Interest Management Middleware for Networked Games

Graham Morgan, Fengyun Lu, Kier Storey
School of Computing Science, University of Newcastle, UK

Abstract

In this paper we present an implementation of an interest management scheme using standard message oriented middleware (MOM) technologies to provide scalable message dissemination for networked games. The aim of all interest management schemes is to identify when objects that inhabit a virtual world should be interacting and to enable such interaction via message passing while preventing objects that should not be interacting from exchanging messages. The time taken by existing interest management schemes to resolve which objects influence each other may be too large to enable the desired interaction to occur. Furthermore, existing interest management implementations tend to be proprietary and are built directly on top of networking protocols. In this paper we present an approach to interest management based on the predicted movement of objects. Our approach determines the frequency of message exchange between objects on the likelihood that such objects will influence each other in the near future. We then demonstrate, via implementation and experimentation, how existing middleware standards provide a suitable platform for the deployment of our interest management scheme.

Keywords: middleware, games, networking

CCS: D.2.12 Interoperability - Distributed objects, I.3.2 Graphics Systems - Distributed/network graphics, C.2.4 Distributed Systems - Distributed applications

1 Introduction

Networked games allow a geographically dispersed set of players to participate in a shared gaming environment. A gaming environment may provide a virtual world that presents some geographic model populated with moving objects with game play achieved by players influencing the state of the virtual world. Typically, a player's console holds a sub-set of game state with players informing each other of their actions via the exchange of state update messages between consoles (either directly or indirectly via some intermediary server). The ability to allow players to maintain a mutually consistent view of the virtual world and still allow players to influence the virtual world in a timely manner are requirements that need to be satisfied to ensure players receive an appropriate gaming experience [Singhal and Zydra 1999]. A game may be played out in real-time, requiring player directed changes of the virtual world to be propagated to other players in real-time. In early versions of such games [Sweany 1999], the number of players that could participate in a virtual world was limited to relatively low numbers (e.g., less than 16) with the geographic placement of players restricted to the same LAN to guarantee the most satisfying gaming experience. Recently, higher bandwidth, lower latency networks to the home have allowed one or two hundred geographically distributed players to participate in games of this genre (e.g., [Sony 2004]). However, this is still viewed as limiting as the desire of players to participate in large, complex, virtual spaces with thousands of other players have provided a substantial demand for large scale networked games.

The scalability issues presented by a networked game relate, primarily, to the number of players and the complexity of the game (e.g., number of objects populating the virtual world). An increase in the number of players may lead to an increase in network traffic as a player's console attempts to propagate local game state updates to other players. If an attempt is not made to identify appropriate message recipients and so inhibit the sending and processing of unnecessary messages, then providing participation for many hundreds or thousands of players will not be possible. The process of identifying appropriate message recipients, referred to as interest management, introduces further computation and must be successfully completed in a timely manner to ensure the real-time and consistency requirements of a networked game are satisfied.

At present, networked games are built on standard protocols provided by existing network infrastructures (e.g., TCP/IP for the Internet) to ensure they are widely available for public use and so commercially viable. This approach results in enabling technologies that are proprietary and tend not to be suited to heterogeneous environments (i.e., deployment across varying platforms). Furthermore, programming distributed applications in this manner is a time consuming, non-trivial exercise. Therefore, middleware standards have been proposed (e.g., [OMG 2003]) that provide standard interfaces to services (e.g., location and discovery, security, inter-process communication) that ease the development of distributed applications. However, even the most recent networked games tend to be built directly above network layer protocols and do not benefit from the interoperability provided by existing middleware technologies.

This paper describes our approach to satisfying the following requirements:

- Provide a scalable interest management scheme suitable for networked games.
- Provide an implementation of our interest management scheme using standard middleware technology.

In the next section we describe existing approaches to interest management and the suitability of existing middleware

technologies for networked game development. In section 3 we describe our approach to interest management and in section 4 we describe our interest management implementation. Performance figures of our system are presented in section 5 with conclusions drawn from our work presented in section 6.

2 Background and Related Work

In this section we describe existing approaches to interest management and the common message dissemination approaches found in existing middleware.

