shared resourced, the system performance exhibits some variation from the ideal trend, which is illustrated in Figure 5.6. These results do, however, show that the performance of the system improves as the number of servers is increased. The average drop rate trend is displayed in Figure 5.7. The performance figures showed in these results are closer to the ideal trend.

5.4 Parameter Selection

The *PIM* system utilises a number of parameters to control its behaviour:

- Future time for generating *PAIs*
- Aura size
- *PUM* message transmission frequency
- $APUM_{admin}$ message transmission frequency

Each of these parameters can be modified to result in different effects on the *PIM* system. For example, increasing the future time value will result in larger *PAIs*, which can reduce the likelihood of missed interactions occurring.

However, this will result in more messages being transmitted and may, therefore, contribute towards network congestion. In the event of network congestion, message transmission delays may be detrimentally affected, decreasing responsiveness and increasing the probability of missed interactions occurring. Similar behaviour can be seen by adjusting the *PIM* system's other parameters. As such, the choice of appropriate parameters for predictive interest management is important and essentially requires the adjustment of these parameters to provide high levels of responsiveness while alleviating the missed interaction problem and minimising *PIM*'s contribution to network congestion.

Recent work has been undertaken to develop a simulator to determine the effect of these parameters on the occurrence of missed interactions [Parkin06]. This work concentrates on identifying under which circumstances missed interactions occur and the frequency of their occurrence. Experiments were conducted by varying aura size and message transmission frequency respectively. The results of these experiments showed that increasing aura sizes and message transmission frequency resulted in fewer missed interactions occurring.

5.5 Summary

This chapter introduced the test environment, which is a shared resource in the School of Computing Science at Newcastle University. In this test environment, each server occupies one machine and nodes are distributed evenly across a set of non-server machines to create synthetic networking traffic. The world simulator and object simulator components were introduced, which are two pieces of software to simulate a simple virtual world and objects' movements respectively. Using these components, it is possible for different objects to observe and navigate through the same virtual world in a pseudo-intelligent fashion. Four different set of experiments were conducted:

- To provide a guideline for the development of DVEs built on top of the *PIM* systems to ensure a minimum level of performance, e.g. drop rate below a threshold for a given number of objects and *PUM* transmission frequency;
- To demonstrate the *PIM* system's scalability.

The first two sets of experiments demonstrated the maximum number of objects a server could support on the test machines. The results also displayed the drop rates corresponding to different *PUM* message transmission frequencies with a given number of objects and servers. Developers of DVEs can use these results to assist in choosing appropriate system variables (e.g. number of servers, PUM transmission frequency, and maximum number of supported objects). It is important to balance the variables as their effects on the overall system performance are inter-related, as can be seen from the results. For example, given the number of servers and maximum number of supported objects, the developers of DVEs can determine the *PUM* transmission frequency to achieve a minimum system performance.

The third set of experiments showed that the additional message introduced by the *PIM* system to alleviate the missed interaction problem, $APUM_{local}$, does not cause a major degradation in the overall system performance, implying that the *PIM* system is scalable. The final set of results showed that the *PIM* system achieves a high level of scalability by employing the de-centralised server communication model, which was discussed in detail in Chapters 2 and 4. These results showed that the system's performance can be improved by increasing the number of servers.

In addition, a brief description of the considerations required when adjusting the parameters the *PIM* system works upon was provided in Section 5.4.

Chapter 6

Conclusions

This chapter provides a summary of this thesis, the contributions it makes, and discusses future work.

6.1 Thesis Summary

A Distributed Virtual Environment (DVE) is a virtual environment which allows dispersed users to interact with each other and the virtual world through the underlying network. To build a DVE, the developer not only needs to handle the issues of a single-user virtual environment, such as collision detection and rendering, but is also required to deal with the issues of a distributed system, such as network latency and bandwidth usage.

Three layers should be provided to build a DVE: application layer, message dissemination layer and network layer.

