Object-Oriented Network Middleware for Massively Multiplayer Online Games

SOCS-TR-2008.5

Alexandre Denault[1], Jörg Kienzle[1], Carl Dionne[2], Clark Verbrugge[1]

[1] School of Computer Science, McGill University, Montreal, QC, Canada
{Alexandre.Denault,Joerg.Kienzle,Clark.Verbrugge}@mcgill.ca
[2] Quazal Technologies Inc., Montreal, QC, Canada
{cdionne@quazal.com}

Abstract. In this paper we present a scalable networking middleware designed for multiplayer and massively multiplayer games. We argue that *objects* can implement an ideal interface between the game logic and the communication middleware. This allows the game developer to maintain current design techniques and apply object-oriented decomposition to partition the game state. Game objects are mapped to *duplicated objects*, our units of distribution and replication of state. Sophisticated, load-balanced, multidimensional interest management is applied to duplicated objects to communicate only relevant updates to player nodes. Optimization of message sending using state prediction techniques is performed at the attribute level of objects. At the lowest level, duplicated objects communicate using a network engine that provides remote method invocation and publish/subscribe capabilities. We outline the architecture of two implementations of our middleware: one designed for experiments in an academic setting, and one designed for use in an industrial setting. Finally, we present experimental results that analyze and compare the performance of our middleware when using 3 different low-level network architectures.

1 Introduction

Compared to a traditional multiplayer game in which usually up to 16 players play a relatively short-lived game, *massively multiplayer games* (MMOGs) offer the possibility for thousands of players to play together in a persistent world. MMOG implementations face huge scalability problems since they have to handle a massive number of connected players, presenting them with a consistent view of the game world, while still providing good performance and hence, immersive and responsive gameplay.

A flexible distributed game architecture, and an efficient and scalable network infrastructure is at the heart of providing an enjoyable MMOG experience. The implementation of a performant networking middleware and its integration with the game logic is, however, a complicated undertaking. In this paper we present a scalable networking middleware designed for multiplayer and massively multiplayer games. We argue that *objects* can implement an ideal interface between the game logic and the communication middleware. Objects can also be used as units of distribution and replication of state. Optimization of message sending using state prediction techniques can be performed at the attribute level

of objects. At the lowest level, our objects communicate using a network engine that provides remote method invocation and publish/subscribe capabilities.

Two implementations of the middleware have been realized, one in a commercial and one in an academic setting, designed specifically for experimentation. To demonstrate the usefulness of such an experimentation platform, we show in this paper how we investigated the scalability and performance of different low-level communication abstraction implementations.

The outline of the paper is as follows. Section 2 summarizes the challenges particular to MMOGs. Section 3 presents our middleware approach and its 4 layers: the game objects layer, the duplicated objects layer, the communication abstraction layer, and the transport layer. Section 4 presents two implementations of our proposed middleware. Section 5 details the experiments we ran in order to determine the effects of different network layer implementations on performance. Section 6 presents related work, and the last section draws some conclusions.

2 Massively Multiplayer Online Games

In a typical multiplayer or massively multiplayer game, players collaborate or compete in a virtual world. Each player sees a graphical representation of the world and controls a character – an *avatar* – which can perform actions. Basic building blocks of such actions are, e.g., moving the avatar, picking up objects, or communicating with other players.

In order to provide a shared sense of space among players, each player must maintain a copy of the (relevant) game state on his computer. When one player performs an action that affects the world, the game state of all other players affected by that action must be updated. This can be done either by sending the action over the network, or by sending the effects of the action, i.e. the state of the game that the action has modified.

2.1 Scalability Issues

The biggest challenge in massively multiplayer games is scalability: the aim is to allow as many players as possible to play together in the same virtual world. Typically, the number of concurrent players in an MMOG is in the thousands. The machines of the players can be located anywhere on the world, connected to the Internet. As a result, the quality of the network connection to individual nodes varies: some connections exhibit a higher *latency* than others, meaning that it takes more time for a message to reach its destination. *Bandwidth*, i.e. the maximum throughput of data to and from a given node, is also limited, and varies depending on the quality of the connection. Finally, any one machine on the network has itself limited processing power and memory. On the other hand, with each player that joins the game, a new machine is added to the game, and hence the total available processing power and memory increases (from now on we will call each machine participating in the game simply a *node*).

2.2 Consistency Issues

Massively multiplayer games are complex distributed systems. Each player interacts with the game in real-time, and therefore his machine must know about the

state of the game world, at least of that part of the game state that is relevant to him. Due to the network latency problem, this game state can unfortunately never be 100% up to date, since the world is constantly concurrently modified by other players. The challenge in MMOGs is to nevertheless provide a consistent view of the virtual world to the players, or provide means to tolerate inconsistencies so that they do not negatively affect the game play.

2.3 Reliability Issues

The probability of failure of a *single* node in a distributed system is low. However, the probability of failure of *some* node in a distributed system grows with the number of nodes. It is therefore almost certain that in a massively multiplayer game with thousands of participating nodes failures will occur, and will occur fairly often. In addition, network connections can temporarily fail, and as a result some nodes might temporarily be isolated from others. An MMOG has to be able to cope with node and network failures in such a way that the disturbance to gameplay is minimal.

3 MMOG Middleware Abstraction Layers

People implementing a multiplayer or massively multiplayer game should be spending most of their time developing the game itself, and hence should be isolated *as much as possible* from low-level network programming. Network transparency can however not be total: in order to be efficient, the game has to provide detailed semantic knowledge to the network layer. Only the game programmer knows which part of the game state is important, i.e. has to be sent over the network and made available to all players. On the other hand, complex techniques that minimize network traffic can be implemented within a middleware layer, and the intricacies of their implementation does not have to be exposed to the game developer.