2.1 Interest Management

Interest management is the term commonly used to describe restricted message dissemination between objects using virtual space division. Interest management schemes may be classified into two broad categories:

- **Regions**: In the region based approach [Miller and Thorpe 1995] the virtual world is divided into well defined regions that are static in nature (i.e., defined at virtual world creation time). The recipient of a message is limited to only interested participants (i.e., reside within the same, or neighbouring, region as the sender). An important consideration in a region based approach is the size of the regions. 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 region membership must be updated (identify a region an object belongs to). Determining a region size that is suitable for all objects in a virtual world may not be possible. For example, if region size is decided when considering the top speed of a fighter aircraft then the presence of soldiers traveling on foot may give rise to unnecessary message exchange between foot soldier objects that are separated by a distance too great for such objects to influence each other. If region size is more suited to foot soldier objects then a fighter aircraft may traverse region boundaries with such frequency that region membership may not be resolved in a timely fashion (object may traverse a region in less time than it takes to realize regional membership changes resulting in an inability for such an object to disseminate messages effectively).

- **Auras**: In the aura based approach each object is associated with an aura that defines an area of the virtual world over which an object may exert influence. An aura may be simply modeled as a sphere that shares its centre with the positional vector of the object it is associated with. An object may potentially communicate their actions only to objects that fall within their aura. This approach is illustrated by the Model, Architecture and System for Spatial Interaction in Virtual Environments projects (MASSIVE) [Greenhalgh and Benford 1995]. In the aura approach there is no need to regionalize a virtual world. However, there is a requirement for all objects to exchange positional update information in order to identify when aura collisions occur (we assume objects may influence each other when their auras overlap). The frequency of these message exchanges must be sufficient to ensure that aura collision may be determined in a timely fashion to allow objects to purposely disseminate messages as and when aura collision occurs. There is the possibility that aura collision may occur but objects are unaware of this as such a collision may not last for a sufficient amount of time to enable appropriate message recipients to be identified before objects move away from each other (aura collision no longer exists).

We describe the problem of resolving interest management in a timely manner to enable meaningful message dissemination between objects as the *missed interaction problem*.

2.2 Middleware

Middleware eases the development of distributed applications by providing a supporting software infrastructure. Popular open standards associated with such technologies are CORBA [OMG 2003] and Java RMI [Sun 1999]. However, the client/server model they provide is not well suited to networked games where the communication is asynchronous and message oriented in nature.

An approach to communications known as *message oriented middleware* (MOM) has been developed that may be more appropriate than the client/server model of communications for satisfying the messaging requirements of networked games. MOM decouples communications between sender and receiver (supplier and consumer), allowing one or more suppliers to issue messages for one or more consumers. MOM has been identified as suitable for providing the basis for message exchange for building distributed multimedia applications [Gore et al. 2001]. Furthermore, the approach appears well suited to providing a platform on which to build networked games. Mercury [Bharambe et al. 2002] is a message dissemination system based on MOM which, although proprietary, provides a positive assessment regarding the practicalities for supporting networked games via MOM. The Java Messaging Service (JMS) [Sun 2002] and the CORBA Notification Service (CORBA NS) [OMG 2000] are two popular MOM standards:

- **JMS** – Provides a standard API that allows Java developers to integrate MOM into their applications. The JMS specification does not indicate how the underlying system implementation is achieved (e.g., architecture) resulting in a number of varying solutions available from different vendors. JMS supports point-to-point and publish/subscribe models of interaction. Point-to-point is based on the notion of queues, with a queue identified as an asynchronous mechanism for passing messages from suppliers to consumers. A client may get all its messages delivered to a queue, allowing a queue to contain a variety of different message types. Publish/subscribe is based on topics, with clients publishing and subscribing to well defined topics. The topic acts as a mechanism for gathering and distributing related messages (as perceived by an application) to clients (publishers and subscribers).

- **CORBA NS** – Provides a standard API that extends the CORBA Event Service to include message filtering and a quality of service framework. The central element of the CORBA NS is a notification channel that assumes the role of propagating events from suppliers to

consumers. The basic function is to propagate all events to all consumers. However, the notification channel has the ability to filter events to provide more customizable message delivery for consumers. In addition to filtering, a number of quality of service parameters can be manipulated to tailor the behavior of a notification channel. CORBA NS is ideally suited to deployment in heterogeneous environments as the specification is based on the CORBA architecture (allowing applications to be written in a variety of languages and deployed on different platforms to interoperate).

