UNIVERSITY OF OSLO
Department of Informatics

Migration of Objects in a Middleware for Distributed Real-time Interactive Applications

Master Thesis

Paul B. Beskow

Contents

List of Figures

Preface

Object oriented middleware provides an application with the possibility of distributing objects to multiple nodes in a distributed system. In this thesis, we have developed a middleware that, in addition to distributing objects, makes it possible to migrate them. As such, it becomes possible to dynamically relocate objects based on the requirements of the application. We use a distributed name service to maintain references to objects, which means that any given object is managed by the node it is currently located at. This middleware was derived from the requirements and characteristics displayed by interactive real-time applications, i.e, applications that are time dependent and event based. To demonstrate the usability of the middleware we have implemented a test application, in form of a chat system derived from a massively multi-player online game (MMOG).

Acknowledgments

I would like to express my gratitude to my supervisors Pål Halvorsen and Carsten Griwodz. Without their expertise, understanding and patience, the process of completing this thesis would not have been as educational and productive.

To the guys at the lab and my friends, thank you for great conversation, good advice and plenty of laughs.

Finally, I would like to thank my family for the support they have provided me with through my entire life.

Chapter 1

Introduction

Middleware is an intermediate layer of software which provides convenient facilities for the development of distributed applications. It has commonly been attempted, by the middleware, to provide a unified view of distributed computing. It has, however, proved necessary to provide specialized middleware depending on the area of use. This is particularly true for interactive applications, which are applications that allow users to interact in real-time. It has been shown that the user experience in interactive applications is highly dependent on the responsiveness of the system [1,2]. This means that a high delay in receiving transmissions can have a profusely negative effect on the performance of the program. As a result, there is an ongoing effort to negate the effects that communication delays induce. Thus, the middleware for interactive applications must focus on providing a high level of responsiveness.

1.1 Background and Motivation

In this thesis, we will isolate a subset of functionality common to interactive real-time applications. This we will accomplish by analyzing the requirements and characteristics exhibited by these applications. We will solidify our efforts by using massively multi-player online games (MMOGs) as an example, and accordingly investigate how we can design and implement a middleware suitable for use with these applications.

Massively multi-player online games (MMOGs) are of particular interest, because they are a genre of interactive applications that have seen considerable growth during the last decade. These applications have become increasingly popular among consumers, and according to the Entertainment Software Association [3], the number of gamers who play online games has increased from 31 percent in 2002 to 44 percent in 2006. Correspondingly, the number of subscribers to MMOGs has steadily increased since 1997, exceeding 13 million in 2006 [4]. Though MMOGs were first popularized by the entertainment industry, they are now finding use in other areas, such as education, training and business [5, 6]. MMOGs are virtual environments, which support thousands of concurrent users. The users interact with the virtual environment and other users through an avatar, and communicate through text, audio or video. The popularity of MMOGs can largely be contributed to their persistent, continuous and interactive nature. Incidentally, the success due to interactivity also raises the most challenging system requirement, i.e, *low latency* for all users.

MMOGs are designed to support thousands of concurrent users, and the real-time interaction requires the users to have a consistent view of the world. This means that the events of the game need to be processed and distributed as fast as possible. With a large number of users in the virtual environment, the number of events generated can easily reach immense numbers. This will cause the response time of the server to increase rapidly. The result is a loss or complete deterioration in the quality of service. To support these virtual environments, with such considerable numbers of interacting entities, there is a need for an efficient architecture, which is able to handle the load generated. As a result, MMOGs commonly deploy an exclusive instance of the virtual environment on a single, centralized, distributed system, such as a cluster or grid. In order to ensure an even distribution of the load, it is common to divide the game world into logical regions. These logical regions can in turn be distributed to various nodes in the system. This partitioning of the virtual environment is possible, because MMOGs by nature are decomposable systems. As in the real world, a user only needs updates for events occurring in the vicinity of him or her. The load is minimized as a result of reducing the number of entities that updates need to be processed for and delivered to, and makes it possible to distribute the users among the nodes composing the system.

For the purpose of this thesis, and the applications we target, we will focus on object oriented middleware. As such, an object should be understood as an instance of a class. An instance has a corresponding state; and a set of methods that can be used to alter or query this state. In order to implement support for distributed objects, there exists a number of middleware solutions that can aid in the development process. The benefit of using one such implementation depends on the type of application it was initially intended to be used with. It would seem that most existing models are intended for use with data driven applications, which have different requirements than MMOGs and their class of applications. For a data driven application, consider a scenario where a user submits a form in a web application, such as posting a message in a forum. The change in state, which occurs when receiving the new post, does not affect other users in such a way that they actively need to be made aware of the change in state. It is sufficient for the application to record the change, and let this be reflected in subsequent queries.

With MMOGs, the interactive nature means that users are now directly affected by a change of state. Consider a scenario where a user is walking across a bridge in the virtual environment, if this bridge is destroyed by a second user, this information must be propagated to the first user. If not, they will have an inconsistent view of the virtual environment. It is necessary for the users to be made aware of any changes to the environment, so they can react to these events accordingly. In the case of the bridge, the first user should fall into the river, which runs under the bridge, instead of safely crossing over to the other side.

It is possible to support the type of communication required by interactive applications using existing middleware, but there is a clear tendency towards lack of support for functionality that is practical and desirable for MMOGs, and this class of applications in general. Primarily, there is a lack of support for facilities that help provide low latency. The main reason for this could be that the demand for low latency exists as a result of human perception. Most applications are not so dependent on fast delivery of messages, but more concerned with the correct and timely delivery of messages. It is important to keep in mind the strong dependency on low latency, which is present in interactive applications, but not as integral for data driven applications. To give an example, if a web page loads slowly it is annoying, but has less of an impact on the user interaction. It is more important, in the data driven application, that the user is sent a web page that has been correctly generated.

A type of functionality that is of particular interest, is migration of objects. As we have mentioned, it is common to partition the virtual environment into regions and have users distributed to nodes in the system according to the region they are located in. Thus, to support the movement of users between regions, it must be possible to transfer the state of a user from one node to another. In situations like this, it would be particularly favorable to be able to migrate the object representing the user. Migration in this context implies that the object's state is packed into a representation that can be transmitted across the network, to be rebuilt at the receiving end. Migration makes it easier to accommodate for continuous execution, in the face of server maintenance. It also makes it easier to perform dynamic load-balancing. Both are easily accomplished by moving groups of objects between nodes in the system.

This functionality there is commonly little or no support for in existing middleware, because most applications are data driven in nature, as we discussed earlier. In those applications, the typical behavior of performing remote method invocation provides most of the functionality required. Remote method invocation, as the name implies, involves calling the method of an object at a remote location. There has been little need to actively move data between nodes. Consider the web application, where the posted message is stored and retrieved from a database back-end, which any node in the system can query to retrieve the latest information. For the web application, a distributed system means that there are multiple nodes that are able to render web pages for users, and as such it is possible to process more requests, faster. For the MMOG, the distribution must exist for the application to run at all, otherwise the number of events to process and synchronize would easily become to great. As a result, migration forms one of the central pillars of our design.

1.2 Contribution

We have seen that interactive real-time applications are highly dependent on low latency in order to provide the user with an optimal experience. The goal of this thesis has been to design and implement a middleware that is optimized to this end. Following this goal, we have identified a number of characteristics exhibited by these applications, which have greatly influenced our design choices.

We have identified the name service as a particularly interesting area. To minimize the traffic on the network, we have decided to use a distributed name service, where each node in the system is responsible for objects managed by it. This removes the overhead of having to register each objects location with a central register, and also performs faster than distributed hash table (DHT) based systems, which are dependent on recursive look ups that take additional time to complete. A distributed name service functions well in light of our grouping, where interacting objects are placed relative to each other. This because it is unlikely that the majority of objects will be accessed from outside the node.

Migration is also of interest, since it provides the application developer with the ability to migrate the state of objects between nodes in the system. This allows for dynamic load balancing, and optimization of resources. We have identified geographical placement of regions, or entire nodes, as a particularly interesting idea. We could, for instance, move the application based on the time of day, to accommodate for the varying activity in any given timezone. This migration should lower the latency for the users active during those hours, since a remote method invocation to a node closer in physical locality removes the overhead introduced by physical distance.

We have implemented a proof of concept, which uses a distributed name service, and has support for migration and remote method invocations. To prove the usability of our middleware we have developed a small chat application.

In order to aid in the development of applications we also provide the developer with a code generator. For our implementation we have decided to focus on a single language, and as such, our code generator does not form an integral part of the middleware. There are, however, middleware implementations where the code generator forms a larger part of the application, an example is the common object relational broker architecture (CORBA), as we will see in section 2.3.

We also realize that distributed systems introduce a number of aspects which must be taken into consideration, such as synchronization of events, partial failures, cache handling and more. We address these topics, and offer viable solutions to solving them, though we consider the implementation of these devices beyond the scope of this thesis.

1.3 Outline

The focus of this thesis is on the development of a middleware that is tailored for use with interactive applications, in particular we are targeting the requirements and characteristics of MMOGs. In chapter 2, we will present work which is related to, or supports, the efforts of this thesis. We will introduce middleware as a concept together with some well known implementations, which is not an attempt at being an exhaustive list, but focuses on introducing some core concepts, many of which are applicable to our middleware also, such as the name service, interface definition language and remote method invocation. We also shortly present the latest research into partitioning of the virtual environment.

In chapter 3, we will thoroughly discuss what has influenced our design. In particular, we will discuss the rationale behind choosing a distributed name service and how it integrates with migration. We also discuss the merits of automatically generating structures that integrate with the middleware.

In chapter 4, we present the implemented middleware, and how the abstract idea from the design chapter (chapter 3) has been turned into a concrete implementation. To solidify the details of the implementation we present a test application we created, which proves the usability of the middleware. We also discuss the implementation of the code generator.

In chapter 5, we discuss the results of our efforts. We will review the perceived benefits of our middleware implementation, and present some example of its use. Additionally, we will address some issues that are raised by distributed systems in general and that should be taken into consideration, such as the repercussions of failing nodes. We will discuss the effects of these implications, and as far as possible, offer viable solutions to solving them; however, the implementation of these facilities are beyond the scope of this thesis.

Finally, in chapter 6, we summarize the results and contributions of this thesis and present some topics for future work.

Chapter 2

Background

Distributed systems are composed of multiple nodes that are connected by a network. The benefit of running an application in such an environment comes from the ability to distribute the load. Depending on the type of application, there are a number of ways to accomplish this load sharing. In data driven applications, such as web servers, it is possible to duplicate the data on each node, and distribute users to nodes in the system depending on the load of each node. A focus of this thesis is interactive real-time applications, and for this genre of applications, particularly massively multiplayer online games (MMOGs), partitioning of the virtual environment into regions is a common way of distributing the load. To make the development process of these distributed applications easier there exists code that provides an abstraction from the low level details of the operating system. This software layer is commonly referred to as middleware, and we will discuss it further in section 2.2, followed by a study of a couple of middleware implementations related to this thesis. First, however, we will take a closer look at partitioning schemes for MMOGs.

2.1 Partitioning Schemes

To handle the large number of concurrently interacting entities in a virtual environment, it is common practice to use a static, region based partitioning scheme. The virtual world is thus divided into smaller, more manageable parts, where each region is hosted on a single node in the system. Some implementations allow several regions to be hosted on a node, such as Anarchy Online [7], while others are more conservative and allow only for one region per node, such as Second Life [8]. A widely accepted problem with the static partitioning scheme is that it does not take into account the dynamic nature of MMOGs. Even if the static partitioning is based on population density trends, and arranged to accommodate this, it is still susceptible to imbalances due to unforeseen events. Thus, a lot of research has been done on how to improve the flexibility of these partitioning schemes, and consequently, algorithms for efficiently distributing entities and regions. There is a clear trend in the research towards systems that can support dynamic load-balancing.

Turck et al [9] have investigated the effects of dividing a game world into dynamic micro-cells. A study with a similar background has been performed by Duong et al [10]. Such micro-cells can be reassigned to servers in a cluster if the load on the server they are currently residing on becomes too large. Three different load-balancing algorithms were used, none of which factored in locality of users, and the number of micro-cells supported per server varied. The test was done on a centralized cluster. The conclusion was that a dynamic approach is preferable, because it will decrease the chance for bottlenecks and lower the overall latency.

Another approach to solving the problems with static partitioning is through maintaining consistency by limiting updates based on an area of interest. IBM has developed a middleware for distributed games called Matrix [11] that uses this approach. It is based on the observation that MMOGs are nearly decomposable systems, and as such, it is usually sufficient to update players with only those events that occur in their zone of visibility. Matrix thus provides pockets of locally-consistent state. Results show that Matrix outperforms static partitioning schemes when the workload exhibits unpredictable and dynamic skews. Matrix makes use of region based partitioning as an underlying foundation, but this is for the purpose of easily distributing the virtual world across multiple servers. Matrix is also intended for a centralized cluster of servers. Another middleware, which implements this area of interest type partitioning is the Colyseus system [12], but this system is designed for first person shooter games and does not utilize the concept of regions.

In summary, the mentioned work on dynamic partitioning considers server load in a centralized cluster. Most of the research tries to optimize the partitioning of the virtual environment into regions that can dynamically accommodate hot-spots. These regions can be user centric, in the area of interest approach, or area centric, in the micro-cell approach. The goal is nonetheless always to minimize the amount of events being distributed, be this through dynamically moving areas when a server becomes overloaded, or by limiting the scope of a user. Regardless, most of these partitioning schemes make the assumption that the system consists of a centralized cluster of servers, grid or similar. The studies indicate that a dynamic approach to the distribution of users and reallocation of regions will provide improved efficiency, and is to be preferred over a static implementation.

2.2 Middleware

A distributed application runs on a system that consists of multiple nodes connected by a network. In order to synchronize the activity between the components of the application, they must be able to communicate. Adding such functionality to an application adds a lot of complexity. As a result, generic code bases, known as middleware, are developed to assist in the development process.

2.2.1 Background

In distributed computing, middleware is an intermediate layer of software between the application and operating system (see figure 2.1). There exists middleware supporting a wide range of programming paradigms, but for the purposes of this thesis we will focus on object oriented middleware. This is because objects are highly suited for supporting migration. We briefly mentioned migration in the introduction, and will discuss it in detail later. Additionally, object oriented programming is widely in use, according to TIOBE Programming Community Index [13], approximately 55 percent

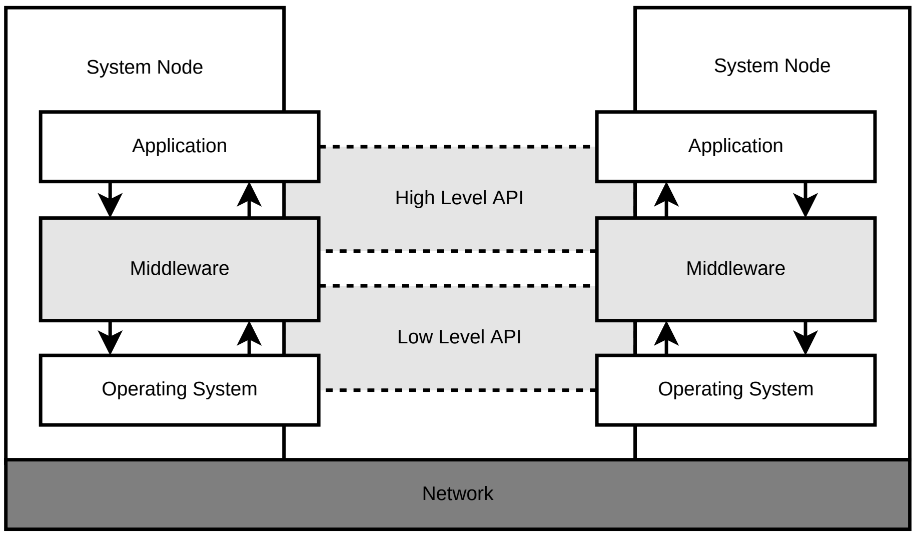


Figure 2.1: Middleware

of programs are written in object oriented programming languages, and the number is rising.

Though the focus of this thesis is on object oriented middleware, there exists a number of middleware implementations predating object orientation, such as Sun RPC [14] and DCE [15]. Any in depth discussion of these is beyond the scope of this thesis, though it is worth mentioning that a lot of the concepts remain the same, and much of what we discuss here has naturally evolved from these. It is also worth mentioning collaborative middleware. This is middleware that is meant to be used in highly heterogeneous environments, such as the Internet. The most prominent example of this is Web services and the SOAP protocol [16]. A primary problem with SOAP is that it is very slow. Parsing XML, which is the format used to exchange messages, puts a heavy load on the CPU, which in turn increases latency, and of course lowers throughput; due to the size of the format. As such, it is highly available and interoperable, but is less efficient as a side effect of that.

There also exists a number of object oriented based middleware, which we will not go into much detail of either, such as the Internet Communication Engine (ICE) [17]. ICE parallels the Common Object Request Broker Architecture (CORBA) [18] in its use of concepts, and accordingly, we consider ICE as sufficiently covered by our description of CORBA in a later section of this chapter. It is also worth mentioning the Distributed Object Component Model (DCOM) [19], which was developed by Microsoft to support the distribution of objects for the Windows platform. The initial restriction to Windows meant DCOM never became very popular. This even when an implementation of DCOM called COMsource [20], which was developed for use with UNIX, was completed at a later point in time by the Open Group [21].

2.2.2 Components

Middleware provides the application developer with three integral services; it neutralizes heterogeneity, provides mechanisms for masking the distribution of objects, and makes available a high level application programmer interface (API). In addition to these components, it is common to implement services that may be of use in the development process. One example would be to provide distributed synchronization of time, for ordering events.

Inherent to distributed systems is heterogeneity. With multiple nodes in a system, one should naturally assume that there are parts which are incompatible. At the hardware level, there is a high probability that some of the nodes have different CPU architectures or memory layouts. At the software layer, there is a probability that the nodes are running different versions of an operating system, or different operating systems all together. As such, one must consider how these systems represent data internally, how this data will look after the network interface has sent it onto the link, and how the receiving computer will interpret it. To hide this heterogeneity, the middleware provides mechanisms for serializing and deserializing the data. This typically consists of converting the data to an agreed upon representation, which can be understood independently of the previously mentioned issues. We will discuss this process in more detail in section 2.2.7, which deals with serialization.

Perhaps the most important function of the middleware is to provide transparent distribution of objects, that is to say, the application should be able to interact with remote objects as though they were local, without being intrusive. In reality, this is an ideal that is hard to accomplish, since the functionality to realize this is tightly coupled to the implementation of the object itself. Programming languages that support reflection and additionally have full control over the compiler or execution environment have an advantage in this area. Reflection gives a class the ability to inspect and modify itself during run-time, most dynamically typed languages support this, together with some interpreted languages such as Java [22]. We will see the benefits of this when we discuss Java remote method invocation (Java RMI) [23] in section 2.4. Still, there is a number of programming languages that do not support reflection, since this is a relatively new programming concept. Even if a language is capable of reflection, it is worth considering if the distribution of objects should be natively supported or implemented by third party libraries. At any rate, in most cases the middleware will compensate for this lack by providing the developer with an interface definition language (IDL). The IDL provides the developer with a simple way of defining an interface to an object. In the interface, the developer can define the methods that support remote method invocation (RMI) and their corresponding arguments and return values. The IDL provides the developer with a simple way of deciding how the application will interact with the middleware. The IDL definitions are parsed and structures that integrate with the middleware are compiled for RMI, migration and so forth.

In order to support RMI and migration, the middleware must implement functionality to intercept and redirect calls to remote objects. To redirect calls correctly we must be able to locate objects; this functionality, for locating and calling a remote object, is commonly referred to as a name service, and we will present and detail three models of this in a subsequent section. The indirection functionality is highly dependent on the programming language, and as such, we will not go into great detail about it in this chapter, but rather in our chapter on the implementation of our middleware.

The high level API is the access point for the developer to the before mentioned functionality, and depending on the implementation, this can be of varying complexity. In the case of Java RMI, it is ridiculously simple, but it also provides less functionality. With CORBA it is ridiculously advanced, but CORBA also provides a broad range of functionality.