Layer	Purposes
Application Layer	Provides users a representation of a virtual world allowing users to interact with it and other users through input/output devices
Message dissemination Layer	• Provides developers a network layer API which eases programming • Provides facilities to ensure interoperability with heterogeneous network architectures and platforms • Provides location and discovery services • Provides filtering mechanisms to reduce the number of unnecessary messages which are transmitted over the underlying network • Provides services to regulate message transmission frequency according to some filtering criteria • Provides the developers the choice of synchronous and asynchronous messaging models
Network Layer	Provides network protocols to enable high-levels of accessibility to the DVE over LANs and public access networks, e.g. the Internet

Table 6.1 Purposes of DVE Layers

Users interact with a DVE through the application layer. These interactions are manifested as events which are passed to the message dissemination layer. Upon receipt of these events, the message dissemination layer processes events into messages and instructs the network layer to send these messages to their required recipients. In order to achieve this, the message dissemination layer can employ a wide variety of techniques, such as message filtering and message frequency regulation, to efficiently utilise the available bandwidth and improve scalability. When the network layer receives a message from the underlying

network, it is passed to the message dissemination layer. Upon receiving a message from the network layer, the message dissemination layer generates a corresponding event, which is passed to the application layer. This event manifests itself as a state update for one or more objects, which is reflected in the output devices to the users.

Middleware can be incorporated into the message dissemination layer to ease DVE development. Middleware is a class of software that resides between an application and the operating system. It shields the application developer from the complexity of networking issues and provides them with services to ease the development of distributed applications. It provides interoperability for a DVE to overcome heterogeneity between networks; it provides easier access to the network layer; it provides the choice of the synchronous and asynchronous message models; it provides location and discovery services.

Interest management can be built on top of middleware to provide message filtering and message regulation mechanisms in the message dissemination layer. There are three interest management approaches: region-based, aura-based and hybrid interest management. Region-based interest management divides a virtual world into different regions; objects residing in the same or neighbouring region can interact with each other through message exchange in the underlying network. Aura-based interest management specifies an aura (a sphere) for each object; objects can interact with each other as long as they fall into each other's auras. Hybrid interest management is the combination of the region-based and aura-based approaches. However, all of the existing approaches do not address the *Missed Interaction Problem*. Missed interactions occur when the time for an interest management approach to resolve the interaction between a pair of objects is longer than the duration of these objects' interaction. For example, a pair of objects, such as high-speed airplanes, might interact with each other, in terms of falling into the same region or their auras overlap, very briefly, say 50 milliseconds. However, it might take the interest management approach 100

milliseconds to detect the interaction. When the interest management approach detects this interaction, these objects may have passed by or crashed into each other. Therefore, a new interest management approach should be provided to alleviate the missed interaction problem.

The choice of communication models can influence the scalability, consistency and responsiveness of a DVE. The peer-to-peer communication model involves direct communication between nodes. This provides minimal network latency; therefore, it should theoretically provide the best responsiveness for a DVE. However, as the number of nodes increases, each node is required to communicate with an increasingly large number of nodes; if the number of nodes becomes sufficiently large, the node will become overloaded and the consistency and responsiveness of the DVE will dramatically degrade. In addition, the scalability of a DVE is limited due to the huge amount of message exchange between every node, which may cause network congestion. The centralised server communication model uses a server to connect all clients. It can provide the most consistent DVE as the server can act as a central repository for object states. However, a single server is potentially a bottleneck; the DVE's scalability is limited by the processing power of its server. Furthermore, if the volume of messages needed to be processed overloads the server, the responsiveness of DVE will significantly degrade. The de-centralised server communication model utilises multiple servers to facilitate intercommunication between geographically dispersed clients. Compared with the other two models, this model provides the best scalability. In addition, due to the participation of multiple servers, the message processing requirements of a DVE are distributed between the servers, resulting in a consistent and responsive DVE.

Chapter 3 described a new interest management approach, termed *Predictive Interest Management (PIM)*, which is intended to alleviate the missed interaction problem in DVEs. The rationale behind *PIM* is to enlarge the objects' auras such that messages will be exchanged before objects' auras

overlap. However, due to the extra message exchange in the underlying network, additional network bandwidth will be consumed. Therefore, it is necessary to regulate the message exchange frequency. The concepts of Predictive Area of Influence (*PAI*), Collision Window (*CW*) and its associated values (*UPV*, *OUPV* and *AUBV*) were introduced to calculate the appropriate message exchange frequency between nodes. These were used to construct a message exchange schema, based on the intersection degree of a pair of objects, to regulate the message exchange types and frequencies between the relevant nodes. Therefore, as two objects approach each other, prior to their auras overlapping, the message exchange frequency should increase until it reaches the maximum message exchange frequency (the message exchange frequency when objects' auras overlap).