This section presents a layered middleware architecture targeted at providing efficient communication for MMOG games that integrates seamlessly with any game that is based on object-oriented design. An overview of the abstraction layers is shown in Fig. 1. The different layers are presented in the following subsections in more detail.

3.1 Game Objects Layer

A game programmer should ideally be able to work with abstractions from the game domain. As we have seen above, a virtual world is usually comprised of many *objects*, e.g. game items and players. Game developers therefore naturally apply object-oriented decomposition techniques to partition the game state.

In our approach, we suggest that game developers maintain this natural object-oriented design philosophy. The resulting *game objects* encapsulate game state, and provide operations to manipulate that state in a consistent way. The developer then proceeds to map game objects to *duplicated objects*, an abstraction provided by our network middleware explained in more detail in the following subsection. This mapping usually is 1 to 1, but can also be 1 to many, or many

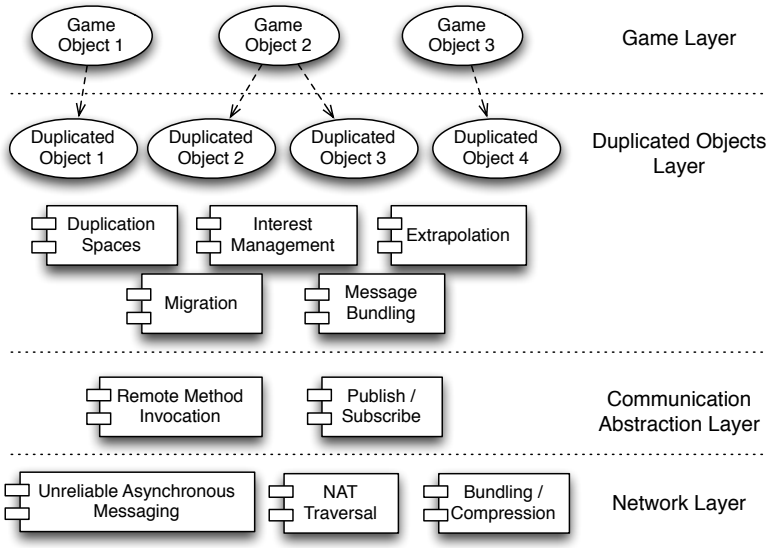

Fig. 1. Middleware Abstraction Layers

to one, if needed. Decisions on which part of the encapsulated state of an object is to be communicated to other players is done at the attribute level. Configuration of the communication quality can also be specified by the game developer by attaching meta-information to object attributes. Possible quality choices affect compression, reliability, and maximal extrapolation errors (see more details in subsection 3.2).

Once this mapping and configuration has been established, the game can simply invoke operations on game objects. The duplicated objects automatically provide efficient communication by using the lower-level network layer.

3.2 Duplicated Objects Layer

Duplicated objects encapsulate that part of the state of game objects that has to be distributed to players. Duplicated objects are also the unit of distribution: every node that needs access to the game state encapsulated by a duplicated object creates a new local instance of the object, a *duplica*. Our middleware makes sure that the state of the duplicas of the same duplicated object are kept up to date.

Masters and Duplicas Using duplicated objects, the state of a game object, for instance a *tomato*, is replicated across player nodes. Whenever the game executes a *read* operation on a game object, for instance `getPosition` on the *tomato*, the state of the local duplica is read.

Modifying operations, however, cannot be executed locally for consistency reasons. If local execution was allowed, it would be possible for concurrent modifications to take place, which could result in serious inconsistencies visible to the players. For instance, if two players simultaneously decide to `pickup` the *tomato*, only one player should succeed.

In our approach, consistency is guaranteed by designating one of the copies of the duplicated objects as being the *duplication master*. Modifying operations are always executed sequentially on the node that holds the duplication master. After the operation finished executing, update messages are broadcast to all duplicas.

The remote execution of modifying operations is completely transparent to the game layer. The game simply invokes the operation on the game object: our duplicated objects redirect the call to the duplication master node, if necessary. This transparency is not only convenient for the programmer. It also makes it easy to migrate the duplication master from one node to another node for load balancing or fault tolerance reasons.

Duplication Spaces The simplest approach of distributing the duplicated objects is for each player to maintain a full copy of the game state, i.e. create duplicas of all game objects on the player's node. The problem with this approach is that it does not scale: as the number of players increases, the number of messages to be sent over the network and the number of messages to be processed by each player's machine increase exponentially.

Since the virtual world of MMOGs is usually vast, one of the most effective strategies to address this problem is to keep on a player's node only the game state that is *relevant* to its avatar. This usually represents only a small subset of all duplicated objects that store the game state of the virtual world.

Of course, *visibility* is usually the most important criteria for determining relevance. Visibility between two objects can be determined based on the position of the two objects in the world, and based on the world geometry between the objects. However, vision is not the only way an avatar can sense its environment. An avatar can also *hear* an object that emits a sound, even if the object is hidden behind a wall. Likewise, an avatar can sense other players using a radar, etc...