JMS and CORBA NS both provide appropriate messaging services on which to deploy an interest management scheme. Only the identity of a topic (JMS) or notification channel (CORBA NS) is required by clients (suppliers and consumers) to enact message passing (the identity of consumers need not be known to suppliers). This provides benefits for large scale applications (such as networked games) as reconfiguration of existing suppliers/consumers is not required to introduce new suppliers/consumers. Furthermore, a JMS or CORBA NS implementation may be enhanced with scalable message dissemination that may be suitable for one-to-many message passing (e.g., [Deering 1989]) without alteration to their standard specifications.

When implementing our interest management scheme we chose CORBA NS over JMS. The main reason for this was the ability of CORBA NS to support applications other than those written in Java. Networked games (and most commercial games) are written in C or C++, often using proprietary game engine technologies. Even though our interest management scheme is implemented in Java (see section 4), we may support applications written in other languages that the OMG have defined language mappings to CORBA (including C++).

3 Predictive Interest Management

In this section we describe an interest management scheme based on the aura approach that attempts to overcome the missed interaction problem by using a three step approach to message exchange: (i) low frequency message exchange used for object discovery; (ii) variable message exchange, with frequency of messages between objects related to the probability of interaction of such objects in the future; (iii) high frequency message exchange between interacting objects. We present here only a brief overview of the technique to enable a reader to gain the basic underlying theory of the work.

3.1 Identifying Scope of Interest

The aura of an object describes an area of the virtual world enclosed by a sphere (Figure 1). The radius of an aura is specified on a per object basis and is fixed at object creation time. Objects have the ability to influence each other when their auras collide via the exchange of messages.

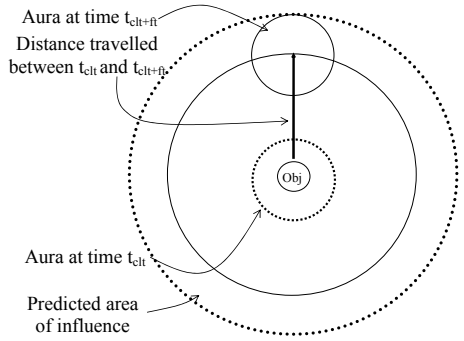

Figure 1: Defining Predicted Area of Influence (PAI).

A predicted area of influence (*PAI*) identifies the extent of an object's aura over a period of time given the distance an object may travel in a straight line in any direction (figure 1). The period of time used to identify a *PAI* is bounded by the current time, say t_{clt}, and some future time (t_{clt+ft}, where *ft* is a positive number and is defined system wide). By this method the distance an object, say obj_1, travels in a straight line identifies the radius of a sphere that encloses all the areas of a virtual space reachable by obj_1 between t_{clt} and t_{clt+ft} with the position vector of obj_1 at time t_{clt} 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* we assume an object is traveling at its highest speed (defined on a per object basis) in a straight line at time t_{clt} and continues at this speed and direction until t_{clt+ft}. This presents a *PAI* that is guaranteed to contain all possible auras of an object between t_{clt} and t_{clt+ft} irrelevant of an object's velocity. Assuming the highest speed remains constant for an object throughout its lifetime allows a *PAI* to be calculated and fixed at object creation time.

When the *PAI*s of two objects collide, but not their auras, there is a possibility that such objects may influence each other and subsequently exchange messages at some point in the near future. A period of time (*window of collision*) may be defined within which the auras of such objects may collide. Using collision detection techniques based on the intersection of spheres we may identify the separating distance between two objects as *sd*. If a collision window exists between two objects then *sd* may be used to determine an approximate upper bound value (*AUBV*) indicating the time taken for the auras of these objects to collide assuming they are traveling towards each other at their respective full speeds. *AUBV* provides a basis for determining frequency of message exchange between two objects within the same collision window.

3.2 Message Exchange

An object is responsible for sending a *positional update message* (*PUM*), identifying its position vector. *PUMs* are sent frequently and form the basic mechanism for message exchange between objects that are influencing each other. To determine if *PAI*s overlap objects must send an *admin PUM* (*APUM*) containing aura radius, *PAI* radius and vector position information. *APUM*s are sent less frequently than *PUM*s and form the basic mechanism for identifying when objects should exchange *PUM*s.