There are a lot of problems which must be addressed in the face of distributed systems. In particular, it is important to consider what happens to the objects located on a node when that node fails. There must exist facilities that a node can fall back to if the node managing an object is not responding. We will discuss this and more in chapter 5. In the remainder of this chapter, we will introduce concepts that are integral to middleware, starting with the name service.

2.2.3 Name Service

A name service exists to provide a uniform entry point for locating objects in a distributed system. In order to accomplish this, a name is mapped to an object. This link between an object and its name is entered into the name service. Following this analogy, references to objects are no longer to addresses in local memory, but to names. This name can be used to query the name service, which will resolve the location of the given object. Multiple schemes for implementing a service such as this exist; as a single centralized service, distributed to each node or as a hybrid, as detailed by Znati and Molka [24]. They are shortly summarized in the following sections.

Centralized

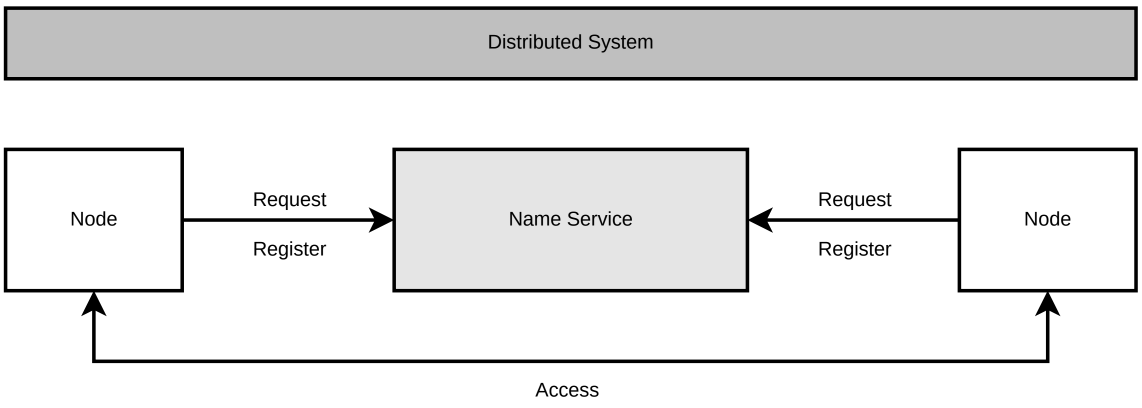


Figure 2.2: Centralized Name Service

In figure 2.2 we can see a diagram of the interaction between the nodes and the name service in a distributed system utilizing a centralized name service. As we can see, in the centralized approach the name service is located on a single, well-known node, all mappings from names to objects occur at this central repository, and in reverse, all location resolutions. The access to objects is performed by the requesting client directly to the node that is managing the object. Examples of centralized name services are the Andrew file system (AFS) [25] and Napster [26].

Advantages of having this service centralized is that there is no replication of the name service functions, making it easier to perform updates, and control consistency. Also, the objects can be governed by one naming scheme.

The shortcoming of the system is that it requires an extra level of indirection before the named object can be accessed. Also, as the average amount of mapping information increases, the average amount of time for processing a request increases, which may cause the name service to become a bottleneck in the system. Additionally, the reliability of the system depends largely on the reliability of the central node.

Distributed

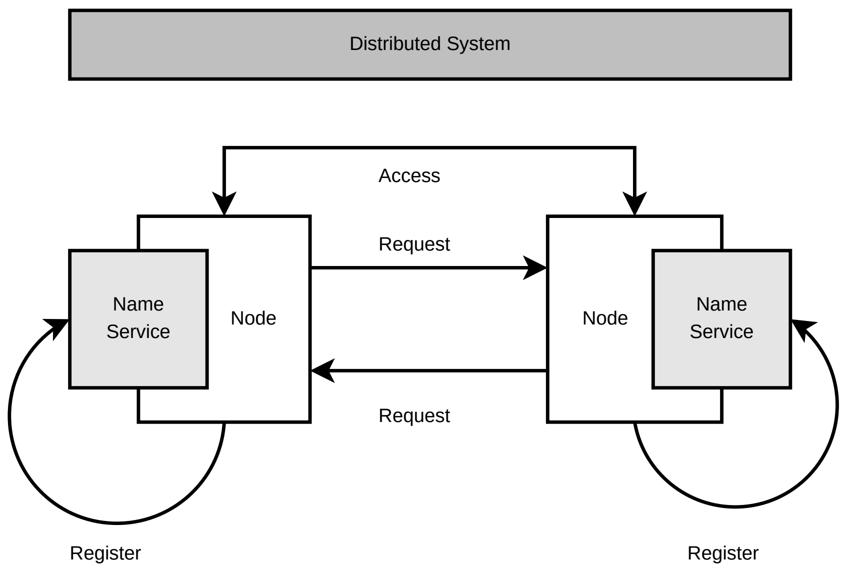


Figure 2.3: Distributed Name Service

In figure 2.3 we have illustrated the interaction in a distributed system utilizing a distributed name service. As we can see, the objects are mapped to their names locally at each individual node. The translation from name to object is thus performed at the location where the object resides, the action of registering is thus performed by the requesting node. Examples of distributed name services are distributed hash table based systems, such as, Chord [27] and Tapestry [28].

A distributed name service offers improved efficiency, since no additional name services need to be contacted to access the object. One gets enhanced reliability, because a named object is always accessible if its manager is accessible. Thus, the failure of a node only disables access to the namespace it managed. The effort of maintaining data consistency is also simplified, because a remote name service does not need to be contacted to update the attributes of a named object.

A disadvantage is that it becomes unclear how a node is initially located based only on a given high level name. This is because accessing the named object requires the identification of the object manager, which in turn can only be achieved by identifying the named object.

Hybrid

In figure 2.4 we see how one possible way of combining centralized and distributed techniques can be accomplished. As we can see, each node has a name service, such as with the distributed approach. It also belongs to a region. Each region manages a domain of objects, and each node in a region registers its presence with a global name service manager. When requests for names that a node does not control are made, a request is sent to the global name service for a resolution, this request is multi-cast to the intended name handling group, upon which only the intended receiver in that group sends a reply. This is how the V system [29] implements its name service. We will discuss these aspects further in view of our design criteria in chapter 3. Another example of a hybrid name service is the domain name system (DNS) [30].

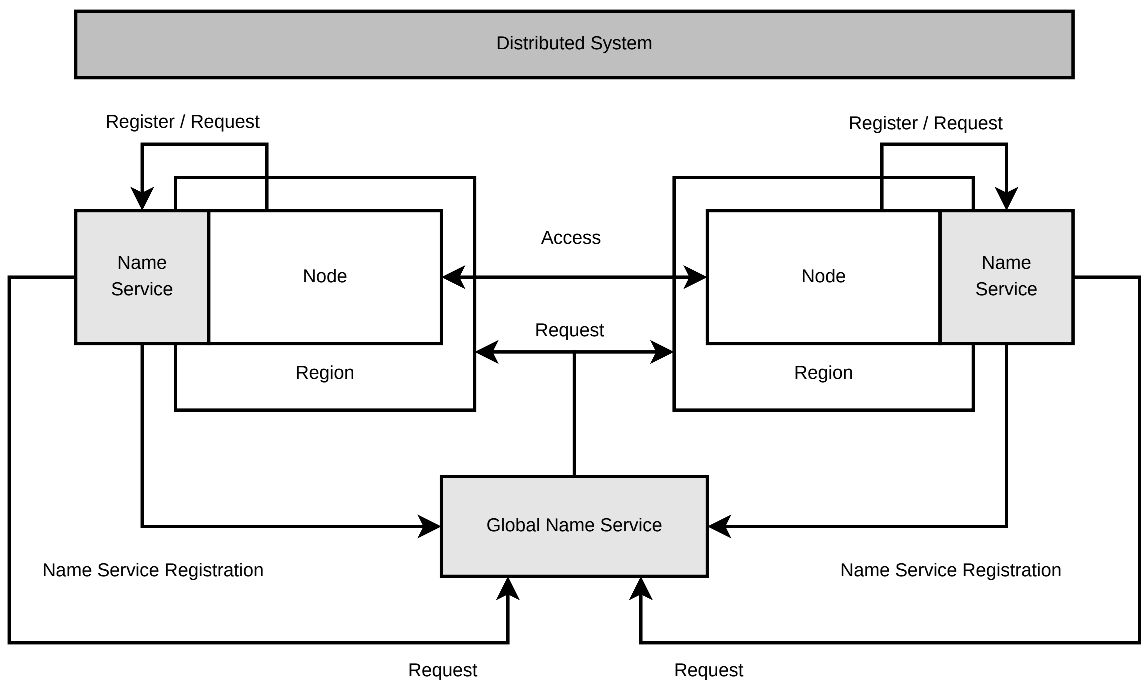


Figure 2.4: Hybrid Name Service

Summary

It is difficult to place the implementation of the name services for existing middleware cleanly into the distributed or centralized approaches. The reason for this is two fold, first, the centralized approach is incapable of scaling well, due to the potentially high number of requests and registrations that will occur. Secondly, the distributed approach requires particular handling during start-up to ensure that objects are located correctly, which can prove cumbersome. As a result, variations between these two are commonly utilized, as is the case with CORBA and Java RMI.

2.2.4 Remote Method Invocation

Since objects can be located on any node in a distributed system there must be a way for these objects to interact even across the network boundaries. This functionality is provided through remote method invocation (RMI) in object oriented middleware. It makes it possible for a method of an object to be called independently of the location of the caller. This type of functionality provides the application with a range of possibilities; it can be used to distribute operations that require a lot of processing (load sharing), screen access to sensitive data (security), offload systems under strain (load balancing) and more.

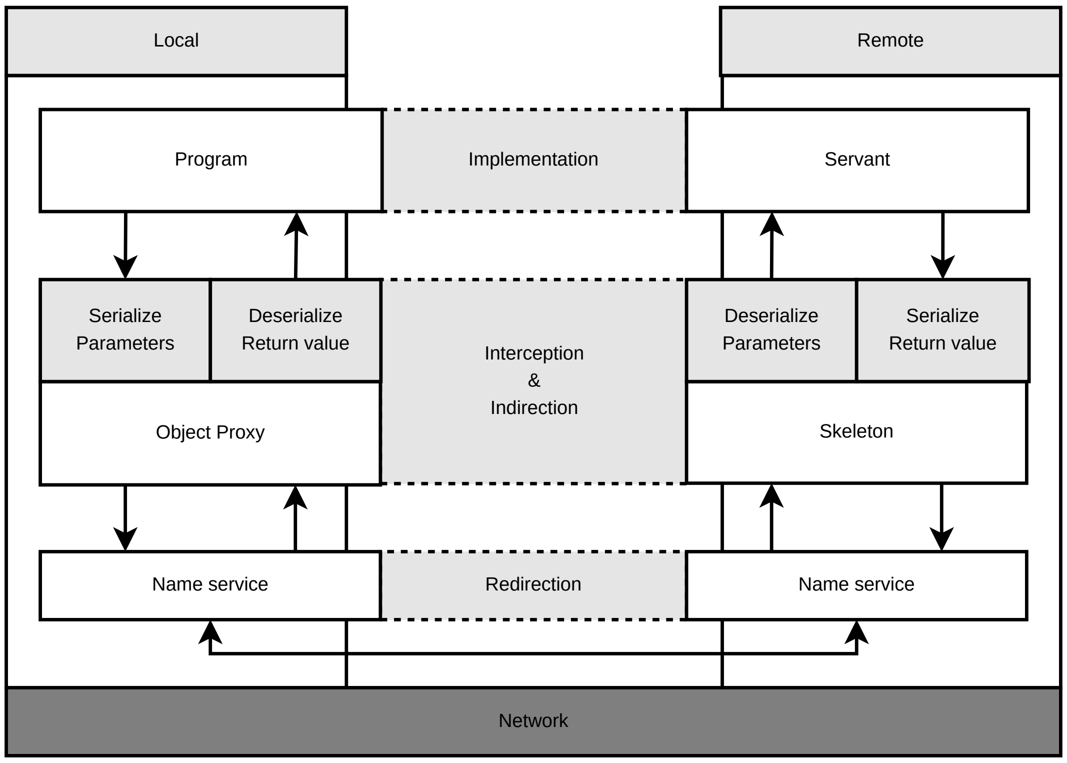


Figure 2.5: Remote Method Invocation

In order to perform RMI, the middleware must intercept and redirect calls to remote objects. Figure 2.5 illustrates how this is accomplished. The program is calling, what it assumes to be, the local implementation of an object. We see that a proxy intercepts the call to the object, and thus, functions as a mediator between the name service and skeleton at the receiving node. In this manner, the proxy behaves like a local object as far as the program is concerned, and thus makes the RMI transparent. Instead of executing a local invocation, it forwards the invocation in a message that can be interpreted by the receiving node. It hides the details of the remote object reference, the serialization of arguments, deserialization of return values and sending and receiving of messages. The proxy consults the name service, in order to ensure that the correct object, on the correct node, receives the invocation. The invocation is sent onto the network, and received by the remote node. The name service resolves the request and dispatches the invocation to the corresponding skeleton. The skeleton deserializes the arguments and invokes the corresponding method in the servant. The servant is an instance of a class that provides the body of a remote object, its behavior is symmetrical to that of the proxy. The skeleton waits for the invocation to complete and then serializes the result, together with any exceptions. This information is then propagated back through the system to the caller.

It is also common to embed a version number with a remote method invocation, this is to ensure that both the sender and receiver agree on the semantics of the called method. Without this version number there is no reasonable way to assure this otherwise.

2.2.5 Interface Definition Language

An interface definition language (IDL) provides the developer with a syntax for defining the interface of an object. The interface will typically be composed by determining which methods of the object should support RMI, which version of the interface it is, perhaps even who or what objects should have access to it. The range of options depend on the complexity of the middleware, and which services are made available. The resulting interface definition file is parsed to generate structures which will easily integrate with the middleware.

The motivation to provide the developer with an IDL are many, where the obvious ones are ease of integration and abstraction of low level details, which will allow the developers to focus their efforts on the development of the application. In addition, it makes it possible to define generic objects that the middleware can use to generate structures for multiple programming languages with, which in turn makes it possible to provide implementations of the distributed objects in different programming languages, depending on the requirements of the applications components.

2.2.6 Migration

In essence, migration involves moving objects from one node in the system to another, and updating the name service accordingly. The advantage of this behavior is that it allows for flexibility. It is not predetermined where objects are located, since they are free to move around. Most classic distributed systems do not require this type of functionality, it is more common to replicate objects at each node. As such, the availability of services is more important. It is, however, perceivable that this type of functionality could be of use to a number of applications, by considering the benefits of being able to dynamically relocate objects. It will, for instance, be possible to dynamically accommodate for high load on one node by moving the busy objects to a node that is experiencing less traffic.

It is important to differentiate migration in view of replication. Replication is by far the more common type of functionality to provide. The reason for this is primarily because existing distributed applications are highly data driven. In light of event based applications, such as interactive real-time applications, it is more interesting to look at functionality such as migration. This is primarily because the performance of the application does not depend on the availability of data, but on the efficiency in processing events.

For migration to function, the name service must provide mechanisms for maintaining references to migrated objects. While migration is perceivably useful for a number of applications, it is not certain that the added complexity provides enough of an advantage to warrant the use.

2.2.7 Serialization and Deserialization

Serialization consists of two sets of operations which reflect each other, serialization and deserialization. Serialization is the process of saving the state of an object, and deserialization is the process of reconstructing the object. When serializing, there is no knowledge of where the objects state will be stored, where it will be restored, what will restore it, or how it will be moved to its point of restoration. As we mentioned earlier, distributed systems are intrinsically heterogeneous. As such, we must assume the worst, in that the object will be restored on a node, which in some way is incompatible with the sending node. To negate the effects of heterogeneity, a common representation for the serialized data is agreed upon. Determining which representation to use can depend on a number of factors, such as readability, size on disk and processing speed. It is possible to use a textual representation, such as XML. This is, as we mentioned, used by SOAP [16]. While XML is a highly interchangeable format, it is also slow to process and requires a lot of space, bandwidth and memory. The alternative is to use a binary format, which will require less space on disk, and take less time to process, but will most likely be less interchangeable and less readable. The ICE middleware [17], for instance, uses binary serialization.

2.3 Common Object Request Broker Architecture

The Common Object Request Broker Architecture (CORBA) [18] is an open distributed object computing infrastructure which is standardized by the Object Management Group (OMG) [31]. CORBA supports multiple programming languages through an interface definition language (IDL). The developer defines the methods that should support remote invocation and their corresponding arguments and return values. The interfaces defined by the developer are compiled, which results in proxies and skeletons that integrate with the CORBA middleware. These structures provide the necessary indirection mechanisms, for intercepting calls to remote objects and handling requests from remote callers. In CORBA, proxies are developed in the client programming language and skeletons in the server language, and as such, any CORBA object can be implemented in a number of different programming languages. CORBA does not support the notion of classes, and accordingly, does not allow for instances of classes to be passed as arguments or returned as results while invoking remote methods. It does, however, support data structures of various types and arbitrary complexity. It is possible to pass CORBA objects as arguments and results, which are the remote implementations of the interfaces, in this case a remote reference is returned, and RMI can be used to query the object.

The middleware component that supports RMI is called the Object Request Broker (ORB). Many different ORBs have been implemented from the OMG specification [32], supporting a wide variety of programming languages. The ORB's role is to help a client invoke a method on an object. This role involves locating the object, activating the object if necessary and then communicating the client's request to the object.

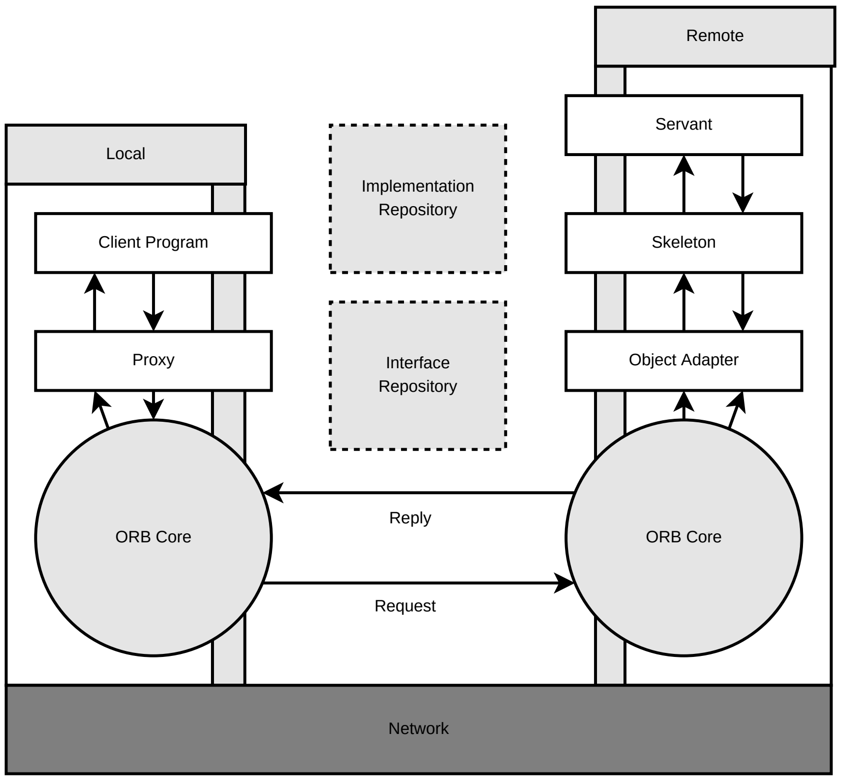


Figure 2.6: The Main Components of CORBA

From figure 2.6 we can see that there are a lot of similarities between the CORBA architecture and the RMI architecture seen in section 2.2.4.

The ORB core provides end to end communication, and additionally provides operations to convert between the internal representation of an object reference and a string representation. It also provides operations to provide argument lists for requests using dynamic invocation. Dynamic invocation is the process of dynamically building an invocation to a remote object, based on querying the interface repository for methods that have certain characteristics.