Chapter 4 described the structure and implementation of the *PIM* system. The *PIM* system utilises predictive interest management to filter unnecessary message exchange between nodes and alleviate the missed interaction problem; CORBA is adopted as the middleware to handle the networking issues and to provide interoperability between heterogeneous networks; the de-centralised server communication model is adopted to improve scalability.

Chapter 5 provided experiments to evaluate different aspects of the *PIM* system:

- The number of objects a single server can support;
- The upper bound of message exchange frequency;
- The overhead of *PIM*'s additional message exchange;
- The scalability of (number of objects which can be supported by) the *PIM* system as the number of servers is increased.

The results from these experiments demonstrate that the *PIM* system provides a scalable middleware for DVEs. They show that the overhead of the additional message exchange required in *PIM* does not have a significant impact on the performance of the system. The results also demonstrate that as the number of

servers is increased, the performance of the system and the number of objects which the system can support increases.

6.2 Contribution of Thesis

The core contribution of this thesis lies in providing a new interest management approach, termed *Predictive Interest Management (PIM)*, its implementation and evaluation.

- *Predictive Interest Management*: an aura-based interest management approach which utilises expanded auras and predictive techniques to initiate message exchange at appropriate frequencies depending on the intersection degree of the objects' expanded auras. This technique utilises variable-frequency message exchange to minimise the impact of the additional message exchange on system scalability. While it is impossible to fully eradicate the missed interaction problem, as network latency may be arbitrarily large in the case of network congestion or network failure, this is a best-effort approach to alleviate the missed interaction problem.

- Implementation: this thesis described the *PIM* system, which implements the predictive interest management algorithm to alleviate the missed interaction problem. This system provides an interoperable interest management middleware, which fulfils the requirements of the message dissemination layer in the three-tier DVE architecture described in Chapter 2. The *PIM* system is built on top of an existing middleware standard, CORBA, which offers platform and language independence. CORBA adopts the Internet Inter-ORB Protocol (IIOP) as its network-layer protocol, which is built on top of TCP/IP, to provide support for Internet deployment. The *PIM* system utilises the asynchronous messaging model to support large-scale message exchange in real-time.

The *PIM* system adopts the de-centralised server communication model, whereby a node connects to its geographically closest server to participate in the DVE. It utilises inter-server message exchange to enable interaction between objects hosted on nodes connected to different servers. This ensures that the computational and network overhead of interest management and message dissemination is fully distributed between the servers. The choice of geographically-closest server removes the overhead of dynamically connecting to servers based on some application-level criteria.

- Evaluation: this thesis provided experimental results to evaluate the *PIM* system in Chapter 5. A number of experiments were conducted to evaluate different aspects of the system to determine:
    - The number of objects a single *PIM* server can support
    - The message exchange frequency upper-bound between a node and its local server
    - The overhead of the *PIM* system's additional message exchange compared with traditional aura-based interest management.
    - The scalability of the system, in terms of the number of objects which can be supported, as the number of servers is increased.

  The experimental results demonstrated that the *PIM* system is scalable. They showed that the *PIM* system's additional message exchange has only a marginal effect on the overall performance of the system. In addition, the results showed that the maximum number of objects the *PIM* system can support increased proportionally to the number of servers.

6.3 Future Work

There are a number of avenues for future work:

- Adaptive message exchange frequency
- Higher-order functions to determine *PAIs*
- Load-balancing on servers
- Integration with a graphics engine

Adaptive Message Exchange Frequency

The current *PIM* system uses initialisation-time constants for maximum message exchange frequency between nodes and servers, and administrative message exchange frequency between servers. In addition, it utilises a system-wide fixed constant value to calculate the time displacement by which the objects' auras are expanded (termed future time). It would be desirable to provide a mechanism to adapt the message exchange frequency to the characteristics of the objects, such that high-speed objects transmit state updates more frequently than low-speed objects. In addition, the use of an adaptive value for future time could help further alleviate the missed interaction problem if network transmission delays rise, whereas network utilisation could be improved by reducing future time if network transmission delays fall.

Higher-order Functions for Predicted Area of Influence (PAI)

Currently, the *PAI* of an object in the *PIM* system is calculated by summing the object's aura's radius with its maximum speed multiplied by a fixed-value future time *(ft)*. However, it may be possible to reduce the radius of the *PAI* by using higher-order functions, such as the rate of change of velocity (acceleration), to predict the distance an object can travel more accurately over *ft*. For example, given an object, such as a tank, which has a relatively high maximum speed (60m/s), but slow acceleration ($4m/s^2$); if this tank is stationary at time *t*, it is unnecessary to use a *PAI* containing the area it can cover over *ft* at top speed, as the tank is not capable of accelerating to top speed before $t + ft$. This could be further extended to consider aspects such as turning circles, braking etc., to further reduce the volume of the *PAI*.