Our approach allows the definition of multiple *duplication spaces* [2], i.e. dimensions in which objects can discover other objects, and be discovered by other objects. For instance, *3D Geometry* is an example of a duplication space commonly used in MMOGs. An object that occupies physical space in the virtual world is a *publisher* in the *3D Geometry* duplication space, an object that can see objects by observing the virtual world is a *subscriber* in the *3D Geometry* duplication space. Objects can simultaneously be publishers and subscribers in one, but also in multiple duplication spaces. For example, an avatar carrying a radio would be a publisher and subscriber in the *3D Geometry* duplication space, and a subscriber in the *Radio Frequency* space.

Customizable Interest Management *Interest management* (IM) is the process of determining what part of the game state (and therefore which duplicated objects) is relevant to each player [13]. The general idea of interest management is explained in [3] as the *aura-nimbus* model. The *aura* is the area of presence of an object in the dimension of interest. For instance, in the *3D Geometry* duplication space, the aura of an avatar would be the physical space that the avatar occupies within the virtual world. The *nimbus* is the space in which an object can perceive other objects within the dimension of interest. In the *3D Geometry*

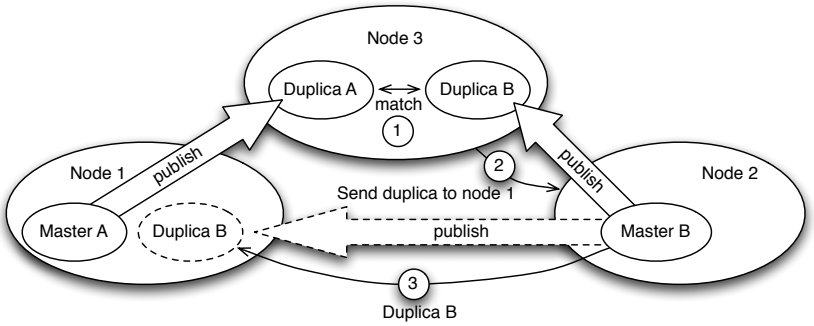

Fig. 2. Interest Management

duplication space, the nimbus of an avatar covers that area of the virtual world that lies within the vision of the avatar. If aura and nimbus is defined for each object, interest can be determined as follows: an object A is interested in an object B if the aura of B intersects with the nimbus of A, i.e. if the area of presence of B lies within the area of perception of A.

In our approach, interest management boils down to matching publishers and subscribers within a duplication space. The matching algorithm depends on the duplication space, on the state of the publisher object and the subscriber object, and potentially even on other game state. For instance, in the night, an avatar wearing infrared vision goggles might see another player even in total darkness. Our approach allows a game developer to define a custom *matching* algorithm for each duplication space. This algorithm, given a publisher object and a subscriber object, has to determine whether or not the subscriber object is interested in the publisher object.

If it is determined that an object A is interested in object B (step 1 in Fig. 2), the node that holds the duplication master of B is instructed to send a duplica of B to the node with the duplication master of A (step 2 and 3 in Fig. 2).

Distributed Interest Management Interest management significantly reduces the amount of state update messages that have to be sent over the network, and hence improves scalability considerably. Unfortunately, interest management itself can easily become a performance bottleneck. As the number of publisher and subscriber objects n in the duplication space increases, the computational effort increases with $O(n^2)$. The computational load to perform interest management soon becomes too high to be performed by a single node. Calculating interest management on a single node is also not a good idea because of the fact that the node that matches a subscriber to a publisher must contain duplicas of the two involved objects. Hence, the node that performs the interest management for the game would have to create duplicas of all objects, and would therefore also receive and process all game state updates.

To distribute the computational effort of interest management across nodes, a duplication space can be split into *cells*. A cell is a subset or subregion of a duplication space. A cell itself is a duplicated object. The node that owns the duplication master of a cell performs the interest management for all the objects

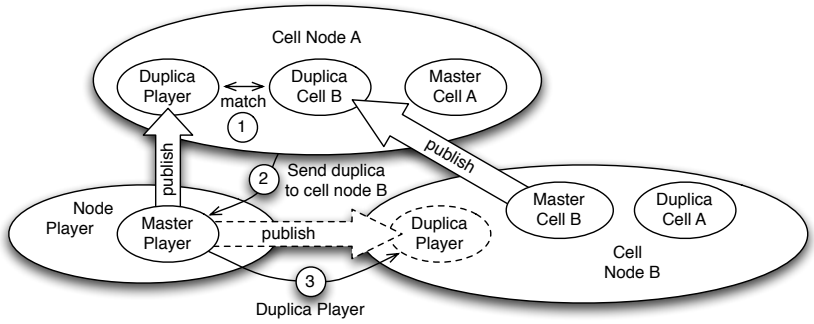

Fig. 3. Dividing a Duplication Space into Cells

located in that cell. This means, of course, that that node must create duplicas for these objects.

Whenever an object moves within a dimension, it is possible that it crosses the boundaries of a cell. In this case, the responsibility for performing interest management for that object has to be transferred to the neighboring cell that the object enters. To do this handoff in a fault-tolerant way, neighboring cell regions have to overlap slightly. In addition, the node that holds a cell master has duplicas of all neighboring cells. Cells are subscribers in the duplication space, and hence the interest management algorithm will trigger whenever an object enters the overlap region of a neighboring cell. In this case, the current cell master node informs the object master node that it must send a duplica of the object to the node that holds the cell master of the cell that the object is entering.

Fig. 3 illustrates the case where a duplication space is split into two cells, *cell A* and *cell B*. A player is currently located in *cell A*. The node that holds the master of *cell A* is performing interest management for the player and therefore has a duplica of the player. It also has a duplica of *cell B*. As the player moves in the duplication space closer to *cell B*, the master node of *cell A* detects a match between the player and *cell B* (step 1). It then instructs the master player node to send a duplica to the master node of *cell B* (step 2 and 3).