The object adapter bridges the gap between CORBA objects with IDL interfaces and the programming language interfaces of the corresponding servant classes. The object adapter creates remote object references for CORBA objects. It dispatches each RMI via a skeleton to the appropriate servant. As with RMI, the skeleton deserializes the arguments and serializes the return value. The proxy also has the same role as with RMI, i.e, deserializing the result and serializing the parameters.

The implementation repository is responsible for activating registered servers on demand and for locating servers that are currently running. The object adapter name is used to refer to servers when registering and activating them.

The role of the interface repository is to provide information about registered IDL interfaces to clients and servers that require it. For an interface of a given type it can supply the names of the methods and for each method, the names and types of the arguments and exceptions. A Dynamic invocation interface allows clients to make dynamic invocations on CORBA objects. The client can obtain from the interface repository the necessary information about the methods available for a given CORBA object. CORBA also introduces the concept of dynamic skeletons that make it possible to dynamically add new CORBA objects during run-time. It is worth noting that in CORBA, once an object has retained a reference it will be associated with that reference for the remainder of its lifetime.

2.4 Java Remote Method Invocation

Java RMI [23] extends the Java object model to provide support for distributed objects in the Java programming language. It makes it possible to invoke methods for objects running on other Java virtual machines (JVMs). The distributed object model is integrated with the Java programming language and could be implemented completely transparently. The language developers have however decided that, for a class to support RMI it must implement the remote interface. This is most likely to minimize overhead when generating code and to make the support for RMI in a class explicit. It becomes apparent that working with distributed objects in Java is relatively simple, much of this can be attributed to the single programming language model and pointer hiding. It is possible to interact with other programming languages using Java, but in this case, it falls back to a CORBA implementation, and one must then define an IDL interface and so forth, which greatly complicates the process.

The semantics of invocations follow that of figure 2.5. With Java RMI any class that implements the serializable interface can be passed as an argument or returned from a remote invocation. When an argument or return type is an instance of a class which implements the remote interface, a remote reference is passed as an argument, otherwise it is passed by value. In the case of the remote reference, RMI can be invoked to gather information about the object, but this might add a lot of extra calls.

Java is designed to allow classes to be downloaded from one virtual machine to another. This is particularly relevant to distributed objects that communicate by means of remote invocation. If the recipient does not already possess the class of an object passed by value, its code is downloaded automatically.

The RMIregistry is the name service for Java RMI. An instance of RMIregistry must run on every server computer that hosts remote objects. It maintains a table that maps textual, URL-style names of references to remote objects hosted on that computer. These remote references look like: //computer:port/object.

Note that there is no inherent support for migration of data. With Java, once an object has received a reference at a name service, it will retain that name for the remainder of its lifetime.

2.5 Summary

Existing implementations of middleware do not accommodate for the dynamic nature of interactive real-time applications. We have seen that a trend in the research regarding massively multi-player online games is focused on dynamically adjusting to the load in the system by migrating regions. It is apparent that there is little or no support for migrating data in CORBA or Java RMI. Their primary concern is to provide facilities for RMI, and discovery of interfaces for performing RMI. This can be seen in the way references are maintained. After an object has received a reference it is hard, if not impossible, to move this object to another location in the system. Migration becomes an extremely complex operation to execute. As a result of our findings we will in chapter 3 outline the design of a middleware that we believe accommodates for the requirements and characteristics displayed by interactive real-time applications.

Chapter 3

Design

The unified approach to developing middleware is no longer as applicable as it used to be. This comes as a result of the increased number of applications with differing requirements and characteristics. The solution has been to develop specialized middleware that is generic for a subset of applications.

3.1 Characteristics and Requirements

Interactive real-time applications have a requirement that is essential for satisfactory performance: low latency. To guide in our development of a middleware that can provide interactive real-time applications with this, we have isolated four components of the middleware that we consider to be of particular importance:

1. *Migration*

   An object in the system can be moved from node to node.

2. *Remote method invocation*

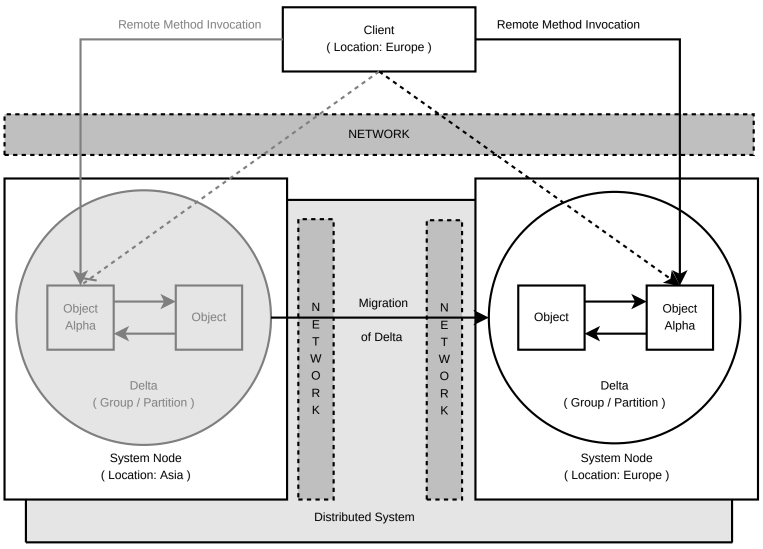
   An object's method can be invoked regardless of the callers location relative to the object.

3. *Distributed name service*

   An object in the system is managed by the node it resides on.

4. *Partitioning and grouping*

   The applications namespace is partitioned so interacting objects are grouped accordingly.



Remote Method Invocation
Client (Location: Europe)
Remote Method Invocation
NETWORK
Object Alpha
Object
Delta (Group / Partition)
System Node (Location: Asia)
NETWORK
Migration of Delta
NETWORK
Object
Object Alpha
Delta (Group / Partition)
System Node (Location: Europe)
Distributed System

Figure 3.1: Middleware Concept

We have illustrated the relationship between these factors in figure 3.1. In short, we can see how the global namespace has been partitioned so that interacting objects are placed in groups corresponding to this. At some point in time, the group *Delta* has been migrated from one node in the system to another and accordingly the client now performs remote method invocations on the object at its new location. Remote method invocations are the primary means of communication in object oriented middleware and extend the local invocation model to function even when objects are located on different nodes in the system. The objects, after migration, are managed by the name service at their new location. References to the new location of the objects are maintained by the original node, which means these objects are still accessible even after they have been moved. Groups of objects are distributed to multiple nodes in the system to distribute the load evenly and accordingly migration makes it possible to dynamically balance the load at a later point in time, if the use of resources becomes skewed.

The four factors we introduced this chapter with, are derived from the following characteristics and requirements. During our research we have noted that these are typical traits for applications that are time dependent, event based and interactive:

1. The system must be able to handle large numbers of concurrent users.
2. A large number of objects will be created, with greatly varying life spans. In an MMOG, for example, consider the life-time of a bullet in comparison to the player firing the weapon.
3. There is intermittent connectivity and varying amounts of transmissions. As such, the density of users per node, and accordingly events, may vary greatly depending on the interaction patterns of the users.
4. To handle the load generated by so many concurrent users, it is necessary to partition the namespace into groups.
5. Clients and servers use the same libraries. This means that code is shared and that only data needs to be migrated. It also means that we can call any function of a remote object directly, without having to discover which function interface to use.

The factors we have outlined, together with the characteristics and requirements described here, each have their impact on the design of the system. In the remainder of this chapter we will discuss in detail the motivation for selecting the design we did, starting with migration.

3.2 Migration

Migration is the process of moving objects from one node in the distributed system to another. This functionality makes it possible to dynamically balance the load in the system, and perform other activities that require moving parts of an active application, without disrupting the flow of execution, between nodes. There are several issues that need to be taken into consideration when migrating objects, for instance, there must exist a way to save the state of the object, transmit it to another node and have that state reconstructed. This functionality, for storing and reconstructing objects, is commonly referred to as serialization, which we will discuss further in section 3.6. Additionally, we need to maintain references to objects after they have migrated. We have made provisions for this in our name service by keeping two identifiers for each object. We will discuss the design of our name service further in section 3.3.

Migration introduces several aspects that need to be taken into consideration. After an object has been migrated, it is likely that there are users still interacting with the object. There are two ways of maintaining this interaction. The first is to have the migrating node pass on requests for the object to the receiving node. This defies the point of migrating objects, since all we have accomplished is to increase the traffic on the network, and occupy two nodes in the system when issuing requests. Nonetheless, there could be scenarios where this is useful. One such scenario comes to mind with regard to MMOGs that support seamless regions. Consider the situation, where a player is interacting in the space where two regions overlap. In this case it could be useful for one region to manage the player and pass on messages to the overlapping region.

The second option is to have users reconnect to the receiving node, and continue their interaction with the object directly. This second solution introduces the problem of how connections to the node are maintained. There are are two main possibilities, to maintain one global connection, and reconnect to nodes depending on what object we wish to interact with, or to maintain one connection per node we are using. To aid in our decision, we need to consider the factors that influenced our design from the beginning. Primarily, we need to consider partitioning, grouping and how the requirement for low latency comes into play. If the objects that communicate are grouped together, and the users interact with the system on a group basis, it is most reasonable to reconnect based on the location of an object. If this is not the case, it is more reasonable to maintain multiple connections.

In our design, we have decided to reconnect, and maintain one global connection. The rationale for this decision is based on the fact that we assume most applications will attempt to group objects that communicate, primarily because the middleware is designed for this purpose.

Another consideration that must be made is in regard to what the object will do once it has been migrated. There are two possibilities, the first is to have the object register itself with a local service, in order for it to be included in the run-time operations of the new node. The second option is to let the object dangle, and only be available for remote method invocation from objects, which are aware of its existence already. This, by all effects, is more of an application design question, then a middleware one, but the middleware must be able to support either case.

It is also important to remember that with our middleware we assume all nodes in the system share the same libraries, and as such, migration only involves moving the data of an object. This means that all nodes in the system must have knowledge of a class of objects. If the receiving node is not able to interpret the type of an object it will dismiss the migration. This also means that it is not possible to dynamically add or query for new classes during run-time, such as is possible in CORBA and Java RMI.

This also raises the question of compatibility. There is a possibility that a node or user in the system ends up running an older version of the libraries. In this case, the middleware should embed each migrating object with the version number of the class. This can become an issue because the serialized object is a flat representation of the objects data, and as such, there is no inherent way of determining, based only on the objects data, if the object carries the same semantic meaning any longer.

3.3 Name Service

When objects are distributed across multiple nodes in a system accessing objects that are not in local memory require special handling. Primarily, it is necessary to locate the node on which a particular object is currently residing so that the call can be redirected correctly. This functionality is provided by the name service that will map an identifier to an object together with information about its location. References to objects in the system are to these identifiers instead of addresses in local memory. The approaches to implementing a name service are centralized, distributed or hybrid.

3.3.1 Architecture

Znati and Molka [24] analyzed three approaches to implementing a name service; in form of centralized, distributed and hybrid versions. Prior to contacting the target object itself, the centralized version contacts a name server to obtain the objects location in the network. As such, the centralized naming scheme adds an extra level of indirection to the name resolution process. The distributed paradigm removes this level of indirection by placing the name of the object with the object itself. The hybrid approach is based on the design principle of keeping names together with the objects they are bound to on the local level, but resorts to multi-casting when resolving names at a regional level. This study indicates that the choice of model for a name service will influence the performance of the service and the throughput of the network. The results showed that the centralized model could achieve acceptable performance only as long as the ratio of remote to local requests was kept reasonable. The performance of the hybrid model highly depended on the efficiency of the cache design. With all other network conditions set equal, they found that, relative to the response times of the centralized simulation, the response time of the simulation of a distributed name service were smaller.

For interactive real-time applications, we described, earlier in this chapter, a set of five characteristics that had a considerable impact on the design of our system. With these requirements in mind, a fundamental problem with the centralized name service is that all object resolution and registration is performed at a single point in the system. This means it easily can become a bottleneck in the system, particularly for systems that have a high object creation frequency or high request frequency. The result is that a centralized name service is not scalable. As the number of users increases, so will the added traffic and eventually the system will come to a stand still. A centralized name service also introduces a single point of failure. If the name service ever crashes, the entire system will halt, since no object resolution or registration can be performed. Naturally, this can be solved by replicating the name service, or something to its effect. A centralized name service still leaves questions unanswered, such as where to place the name service relative to the participating nodes in the system.

A hybrid name service solves a few of these issues, but introduces a few of its own. The name service is now distributed at a local and regional level, which deals with the placement of the name service. Though, instead of a single point of failure, we now are faced with partial failures. A partial failure means only the objects managed by a crashing node will become unavailable when a node crashes. There is, however, a possible weakness in the way objects are located. The look up method relies on multi-casting, or its equivalent, and there is a high probability that the response time for such a request will be too high for applications which are time dependent. Since objects are bound locally, rapid object creation and destruction no longer create the same problems as with the centralized name service. We no longer have to worry about the name service becoming a bottleneck in the system.

Last, there is the distributed approach, which raises an interesting issue; in that there is no clear way to show were an object is located without first contacting its name service and how that name service is located given only a high-level name. For our middleware, this is not an issue since at start-up the system takes on a centralized approach, where a single node in the system initiates all communication. Thus, there is always a known path to an object. A suitable analogy is to ripples in a pond that expand from a single point. The distributed version leaves the issue with partial failures unresolved, but apart from this it serves the purpose of our middleware well. It also shows the most potential, in that the tests of Znati and Molka [24] revealed this to have the highest performance benefits, without the need to optimize caching etc, which can become an issue, considering migration.

3.3.2 Interoperation with Migration

In addition to choosing where the mapping and resolution of objects occurs, we also need to consider how these mechanisms will interoperate with migration. In this section, we outline the design of our distributed name service, using terms from mobile IP [33,34] to solidify how the migration is combined with the distributed name service. Mobile IP addresses the desire to have continuous network connectivity to the Internet irrespective of the physical location of a node. This coincides with our goals in that the objective is to make mobility transparent to the application. The analogy is suitable, because where mobile IP is used to find a route to a mobile computer, moving from network to network, we need to find a route to a mobile object, moving from node to node. A prerequisite to accomplish this is to be able to uniquely name a computer or object in the context that it is used in. We find that the taxonomy used to describe these processes overlap. In the following discussion, we will primarily focus on the definitions of mobile node, home agent, foreign agent, care-of address and home address.

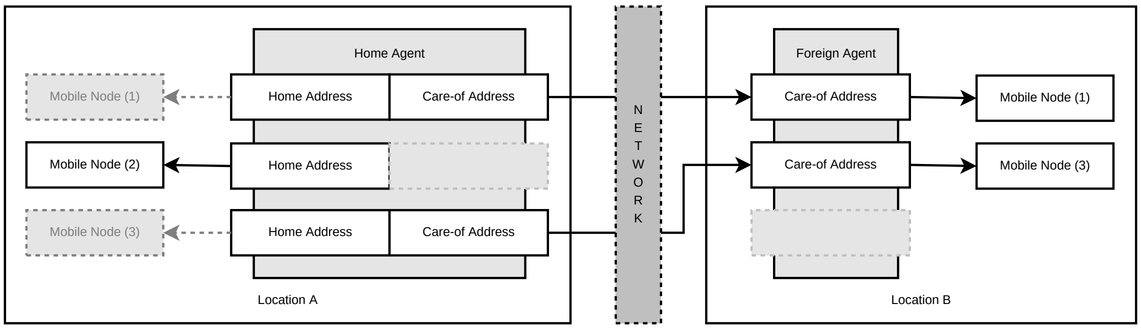
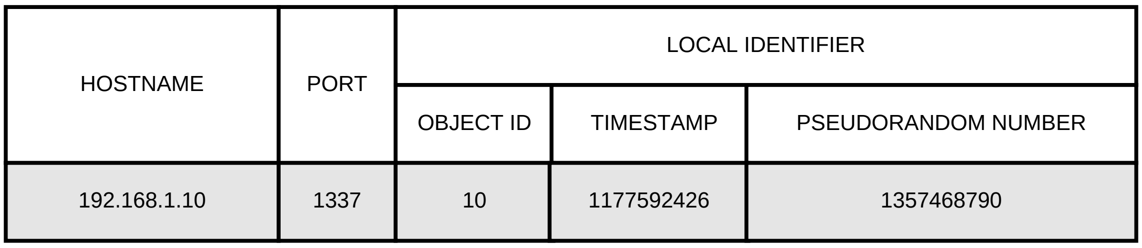


Figure 3.2: Mobile IP

Following figure 3.2 we can see the interaction between the elements that compose mobile IP. When a mobile node (object) has migrated to another node in the system, it registers its presence with the foreign agent (name service) at its new location. The foreign agent issues a message to the mobile node's home agent (name service), in the form of a care-of address (local object identifier), which the home agent can use to forward requests to the mobile node. Each node in the system has an active name service. This name service acts as a foreign agent when receiving migrated objects and a home agent otherwise. Our implementation diverges from the approach of mobile IP at one point. In mobile IP, a mobile node has one home agent throughout its lifetime, and thus one single entry point. This is not the case in our implementation, where a mobile node can have multiple entry points, depending on how many nodes it has been moved to, and thus been registered with. As a result, our version will leave a trail of references that can be unraveled to reach the mobile node at its current location. Following the mobile IP example again, we can consider how a request for the mobile node marked as *(1)* in figure 3.2 at *location A* will be handled. The request will be processed by the home agent that discovers that the mobile node has a care-of address associated with it. As an effect of this, the request is forwarded to *location B* that is currently managing that mobile node. The foreign agent at *location B* will in turn forward the request to the mobile node that in turn will process the request and send a reply directly in return to the requesting party.

3.3.3 Reference Maintenance

Since we have a distributed name service, a single node initializes the system. Objects that compose the system are propagated to other nodes based on necessity, and references to the propagated objects are maintained in the name service of the corresponding nodes. If this were not the case, it would be impossible to maintain references between objects, because we would have no way of knowing they existed. To identify objects in the system, we use an identifier that looks like that depicted in figure 3.3.



HOSTNAME	PORT	LOCAL IDENTIFIER		
		OBJECT ID	TIMESTAMP	PSEUDORANDOM NUMBER
192.168.1.10	1337	10	1177592426	1357468790

Figure 3.3: Format of the Home Address and Care-of Address

The care-of address and home address, which are used to identify objects in the system, have a format as seen in figure 3.3. The address uniquely identifies an object in the system throughout the lifetime of the application, since we assume that the application is designed to run indefinitely. We see that there are three main sections which compose the address. The hostname identifies the node where the object is located. The port provides an access point to the name service, which manages the nodes objects. The local identifier is specific to an object in the name service. The object-id is an index to more information about the object, such as the pseudo-random number and timestamp. The timestamp is required to identify the object temporally, but since this does not guarantee it to be unique over time, because of uncertainties related to computers and time keeping, we have a pseudo-random number in addition.

As a result, we end up with a name service that looks like that depicted in figure 3.4. We can see how the system will function by looking at the interaction between the objects *ChannelMngr* and *Channel*. We can see that *ChannelMngr* has a reference to *Channel*, this reference is not to an address in local memory, as we normally would expect, instead the reference points to the home address of *Channel*. We have seen in figure 3.3 what the home and care-of addresses for objects will look like. As we can see from the figure, *Channel* has at some point in time been migrated to another node in the system. As a result of the migration *Channel* has been registered with the name service at its new location and has as a new home address. This home address has been returned to the node that previously managed *Channel*, and has been placed in the care-of address. Due to indirection mechanisms in the middleware, any calls made by *ChannelMngr* to *Channel* will be intercepted and redirected in a correct manner. It is worth noting how the name service at *Node 1* has the same view of *Channel* as the name service at *Node 0* has of *ChannelMngr*. As far as *Channel* and *Node 1* are concerned, it was first created at this location. It is only the name service at *Node 0* that is aware of the fact that at some point in time it was managing the *Channel* object.