We assume the existence of a communications sub-system capable of providing reliable FIFO channels that may facilitate

inter-object communications via the sending and receiving of *PUM*s and *APUM*s:

- **Admin**: Used to disseminate *APUM*s to all objects.
- **Local**: Created on a per object basis to provide mechanism for passing *APUM*s and *PUM*s between objects without the need to send messages to all objects.

Three local timers, associated on a per object basis, are responsible for regulating the frequency of publishing *APUM*s on the admin channel (*ta*), *APUM*s on a local channel (*tal*) and *PUM*s on a local channel (*tp*) respectively. The value of *ta* should be set to a value that ensures *APUM*s may be received and processed by all objects in time to determine potential aura overlaps. The time interval *tal* used to define the frequency an object publishes *APUM*s on the local event channel is determined by *AUBV*. The time interval *tp* is defined on a per object basis at a developer's discretion (message exchange associated with interacting objects).

The modeling of variable frequency message exchange via admin and local channels is achieved as follows:

- **Low frequency** – all objects periodically send *APUM*s to all other objects via the admin channel at a frequency determined by *ta*.
- **Variable frequency** – an object, say O_i, sends *APUM*s to another object, say O_j, using O_i's local channel at a frequency determined by *tal* if O_j determines that interaction between O_i and O_j may occur in the near future.
- **High frequency** – an object, say O_i, sends *PUM*s to another object, say O_j, using O_i's local channel at a frequency determined by *tp* if O_j determines that interaction between O_i and O_j is occurring now.

An object decides on subscription and *APUM* frequency based only on local information (derived from *PUM*s and *APUM*s). Therefore, it may be possible for an object, say O_i, to subscribe to another object's, say O_j's, local channel without O_j subscribing to O_i's local channel. However, if such objects are moving closer to each other then we may assume that O_j will receive the appropriate *APUM* on the admin channel and subscribe to O_i's local channel before *PUM* message exchange is required. If objects are moving apart then it may be that O_i will remove its subscription from O_j's local channel with O_j never subscribing to O_i's local channel. This is acceptable behaviour and an overhead which is a result of attempting to prevent missed interactions.

4 A Distributed Implementation

We now describe an implementation using Java, the OpenFusion CORBA NS [Prism 2004] and JacORB [Brose and Noffke 2002] that realizes the conceptual model described in the previous section.

4.1 System Architecture

A network of messaging servers is responsible for implementing our interest management scheme on behalf of one or more local clients with a client associated with a single messaging server. Clients do not have to make calculations based on *APUM*s as this is the responsibility of the messaging server network. The scalability of our approach is reflected in the selective nature in which *APUM*s and *PUM*s are propagated between messaging servers, with local messaging servers acting as filters when determining which clients should be the recipients of *PUM*s. Scalability is further promoted as clients are unaware, at the network level, of other clients. Clients may communicate with a messaging server via a number of communication primitives (CORBA RPC, TCP and UDP). The use of CORBA allows clients to take advantage of CORBA services (e.g., security, transactions, location and discovery). We include TCP and UDP as they are commonly used, via socket programming, in current network game development. For brevity, we consider only clients using CORBA RPC in the remainder of this paper. Inter-messaging server communication is achieved via the CORBA NS. *PUM*s and *APUM*s published by a messaging server may be required by multiple recipients (messaging servers). A multicast mechanism may be employed below the CORBA NS for optimum message dissemination. However, discussion on the appropriateness of such a mechanism is beyond the scope of this paper as we assume this optimisation is associated with the CORBA NS implementation. In the OpenFusion CORBA NS implementation, suppliers and consumers are connected directly (without intermediary servers) over best effort (TCP) connections.

The two different message types associated with a local event channel (*PUM* & *APUM*) are accommodated for by a messaging server using two different interfaces to connect to the same event channel. This provides the subscription semantics as described in 3.2. To accommodate message passing between messaging servers, rather than have local channels created on a per object basis, local channels are created on a per server basis. Periodically, individual *APUM*s are combined into single messages and distributed on a per-server basis using local channels. Similarly, *APUM*s from a server destined for the admin channel are combined into a single message. For clarity, we continue describing local and admin channels as mentioned in section 3 and refrain from discussing message aggregation policies.