Load Balancing on Servers

In the current *PIM* system, each server is connected to nodes which are geographically closest to it. However, it is possible with this architecture for a particular geographical region's server to become more heavily loaded than others, which could compromise the system's performance. To alleviate this issue, multiple servers could be applied for a given region. This would enable a server to redirect a connecting node to another local server if its load was greater than some predefined threshold value to avoid itself from becoming overloaded. This would enable the scalability, consistency and responsiveness of the DVE to be further improved by adding additional servers to geographical regions with large numbers of users.

Integration with a Graphics Engine

To reiterate, the *PIM* system is an experimental system, which only provides a complete implementation of the message dissemination and network layers. Its current application layer does not provide a graphical user interface, but instead simply propagates state update messages to its server using random way-point generation to move its objects throughout the DVE. It would be desirable to provide a more fully-featured application layer, integrated with a graphics engine, to produce a complete DVE for dispersed users to interact within.

BIBLIOGRAPHY

[Abrams99]

H. A. Abrams, "Extensible Interest Management for Scalable Persistent Distributed Virtual Environments", PhD Thesis, Naval Postgraduate School, Monterey, California, 1999.

[Barrus96]

J. W. Barrus et al, "Locales and Beacons: Efficient and Precise Support For Large Multi-User Virtual Environments", a mitsubishi electric research laboratory (MERL), TR-95-16a, 1996.

[Benford94]

S. Benford et al, "Managing mutual awareness in collaborative virtual environments", VRST'94, 1994.

[Benford97]

S. Benford et al, "Introducing Third Party Objects into the Spatial Model of Interaction", CRG Internal Report, 1997.

[Bharambe02]

A. R. Bharambe et al, "Mercury: A Scalable Publish-Subscribe System for Internet Games", Netgames2002, 2002.

[Bharambe04]

A. R. Bharambe et al, "Mercury: Supporting Scalable Multi-Attributed Range Queries", SIGCOMM'04, Portland, Oregon, USA, 2004.

[Brose01]

G. Brose et al, "Java Programming with CORBA: advanced techniques for building distributed applications, Third Edition", John Wiley and Sons, 2001.

[Castro01]

M. Castro et al, "SCRIBE: A large-scale and decentralized publish-subscribe infrastructure", NGC2001, 2001.

[Cerf74]

V. G. Cerf and R. E. Kahn, "A protocol for packet network interconnection", IEEE Transaction on Communication, Vol. COM-22, V5, pp. 627-641, May 1974.

[Churchill01]

E. F. Churchill, D. N. Snowdon and A. J. Munro, "Collaborative Virtual Environments: Digital places and spaces for interaction", Springer-Verlag London Limited, 2001.

[Cohen94]

D. Cohen, "NG-DIS-PDU: The next generation of DIS-PDU (IEEE 1278)", in the proceedings of the 10th Workshop on Standards for Distributed Interactive Simulations, 735-742, March 1994.

[Comer91]

D. E. Comer, "Internetworking with TCP/IP Vol.1: Principles, Protocols, and Architecture (2th Edition)", Chapter11, pp. 159-168, Chapter, pp. 171-202, Chapter17, pp. 281-290, Prentice-Hall, inc, 1991.

[Das97]

T. K. Das, et al, "NetEffect: A Network Architecture for Large-scale Multi-user Virtual Worlds", ACM VRST' 97, Lausanne Switzerland, 1997.

[Frecon98]

E. Frécon and M. Stenius, "DIVE: A Scaleable network architecture for distributed virtual environments", Distributed Systems Engineering Journal special issue on Distributed Virtual Environments, Vol. 5, No. 3, pp. 91-100, Sept. 1998.

[Funkhouser95]

T. A. Funkhouser, "RING: A Client-Server System for Multi-User Virtual Environments", AT&T Bell laboratories, 1995.

[Greenhalgh95]

C. Greenhalgh et al, "MASSIVE: a Distributed Virtual Reality System Incorporating Spatial Trading", the 15th International Conference on Distributed Computing Systems (DCS'95), pp.27-34, 1995.