The result is a name service that is capable of maintaining references to objects independent of their location in the system, even after they have been migrated. We will discuss migration in detail in section 3.2. The name service also provides scalability, as long as the name space can be reasonably partitioned.

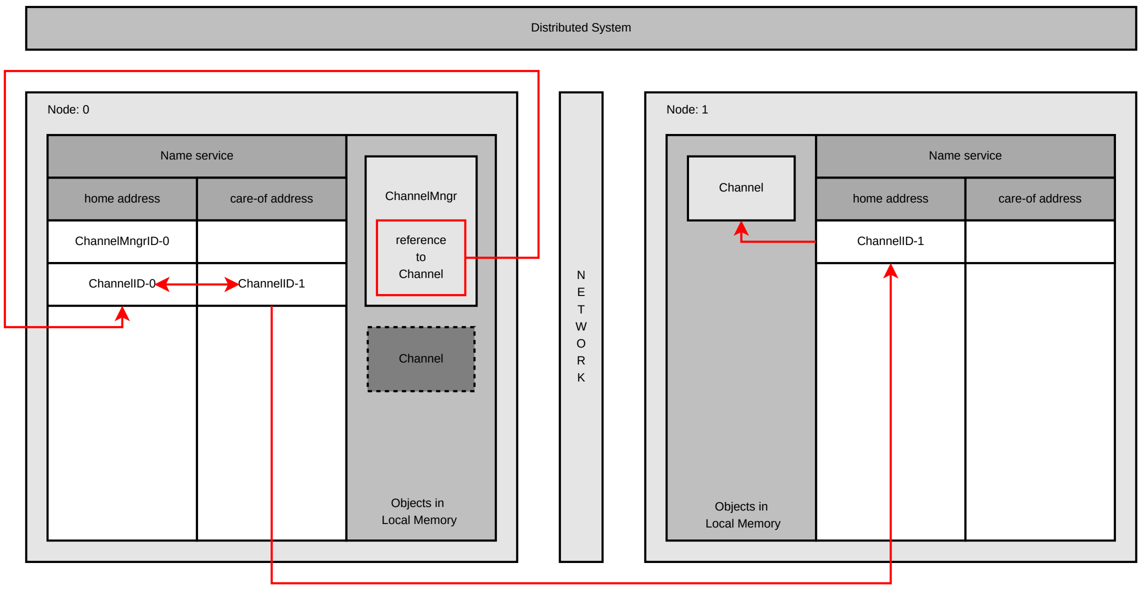


Figure 3.4: Design of the Middleware

3.4 Remote Method Invocation

As we have seen, it is necessary to intercept calls to objects that are located at a remote node. The name service is one of the components that we require to accomplish this. In addition, it is necessary to provide an indirection mechanism that queries the name service to determine if an object is local or remote prior to a method is invoked. Once it has been determined if an object exists in local memory or not, there are two scenarios that can unfold. If the object is local, the handling is quite easy, it follows the steps for a local invocation, where, upon successful completion the method returns a result. In figure 3.5, by following the black arrows, we can see how a local invocation is performed.

In the case where the object is remote, the process becomes more complicated, there are suddenly a number of additional scenarios that must be taken into consideration. In figure 3.5, by following the red and blue arrows, we can follow the successful invocation of a remote object. The red arrows show the path to the method invocation at the remote node, and the blue arrows show the path of the return value. We will use figure 3.5 to explain the process involved and outline at which points failures can arise. We see that the indirection mechanism intercepts the method invocation, it queries the

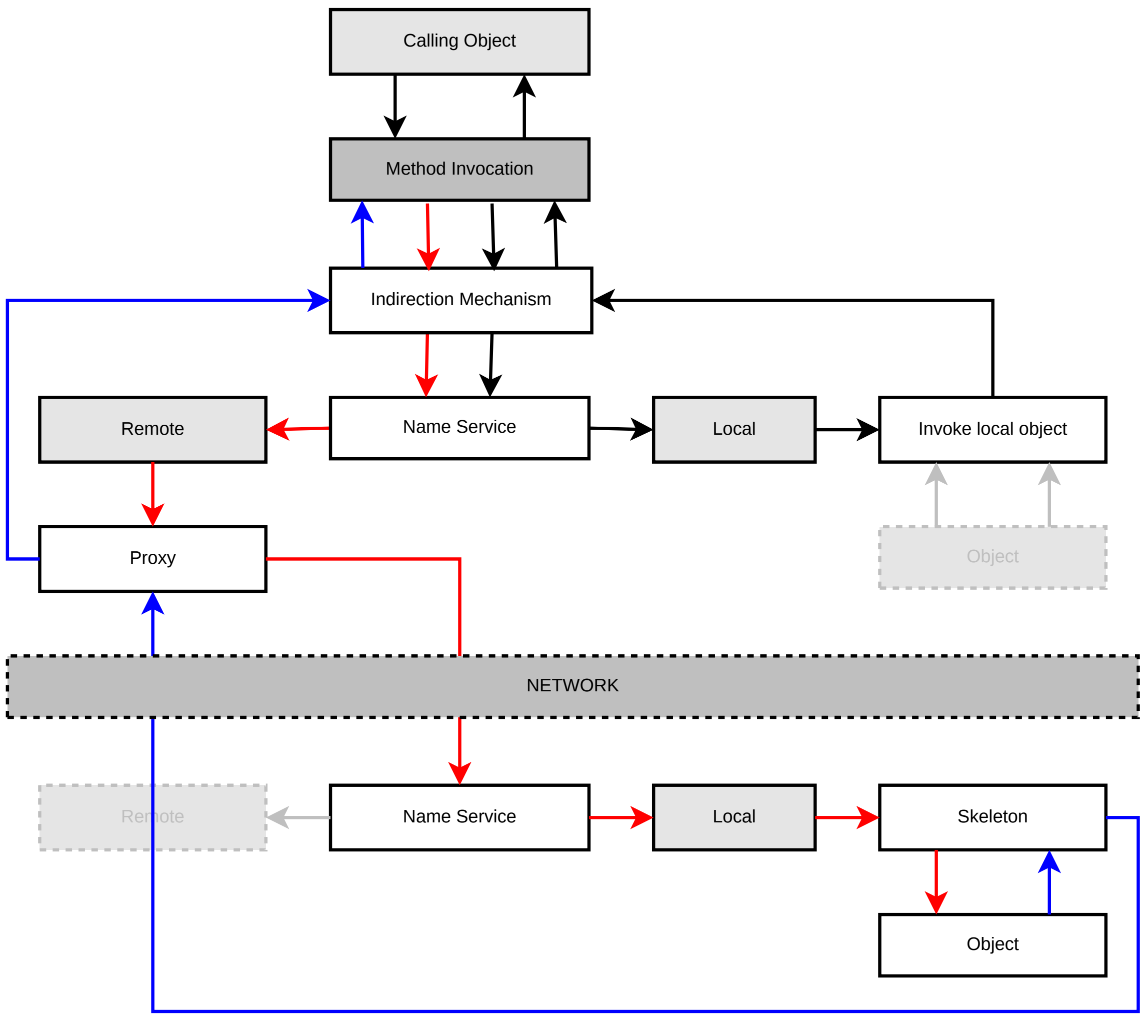


Figure 3.5: Indirection Mechanism

name service to determine if the reference is to an object at a remote node, which it is in this case. As we can see the call is redirected to the objects proxy. The architecture is conceptually identical to the remote method invocation architecture outlined in section 2.2.4. The proxy is responsible for serializing the arguments of the method invocation, locating the connection to the node managing the object and transmitting the request to the receiving node. At this point in time, it is not certain whether we have a connection to the node. If this is not the case we need to establish a connection prior to transmitting the invocation. At the receiving node, the invocation is received and the name service determines if the object is local to the node or not. There is a possibility that the object has been migrated again, in which case, the name service must transmit a message back to the sending node with the new location information. We assume, in this scenario, that the object is located on this node. We can see how the call is redirected to a skeleton by the name service. This skeleton deserializes the arguments and invokes the method for the object it manages. By following the blue arrows, we can see how the result from the method invocation is propagated back to the caller. First, we see that the result goes back to the skeleton that serializes the return value and that the skeleton sends a message back to the calling proxy. The calling proxy deserializes the result and returns the corresponding result to the caller.

From these discussions, we can see that there are two return values for each remote method invocation, one from the invoked object and one for the middleware itself. The return value from the middleware can be any one of a number of exceptional states. First, it can be a message that the object has migrated again, in which case the new location of the object needs to be transmitted to the calling node. We also need to ensure that the caller and callee both are running the same versions of the libraries, and as such we have added the version number of the remote method that we are expecting to call. When the receiving application receives the message containing the remote invocation it will make sure that the expected version of the method matches the one the application is using before invoking it. If it is not a match, the middleware will abort the invocation and an error message stating that the wrong version number was sent will be transmitted to the calling node. There is also the possibility that the caller is attempting to invoke a method that does not exist. In this case, a message to such effect is returned. If the call passes all these tests, the method is invoked. If the method is called successfully, the corresponding return value is handled and returned to the calling node.

In addition to these exceptional states, there are additional considerations to make. We need to consider the scenario where the node managing the object has crashed. It is also desirable to implement caching, to improve the speed of lookups. We do not address these topics as a part of the design, as they are beyond the scope of this thesis. We do, however, realize the importance of these facilities, and as such, we will discuss these topics further in chapter 5.

3.5 Partitioning and Grouping

In order to make the most of the functionality that we have discussed in the preceding sections, we consider the partitioning of the applications namespace into groups, based on communication patterns, to be of utmost importance for the efficiency of the developed system. There are a number of applications where the interaction between entities is dependent on a subset of the name space. This is particularly the case for MMOGs, where the user interaction is highly dependent on an area of interest. A suitable analogy is to how we, as humans, perceive only a small part of the physical world. Since there are a number of situations where this is the case, we can also reasonably assume that once these groups of interacting objects have been formed, the groups can be placed on different nodes in the distributed system. Initially the groups can be distributed to share the load evenly across the nodes. Grouping coupled with migration also makes it possible to perform load-balancing of various types. Grouping objects that interact also means that you are lowering the potential traffic over the network. Invoking a method on a remote object will typically induce a considerable factor of added time in the processing of that invocation. By grouping objects that communicate with each other, we are able to lower the number of (long distance) remote method invocations considerably and thus the latency. Another benefit of grouping is related to concurrency, it is much easier to synchronize the interaction between objects on a local node than it is to synchronize the events between objects located at different nodes. There are additional benefits to partitioning and grouping, though these are unrelated to the design of the middleware. As such, we will discuss the possibilities introduced by grouping further in chapter 5.

To end this chapter on our design we will first discuss our choice of serialization mechanism followed by a discussion about our code generator.

3.6 Serialization

There are three primary concerns that must be taken into account when selecting a mechanism for serializing our data. First, it should be a format that is easy to implement or already is highly available, since there is a high probability that nodes in the system will be heterogeneous. Second, it should be a compact and quick format, because we want to minimize the amount of resources used. Finally, the format should be portable, by this we mean that there should be no problem in transmitting data across networks using the format. We have decided to use the external data representation (XDR) as it meets these requirements. This is based on the discussion from section 2.2.7. The XDR format, as described in RFC 1832 [35] and RFC 1014 [36], is a standard for the description and encoding of data. It is useful for transferring data between different computer architectures. We have decided to use this format, because it is highly available, and comes as default on most Linux distributions. XDR is based upon implicit typing, where the sender and receiver must agree on the order and type of all data. XDR thus makes use of symmetric data conversion, since both the client and server convert from/to a common representation. XDR routines are direction independent, the same routines are called to serialize and deserialize data.

XDR libraries are based on a stream paradigm. This implies an ordered stream of bytes without message boundaries. XDR streams can be attached to a file, pipe, socket or memory. Values are passed through filters before being pushed onto the stream. A stream can either decode or encode data. Encode means changing from the local format to the XDR format, decode means changing from the XDR format to the local format. A strength of XDR is how it is easy to describe complex data types by combining a set of lower level data types. This allows for flexibility, since it does not attempt to be everything at once. The XDR standard makes the following assumption: that bytes (or octets) are portable, where a byte is defined to be 8 bits of data. A given hardware device should encode the bytes onto the various media in such a way that other hardware devices may decode the bytes without loss of meaning. For example, the Ethernet standard suggests that bytes be encoded in "little-endian" style, or least significant bit first.

3.7 Code Generation

Code generation is commonly provided to the application developer for easily generating structures that integrate with the middleware. The goal of these structures is to hide the complexity introduced by the middleware. While most of the time it is impossible to make the distribution transparent, even though there are examples of this, such as we saw with Java RMI, most of the time it is only partially transparent.

In this version of the middleware, we only support a single language, as such our code generation tool does not form an integral part of the middleware, such as is the case with CORBA. We will be generating structures which fit into the middleware we have designed. This is primarily to aid the developer in the development process.

We provide the developer with an interface definition language, which in essence is C++ declarations expanded with additional syntax that we provide the developer with. The developer can decide if a class is capable of being migrated, and define which methods are supposed to be capable of being invoked remotely.

The code generation tool is basic at this point in time, but it is quite possible to expand on the semantics. One thought is to allow the developer to provide hints to support automatic grouping of objects.

3.8 Summary

In this chapter we have outlined the components of the middleware. We have seen how migration and grouping of objects have had an impact on the design of our name service and how we perform communication between objects using remote method invocation. There are several implications of choosing a design such as the one we have, many of which introduce issues that should be handled. While we have attempted to address as many of these as possible we have not been able to provision for each and every one. In chapter 5, we will discuss the implications of our design, and provide viable solutions for minimizing any negative effects the middleware might introduce.

First, however, we will see how this abstract idea has been translated into a functioning system. Thus, we next present our prototype implementation.

Chapter 4

Implementation

While the concepts introduced in the preceding chapter might be relatively simple to understand, it is not an easy task to bridge the gap between an abstract idea and concrete implementation. As a result, we have aimed at implementing a basic, functioning version of the middleware as a proof of concept. The functionality of the middleware consists of the fundamental components outlined in chapter 3. As such, it has a distributed name service and supports remote method invocation and migration. In the remainder of this chapter, we will describe in detail the implementation of the various components. First, we will shortly summarize the interaction between the primary components of the middleware, followed by a discussion on how we have implemented remote method invocation (RMI) and migration. To demonstrate the usability of the middleware we have developed a test application and explain a scenario where the application is used; we will cross-reference between the implementation discussion and explanation of the test application in order to solidify our understanding. Finally, at the end of this chapter we outline how we provide facilities for automatically generating code.

During these discussions we will refer to class names from the implementation. In this context, we would like to refer to the documentation of the source code that accompanies this document[1]. There are two versions of it, one as a pdf document and

[1]The source code is available for download at: http://paul.beskow.no/master/paulbb.middleware-source.tar and the documentation is available for download at:

one as a browsable web page. Additionally, it is possible to look directly at the source code, since the documentation was generated from the comments written in the source files using doxygen [37].

4.1 Programming Language

For developing the middleware, we choose to use the object-oriented programming language C++ [38], since it provides fine grained interaction with low level interfaces, such as sockets (for network programming). Thus, it allows the developers to have relatively good control of the environment they are programming in. Since C++ is a compiled language, with few built-in abstractions, such as dynamic memory management, its programs are generally quite fast. Naturally, this places more responsibility on the programmer, for coding correctly, but the trade-off can be well worth the effort. Finally, C++ is the most actively used programming language for developing computer games. This is relevant since massively multi-player online games have formed a focus for this thesis.

The use of C++ has had its impact on the implementation of the system, and throughout the discussions of this chapter we will be using terms from C++, such as templates and run-time type identification (RTTI), and terms from object-oriented programming, such as polymorphism and inheritance. We will introduce these concepts as they appear in the context of the implementation.

4.2 Primary Interaction

The middleware consists of many components that interact to provide the application with the necessary functionality to migrate objects and perform RMI. At the lowest level, we have the communication facilities that provide mechanisms for serializing and transmitting data across the network. In our middleware, we combine a wrapper class to C sockets [39], which are implemented using the transmission control protocol

http://paul.beskow.no/master/paulbb.middleware-documentation.tar

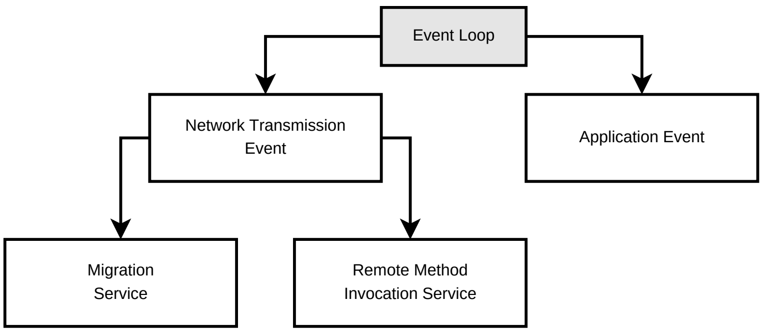


Figure 4.1: Middleware Event Loop

(TCP) [40] with a wrapper class to an external data representation (XDR) stream. We discuss the details of the XDR component further in section 4.3. TCP is a connection-oriented, reliable transmission protocol and is, for example, used actively in MMOGs. The connection-oriented nature implies that after a connection has been established, a link exists between the sender and receiver that can be used to transmit messages. This link will function even if the sender is sitting behind a network address translator (NAT) [41] or firewall [42]. In short, the combined wrapper classes make it simple to create and send messages between nodes. These messages consist of a header and body, where the header provides hints about the intended action of the messages, which, once received by the middleware, are handled as displayed in figure 4.1. The middleware is waiting for events from the network or application. If data arrives on the network, the message is read from the stream, and the first part of the header is consumed. The result of reading the header ends with redirecting the message to the migration or remote method invocation services. We will discuss these services further in section 4.4.

In addition to these components, the sending node requires an indirection mechanism that can determine if a call to an object is local or not and redirect accordingly. We discuss this functionality in section 4.5. For an object to support migration and RMI, it must be associated with a reference that can be understood by the name service. This is accomplished by associating an object identifier with the object. In our middleware, the class *Object* contains a pointer to an instance of the class *Objectid*. When an instance of an object that inherits *Object* is created, the default action is to have the *ObjectRe-*

gister generate an identifier for it. All objects that support migration and RMI need to inherit the class *Object*. We will not go into discussions about the registration process here, since it has little impact on the middleware itself, and is well documented in the source code. We will ,however, go through the actions performed in the middleware more closely when discussing our example application in section 4.6. First, we will describe our serialization mechanism. It is important to remember that the functionality described here is identical for every node in the system, including potential clients.

4.3 Serialization

While most modern programming languages support serialization natively, C++ does not. Thus, it needs to be implemented independently. Since there is no easy way to decouple the serialization mechanisms from the class implementation, they will become quite integrated. Following the discussion in section 3.6, we have decided to use the external data representation (XDR) format for serializing data. There are other options available for C++, with varying degrees of complexity. One example is the Boost C++ serialization libraries [43]. A problem with this library is that it is quite complex, since it is possible to serialize a broad range of C++ structures, many of which we do not need to support at this point in time. As such, XDR provides the level of detail we desire at this point in time. It is compatible with C++, in that it was originally written to be used with C, which is a subset of C++. XDR provides the level of control we need and minimizes overhead. Additionally, it is a highly portable format as it is only dependent on ANSI C. A downside with XDR is its readability when compared to formats such as XML-RPC [44], which are text based, the byte based XDR format is hard to interpret directly. With a loss of readability comes the benefit of a format that is more compact and easier to process.

To conceal the C nature of the XDR libraries, we have implemented the serialization mechanisms by creating a wrapper class called *XDRHandler*. This class creates a buffer of 4096 bytes and creates an XDR memory stream that points to its location. With XDR one can serialize data to an area in memory, such as we do, or directly to a file descriptor; this file descriptor can refer to a file, standard in or out, a socket and more. The direction of the stream can be changed depending on the context, the possible actions are to read, write and free data from the stream. We have implemented methods to modify the behavior of the XDR stream, this includes methods for resetting the stream and clearing the buffer. Since messages in our system will be sent to remote nodes in the system we associate an XDR stream with a file descriptor that references a socket. This socket is used to move data to or from the network into or out of the buffer. An instance of *XDRHandler* is passed around in the system and used to serialize and deserialize data to and from. For an exhaustive overview of the functionality available from the XDR libraries, see Sun's developer's guide [45].