4.2 High Frequency Message Exchange

Figure 2 describes the flow of *PUM*s within a messaging server and associated event channels. A client may only issue *PUM*s associated with objects that are already registered with the local messaging server. Registration is by a method call invoked on the local messaging server by a client via the sending of a CORBA sequence with each element of the sequence identifying individual object properties (as a client may register more than one object at a time). The return value of such a method call is a CORBA sequence of identifiers that enables each object to be uniquely identified within the virtual world. Clients must use such an identifier when informing the local messaging server of changes to an object's state (e.g., position update, deletion from virtual world). An object's identifier is constructed from a client's IOR (interoperable object identifier – unique identifier used by CORBA) coupled with a timestamp that is derived from a logical clock that is incremented by a messaging server each time an object is registered (irrelevant of client). This logical clock has a maximum value that relates to the value an administrator deems appropriate for the maximum number of objects a server may manage. This approach provides a clear distinction between the application level object identification from that used by interest

management. Considering that the client application may be heterogeneous in nature (e.g., different naming conventions for objects), the naming conventions for objects may not be interoperable at the application level and so this distinction between application and interest management identification is necessary.

Once registration has occurred a client may issue *PUM*s to their local server. This is achieved via a method invoked by a client on the local messaging server consisting of a parameter containing a CORBA sequence with each element of the sequence associated with a single *PUM*. This allows a client to send multiple *PUM*s at the same time (as a client may be hosting multiple objects). The frequency of sending a *PUM* is decided by a client and relates to the high frequency message exchange expected between clients. A thread pool in the messaging server is responsible for accepting *PUM*s from clients and depositing *PUM*s into the appropriate inward message buffers held in the server.

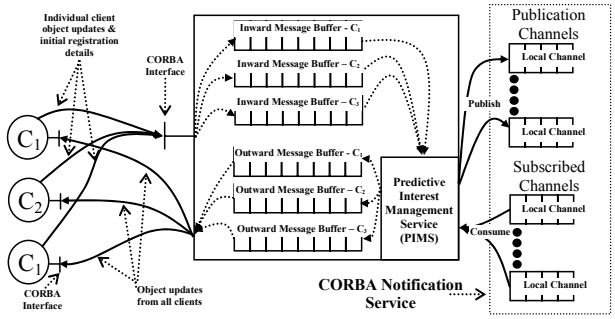

Figure 2: High fidelity message management.

An inward message buffer is present on a per-client basis with each element of an inward message buffer representing the last known position of an object. When a *PUM* is received the appropriate inward message buffer elements are updated with new object positional information. Periodically the *Predictive Interest Management Service* (PIMS) retrieves information from all message buffer elements that have been updated with new object positional information. The PIMS implements predictive interest management. There may be a possibility that the PIMS may not retrieve object information as quickly as *PUM*s refresh the inward message buffers. Therefore, an object's positional update information may be lost. However, this loss is acceptable as it is assumed that an application places primary importance on the most recent position of an object. The ability to overwrite inward message buffer information in this manner minimizes the need to introduce flow control at the client side.

On receiving *PUM*s from inward message buffers, the PIMS places *PUM*s on the appropriate local event channel. The action of publishing *PUM*s on local event channels allows subscribers (other messaging servers) to consume *PUM*s and so enables servers to forward *PUM*s to their own clients.

Messages consumed by the PIMS from local event channels belonging to other messaging servers are placed in the appropriate outward message buffers. These outward message buffers act in a similar way to the inward message buffers with regard to client association. A thread pool is responsible for periodically consuming object information from the elements of the outward message buffers and uses the reply parameters in a client call to return *PUM*s to clients. A messaging server may send *PUM*s to clients that have not sent *PUM*s for a substantial length of time (i.e., due to object inactivity – timeout determined by client).