[Greenhalgh96]

C. Greenhalgh, "Dynamic, embodied multicast groups in MASSIVE-2", Technical Report NOTTCS-TR-96-8, 1996.

[Greenhalgh00]

C. Greenhalgh et al, "Inside MASSIVE-3: Flexible Support for Data Consistency and World Structuring", CVE2000, 2000.

[Kuhl99]

F. Kuhl, R. Weatherly and J. Dahmann, "Creating Computer simulation systems: an introduction to the high level architecture", Prentice Hall PTR, 1999.

[Lee02]

D. Lee et al, "ATLAS – A Scalable Network Framework for Distributed Virtual Environments", CVE'02, September 30-October 2, Bonn, Germany, 2002.

[Lengyel02]

Eric. Lengyel, "Mathematics for 3D Game Programming and Computer Graphics", Charles River Media Inc, Hingham, MA, 2002.

[Lopez02]

P. G. Lopez et al, "MOVE: Component Groupware Foundations for Collaborative Virtual Environments", CVE'02, September 30-October 2, Bonn, Germany, 2002.

[Lu03]

F. Lu et al, "Predictive Interest Management: An Approach to Managing Message Dissemination for Distributed Virtual Environments", In Proceedings of the First International Workshop on Interactive Rich Media Content Production: Architectures, Technologies, Applications, Tools (Richmedia2003) 2003.

[Lu05]

F. Lu et al, "Interest Management Middleware for Networked Games", ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, Washington, DC, April 3-6, pp 57-63, ACM SIGGRAPH 2005.

[Macedonia95]

M. Macedonia, "A Network Software Architecture for Large-Scale Virtual Environments", Ph.D. thesis, computer science department, naval postgraduate school, Monterey, CA, USA, 1995.

[Milgram99]

P. Milgram, Herman W. Colquhoun Jr., "A FRAMEWORK FOR RELATING HEAD-MOUNTED DISPLAYS TO MIXED REALITY DISPLAYS", proceedings of the human factors and ergonomics society 43rd annual meeting.

[Ng02]

B. Ng et al, "A Multi-Server Architecture for Distributed Virtual Walkthrough", ACM VRST'02, Hong Kong, November 11–13, 2002.

[Okanda05]

P. Okanda and G. Blair, "OpenPing: A Reflective Middleware for the Development of Adaptive Networked Game Applications", SIGCOMM'04 Workshops, Aug. 30 & Sept. 3, 2004, Portland, OR, USA, ACM.

[Parkin06]

S. E. Parkin, P. Andras and G. Morgan, "Managing Missed Interactions in Distributed Virtual Environments", Proc. of the 12th Eurographics Symposium on Virtual Environments, pp 27-34, Lisbon, Portugal, 8th - 10th May 2006.

[Ruh99]

W. Ruh, T. Herron and P. Klinker, "IIOP Complete: understanding CORBA and Middleware Interoperability", Addison-Wesley, 1999.

[Savetz96]

K. Savetz et al, "MBONE: Multicasting Tomorrow's Internet", Hungry Minds Inc, 1996.

[Sedgewick01]

R. Sedgewick and P. Flajolet, "An Introduction to the Analysis of Algorithms", Addison Wesley, 2001.

[Sedgewick03]

R. Sedgewick, "Algorithms in Java – 3rd Ed", Addison Wesley, 2003.

[Singh95]

G. Singh et al, "Bricknet: Sharing object behaviours on the Net", In Proceedings of the Virtual Reality Annual international Symposium (VRAIS'95), pages 19-25. Los Alamitos, CA, IEEE Computer Society Press, March 1995.

[Singhal99]

S. Singhal, M. Zyda, "Networked Virtual Environments: Design and Implementation", ACM Press, SIGGRAPH Series, Addison-Wesley, 1999.

[Smed02]

J. Smed, T. Kaukoranta and H. Hakonen, "A Review on Networking and Multiplayer Computer Games", TUCS Technical Report No 454, TurKu Centre for Computer Science, April 2002.

[Tanenbaum97]

A. S. Tanenbaum, "Computer Networks Third Edition", Prentice-Hall International Inc, 1996.

[Watt01]

A. Watt and F. Policarpo, "3D Games: real-time rendering and software technology", Chapter14, pp. 415-418, Chapter15, pp. 437-466, Addison-Wesley, 2001.