Minimizing Transmitted Data As mention in section 2, limited bandwidth and latency is one of the important problems game developers of multiplayer and massively multiplayer games have to cope with. Our middleware provides customizable extrapolation of attribute values of duplicated objects to minimize bandwidth usage. Rather than updating the attribute values in duplicas each time the value in the duplication master changes, values are predicted on the nodes that hold duplicas using a user-customizable function. Only if our middleware realizes that the prediction is not accurate enough is the new attribute value broadcast to all duplicas. The required precision is again user-definable, and can be set to a constant value or even to vary according to the "distance" between the avatar and the duplicated object within the duplication space.

Another simple way to minimize transmitted data is to publish update messages pertaining to the same object together. This saves bandwidth because the object header does not have to be sent multiple times.

Load Balancing and Fault Tolerance Since modifying operations are always executed on the node that holds the duplication master, the CPU usage on machines that carry many masters could become an issue. Our middleware, however, allows duplication masters to migrate from one node to another node upon request. This migration is transparent to the rest of the program. Migration allows the developers to define elaborate load balancing algorithms, if necessary.

In case of node crashes, migration is performed automatically. When a node crashes, any duplication master that used to run on that node disappears. Our middleware can detect such a failure, and will start to initiate a fault recovery process. For each object that is without duplication master because of the failure, a new duplication master is elected among the nodes that owned duplicas[3]. Any pending modification requests are forwarded to the new master as soon as possible.

Another form of fault tolerance is applied at the attribute level of duplicated objects. For each attribute, the game developer can specify if the updates are to be transmitted reliably or unreliably, unreliably of course being more efficient. But even if unreliable publishing is chosen, some updates are published reliably in order to ensure that the extrapolation algorithm mentioned in section 3.2 does not yield wrong results in case of lost messages.

3.3 Communication Abstraction Layer

Our duplicated objects middleware layer needs to communicate with other nodes in two situations: 1) when a modifying operation is called by the game layer on a duplica, and 2) when the state of a duplication master object changed significantly. For the former, a *remote method execution* (or reliable, synchronous messaging) communication abstraction is needed. For the latter, *publish / subscribe-based messaging* is the ideal communication abstraction. The two communication means are described in the following subsections.

Remote Method Invocation (or Reliable, Synchronous Messaging) When an operation is called on a duplicated object that modifies the object's state, the operation has to be executed on the state of the duplication master object for consistency reasons (see section 3.2). To facilitate fault tolerance and master migration, remote execution should however be transparent for the caller of the operation[4].

Method calls are usually *synchronous*, i.e. the calling thread returns from the call once the method has executed, possibly carrying a return value. In addition, method calls provide *exactly once* semantics, i.e., when a call returns, then the method was executed exactly one single time. Finally, remote method executions must be *reliable*, i.e., even in case of node failures, a remote method call that successfully returns has indeed changed the state of the object on the node of the duplication master.

[3] It is possible to force our middleware to at least maintain n duplicas for each duplicated objects in order to be able to tolerate *n-1* crash failures.

[4] Apart, of course, from the additional communication time needed.

In case an operation does not have a return value (which includes any exceptions that the method could throw), it is possible to optimize performance by executing the call asynchronously. In such a case, however, reliability and exactly once semantics are still required.

Publish / Subscribe After a modifying operation has changed the state of a duplication master significantly[5], the state of all duplicas has to be updated. A *channel* or *topic-based publish / subscribe* communication abstraction can achieve this task in an elegant way.

In a publish / subscribe system, a large number of subscriber nodes can express interest in a certain channel or topic. Once subscribed, a node receives any publications or events published on that channel or topic. The publish / subscribe communication paradigm is highly flexible and scalable due to the fact that publisher nodes and subscriber nodes are completely decoupled, and communication is asynchronous. Publisher nodes do not know about subscriber nodes, and subscribers do not have to know about publishers. Publishers do not have to wait for all the subscriber nodes after announcing an event. Finally, there is no need for any node, be it publisher or subscriber, to have global knowledge about the network topology. This property is essential for implementing a fault-tolerant publish / subscribe system.

In our middleware, a new topic is created and assigned to each duplicated object. The duplication master node publishes every significant state change of the duplication master object on the corresponding topic. Nodes that have a duplica of an object subscribe to the corresponding topic and thus receive all important state change events.

Integrating RMI and Publish / Subscribe A special situation that compromises consistency if not addressed properly arises when a modifying operation is initiated on a node that contains a duplica. In general, after a method has executed, the caller of the method expects that the state of the object reflects the execution of the call, e.g., if a read-only method is called on the object subsequently. In our system, however, the duplication master publishes the new state of the object when the call has executed on the master node, and the state of the duplica is only updated when the publication reaches the node of the duplica. Therefore, our middleware must make sure that read-only methods executed after a modifying operation are only allowed to proceed on the duplica once the state update corresponding to the modifying operation has been received from the master node.

3.4 Network Layer

At the lowest level, the network layer provides point-to-point unreliable messaging. The NAT traversal component helps to establish connections even between clients that are behind firewalls. A message bundling component groups messages destined to the same node together. Finally, a compression components compresses messages before they are sent, and decompresses them upon arrival at the destination node.

[5] Significantly means here that the predicted value based on the extrapolation techniques described in section 3.2 would lie outside the acceptable tolerance interval.