4.3 Variable Frequency Message Exchange

Figure 3 describes the components of our system that are responsible for managing *APUM* message exchange between servers. On registration of an object by a client the *object registration and deregistration service* (ORDS) registers an object's details in a database. Once registered, these details are used by the PIMS when determining local event channel subscriptions. This indicates that aura and PAI radii are static throughout an object's lifetime. If this is not the case then a client may reflect changes in aura and/or PAI radii by deregistering an object and registering the object again with different aura and/or PAI radii. The ORDS is also responsible for amending the structure of inward message buffers to ensure any recently introduced objects have an element associated with them for recording incoming *PUM*s. The object database may be implemented using existing database technologies and be persistent. For our purposes we used mySQL [Widenius and Amark 2002].

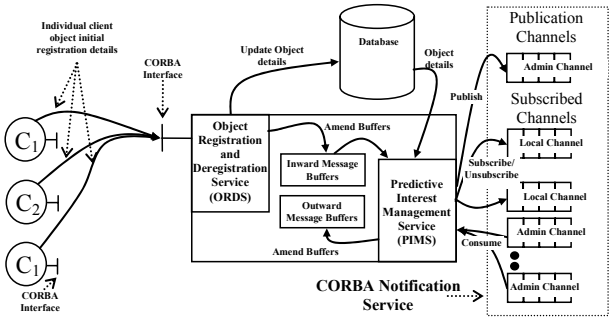

Figure 3: Variable frequency message management.

At a frequency determined by an administrator (denoted by timer *ta* in section 3.2), the PIMS will construct an *APUM* for publication on the admin event channel using information retrieved from the inward message buffers and associated object details from the database. Due to the variable frequency on which *APUM*s are published on the local event channels, the PIMS reuses information gained when constructing the last *PUM* together with information gained from the database when publishing *APUM*s on the local event channel.

On receiving *APUM*s (irrelevant if they arrive on local or admin event channels) the PIMS carries out the calculations described in 3.2 and determines appropriate frequency of *APUM* publication for each local event channel and ensures such a frequency is realized via the setting of the appropriate timeouts. The receiving of an *APUM* also requires the PIMS to instruct the messaging server to maintain the appropriate local event channel subscriptions.

5 Experiments and Results

This section describes the experiments carried out to assess the benefit of our approach to interest management. The primary reason for our experiments is to deduce if our approach is scalable. Ideally, we would like the addition of messaging servers to allow an increased number of clients to be serviced successfully.

There are two performance measures that are of interest: (i) throughput; (ii) successful message delivery. When considering throughput, we record the number of messages sent and the number of messages received by clients during a single run of the system. That is, when measuring throughput we only consider the *PUM* message exchange that occurs between clients and their local messaging servers, we do not count message exchange between messaging servers (event channel traffic). In this manner we derive performance of our overall system as perceived by clients. In order to identify the success rate of message delivery we record the number of messages lost due to buffer overwrites in the messaging servers during a single run of the system. Due to the quality of sevice provided by the OpenFusion CORBA NS, we assume no messages are lost in transit across admin and local event channels (which is what happened during our experiments).

In all our experiments we use messaging servers located on nodes in different LAN segments. Clients are co-located on different nodes with their local messaging server (same LAN segment), with 10 nodes used for creating synthetic client traffic. Each client hosts ten objects. Client numbers are increased using increments of 500 from 500 to 6000, with measurements taken at each increment. Clients are distributed as evenly as possible between servers. Each experiment is repeated for 1 server through to 10 servers (inclusive). The frequency at which *PUM*s are sent by all clients is set at approximately 3 per second. As the network environment is shared with regular traffic, the experiments were repeated a number of times throughout a day and the appropriate mean values for the measurements recorded. Experiments were conducted on Pentium 3GHz PCs with 512MB RAM running Red Hat Linux 7.2 connected via 100 Mbits fast Ethernet.

An attempt is made to provide realistic movement of objects within the virtual world. A number of targets (T) are positioned within the virtual world that objects (O) travel towards. Each target has the ability to relocate during the execution of an experiment. Relocation of targets is determined after the elapse of some random time (between T_{min} and T_{max}) from the time the previous relocation event occurred. Furthermore, objects may change their targets in the same manner (random time between O^t_{min} and O^t_{max}). Given that the number of targets is less than the number of objects and T_{min}, T_{max}, O^t_{min} and O^t_{max} are set appropriately, objects will cluster and disperse throughout the experiment. This provides a realistic movement of objects in a virtual world.