[Zyda93]

M. Zyda et al, "NPSNET: REAL-TIME VEHICLE COLLISIONS, EXPLOSIONS AND TERRAIN MODIFICATIONS", The Journal of Visualization and Computer Animation, Vol. 4, No. 1, 1993, pp. 13-24.

[Zyda93_2]

M. Zyda et al, "Hypermedia and Networking in the Development of Large-Scale Virtual Environments", In the Proceedings of the International Conference on Artificial Reality and Tele-existence, Tokyo, Japan, 6-8 July 1993.

[ActiveWorlds05]

http://www.activeworlds.com/tour.asp#, as viewed 8 Jan 2005

[Blizzard05]

Blizzard Entertainment, http://www.blizzard.com/, as viewed 17 July 2005.

[Bungert05]

Christof Bungert, http://www.stereo3d.com/hmd.htm as viewed 26 Nov 2005.

[CFAINET05]

CFAINET, http://www.cfainet.org/home/virtual_reality/vrprogram.asp, as viewed 8 Jan 2005

[Epic05]

Epic Games, http://www.epicgames.com, as viewed 19 July 2005.

[EverQuest99]

EverQuest,
http://www.absoluteastronomy.com/encyclopedia/e/ev/everquest1.htm, as viewed 18 July 2005.

[GIST05]

GIST, http://www.dcs.gla.ac.uk/~steven/haptics.htm, as viewed 18 July 2005.

[IBM05]

IBM, http://www.redbooks.ibm.com/abstracts/sg246842.html, as viewed 8 Jan 2005

[Id05]

Id Software, http://www.idsoftware.com, as viewed 19 July 2005.

[InternetSociety05]

Internet Society, http://www.isoc.org/internet/history/brief.shtml, as viewed 19 July 2005.

[Javvin05]

Javvin, http://www.javvin.com/protocols.html, as viewed 19 July 2005.

[Kuo01]

A. Kuo, "A (very) brief history of cheating", http://shl.stanford.edu/Game_archive/StudentPapers/BySubject/A-I/C/Cheating/Kuo_Andy.pdf , as viewed 16 July 2005.

[Kushner05]

D. Kushner, "Engineering EverQuest: Online gaming demands heavyweight data centers", http://www.spectrum.ieee.org/WEBONLY/publicfeature/jul05/0705eq.html, as viewed 15 July 2005.

[MathWorld05]

MathWorld http://mathworld.wolfram.com/Derivative.html as viewed 11 Dec 2005.

[Microsoft05]

Microsoft http://www.microsoft.com/com/default.mspx as viewed 11 Dec 2005.

[Mogul84]

Jeffrey Mogul, http://rfc.dotsrc.org/rfc/rfc919.html, as viewed 9 June 2005.

[Monty97]

Monty, http://www.bigkid.com.au/articles/00_07/diablo_ii_3.htm, as viewed 17 July 05.

[OMG05]

OMG, http://www.omg.org, as viewed 11 Dec 2005.

[Origin05]

Origin, http://www.owo.com/, as viewed 18 July 2005.

[Reilly99]

D. Reilly, "Inside Java: The Java Programming Language", http://www.javacoffeebreak.com/articles/inside_java/insidejava-nov99.html, as viewed 9 June 2005.

[Sun05]

Sun, http://java.sun.com/products/jms/overview.html, as viewed 11 Dec 2005.

[Sweeney99]

T. Sweeney, "Unreal Networking Architecture" http://unreal.epicgames.com/Network.htm, as viewed 16 July 2005.

[Ultima97]

Ultima Online, http://www.absoluteastronomy.com/encyclopedia/u/ul/ultima_online.htm, as view 18 July 2005.