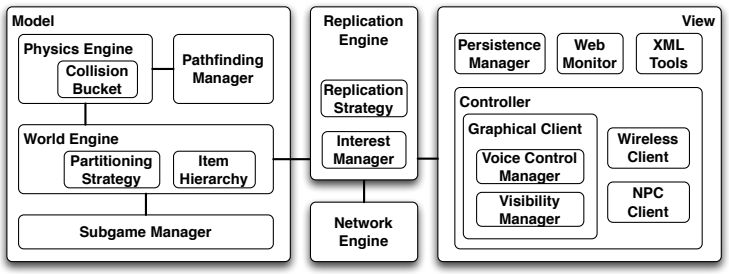

Fig. 4. Components of the Mammoth Framework

4 Middleware Implementations

This section presents two implementations of the middleware described in section 3. The first one was created at McGill University in an academic setting as part of the Mammoth framework. The second one, Net-Z, is a commercial middleware for multiplayer games provided by Quazal.

4.1 Mammoth

Mammoth is a massively multiplayer game research framework. It was created as a collaborative project between a group of McGill professors and students in early 2005, and has evolved considerably during the last 3 years. Its goal is to provide an implementation platform for academic research related to multiplayer and massively multiplayer games in the fields of distributed systems, fault tolerance, databases, networking, and concurrency.

In order to allow researchers to easily conduct experiments, the Mammoth framework has been designed as a collection of collaborating components that each provide a well-defined set of services. The components interact with each other through two types of well-defined interfaces, engines and managers. The general architecture is depicted in Fig. 4.

The architecture of Mammoth follows the Model-View-Controller paradigm. The state of the world is contained inside model objects, which are then distributed to various view and controller components. View components are used to monitor and record the state of the world, as opposed to controller components who alter the state of the world.

Our middleware approach, implemented in the *Replication Engine* and the *Network Engine* (see central components of Fig. 4), is at the heart of this architecture, as it provides the bridge between the model, the views and the controllers. The *Replication Engine* implements duplicated objects as described in section 3.2. As such, it takes care of managing the distributing duplicated objects, managing the duplication spaces, object migration, and initiating interest management. The actual interest management algorithms are coordinated by the *Interest Manager* component, located within the *Replication Engine*. The *Network Engine* implements remote method invocation and a publish / subscribe service as described in section 3.3. Details on 3 different implementations of the network engine are given in section 5.

The mapping between game objects and duplicated objects in Mammoth is done using a pre-processor. The developer implements his game objects in Java, and annotates remote methods using Java attributes. The pre-processor reads the game object class as input, and automatically generates proxies for the master and replica objects using Java reflection.

In addition to the many functional components, Mammoth also provides multiple monitoring and logging tools to facilitate benchmarking of various characteristics, such as CPU load, network performance and latency.

The implementation of Mammoth is done almost exclusively using the Java programming language. This was a practical decision. Many researchers at the School of Computer Science of McGill University use Java for their experiments, and many tools have been developed for research and performance analysis in Java. Furthermore, the cross-platform nature of Java facilitates access to Mammoth for the students, and makes maintenance easier. We are of course aware that an industrial implementation of our middleware architecture using a non-interpreted language such as C++ would provide even better performance. However, our experiments are still valid, since they provide insight into the complexity of our algorithms and techniques as the number of players, game objects and nodes increases.

4.2 Net-Z

Quazal's Net-Z is a commercial product implementing a middleware for multiplayer games following the abstraction layers described in this paper.

The two main components of Net-Z are the runtime and the data definition language (DDL) compiler. The runtime is a portable C++ library available on the various gaming platforms (Windows, MacOSX, Linux as well as for the Sony, Microsoft and Nintendo consoles). This runtime implements a network infrastructure with the choice of a fully connected peer 2 peer model or a client/server model, and implements the various network optimizations described above (choice of reliable or unreliable messaging, bundling of the messages at various levels, configurable encryption and compression, different extrapolation models, remote method invocation).

In Net-Z, the interface between the game objects and the duplicated objects is implemented using a data definition language (DDL) [2]. The DDL is used to define the classes of duplicated objects that will be available to the game, along with the data replicated for each, the policies specifying how this data is replicated (reliable or not, using extrapolation or not), the set of the method calls available on each duplicated object. Interest management is also specified in this data definition language.

A sample DDL specification is given in Fig. 5. The first dataset definition specifies a *position* data structure. Position updates are published unreliably, and extrapolation techniques can be applied in order to minimize network traffic. The second dataset definition specifies a *description* data structure for which updates are published reliably, and only when requested by the game layer. The next line defines a duplication space *Geospace*, for which interest management has to be implemented by the game developer in form of a C++ method. Finally, the

```
dataset Position
 {double x, double y, double z} unreliable, extrapolation_filter;
dataset Description {string name, uint32 type} upon_request;
dupspace GeoSpace;
doclass Avatar {
  Position m_pos;
  Description m_desc;
  void damage(uint32 damage);
  publisher in GeoSpace;
  subscriber in GeoSpace;
};
```

Fig. 5. Sample DDL Code

doclass Avatar describes a duplicated object that has a *position* and a *description* data structure, and a remotely callable method *damage*. An object of class Avatar can be discovered, i.e. is a *publisher* in the *Geospace*, and can discover others, i.e. is a *subscriber* in the *Geospace*.

The DDL compiler reads this definition as input and outputs a duplicated objects layer specific to the game, in the form of a set of C++ classes corresponding to the duplicated object classes that were defined in the DDL. Game developers integrate the duplicated objects into their game by inheriting from the generated C++ classes. The only additional required step is to link the Net-Z run time with the executable.