Objects are uniform in size and their sizes do not change throughout the experiment. Objects may move freely in any direction. The auras of objects are varied and are chosen at random (between 10 and 100 meters) and their maximum speeds are also chosen at random (between 5 and 100 meters per second) at initialization time. The value for determining *PAI* is standard for all objects and is determined by how far an object may travel at full speed in 20 seconds. The experiments were carried out with a 5% level of coverage of object auras. That is, assuming no object auras overlap then 5% of the virtual world will be covered by object auras.

Before we discuss the figures we must state that the requirement that a client sends 3 messages a second could not be satisfied in some instances. Such instances occur when client numbers are high and server numbers are fewer in number (messaging server becomes overloaded). This is because synchronous calls are made by a client to the messaging servers. A client may not issue the next call until a previous call returns. Therefore, in some instances clients could only submit at most 1 call a second. Another item to note is that we are measuring *PUM* exchange, not message exchange (as a single message may contain multiple *PUM*s).

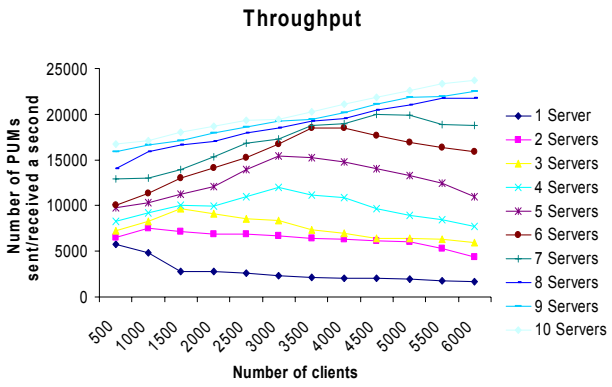

Figure 4: Throughput.

Figure 4 shows the graph that identifies the throughput of our system with rising client numbers for 1 through 10 server configurations. The first observation to be made is that the addition of servers allows greater throughput to be achieved. This is more noticeable when client numbers are high, indicating that the addition of servers provides more scalability in terms of client numbers. For each server increment there is a noticeable maximum throughput. That is, after a certain number of clients have been reached there is a drop in throughput. This "client threshold" is greater given more servers. For example, when 3 servers are present the client threshold appears to be around 1500 clients whereas when 7 servers are present this threshold has risen to around 4500 clients. For higher server numbers, the drop off rate after client threshold is noticeably less than that experienced by lower server numbers. From our observations we may assume our system is scalable in the context of our experiments as the addition of servers result in: (i) higher peak throughput; (ii) less pronounced drop off rate for throughput.

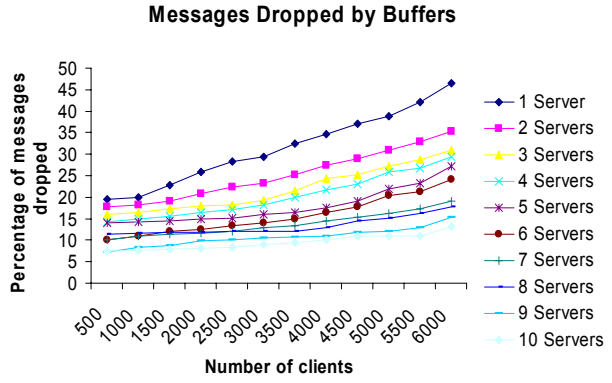

Figure 5: Messages dropped.

Figure 5 shows the graph that identifies the percentage of messages dropped by our system given rising client numbers for 1 through 10 server configurations. The graph clearly shows that

increasing server numbers results in fewer dropped messages for the same number of clients. Furthermore, the rate of increase of dropped messages when client numbers are increasing is less given additional servers.

From our observations we may assume our system is scalable in the context of our experiments as the addition of servers result in: (i) percentage of messages dropped decreases; (ii) decline in the rate of messages dropped when client numbers increase.

6 Conclusion and Future Work

We have presented an approach to interest management suitable for networked games that, we believe, will alleviate the problem of missed interaction common in existing interest management schemes. Only standard middleware technologies have been used in our interest management scheme implementation, promoting code reuse and benefiting from the interoperability provided by existing middleware standards making our system suitable for deployment in heterogeneous environments.