4.4 Object model

An object model defines the structural relationships and interaction between a group of related objects. In figure 4.2, we see an illustration of the object model that integrates with the middleware and provides a class of objects with the ability to be migrated and accept remote method invocations. While it is relatively difficult to decouple the functionality for migration and remote method invocation from the class itself, we have tried to minimize the integration as much as possible. From the figure, we can see that *Class* inherits the functionality from three parent classes, as such *Class* is said to be derived from these classes. In addition, *Class* has a child class *ClassRPC*, that inherits the methods and data from its parent (together with the functionality inherited by that classes parents). *Class* is among other derived from *Object*. This means, as we have mentioned earlier, that any instance of *Class* has an object identifier associated with it. This identifier can be used to reference an object independently of its location in the system. For a class to support migration or remote method invocation, it needs to inherit *Object*, since it in effect means that objects of that class are managed by the name service on that node. This makes it possible for the middleware to move these objects from node to node and redirect remote method invocations to the correct instance of an object. *Object* does not implement this functionality all on its own, but there are also additional semantics that need to be in place. To further solidify our understanding of the implemented migration and RMI components, we will in section 4.6 refer to the discussions that follow in this section, in view of a test application that we created to demonstrate the usability of the middleware. First, we will describe how migration is implemented in our middleware, after which we will discuss the implementation of RMI.

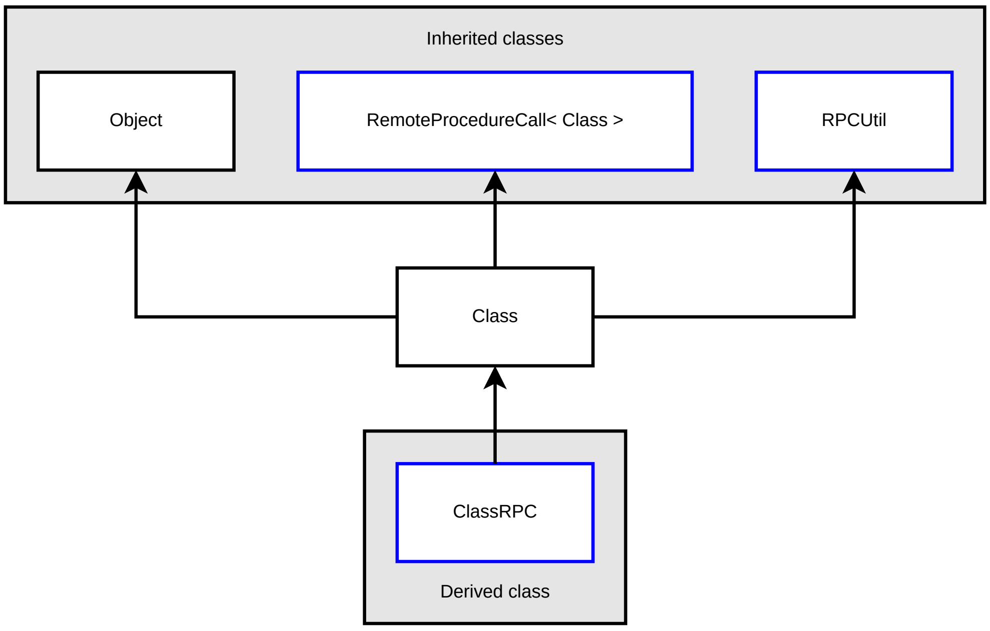
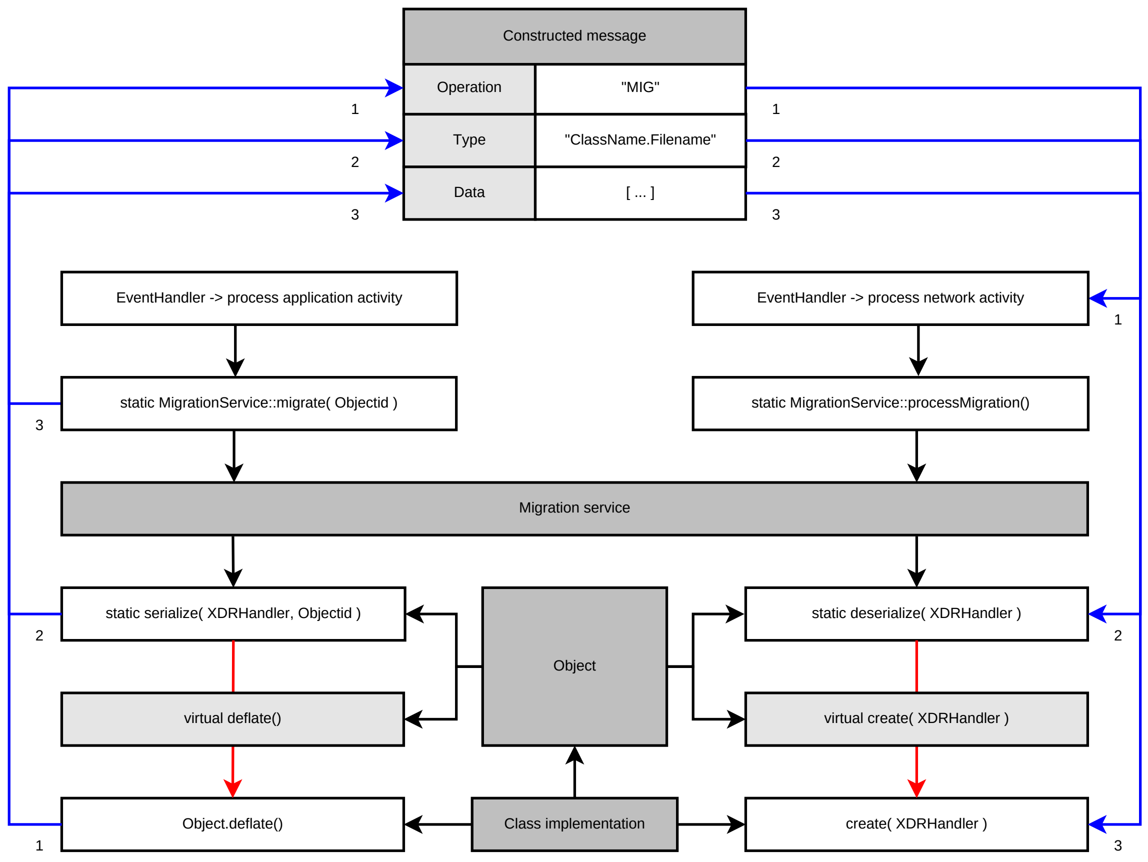


Figure 4.2: Object Model Inheritance Diagram

4.4.1 Migration

In order for an object to be migrated, it needs to be managed by the name service, so that references to the object can be maintained even after the object has been moved. In addition, the object must be serializable so that its data can be added to a message that will be interpretable by the receiving node. Both these tasks are supported by inheriting the abstract class called *Object*. An abstract class in C++ means that instances of *Object* cannot be created directly, but that any class must inherit the interface and functionality defined by it. As we have already mentioned, any class that inherits *Object* will, at creation time, have an object identifier generated for it. For a class to be abstract in C++, it must define a pure virtual function. This is a function that any derived class must provide an implementation for. Additionally, when defining a method as virtual



Constructed message
Operation
"MIG"
Type
"ClassName.Filename"
Data
[...]
1
2
3
EventHandler -> process application activity
EventHandler -> process network activity
static MigrationService::migrate(Objectid)
static MigrationService::processMigration()
Migration service
static serialize(XDRHandler, Objectid)
Object
static deserialize(XDRHandler)
virtual deflate()
virtual create(XDRHandler)
Object.deflate()
Class implementation
create(XDRHandler)

Figure 4.3: Object Model Migration

in C++, it is implied that classes derived from that class are polymorphic. Polymorphism is used to dynamically redirect the call to a method based on the class that the pointer is an instance of.

We can follow this discussion by looking at figure 4.3. The class *Object* defines two static methods, *serialize(...)* and *deserialize(...)*, and two pure virtual functions, *deflate(...)* and *create(...)* that are used to support migration. Static member class methods in C++ can be called without requiring an instance of the class to be available, while pure virtual functions mean that the derived class must provide an implementation. As we can see from the figure, *deserialize(...)* and *serialize(...)* are used by the migration service to perform and process migrations. When a migration is being performed, we can see that the migration service adds a header that identifies the message to contain data about a migration. This is followed by a call to the method *serialize(...)*, as we can see this method adds the type name of the class to the message.

It is necessary to add the type name so that when *deserialize(...)* is called at the receiving end, we are creating an instance of the correct type of object. To accommodate for the type name, we have a type register that consists of a mapping between a string representation of the type name and an instance of the corresponding class for all serializable classes. The type name of a class is determined by the code generator and is a combination of the filename and class name. It is done this way because RTTI type names are not portable. RTTI makes it possible to identify the type of object a generic pointer is referencing during run-time. After adding the type name, *serialize(...)* calls the pure virtual function *deflate(...)*, that all derived classes from *Object* need to provide an implementation for. As such, the *serialize(...)* method can assume that the implementation of *deflate(...)* is provided by the object being serialized and that it is capable of adding its data to the message. At this point, the message is ready to be transmitted to the receiving node. The composition of the resulting message can be seen in figure 4.3.

This message is then transmitted to the receiving node, and read into an *XDRHandler* instance. First, the middleware determines the type of action associated with the message, and it reads it to be a migration and passes the message on to the migration service. The migration service calls the *deserialize(...)* function that reads the type name of the class from the message. The type register is queried for an instance of the class corresponding to the type name. In turn, the *create(...)* function for that class of objects is called and an object is constructed and initialized with the data read from the message. Once the object is created, an object identifier is generated for it by the object register, because, as we have mentioned, all classes that inherit *Object* automatically have object identifiers generated for them during their construction. This object identifier is returned to the calling node, and is registered with the object register as a care-of address for the migrated object. At this point in time, the object at the sending side can be deleted, since it now is safely placed at its new location. The process for returning the *objectid* is identical to that explained here, just that the header and type name are not added. Only the data for the object identifier is serialized and returned to the sender. This is possible because the sender is waiting to receive this identifier in return. This concludes the process for migrating objects, and we will now look at how remote method invocation is implemented in our system.

4.4.2 Remote Method Invocation

Remote method invocation (RMI) is quite a bit more complicated to implement then migration. The classes with a blue border from figure 4.2 provide part of the necessary interaction for RMI to function correctly. When a call to a remote object is made, it will be intercepted and redirected to a proxy that is implemented by the class *ClassRPC*. This class will ensure that a message is constructed and sent to the receiving node. At the receiving side, the message is interpreted and dissected by the skeleton, which acts as an intermediate for remote invocations. The skeleton is implemented partially by *RemoteProcedureCall<Class>* and partially by virtual functions defined in *Object* that *Class* needs to implement. Finally, there is the supporting class *RPCUtil* that provides functionality to build a header for the message and that creates and handles return values generated by the middleware. We can follow the remainder of this discussion by looking at figure 4.4.

We will start this discussion by looking at how the proxy for a class of objects in our system is implemented; this is accomplished by *ClassRPC* deriving from *Class*. The reason *ClassRPC* inherits *Class* is so that the interception mechanism can call the method of either the local or proxy implementation, depending on the current location of the object. We will discuss this interception mechanism further in section 4.5. Any method that supports RMI is virtual in the base class *Class*. In this way, the derived RMI implementation of the class can provide an implementation of the method that is capable of constructing a message that will invoke the method at the receiving node. The reason for implementing the RMI handling at the calling side in this way is that C++ is a strongly typed language. This means that if you have a pointer to an object, it is not possible to change the type of that pointer at a later point in time, since it is determined at compile time what the type of a variable is. There is, however, a slight variation to this truth when polymorphism is involved. In this case, it is possible to have a single pointer of a base class type, and by using RTTI, one can have this pointer

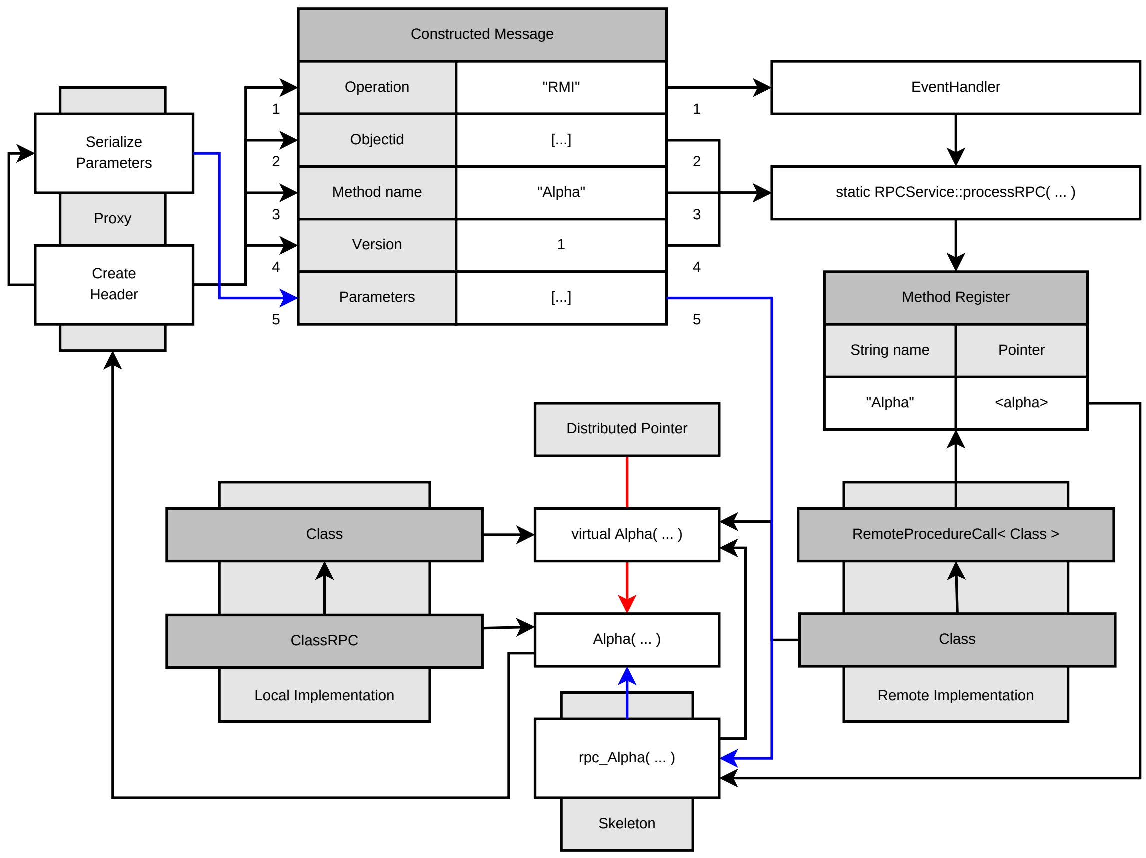


Figure 4.4: Implementation of Remote Method Invocation

refer to any object that is an instance of a class derived from the base class.

At this point in time, the call to the method has been redirected to the proxy implementation of the derived class *ClassRPC*. The proxy first adds a header, that describes the type of event the message contains; this together with the name and version of the method that we are calling. Finally, the parameters of the method are added to the constructed message. The constructed message looks like that depicted in 4.2. At the receiving end the message is processed by the event handler that determines what type of event it is, in this case a remote method invocation. The message is forwarded to the RMI service that reads the object identifier from the message. The RMI service queries the object register to obtain a local reference to the object in question. Once this is accomplished it calls a method that can locate and invoke methods based on the string representation of the name. The name and version of the method are here checked to ensure a match. The version number, as we discussed earlier, together with the method name, is necessary to guarantee type safety across network boundaries. If an older version of the method is being called, the semantics of the method might have changed, and result in unexpected behavior. This mapping of method name to implementation is performed during construction of the object, and is implemented by the template class *RemoteProcedureCall< Class >*. This class could be embedded with each class of objects that perform RMI, but we wish to provide as little coupling between the implementation of RMI and the object itself. *RemoteProcedureCall< Class >* is implemented as a template because C++ function pointers must know what type of class they reference at compile time, even though the instances share the same base class. As such, we can use templates, because they allow code to be written without consideration of the data type with which it will eventually be used. Once the corresponding method has been located, it is invoked, and the remainder of the message is passed to it as a parameter. The method that is invoked does not contain the implementation, but acts as the skeleton. It handles the deserialization of the parameters and invokes the corresponding implementation of the method, which we have referred to as the servant earlier. The return value from the invoked servant is serialized by the skeleton and passed directly to the proxy, and is in turn returned to the caller. The proxy and skeleton also add and process return values generated by the middleware itself. This concludes the discussion for our implementation of remote method invocation. We will now look at the distributed pointer.

4.5 Distributed Pointer

The purpose of the distributed pointer is to determine whether an object exists in local memory or not and then redirect a call either to the local instance of the object or to a proxy for the object when a remote method invocation is being performed. To accomplish this in C++, the distributed pointer is implemented as a template and additionally makes use of polymorphism. The class being called needs to have a derived class that can function as a proxy, as we have discussed earlier. We use these two concepts because C++ allows for pointers to be dynamically substituted between classes that are polymorphic (classes that are derived from, or are instances of, the same base class). In addition, we want this distributed pointer to function with any type of class, which is why we use templates that allow us to program independently of type. As a result, we end up with the flow as depicted in figure 4.5.

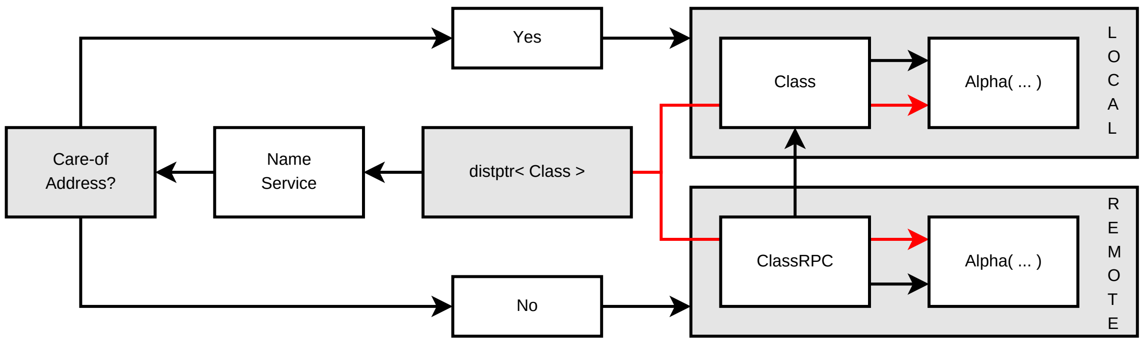


Figure 4.5: Distributed Pointer

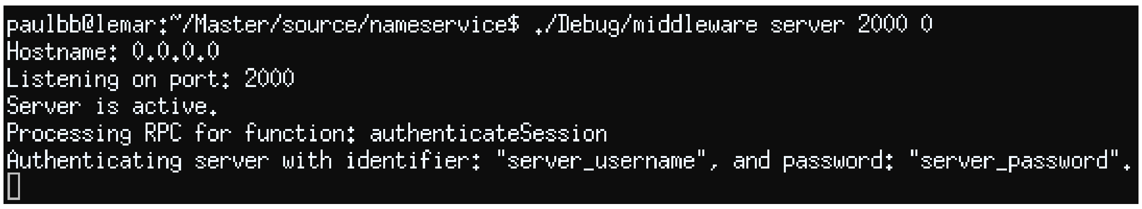
The distributed pointer is associated with a class type and the *objectid* of the instance. It is important to remember that even though access to the object can be performed directly, this should be avoided. Since there is no way of determining whether the object referenced by an ordinary pointer has been migrated or not, we have overloaded the pointer operator for our distributed pointer, and as such, the implementation will query the *ObjectRegister* and ask if the object has a care-of address associated with it. If it has a care-of address, a static method is called that returns a pointer to the instance of the proxy for that class of object. This results in the method implemented by the proxy to be called. If the object is local, the implementation of the method is called directly.