5 Network Layer Experiments

Early experiments have shown that a crucial part of the middleware is the efficiency of implementation of the communication abstraction layer. In particular, we wanted to investigate the effect different network topologies would have on the performance of our middleware. In typical client/server topologies, all clients connect to a central hub. In contrast, in a fully connected network, every node is directly connected to every other node. To better understand the scalability issues of our middleware approach, these two extreme network topologies, along with a third adaptive topology, were implemented. To evaluate the impact of the different strategies, we used Mammoth to run scalability tests on top of these 3 different communication layer abstraction implementations.

5.1 Toile

Toile is a peer-to-peer network topology implementation of the communication layer for the Mammoth framework. In *Toile*, every node is directly connected to every other node. To simplify the implementation, a centralized rendezvous node/application is used to manage the arrival of new members. However, this rendezvous node is only needed when a new node wants to connect, and therefore does not affect experimental results once the node is joined.

As is required by the Mammoth framework, *Toile* also provides publish/subscribe services. Publishers are stored on a specific node, as are subscriptions to that publisher. Thus, all traffic, include publications, subscriptions and unsubscription requests for a given publisher must be directed to this particular node.

The *Toile* engine is fairly fault-tolerant when dealing with node failure. The loss of a node is immediately detected, given that all nodes are interconnected. In addition, the rendezvous node broadcasts an updated version of its connection lists upon failure, so that clients can discover any missing members. However, the rendezvous node itself is a single point of failure. Although the loss of this node does not affect current members, no new members will be able to join the network. In addition, *Toile* does not cope well with network firewalls. If a node is unable to receive incoming connections, it cannot be part of the network.

5.2 Stern

Stern mimics a client/server style network topology implementation for the Mammoth framework. All nodes connect to a central hub on startup, which handles all network traffic from then on, thus forming a star topology. This reduces the load on the individual nodes, but of course dramatically increases the burden on the central hub.

The implementation of a publish/subscribe system in such a network topology is fairly straightforward: the hub also manages publications and subscriptions. Although this also increase the load on the central hub, it is the most efficient way of dispatching messages when dealing with publications. When a node wants to publish an event, it simply has to send a publication message to the central hub. The hub then takes care of forwarding the publication to all subscriber nodes.

Stern has an important single point-of-failure: if the central hub fails, the communication stops immediately. Other node failures are immediately detected by the hub, which in turns warns remaining clients. On the other hand, the star topology provides an efficient way of bypassing most firewalls: as long as a client can establish an outgoing connection to the hub, it can join the network.

5.3 Postina

Postina is a peer-to-peer networking engine for the Mammoth framework designed to address the scalability issues that arise when publishing states to a large number of clients. *Postina* is built on top of *Scribe* [15, 6], a publish/subscribe service which itself is built on top of *Pastry* [14], a generic, scalable and efficient substrate for wide-area peer-to-peer applications. *Postina* implements the additional required communication abstractions not implemented by *Scribe* and *Pastry*, such as for example reliable remote method invocations, on top of the unreliable communication provided by *Pastry*.

Nodes within *Pastry* form a decentralized, self-organizing and fault-tolerant overlay network. Nodes are assigned random node *ids* when joining the network. Routing tables are created at each node that allow a *Pastry* node to forward a message based on its destination node *id* using the most efficient link to a node that is "closer" to the destination. These routing tables are constantly maintained to compensate for network quality fluctuations. The expected number of hops for a message to reach its destination is $log_{2^b} n$, where n is the number of nodes in the network, and b is a configuration parameter.

Scribe implements publish/subscribe on top of *Pastry* by organizing the nodes for each topic into a tree-like structure. Publications are forwarded to the top of the tree, and then distributed through the branches to all subscribers. This greatly decreases the burden on individual nodes when distributing updates, but increases the time required for updates to fully propagate.

Since the Pastry network topology is decentralized, there is no single point-of-failure. However, maintaining the routing tables in the presence of network congestions and node failures incurs significant overhead. From our experiments we know that joining the network requires a couple of seconds, and detecting a faulty node can require several minutes. A severe drop in performance was also noticed at regular intervals, when the overlay network is reorganizing itself. In addition, *Postina* currently has problems similar to *Toile* with respect to firewalls. Although a more firewall-friendly version of *Pastry* is in development, the current version of *Postina* requires that all nodes be able to receive incoming communications.

5.4 Experimental Setting

We used the Mammoth platform to run our experiments on the 200 lab machines of the School of Computer Science at McGill University. All the machines have Pentium 4 processors, at least 2GBs of RAM, and are running Linux. We ran 4 separate experiments for each of the communication layer implementations: all duplication master objects were distributed onto 1 server, 2 servers, 4 servers or 8 servers. In the case of *Stern*, an additional machine was used as the communication hub. The virtual game world included multiple obstacles, simulating a moderately-sized semi-urban scenario.

Client machines were started gradually, one by one, to join the game and take control of a player. Since previous experiments have shown that carefully designed computer-controlled players generate similar network traffic to real human players [5], we used our AI component to control the avatar on the client. The AI component was instructed to move the avatar within the world, changing direction every 1.5 to 2.5 seconds. Since the duplication master of an avatar is located on the server, each change in direction requires a remote method invocation. On the server side, the call is executed, and the new state is published to all interested clients. Then, the return value of the call is sent back to the client. In all our experiments we measured the round trip time of such a RMI/Publish call, i.e. the time spent from the client sending the RMI request to the reception of the return value by the client.