Our initial experiments identify that our approach is scalable as the addition of servers improves the performance of our system when satisfying increasing client numbers. We have successfully demonstrated that standard middleware, in particular MOM, may be incorporated into a scalable messaging dissemination scheme for networked games and still achieve performance that is acceptable for many game types.

Future work will enhance our interest management scheme to allow the modeling of object influences that are not only based on an object's position but also on message and object types. As an object may interact within a virtual world in a number of ways, there may be many different types of influence an object may exert. Furthermore, objects may exhibit varying degrees of susceptibility to different types of influences. For example, with the aid of a radio transmitter a soldier may communicate information to all other soldiers holding a radio receiver within a two kilometer radius of a radio transmission mast. In this instance a soldier transmitting radio signals exerts influence on a radio mast that, in turn, exerts influence on a subset of soldiers (those holding radio receivers). Modeling such interaction requires identification of message recipients not only on their location, but on the message type (radio transmission) and object type (capable of receiving radio transmission).

In addition to enhancements to our interest management scheme, we will investigate applying quality of service parameters on a per-message basis. For example, a type of message must be reliably sent to all recipients whereas other types of messages may be able to suffer a percentage of non-delivery. Furthermore, we aim to apply quality of service guarantees to other key aspects of our system (e.g., collision detection [Storey et al. 2004]).

Acknowledgements

This work is funded by the UK EPSRC under grant GR/S04529/01: "Middleware Services for Scalable Networked Virtual Environments".

References

BHARAMBE, A., RAO, S., SESHAN, S. 2002. *Mercury: A Scalable Publish-Subscribe System for Internet Games*, In *Proceedings of the 1st workshop on Network and system support for games*, ACM Press, 3 - 9

BROSE, G., NOFFKE, N., 2002. *JacORB 1.4 Programming Guide*, http://www.jacorb.org/docs/ProgrammingGuide_1_4_1.pdf, as viewed October 2004

DEERING, S. 1989. *Host Extensions for IP Multicasting*. RFC 1112, IETF Network Working Group

GORE, P., CYTRON, R., SCHMIDT, D., O'RYAN, C. 2001. *Designing and Optimizing a Scalable CORBA Notification Service*. In *Proceedings of the ACM SIGPLAN workshop on languages compilers and tools for embedded systems*, ACM, 196 – 204

GREENHALGH, C., BENFORD, S. 1995. *MASSIVE: A Distributed Virtual Reality System Incorporating Spatial Trading*. In *proceedings 15th International Conference on distributed computing systems* (*DCS 95*), IEEE Computer Society, 27 - 35

MILLER D., THORPE J. A. 1995. *SIMNET: The advent of simulator networking*, In *Proceedings of the IEEE 83(8)*, IEEE, 1114 - 1123

OMG 2000, *Notification Service Specification*, OMG (Object Management Group) TC Document telecom/99/07/01

OMG 2003. *The Common Object Request Broker: Architecture and Specification*, 2.4 ed.

SINGHAL S,. AND ZYDRA M. 1999. *Networked Virtual Environments, Design and Implementation*, Addison Wesley

SONY ENTERTAINMENT 2004. *Planet Side web Site*, http://planetside.station.sony.com, as viewed March 2004

STOREY, K., LU, F., MORGAN, G. 2004. *Determining Collisions Between Moving Spheres for Distributed Virtual Environments*. *In Proceedings of Computer Graphics International*, IEEE Computer Society, 140-147

SUN (SUN MICROSYSTEMS) 2002. *Java Message Service Specification – Version 1.1*. http://java.sun.com/products/jms/docs.html, as viewed October 2004

SUN (SUN MICROSYSTEMS CORPORATION) 1999. *Java RMI Specification*, ftp://ftp.javasoft.com, as viewed October 2004

SWEENEY T. 1997, *Unreal Networking Architecture*. http://unreal.epicgames.com/Network.htm, as viewed October 2004

PRISM TECHNOLOGIES. 2004. *OpenFusion CORBA Notification Service*. http://www.prismtechnologies.com, as viewed October 2004.

WIDENIUS, M., AXMARK, D., 2002. *MySQL Reference Manual*, O'Reilly & Associates