4.6 Example Application

To demonstrate the usability of the middleware, we have implemented a test application. The application is modeled as a chat system in a MMOG, where the clients are able to create and join channels of their choice. The application has three main components, the primary server, secondary server and clients. The primary server functions as the initializing node, which is necessary for the activity between the servers to be synchronized. Initially, all connections by clients and secondary servers need to be established with the primary server. To make the process of explaining how the application functions easier, we will follow an example of its usage. We can start by looking at the output from the primary server (figure 4.6) and the secondary server (figure 4.7). We can see that the secondary server establishes a connection with the primary server. At this point in time, the secondary server receives a migration from the primary server, in the form of an instance of the class *Authentication*.

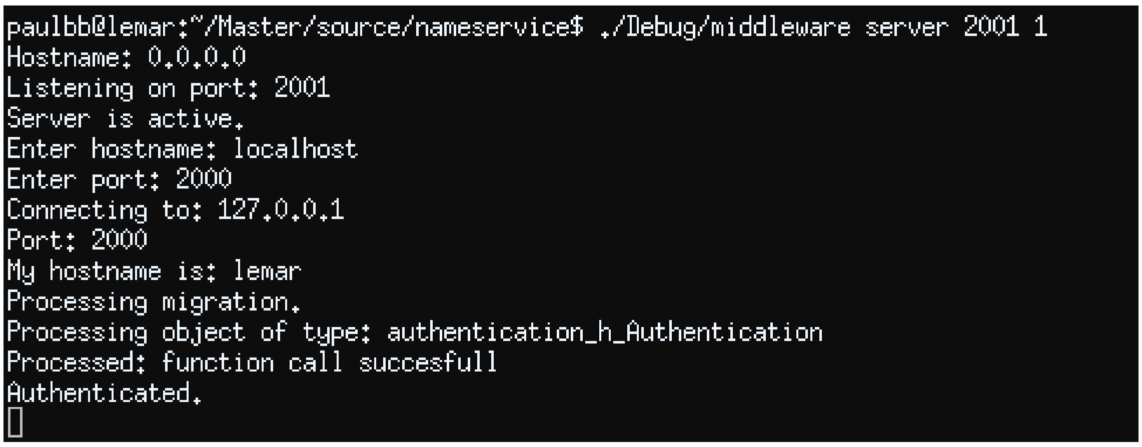
Looking at figure 4.3, which was explained in detail in section 4.4.1, we can follow the process involved in migration an object. First, we see that the *EventHandler* receives a request to migrate the object, which it forwards to the *MigrationService*. The migration service queries the name service for a local reference, to determine whether the object is currently located on this node or not. Once it determines that the object is local, it begins construction of the message. First it adds the operation identifier, in this case *MIG*. This is followed by the type name of the object. Each object has a method *getTypeName(...)* that returns the identifier for the object. In this case, as we can see from figure 4.7 that the identifier is *authentication_H_Authentication*. Finally, the *deflate(...)* method is called for the object which results in the data associated with the object being added to the message. At this point in time, the constructed message is transmitted to the receiving node. The message is identified as a migration and is forwarded to the migration service. The migration service reads the type name from the message and queries the type register for an instance of such an object. The *create(...)* method constructs a new object of its type and initializes it with the data stored in the message and finally returns a pointer to the newly created object. The migration service uses the pointer to request the *objectid* of the newly created object, which it returns to the sending node.

The *Authentication* class is used by the connecting party to identify as a client or server and provide a username and password for authentication. This is accomplished by a remote method invocation to the method *authenticate(...)*. It is worth noting that in this case, the primary server, after migrating the instance of *Authentication* to the secondary server, does not register the care-of address it receives in return from the secondary server as we normally would expect it to do. This makes sense because all we want is for the connecting party to establish a means of communication with the primary server, which means that we need to have a remote object that we can communicate with. As such, *Authentication* is implemented as a singleton, which means each node will never have more than one instance of it running, and as such all method invocations will be handled as remote. We can now see that the secondary server automatically performs a remote method invocation that identifies it as a server, and passes its designated username and password. We can see that two return values are received, one that indicates a successful completion of the invocation, this is a middleware return value. The second return value indicates that the authentication was successfully, this is a return value from the method invocation.



```
paulbb@lemar:~/Master/source/nameservice$ ./Debug/middleware server 2000 0
Hostname: 0.0.0.0
Listening on port: 2000
Server is active.
Processing RPC for function: authenticateSession
Authenticating server with identifier: "server_username", and password: "server_password".
```

Figure 4.6: Example Application: Output from primary server, secondary server connecting

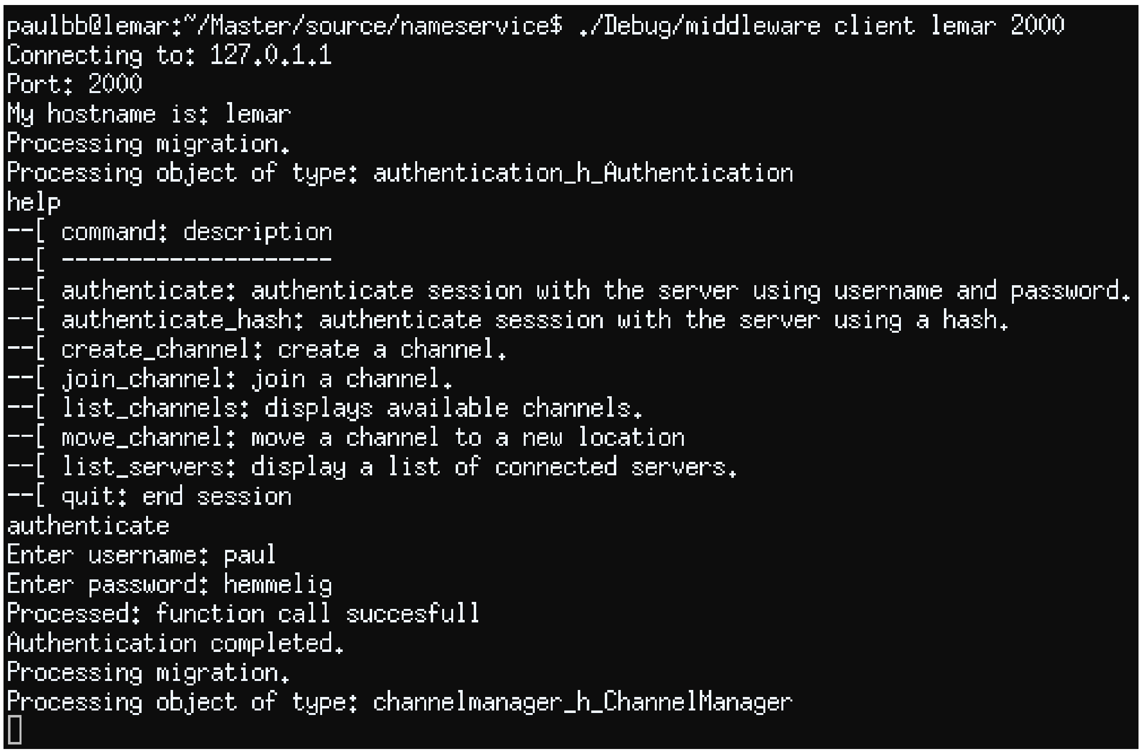


```
paulbb@lemar:~/Master/source/nameservice$ ./Debug/middleware server 2001 1
Hostname: 0.0.0.0
Listening on port: 2001
Server is active.
Enter hostname: localhost
Enter port: 2000
Connecting to: 127.0.0.1
Port: 2000
My hostname is: lemar
Processing migration.
Processing object of type: authentication_h_Authentication
Processed: function call succesfull
Authenticated.
```

Figure 4.7: Example Application: Output from secondary server, connecting to primary server

To further solidify our understanding of how remote method invocations are implemented in our middleware we can consider figure 4.4, which was explained in detail in section 4.4.2. Following this figure, we see that in our example application we have a distributed pointer to the *Authentication* instance. In this case, this pointer has determined, after querying the name service, that our call to *authenticate(...)* needs to be redirected to the proxy implementation of the method. This proxy is implemented by a class *AuthenticationRPC* that inherits from *Authentication*. Since the method *authenticate(...)* is implemented as a virtual function in *Authentication* and *AuthenticationRPC* provides its own implementation the call will be made to the derived version. This derived implementation starts the construction of a message, since the invocation is remote. First it adds the type of operation, which is *'RMI'*. This is followed by the *objectid* of the object we are requesting and the method name (*"authenticate"*) and version (*1*). Finally, we add the parameters that accompany the method, if any. In this case, there are two, namely the username and password. At this point in time, the message is transmitted to the receiving node. There it is identified by the event handler as a remote method invocation and is forwarded to the *RPCService*. This service first reads the *objectid* and determines whether the object is local or not, then reads the method name and version and determines if the object will allow the invocation. If the invocation is legal the call will be forwarded to a skeleton that can handle the request, in this case the request is forwarded to the method *rpc_authenticate(...)* in the class *Authentication*. This method, which functions as a skeleton, deserializes the parameters, calls the local version of *authenticate(...)* and serializes the return value from the invocation. The return value is in turn returned to the requesting node.

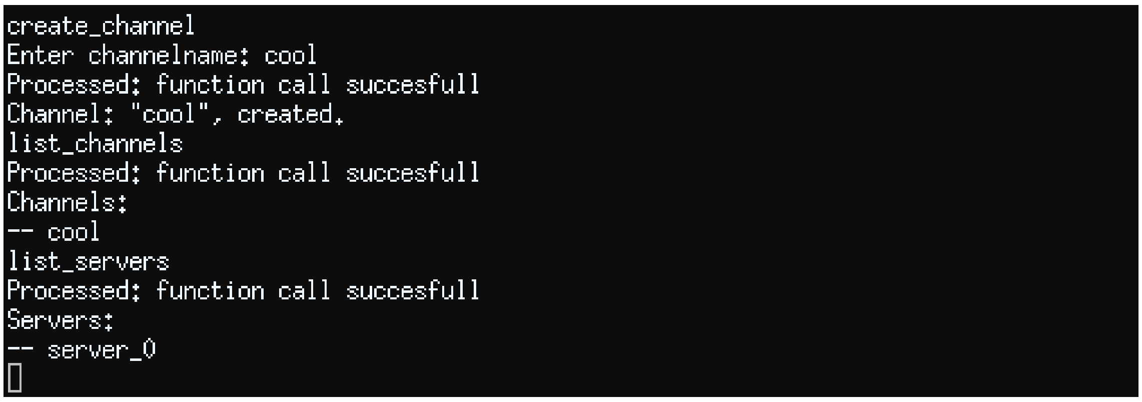
The client, which we refer to as the primary client, now connects to the primary server and the output from this action can be seen in figure 4.8. We can see that the client also receives the instance of *Authentication*; we have issued the help command at this point in time, to display the actions available to the client. We can see that the authenticate method is among these, and as we can see the client performs an authentication. Once again, we can see that the authentication is successful, and unlike with the secondary server, an additional action occurs where a *ChannelManager* instance is migrated to the client. This class is also implemented as a singleton and functions the same way as *Authentication*. After we have received the *ChannelManager* we are able to create, move and list channels. An unauthorized client is not able to invoke remote methods other than authenticate. In figure 4.9, we can see the primary client create a channel *cool* and display a list of servers and channels. The activity on the server can be seen in figure 4.10. From this figure we can also see that a secondary client has connected.



```
paulbb@lemar:~/Master/source/nameservice$ ./Debug/middleware client lemar 2000
Connecting to: 127.0.1.1
Port: 2000
My hostname is: lemar
Processing migration.
Processing object of type: authentication_h_Authentication
help
--[ command: description
--[ --------------------
--[ authenticate: authenticate session with the server using username and password.
--[ authenticate_hash: authenticate sesssion with the server using a hash.
--[ create_channel: create a channel.
--[ join_channel: join a channel.
--[ list_channels: displays available channels.
--[ move_channel: move a channel to a new location
--[ list_servers: display a list of connected servers.
--[ quit: end session
authenticate
Enter username: paul
Enter password: hemmelig
Processed: function call succesfull
Authentication completed.
Processing migration.
Processing object of type: channelmanager_h_ChannelManager
▯
```

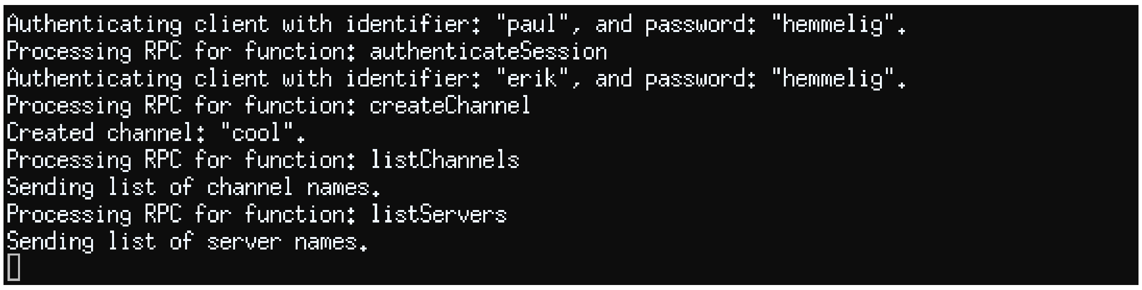
Figure 4.8: Example Application: Output from primary client, connecting to primary server

We can in figure 4.11, 4.12 and 4.13 see how the clients join a channel and send messages to each other. The channels form the communication groups in the system, and the clients are grouped based on their participation in channels. In this case both the clients have joined the channel *cool*. This means that both clients have a distributed pointer that is associated with the *objectid* of that channel. We can also see that the primary server is accepting remote method invocations for the method *sendMessage(...)*. This is a method associated with the *Channel* object, and when invoked, it will



```
create_channel
Enter channelname: cool
Processed: function call succesfull
Channel: "cool", created.
list_channels
Processed: function call succesfull
Channels:
-- cool
list_servers
Processed: function call succesfull
Servers:
-- server_0
```

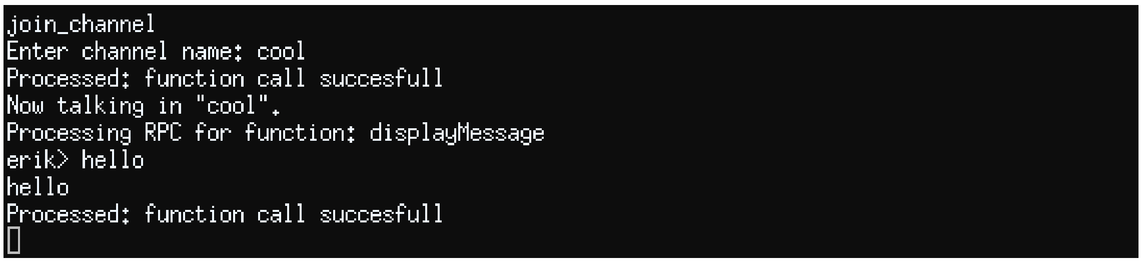
Figure 4.9: Example Application: Output from primary client, sending commands



```
Authenticating client with identifier: "paul", and password: "hemmelig".
Processing RPC for function: authenticateSession
Authenticating client with identifier: "erik", and password: "hemmelig".
Processing RPC for function: createChannel
Created channel: "cool".
Processing RPC for function: listChannels
Sending list of channel names.
Processing RPC for function: listServers
Sending list of server names.
```

Figure 4.10: Example Application: Output from primary server, received commands

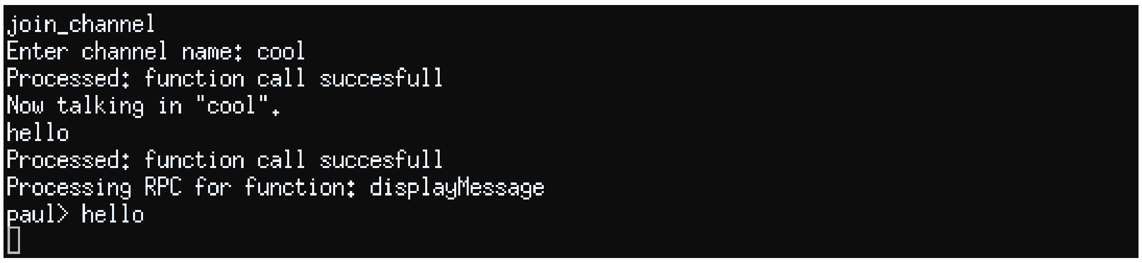
send a message to all clients associated with it. As we can see, this is accomplished by calling the method *displayMessage(...)* that is a method associated with the *ChannelAction* object. Each client has an instance of such an object and passes it as an argument to the *ChannelManager* when requesting to join a channel.



```
join_channel
Enter channel name: cool
Processed: function call succesfull
Now talking in "cool".
Processing RPC for function: displayMessage
erik> hello
hello
Processed: function call succesfull
```

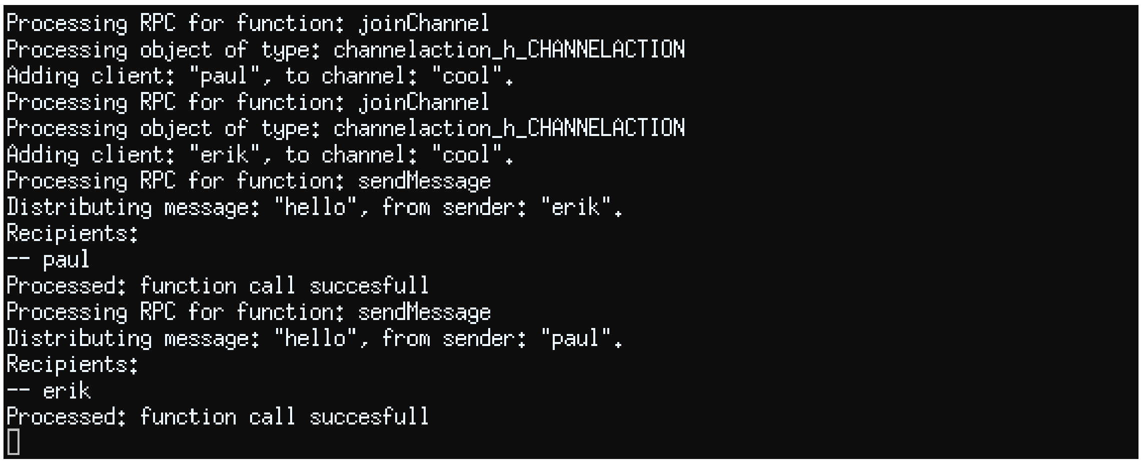
Figure 4.11: Example Application: Output from primary client, channel actions

Finally, we will see how migrating the channel is performed, and how it affects the servers. We can follow the remainder of this discussion by looking at the figures for the primary server (4.16), secondary server (4.17), primary client (4.16) and secondary



```
join_channel
Enter channel name: cool
Processed: function call succesfull
Now talking in "cool".
hello
Processed: function call succesfull
Processing RPC for function: displayMessage
paul> hello
```

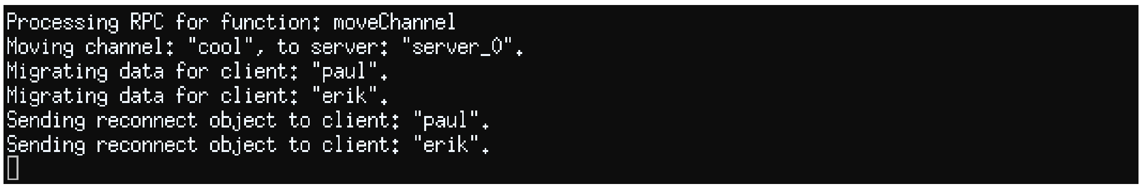
Figure 4.12: Example Application: Output from secondary client, channel actions



```
Processing RPC for function: joinChannel
Processing object of type: channelaction_h_CHANNELACTION
Adding client: "paul", to channel: "cool".
Processing RPC for function: joinChannel
Processing object of type: channelaction_h_CHANNELACTION
Adding client: "erik", to channel: "cool".
Processing RPC for function: sendMessage
Distributing message: "hello", from sender: "erik".
Recipients:
-- paul
Processed: function call succesfull
Processing RPC for function: sendMessage
Distributing message: "hello", from sender: "paul".
Recipients:
-- erik
Processed: function call succesfull
```