5.5 Experimental Results

The results of our experiments are presented in the three graphs of Fig. 6. First of all, the numbers are always above 10ms per RMI/Publish call. The performance overhead generated by the duplicated objects layer is an order of magnitude smaller than the time spent in the network layer and can therefore safely be ignored. In addition, the measurements clearly show that network topology has an important impact on performance and scalability.

For *Toile* the average RMI/Publish call takes over 100ms, even with only 10 connected clients. This is actually not surprising, since the master node has to take care of all the update publications on its own. So even in a 10 client scenario, the master node has to send 11 messages (10 updates plus the return value of the call). With only 1 server that contains all the master objects, *Toile* can handle around 80 clients before breaking down. By adding more servers and distributing the master objects, this limit can be augmented to 120 clients for 2 servers, 150 clients for 4 servers, and 160 clients for 8 servers.

Stern on the other hand has a low average call time when only few clients are connected. With only 10 connected clients, for instance, the call time is below 40ms. This is due to the fact that a server only needs to send one message to the communication hub when publishing state updates. The scalability test results are also not surprising: communication breaks down in *Stern* at around 130 connected clients, regardless of how many servers are used. This is due to the fact that even with multiple servers, all communication is routed through the central hub.

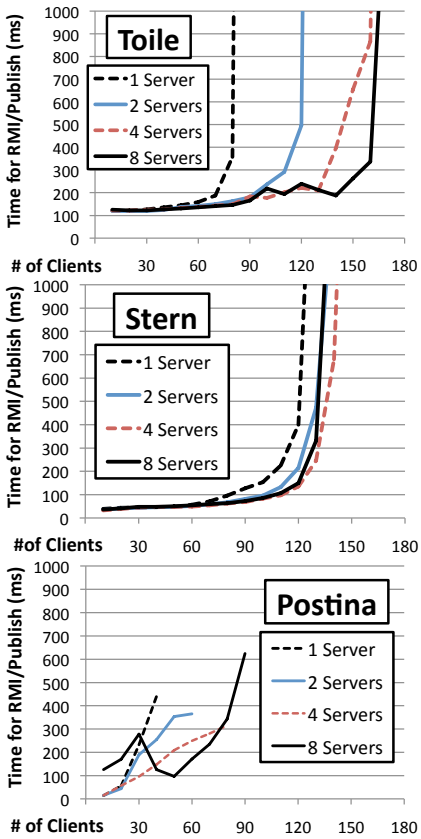

Fig. 6. Time for RMI/Publish Call

The test results for *Postina* were unfortunately not reliable enough to draw clear conclusions. In a single server setting we were not able to connect more than 40 clients. We were not able to determine the cause for this problem. Pastry simply did not allow more clients to connect, flagging them all as faulty and ignoring them. For 2 servers the problems started at 60 clients, for 4 servers at 80, and the 8 server experiment choked at 90 connected clients. Another problem that made our experiments less reliable is that a small percentage of messages took *very* long to complete, i.e. from 3s up to 30s! We believe that this is due to the fact that Pastry does not use FIFO waiting queues. Hence, a message can be starved on the queue in case of a congested network link.

In order to still be able to compare *Postina* to the other network implementations, we removed any messages taking over 3s. The results presented in the 3rd graph of Fig. 6 are interesting. First of all, *Postina* provides 15ms average RMI/Publish calls with only 10 connected clients, i.e. faster even than *Stern*. Next, the more servers are added, the better the performance in a long run.

Finally, the shape of the 8 server curve reveals an interesting fact. The time for an RMI/Publish call increases up to 30 clients, but then decreases again to reach a new minimum at 50 clients, and then increases again. This is due to the fact that during this experiment we allowed the network around 10 minutes of time to self-organize, i.e. optimize the network routing tables of each node to adapt to the current network load.

We realize that a deeper understanding of the inner workings of *Pastry* is required to overcome the client limits and design new experiments that can better reveal the potential of a decentralized, self-organizing network layer.

6 Related Work

Our work aims at an appropriate middleware design for MMOGs. Several other authors have proposed designs in this area, covering various perspectives on suitable game design. The most common goal is to abstract a complex network architecture and provide a simple model or API to the game developer. Hsiao and Yuan's *DoIT* game middleware, for example, abstracts the communication architecture and provides automatic code generation features to facilitate easy integration of the actual game logic and the network system [10]. They base their design on a clustered server environment, although the use of proxies for client communications should permit arbitrary network modules.

An associated general design principle is to present an overall client/server model, however the underlying system is implemented. The *Lucid* middleware platform, for instance, provides multiple layers of abstraction for game design [12]. The underlying network components can make use of advanced routing, such as provided through interest management modules, but the overall model presented is client/server. *ATLAS* provides a fairly comprehensive virtual world framework based on either a client/server or hybrid peer/server environment [11]. Here the virtual environment is assumed to be region-partitioned among servers, with ATLAS supplying a uniform interface as well as various components to facilitate scalability, including region and aura-based interest management, replication (caching) control, simple mutual exclusion requirements, and load balancing.

The *RTF* middleware game framework is closer to our design in its use of a publish/subscribe system for interest management, replication and migration system [8]. Replicated shadow object updates are automatically handled, and RTF allows for object-based messaging through serialization, although the design is fundamentally geared toward server-based management rather than a fine-grained object replication per se. Our model directly exposes object replication to the network system, permitting data-specific optimizations such as field-based dead-reckoning.

The design of *Colyseus* is perhaps closest to our approach, replicating data at the object level and filtering communication based on interest management [4]. Their design incorporates interest-based range queries at the DHT level and an object discovery mechanism is then responsible for gathering relevant data at runtime, optimized by a predictive object and replica migration strategy.