Appendix A

messageservice.idl

```
module ms
{
      module idl
      {
          interface util
          {
           /*
            Vector3D is defined as the vector of an object
            @param x=x coordinate
            @param y=y coordinate
            @param z=z coordinate
           */
           struct Vector3D
           {
            double x;
            double y;
            double z;
           };

           struct SinglePumMessage
           {
            long user_id;
            long object_id;
            long world_id;
            Vector3D position;
           };

           struct ObjectProperty
           {
            long object_id;
            double aura;
            double object_radius;
            double tipl;//time interval of pum in local channel
            double topVelocity;
            Vector3D initialPosition;
           };

           struct SingleApumMessage
           {
            long ms_id;
            long world_id;
            long supplier_id;
            long object_id;
            double pia;
            double aura;
```

```
    Vector3D position;
    };


    struct MserviceToUserPumMessage
    {
     long ms_id;
     long world_id;
     SinglePumMessage userMessages;
    };

    typedef sequence<SinglePumMessage>  AggregatedPumMessage;

    typedef sequence<SingleApumMessage> AggregatedApumMessage;

    typedef sequence<MserviceToUserPumMessage> AggregatedMTUPumMessage;


    struct MSExchangeAdminApumMessage
    {
     long ms_id;
     long world_id;
     long maxSupllierId;
     AggregatedApumMessage userApumMessages;
    };

    struct MSExchangePumMessage
    {
     long ms_id;
     long world_id;
     long maxSupplierId;
     AggregatedPumMessage pum;
    };

    struct WorldInformation
    {
     long world_id;
      string url;
      string description;
     };

    typedef sequence <long> objectIds;

    typedef sequence<WorldInformation> WorldsInformation;

     typedef sequence<ObjectProperty> ObjectsProperties;

};

module excep
{
    exception WorldNotExist {string errorDetails; long world_id;};
    exception ObjectNotExist{string errorDetails;long object_id;};
    exception PermissionDenied {string errorDetails;};
    exception SubscriptionExceeded{string errorDetails;};
};

module clients
{
    interface MessageServiceUser: util
    {
     //called by ms server, to push the aggregated pum message to node
     oneway void receive_status_messages(in  AggregatedMTUPumMessage status_message);
```

```
    oneway void receive_status_message(in  MserviceToUserPumMessage status_message);

    void server_ready_signal(in long ms_id);
   };
};

module servers
{

   interface MessageService: util
   {
    //called by node, subscribe itself to message service
    string user_subscribed(in string subscriber, in string user_ior, out long user_id) raises
   (excep::SubscriptionExceeded);

    //called by node when it has been subscribed
    long get_ms_id();

    //called by node, unsubscribe itself to message service
    void user_unsubscribed(in long user_id);

    //called by node, subscribe itself to a certain world
    void world_subscribed(in long user_id, in long world_id) raises(excep::WorldNotExist);

    //called by node, unsubscribe itself to a certain world
    void world_ubsubscribed(in long user_id, in long world_id);

    //called by node,declaim the new object created by node to message service
    void add_object(in long user_id, in long world_id, in ObjectProperty pro);

    //called by node, declaim to remove one object to message service, if the object doesn't belong
    //to the node or the object isn't a shared object, PermissionDenied exception will be raised.
    void remove_object(in long user_id, in long world_id, in long object_id) raises
   (excep::PermissionDenied, excep::ObjectNotExist);

    //called by node and remove the specified world in message service, if the node isn't the owner of
    //the world, PermissionDenied exception will be raised
    void remove_world(in long user_id, in long world_id) raises (excep::PermissionDenied, excep::
    WorldNotExist);

    //called by node or other message service, get the information about the existing worlds
    WorldsInformation get_worlds();

    //called by node and specify information about the world and define a world id by it
    //own. If the world id has existed, message service will assign a new world id
    long create_world(in long user_id, in any world);

    //called by node, send structured aggregated (all objects information in one package) message
    //message is an any cast to UserPushStructuredMessage[].
    oneway void send_pum(in SinglePumMessage message);

    //called by other message service, indicate the host message service the new subscribed world
    void append_world(in long ms_id, in any world);

    //when user remove the his own world from the other message service, other message serivce call
    //this method to tell the host message service the world removal
    void displace_world(in long ms_id, in long world_id) raises(excep::PermissionDenied, excep::
    WorldNotExist);

    //called by other message service, declaim the entering status to the host message service
    void ms_subscribed(in string ms_domain, in string ms_name, in string ms_ior, in long ms_id);
```

```
        //called by other message service, declaim the leaving status to the host message service
        void ms_unsubscribed(in long ms_id);

        void different_ms_user_subscribed(in long ms_id, in long world_id, in SingleApumMessage
        initialInfo);

        void different_ms_user_unsubscribed(in long ms_id, in long world_id, in long user_id);

        void push_aggregated_ms_adminApum(in MSExchangeAdminApumMessage apum);

        oneway void push_aggregated_ms_apum(in SingleApumMessage apum);

        oneway void push_aggregated_ms_pum(in MSExchangePumMessage pum);
       };
    };
  };
};
```