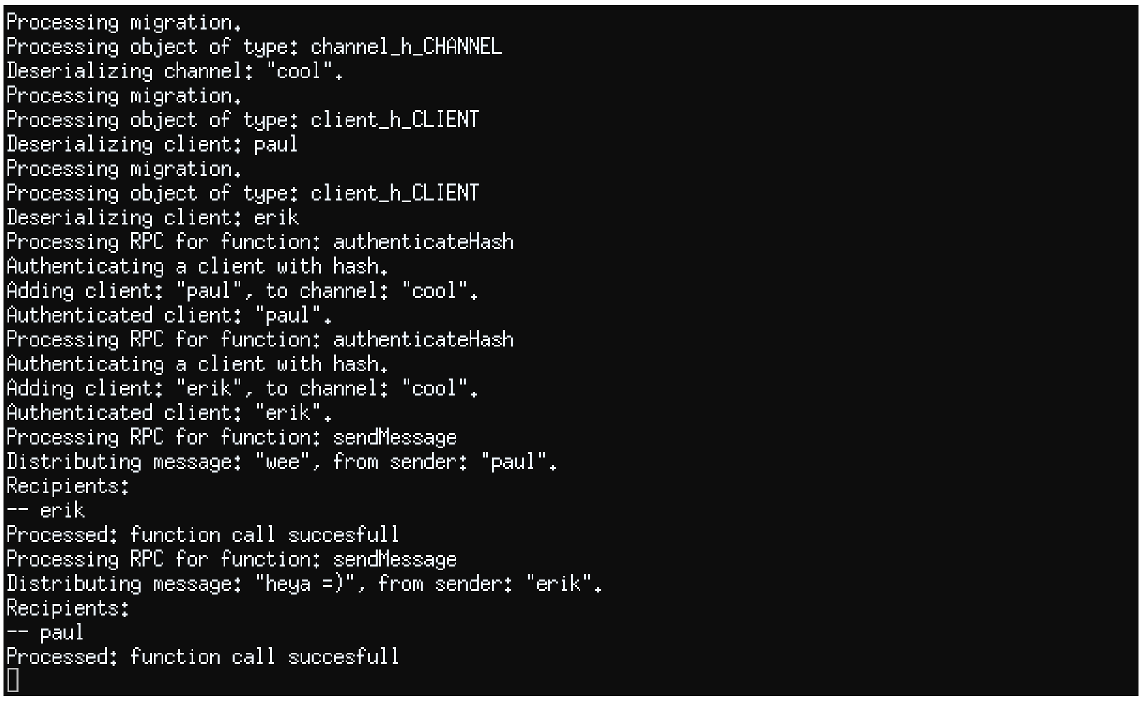
Figure 4.13: Example Application: Output from primary server, channel actions

client (4.17). We can see that the primary server receives the move command and migrates the channel *cool* to the secondary server. This is then followed by migrating the data for the connected clients. Finally, we migrate reconnect objects to the clients, that contain the hostname and port of the server they should reconnect to. When this object is received by the client and deserialized, it registers itself with an event queue. Once the reconnect event is handled the client will connect to the new server. At this point, the client authenticates with the new server. Earlier, when the client connected to the server it received a hash value generated by the server. Now, when reconnecting, it can authenticate using this hash value, instead of supplying a username and password. At the secondary server, we can see that the user, after reconnecting, is associated with the channel cool on new. This is accomplished because the client data includes channel associations. Finally, we can see that the distribution of messages now is performed at the secondary server, and the primary server is offloaded. This concludes the test application, which has shown that migration is feasible, and the distribute name service functions as it should.



```
Processing RPC for function: moveChannel
Moving channel: "cool", to server: "server_0".
Migrating data for client: "paul".
Migrating data for client: "erik".
Sending reconnect object to client: "paul".
Sending reconnect object to client: "erik".
```

Figure 4.14: Example Application: Output from primary server, migrating channel



```
Processing migration.
Processing object of type: channel_h_CHANNEL
Deserializing channel: "cool".
Processing migration.
Processing object of type: client_h_CLIENT
Deserializing client: paul
Processing migration.
Processing object of type: client_h_CLIENT
Deserializing client: erik
Processing RPC for function: authenticateHash
Authenticating a client with hash.
Adding client: "paul", to channel: "cool".
Authenticated client: "paul".
Processing RPC for function: authenticateHash
Authenticating a client with hash.
Adding client: "erik", to channel: "cool".
Authenticated client: "erik".
Processing RPC for function: sendMessage
Distributing message: "wee", from sender: "paul".
Recipients:
-- erik
Processed: function call succesfull
Processing RPC for function: sendMessage
Distributing message: "heya =)", from sender: "erik".
Recipients:
-- paul
Processed: function call succesfull
```

Figure 4.15: Example Application: Output from secondary server, migrating channel

4.7 Code generation

We provide the developer with a code generation tool to help create the structures described in section 4.4. The code generator is implemented as a python script and makes extensive use of the pygccxml [46] module. This is a python language binding module for the GCC-XML [47] parser, which is a tool that extends the open source GCC compiler [48], using its internal representation of a C++ source file to produce XML output. The generator expects a C++ class declaration as its input, as illustrated

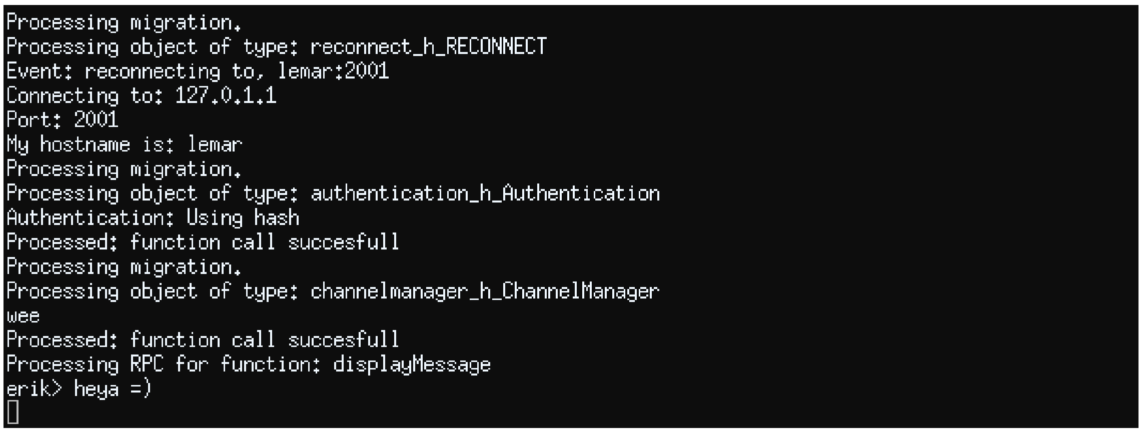

```
Processing migration.
Processing object of type: reconnect_h_RECONNECT
Event: reconnecting to, lemar:2001
Connecting to: 127.0.1.1
Port: 2001
My hostname is: lemar
Processing migration.
Processing object of type: authentication_h_Authentication
Authentication: Using hash
Processed: function call succesfull
Processing migration.
Processing object of type: channelmanager_h_ChannelManager
wee
Processed: function call succesfull
Processing RPC for function: displayMessage
erik> heya =)
```

Figure 4.16: Example Application: Output from primary client, migrating channel

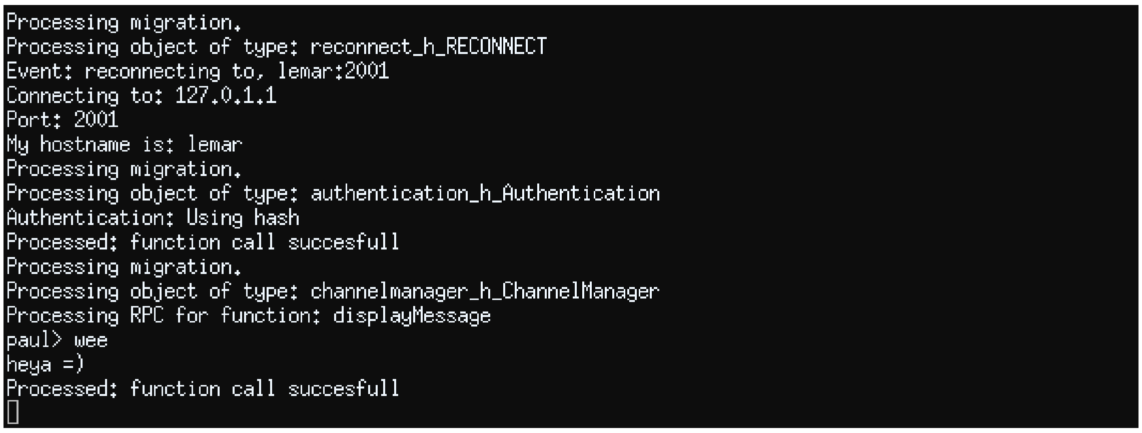
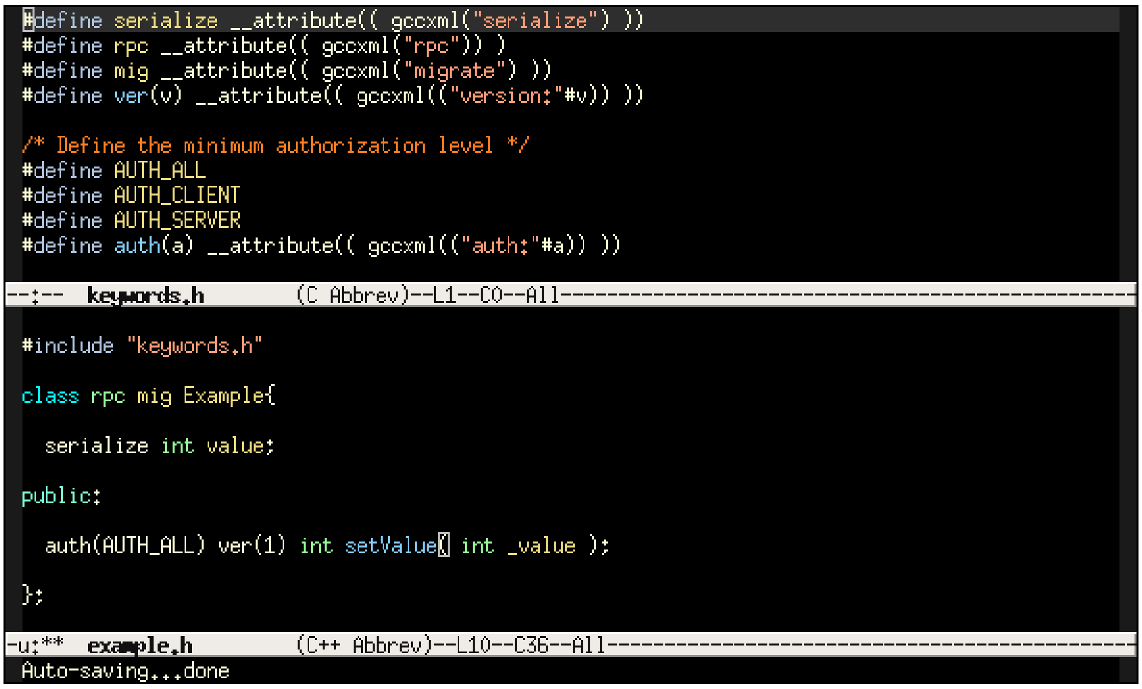

```
Processing migration.
Processing object of type: reconnect_h_RECONNECT
Event: reconnecting to, lemar:2001
Connecting to: 127.0.1.1
Port: 2001
My hostname is: lemar
Processing migration.
Processing object of type: authentication_h_Authentication
Authentication: Using hash
Processed: function call succesfull
Processing migration.
Processing object of type: channelmanager_h_ChannelManager
Processing RPC for function: displayMessage
paul> wee
heya =)
Processed: function call succesfull
```

Figure 4.17: Example Application: Output from secondary client, migrating channel

in figure 4.18. In addition to normal C++ class syntax, GCC-XML allows for defining additional attributes, and this was not supported natively by the pygccxml module. We have therefore extended it with functionality to handle this. We make use of this ability to extend the C++ syntax with our own keywords. We can see how this is accomplished by looking at figure 4.18. As we can see, the class *Example* is marked as a class that should support migration and remote method invocation. The variable *value* has been marked with the attribute *serialize*, which implies that during the generation process structures for serializing, *value* should be generated. Variables that are not marked with this keyword will not have structures for serialization generated for them, and as such, will not be transmitted to the receiving node. This makes sense, because not all the data associated with an object is necessarily of interest after migration has been performed. In addition, we have the method *setValue(...)* that assigns *value* with a new value. This method is implemented as a remote method invocation because it is public. We have made the assumption that all public methods should support remote invocation, since there is a probability that they might be called at a point in time. The *auth(AUTH_ALL)* is implemented because it is useful for the application and is not so much a part of the middleware. It provides a simple way of determining whether a server or client has access to a given method. The version number requires no further explanation.



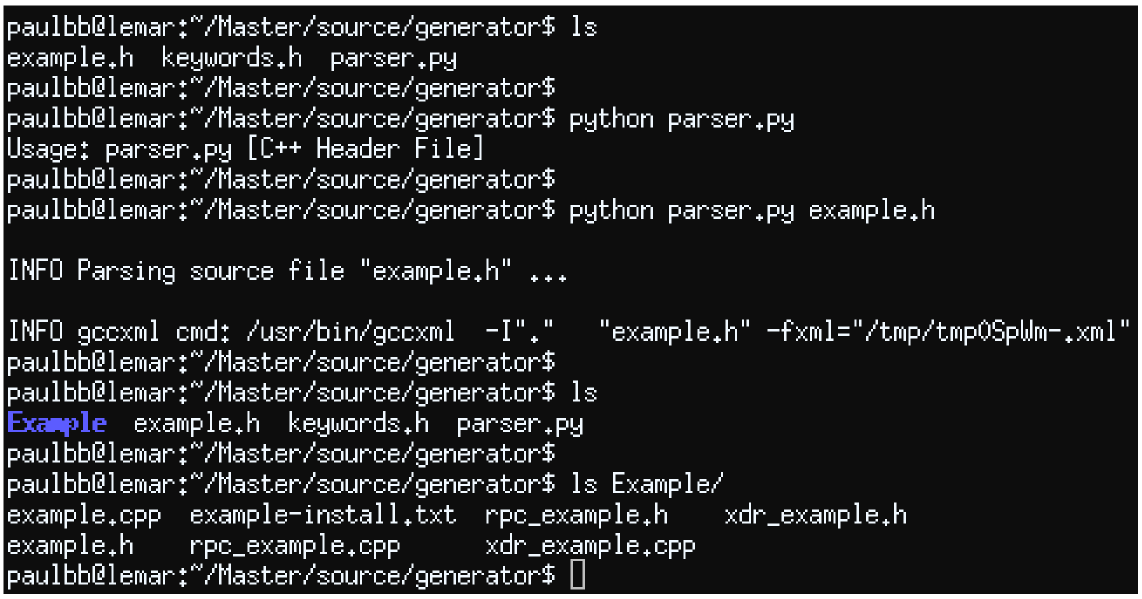
```
#define serialize __attribute(( gccxml("serialize") ))
#define rpc __attribute(( gccxml("rpc")) )
#define mig __attribute(( gccxml("migrate") ))
#define ver(v) __attribute(( gccxml(("version:"#v)) ))

/* Define the minimum authorization level */
#define AUTH_ALL
#define AUTH_CLIENT
#define AUTH_SERVER
#define auth(a) __attribute(( gccxml(("auth:"#a)) ))

--:--  keywords.h        (C Abbrev)--L1--C0--All------------------------------------------

#include "keywords.h"

class rpc mig Example{

  serialize int value;

public:

  auth(AUTH_ALL) ver(1) int setValue( int _value );

};

-u:**  example.h         (C++ Abbrev)--L10--C36--All----------------------------------------
Auto-saving...done
```

Figure 4.18: Code Generation Header Files

In figure 4.19, we can see how the generator is run. We can see that the result is a directory containing a number of files, together with a text file that describes the installation procedure. In appendix A, you can see the files that resulted from the generation process.

4.8 Summary

We have seen how the ideas presented in the design chapter (chapter 3) have been realized in the form of a functioning middleware. To solidify the validity of the imple-



```
paulbb@lemar:~/Master/source/generator$ ls
example.h  keywords.h  parser.py
paulbb@lemar:~/Master/source/generator$
paulbb@lemar:~/Master/source/generator$ python parser.py
Usage: parser.py [C++ Header File]
paulbb@lemar:~/Master/source/generator$
paulbb@lemar:~/Master/source/generator$ python parser.py example.h

INFO Parsing source file "example.h" ...

INFO gccxml cmd: /usr/bin/gccxml  -I"."   "example.h" -fxml="/tmp/tmpOSpWm-.xml"
paulbb@lemar:~/Master/source/generator$
paulbb@lemar:~/Master/source/generator$ ls
Example  example.h  keywords.h  parser.py
paulbb@lemar:~/Master/source/generator$
paulbb@lemar:~/Master/source/generator$ ls Example/
example.cpp  example-install.txt  rpc_example.h    xdr_example.h
example.h    rpc_example.cpp      xdr_example.cpp
paulbb@lemar:~/Master/source/generator$
```

Figure 4.19: Code Generation Output

mentation, we have created an application that mimics the communication patterns of MMOGs. While we have attempted to provide for most eventualities, we realize that the limited time available for completing this thesis has its restrictions. As such, we will discuss some aspects that we consider important, but which have been out of the scope of this thesis. Additionally, we will look at the possibilities offered by the design.

Chapter 5

Discussion

We have presented the design (chapter 3) and implementation (chapter 4) of our middleware, which in addition to supporting distributed objects, makes it possible to dynamically relocate objects. Since we already have discussed the rationale for our design and implementation in detail in the preceding chapters we will not reiterate these in this chapter. Instead, we will look at the possibilities that our middleware presents us with. This in effect means that we will take a closer look at the migration facility, which provides the developer with several interesting opportunities. As such, we will discuss a couple of scenarios where we consider it to be of particular use. While we have aimed at implementing a proof of concept in this thesis, we do realize that there are a number of issues, raised by distributed systems in general, that need to be considered. We will examine these implications and provide viable solutions for their resolution. Finally, we will consider the possibilities for expanding the functionality of the middleware. First, however, we will start by examining the migration facility.

5.1 Migration

Migration is the process of moving objects from one node in the distributed system to another. While a distributed system makes it possible to distribute the load, migration makes it possible to dynamically relocate objects to balance the load as it shifts. Throughout this thesis, we have presented this functionality as an integral part of our middleware. It certainly has had a great impact on the design we have selected, considering how we had to specifically accommodate for this type of behavior in the name service. The benefit of migration comes from the possibility to dynamically alter the state of the distributed application. With migration we are able to move objects between nodes and reconnect users based on their interaction with the migrated objects. We will now present a couple of examples that demonstrate the possibilities that migration offers the developer. This will be followed by the discussion about an additional communication model that migration introduces to object-oriented middleware.

5.1.1 Latency Reduction

We have seen that interactive real-time applications are particularly dependent on low latency in order to provide the user with an optimal experience. As such, we consider one scenario, where migration is involved, to be of particular interest. To solidify this example, we will consider the behavior of massively multi-player online games (MMOGs), but the low latency requirement also applies for a range of other interactive applications.