The resulting consistency model is slightly weaker than ours, accommodating tentative writes to object replicas as well as progress in the face of incomplete replication, but allows for good, scalable performance in the FPS genre, albeit with greater consistency concerns. MMOGs and other less real-time intensive games, as Bharambe et al. postulate, are not as demanding and can coexist with stronger consistency models, as we target in our design.

General frameworks have also been proposed with the specific goal of prototyping more than actual game development. Fletcher et al. describe *plug-replaceable* concurrency and consistency control with the goal of providing a flexible means for exploring different game consistency models [7]. For exploration of overall game network design, *NGS* allows for prototyping a variety of region-based network architectures [16], including P2P and client/server designs. Such designs allow for rapid evaluation of different parameters and designs, although full game implementations are still required to consider other in-game aspects such as realistic player movement or the impact of visibility.

Our design is partly inspired by the approach in Quazal's commercial middleware [2], but naturally several other commercial products for game network middleware also exist. Among the more popular is *BigWorld*. BigWorld is primarily aimed at server cluster approaches, supporting the more traditional zone or shard-based approaches to MMOGs with load-balancing and other scalability improvements [1]. *ZeroC* provides open source and commercial variants of a distributed communication engine which can be applied to MMOGs [9]. ZeroC has some features similar to our design, but offers a more generic model to accommodate a wide variety of potential uses—like ATLAS, ZeroC does not target games exclusively.

7 Conclusion

A non-trivial MMOG framework is a useful vehicle for the rapidly growing worlds of game research and development, and the design we present here accommodates both research interests and practical, industrial game design. A middleware approach alleviates game programmers from the complex issues of integrating consistency, fault-tolerance, and other modules within a scalable system. Our particular strategy is based on a strongly object-oriented model, which fits well with various components, and moreover maps quite naturally to MMOG design. By appropriately abstracting core communication issues the overall system provides a clean development model with efficient execution; modular design further permits, as we demonstrate, an effective means for research experimentation or design prototyping. Finally, game developers find the object-oriented approach appealing, which is demostrated by the fact that Quazal's Net-Z is used currently in more than 50 commercial games.

Of course improvements to scalability are continuous in MMOGs. As we show, high-level game topology can introduce bottlenecks in the communication system, even if the underlying architecture is highly scalable. Making a P2P system more aware of the overlaying publish/subscribe model may improve performance in this situation, although the impact on modularity must also be

considered for reasonable and maintainable design. Our current work focuses on extending the Mammoth framework to accommodate *instancing* and other MMOG models that imply multiple, dynamic levels of game consistency.

References

1. BigWorld. http://www.bigworldtech.com/index/index.php, 2007.
2. Quazal Technologies Inc., duplication spaces patent # 6,907,471, DDL patent # 7,096,453. http://www.quazal.com, 2008.
3. Steve Benford and Lennart E. Fahlen. A spatial model of interaction in large virtual environments. In *Third European Conference on Computer Supported Cooperative Work*, pages 107–123, 1993.
4. Ashwin Bharambe, Jeffrey Pang, and Srinivasan Seshan. Colyseus: a distributed architecture for online multiplayer games. In *NSDI'06: Proceedings of the 3rd Symposium on Networked Systems Design & Implementation*, pages 155–168, Berkeley, CA, USA, 2006. USENIX Association.
5. Jean-Sébastien Boulanger, Jörg Kienzle, and Clark Verbrugge. Comparing interest management algorithms for massively multiplayer games. In *NetGames '06*, pages 1 – 6, New York, NY, USA, 2006. ACM.
6. Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):100–110, 2002.
7. Robert D. S. Fletcher, T. C. Nicholas Graham, and Christopher Wolfe. Plug-replaceable consistency maintenance for multiplayer games. In *NetGames '06*, page 34, New York, NY, USA, 2006. ACM.
8. Frank Glinka, Alexander Ploß, Jens Müller-lden, and Sergei Gorlatch. RTF: a real-time framework for developing scalable multiplayer online games. In *NetGames '07*, pages 81–86, New York, NY, USA, 2007. ACM.
9. Michi Henning. Massively multiplayer middleware. *Queue*, 1(10):38–45, 2004.
10. Tsun-Yu Hsiao and Shyan-Ming Yuan. Practical middleware for massively multiplayer online games. *IEEE Internet Computing*, 9(5):47–54, 2005.
11. Dongman Lee, Mingyu Lim, Seunghyun Han, and Kyungmin Lee. ATLAS: a scalable network framework for distributed virtual environments. *Presence: Teleoper. Virtual Environ.*, 16(2):125–156, 2007.
12. Elvis S. Liu, Milo K. Yip, and Gino Yu. Lucid platform: applying HLA DDM to multiplayer online game middleware. *Comput. Entertain.*, 4(4):9, 2006.
13. Katherine L. Morse. Interest management in large-scale distributed simulations. Technical report, CS Department, UCLA, Irvine, 1996.
14. Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01*, pages 329–350, London, UK, 2001. Springer-Verlag.
15. Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: the design of a large-scale event notification infrastructure. In *NGC '01: Proceedings of the 3rd Workshop on Networked Group Communication*, pages 30–43, London, UK, 2001. Springer-Verlag.
16. Steven Daniel Webb, William Lau, and Sieteng Soh. NGS: an application layer network game simulator. In *IE '06: Proceedings of the 3rd Australasian conference on Interactive entertainment*, pages 15–22, 2006.