As we have mentioned, it is common for MMOGs to partition the virtual environment into regions. These regions partition the virtual environment into groups of objects that interact. Accordingly, it is reasonable to assume that the users interacting in MMOGs can be from widely different areas of the world. In many distributed systems, the effect of this is not necessarily prevalent, mainly because the architecture of the application can adapt to this by distributing users to servers accordingly. With MMOGs, users cannot necessarily be separated to accommodate for this, because of the interaction that occurs with other users in the virtual environment. As a result, users cannot be placed in the virtual world according to their physical location. It becomes apparent that the static region based architecture on a centralized cluster, while efficient and relatively easy to maintain, does not cater to the varying geographical locations of the users. It can be argued that games, such as World of Warcraft (WoW) [49] take this into account, to a certain degree. They have shards, which are single, unique instances of the game world, located at multiple locations across the world, where users generally connect to one close in proximity. The distribution of users, however, occurs as a result of the availability of servers, not because of how the middleware is implemented. Ideal examples of this are EVE online [50] and Anarchy online [7], with their single shard structure, where all users are connected to the same centralized cluster. Users from all over the world interact in the same logical regions, with one region hosted by a single server. The result is an architecture that cannot adjust itself to the difference in latency among its users.

Thus, it would be better to have a virtual world where the regions could be managed by nodes geographically located closer to the majority of the users. In this context, a recent study [51] of the MMOG Anarchy Online [7], analyzing the round-trip times (RTTs) in traces from one of several hundred regions composing the virtual environment, three distinct groupings of users were revealed. Based on the US location of the server, these user groups were in USA, Asia and Europe. It is safe to assume that one of these groups will be dominant, depending on the time of day. Thus, the assumption is that by analyzing the latency of users in a region from the virtual environment, one can determine where they are approximately located geographically. One is not limited to analyzing RTTs to obtain this information. Similarly, one could use the IP addresses of the users, if available, to obtain this information. Though with a lot of research being done on integrating peer to peer based systems into MMOGs [52], there is no guarantee that such information is readily available. As far as we can determine, there has been little research into the possibilities connected to load-balancing the regions of an MMOG based on the geographical location of users currently located within it. Most of the research is focused on effective load-balancing within a centralized cluster, by dynamically re-locating regions based on overall load. Since this has not been investigated, we propose to integrate migration with facilities that are aware of the physical locality of its users. The intent is to lower the response time of remote method invocations for the majority of users connected to a given region, which in turn should lower the overall latency. As we have proposed, this can be accomplished by migrating the region to a server closer in physical locality.

This is one possible application of migration, where the intent is to lower the overall latency in a distributed application. As we will now see, we do not consider the use of migration to be limited to this type of operation.

5.1.2 Redeployment

While the design of our middleware has been influenced by MMOGs, there is no requirement that an application exhibit characteristics similar to MMOGs for this middleware to be of use. If an application can be partitioned into groups that require real-time interaction, this middleware is suitable. The test application developed for this thesis is for instance a chat application, where the groups are formed based on channel membership. To give an example that does not involve human interaction, consider a sensor network that is setup to monitor the temperature of a building. The sensors can be added to groups based on the floor they are located on and setup as a warning system. If the sensors register a rise in temperature above a certain level, they dispatch a message to the floor (group) and each sensor on that floor sounds an alarm. If at some point in time the server managing the sensors requires maintenance and it is essential to maintain the interaction between the sensors, we can use the migration facilities to redeploy the sensors while maintenance is performed and migrate them back when the maintenance is completed.

5.1.3 Communication

In addition to providing the developer with a simple way of moving objects in a system, migration can also be used as a way to communicate. This as an addition to remote method invocation, which is the classical way for nodes in object oriented middleware to interact. Consider when an object is migrated, its state is serialized and transmitted to a new node in the system. At the receiving node, the object is recreated and initialized with the state it had at the sending node. As such, the construction of an object that has been migrated differs from that of an object being constructed as a normal instance of a class. This makes it possible to add additional behavior to an object that is constructed from migration. It is, for example, possible to have this object register itself with an event handler, which upon processing the object will react with an action corresponding to the information communicated by the object. The following example is taken from the test application we developed. When a user issues a command to move a channel from one node in the system to another it is required that the users be made aware of this change. As such, they must be told to reconnect to the new node. Instead of using RMI to accomplish this task, we migrate an instance of a class called *Reconnect*. This object contains the hostname and port of the node we want the user to reconnect to. When the object is recreated at the user, the reconnect object registers an event with the event handler for the system. Thus, when the event loop of the system processes the event, it reconnects to the new location, and continues execution from there. This type of messaging differs quite a bit from RMI. With RMI we expect an explicit reply, we want a confirmation from an action. With the migration form of communication, we expect the event to be processed, but the communication is implicit, there is no need for a reply. The functionality described here can be related to mobile agents [53], which are programs that include both code and data that travel from computer to computer in a network carrying out a task on someone's behalf. There are several implications raised because of such agents, especially in systems where security is a vital issue, in that these agents are widely considered to be a security risk [54]. This comes as a result of being able to run arbitrary code on the computers that accept the mobile agents. Our migration differs from mobile agents with regard to this, because in our middleware it is only the state of an object that is transmitted between nodes. Each node already has the code related to the object. There is also a side effect to this form of communication, when we compare the figures for migration (4.3) and RMI (4.4). We can see that there are more actions involved when performing RMI. As such, it is conceivable, but not tested and analyzed, that there is less overhead for this type of communication.

5.2 Partial Failures

A distributed system consists of multiple nodes connected by a network. When a node in such a system becomes unavailable, due to a system crash or network failure, it is referred to as a partial failure. The extent of such a failure depends on the implementation of the system. If the system is data driven and each node in the system has a replica of the available objects, such failures are not particularly serious, because further requests can be forwarded to the nodes still operating. In a system using a distributed name service, the scenario is different. In this case, a node failure means that the objects managed by that node will become unavailable. While the current architecture does not accommodate for partial failures, there are ways to minimize the repercussions of these incidents.

One possibility is to utilize a central registry, where all object migration is recorded. In the case of a miss, when attempting to access a remote object, the central register can be queried instead, as a backup solution. Once the node has recovered, normal object access can resume. A flaw with this approach, is that the central register becomes a single point of failure in the case of a crash. In addition, it must be capable of handling the traffic generated by all migrations, including any misses. This can be a weak point, since in a lot of situations migration is activated as a result of heavy load on a node.

A different approach has its roots in peer-to-peer based file systems, where copies of an object will be distributed to several nodes in the system. PAST [55] and OceanStore [56] have, for example, implemented such systems with success. PAST copies objects to the *k* numerically closest nodes, based on a comparison of the *nodeid* and *fileid*. OceanStore uses a more deterministic approach, and places the objects close to nodes that access the objects. Look up of objects in the system can then be implemented in a fashion similar to that of Chord [27] or Tapestry [28]. These implementations are based on the principle of incrementally forwarding messages from point to point, until they reach their destination. Each node in the system keeps a small routing map, which is used to determine which nodes to forward the message to. A problem with this type of look up is that the response time might be too high for time-dependent applications. Both the centralized and distributed failsafe techniques offer their own set of advantages and disadvantages.

5.3 Implications

The implications of distributed systems are not limited to the handling of node failures. With objects distributed to multiple nodes the name service will be queried frequently, as such, caching mechanisms can increase the efficiency of the look up process. Additionally, the use of distributed objects raises the question of how to efficiently reclaim the resources used by an object after its lifetime.

When objects are distributed to multiple nodes in the system, there are several considerations that need to be made. First, we need to consider the possibilities for increasing the efficiency of the name service, by expanding it with the ability to cache resources that are frequently in use. While the implementation of caching mechanisms are beyond the scope of this thesis we consider such mechanisms to be integral for the middleware. The reason for this is that it can greatly increase the speed of the processing, particularly if there is a subset of objects that are frequently being accessed. In the case of MMOGs, this is not an unlikely scenario. Consider the long lifetime of the objects representing the interacting players. There is a high likelihood that much of the traffic generated involves these objects, and as such, it could be useful to have them cached. This is particularly the case when we consider migration in view of moving players between regions of the game. This also introduces a consideration about having tiered caching, where the application developer can assign objects to caching tiers dependent on their perceived importance, or number of expected look ups. The tiering system is comparable to how a central processing unit commonly has a L1, L2 and L3 cache.

While caching of frequently used resources can improve the speed of the application, it is just as important to reclaim obsolete objects. If this is not done, the system resources can become depleted. Garbage collection, as it is commonly referred to, is a form of automatic memory management. The garbage collector reclaims memory used by objects that will never again be accessed by the application. As such, the purpose of the garbage collector is to manage the life time of an object and free the resources used by it. In the face of a distributed system, this can become particularly difficult, and a number of solutions to the implementation of a reliable and effective system has been proposed [57–59].

5.4 Summary

We believe that the middleware we have implemented offers some exciting possibilities to the developers of distributed applications. It provides facilities that are, as far as we have been able to determine, non-existent in existing object-oriented middleware. We are referring to the ability to migrate objects between nodes. To implement this functionality, we have had to expand the design of the name service to include mechanisms for maintaining references to the objects, even after multiple migrations. The name service itself is implemented as a distributed name service, which is not as uncommon in object oriented middleware. We do realize that this middleware is not suitable for all types of applications. There are a number of object oriented middleware implementations available that are more suitable depending on the application. In particular, we consider this middleware, in its current state, to be unsuitable for applications that are data driven and require reliable transactions. Consider the requirements of a web application, where being available and reliable is more important then being able to deliver a result as fast as possible. Furthermore, we have seen examples of middleware, such as CORBA (see section 2.3), that attempt to solve all middleware problems, and as a result end up becoming unwieldy and complex. We have demonstrated a few areas of use for our middleware and believe there are a number of additional uses for it. In the next chapter, we will summarize the work accomplished and knowledge gained through the time working on this thesis.

Chapter 6

Conclusion

In the course of this thesis, we have designed and implemented a middleware that meets the needs of real-time interactive applications. Here we will shortly summarize the results of our efforts and the most significant contributions of our work. Finally, we will present issues that we consider worthwhile pursuing.

6.1 Summary and Contribution

Middleware is an important tool in the development of distributed applications, and equally important is selecting an appropriate middleware. The selection process should be guided by the characteristics and requirements of the application in question. During our research, we noted that interactive real-time applications, which are time-dependent and event based applications, were inadequately supported by existing middleware. As such, we isolated a number of factors pertaining to this genre of applications that we considered fundamental to the development of a middleware that could accommodate for the behavior of these applications. As a result of analyzing these factors, we designed and implemented a middleware that supports the migration of objects using a distributed name service to maintain references. We did this, because we noticed that these applications, due to their interactive nature, could achieve performance benefits if they were able to dynamically accommodate for changes in the environment. The middleware we developed supports this dynamic nature, as we demonstrated by implementing an application that mimics the chat system of a

MMOG. The migration facilities provide the developer with several interesting opportunities, as we saw in section 5.1. The scenario involving latency reduction in MMOGs, by migrating objects based on geographical composition of user groups, we considered to be particularly interesting. As such, we have written an article [60] about it, which will appear in the proceedings of the ITA' 07 conference, held in Wrexham, North Wales. We have also reviewed the implications of our design, in light of distributed systems. Accordingly, we have considered possibilities for avoiding the effects of partial failures. We have also considered additional facilities that distributed systems should have in place, such as garbage collection and caching. The development of this middleware has provided us with valuable insight, and we realize that there are a number of additional services that could be useful for the developer. We have provided one service, in the form of a code generator, which easily can be expanded to include support for new services.

6.2 Future Work

In addition to the functionality mentioned in the preceding section, there are a number of useful expansions to the middleware that could be implemented. In this section we will review some of the more obvious ones.

Interactive real-time applications, such as MMOGs, are designed to run indefinitely, as such, it would be beneficial to be able to dynamically load new classes during runtime. This would make it possible to update the implementation of a class if it became necessary.

Another aspect is related to partitioning. At this point in time, there are no mechanisms that make it possible to dynamically form groups. Though we consider automatic grouping of objects, based on examining the interaction patterns, to be an interesting area that is worth investigating in the future.

Finally, it would be useful to provide the developer with language bindings to a high level programming language, such as Python, Ruby, Perl or similar. This could make the development process a lot faster, while still retaining the low level interfaces provided for by C++.

Appendix A

Result from Code Generation Example

A.1 example.h

```
/* Output automatically generated. */

#ifndef EXAMPLE_H_
#define EXAMPLE_H_

#include <string>
#include "object.h"
#include "xdrhandler.h"
#include "rpcresult.h"
#include "rpcutil.h"
#include "remoteprocedurecall.h"

class ExampleRPC;

class Example : public Object, public RemoteProcedureCall<Example>, public RPCUtil
{
protected:

        friend bool_t xdr_Example( XDR* xdrs, Example* objp );

        int value;

        friend bool_t xdr_rpc_setValueReturnHandler (XDR *xdrs, Example *objp);
        int retval_setValue;


public:

        static ExampleRPC* createRPCHandler( Objectid* );

        Example( XDRHandler& xdrh, bool createObjectid=true );
        char * getTypeName();

        int executeFunction( std::string, XDRHandler& );
        int getFunctionAuthLevel( std::string );

        Object& create( XDRHandler& );
        void deflate( XDRHandler& xdrh );

        Example( bool createObjectid=true );
        virtual ~Example();

        virtual int setValue( int _value );
        void rpc_setValue( XDRHandler& );
        void rpc_setValueReturnHandler( XDRHandler& xdrh, int returnValue );

};

#endif /*EXAMPLE_H_*/
```

A.2 example.cpp

```
/* Output automatically generated. */

#include "example.h"
#include "rpc_example.h"
#include "rpcresult.h"

Example::Example( bool createObjectid/*=true*/ ) : Object( *this, createObjectid )
{
        registerFunction( std::string("setValue"), &Example::rpc_setValue, 0, 1);
}

void Example::deflate( XDRHandler& xdrh )
{
        xdrh.setStreamToWrite();
        xdr_Example( xdrh.getStream(), this );
}

Object& Example::create( XDRHandler& xdrh )
{
        xdrh.setStreamToRead();
        return new Example( xdrh, 0);
}

Example::Example( XDRHandler& xdrh, bool createObjectid/*=true*/ ) : Object( *this, createObjectid )
{
        xdr_Example( xdrh.getStream(), this );
}

int Example::executeFunction( std::string functionName, XDRHandler& xdrh )
{

        if( findFunction( functionName ) ){
                callFunction( functionName, xdrh );
        }
        else{
                RPCResult res( "NoSuchFunction" );
                xdrh.resetHandler();
                res.serialize( xdrh );
                xdrh.writeToSocket();
                return 1;
        }
        return 0;
}

int Example::getFunctionAuthLevel( std::string functionName )
{
        return getFunctionAuthorizationLevel( functionName );
```

```
}

Example::~Example()
{
}

ExampleRPC* Example::createRPCHandler( Objectid* oid )
{
        ExampleRPC* erpc = ExampleRPC::getInstance();
        erpc->setObjectid( oid );
        return erpc;
}

char * Example::getTypeName()
{
        return "example_h_EXAMPLE";
}







//--------------------[ begin: setValue

int Example::setValue( int_value )
{
        /* your code here */
}

void Example::rpc_setValue( XDRHandler& xdrh )
{
        ExampleRPC* erpc = ExampleRPC::getInstance();
        erpc->setValueParameters( xdrh );
        setReturnValue( "Processed", &xdrh );
        rpc_setValueReturnHandler(
                xdrh,
                setValue( erpc->getParam_value() )
        );
        xdrh.writeToSocket();
}

void Example::rpc_setValueReturnHandler( XDRHandler& xdrh, int returnValue )
{
        xdrh.setStreamToWrite();
        retval_setValue = returnValue;
        xdr_rpc_setValueReturnHandler( xdrh.getStream(), this );
}

//--------------------[ end: setValue
```

A.3 rpc_example.h

```
/* Output automatically generated. */

#ifndef EXAMPLERPC_H_
#define EXAMPLERPC_H_

#include "example.h"
#include "rpcutil.h"
```

```

class Objectid;

class ExampleRPC : public Example
{

        int _value;

        friend bool_t xdr_rpc_setValue( XDR * xdrs, ExampleRPC * objp);
        friend bool_t xdr_rpc_setValueReturnHandler( XDR * xdrs, ExampleRPC * objp);


        static ExampleRPC* _erpcInstance;

        ExampleRPC( bool createObjectid=false );
        ExampleRPC( const ExampleRPC& );
        void operator=( const ExampleRPC& );
        ~ExampleRPC();


public:

        static ExampleRPC* getInstance();
        void setObjectid( Objectid* );



        int setValue( int __value );
        void setValueParameters( XDRHandler& xdrh );
        int getParam_value();

};

#endif /*EXAMPLERPC_H_*/
```

A.4 rpc_example.cpp

```
/* Output automatically generated. */

#include "rpc_example.h"
#include "rpcresult.h"

ExampleRPC* ExampleRPC::_erpcInstance = 0;

ExampleRPC* ExampleRPC::getInstance()
{
        if( _erpcInstance == 0 )
        {
                _erpcInstance = new ExampleRPC();
        }
        return _erpcInstance;
}

ExampleRPC::ExampleRPC( bool createObjectid/*=false*/ ) : Example( createObjectid )
{
}

void ExampleRPC::setObjectid( Objectid* _oid )
{
        oid = _oid;
}
```

```

ExampleRPC::~ExampleRPC()
{
}









//------------------[ begin: setValue

/* serialize */
int ExampleRPC::setValue( int __value )
{
        _value = __value;

        bool retry = true;
        do {
                XDRHandler* xdrh = createHeader( "setValue", getFunctionVersion("setValue"), getObjectid() );
                xdr_rpc_setValue( xdrh->getStream(), this );
                xdrh->writeToSocket();
                XDRHandler* xdrh_ret = new XDRHandler( xdrh->getSocket() );
                int result = handleReturnValue( xdrh_ret );

                if( result == 1 )
                {
                        Objectid* oid = Objectid::deserialize( *xdrh );
                        ObjectRegister::setCareofAddress( getObjectid(), *oid );
                }
                else if( result == -1 )
                {
                        std::cout << "setValue,␣result:␣-1" << std::endl;
                        std::cout << "Killing␣application." << std::endl;
                        exit(1);
                }
                else
                {
                        xdr_rpc_setValueReturnHandler( xdrh_ret->getStream(), this );
                        return retval_setValue;
                }
                delete xdrh;
                delete xdrh_ret;
        } while( retry );
        return 0; /* should never reach this point */
}

/* deserialize */
void ExampleRPC::setValueParameters( XDRHandler& xdrh )
{
        xdrh.setStreamToRead();
        xdr_rpc_setValue( xdrh.getStream(), this );
}

int ExampleRPC::getParam_value()
{
        return _value;
}

//------------------[ end: setValue
```

A.5 xdr_example.h

```
/* Output automatically generated. */

#ifndef XDR_EXAMPLE_H_
#define XDR_EXAMPLE_H_

#include <rpc/xdr.h>
#include "example.h"
#include "rpc_example.h"

bool_t xdr_rpc_setValue( XDR* xdrs, ExampleRPC* objp );

bool_t xdr_Example( XDR* xdrs, Example* objp );

#endif /*XDR_EXAMPLE_H_*/
```

A.6 xdr_example.cpp

```
/* Output automatically generated. */

#include "xdr_example.h"

bool_t xdr_rpc_setValue( XDR* xdrs, ExampleRPC* objp )
{
        if( !xdr_int( xdrs, &objp->_value ) )
        {
                return false;
        }
        return true;
}

bool_t xdr_rpc_setValueReturnHandler( XDR* xdrs, ExampleRPC* objp )
{
        if( !xdr_int( xdrs, &objp->retval_setValue ) )
        {
                return false;
        }
        return true;
}

bool_t xdr_Example( XDR* xdrs, Example* objp )
{
        if( !xdr_int( xdrs, &objp->value ) )
        {
                return false;
        }
        return true;
}
```

A.7 example-install.txt

```
Usage instruction.
------------------
Edit the following files:

In file: "distptr.h" add the following lines:
```

```

#include "example.h"
#include "rpc_example.h"

In file: "main.cpp" in function: "static␣void␣registerClassTypes()" add the following line(s):

Example* type = new Example();
TypeRegister::addType( type->getTypeName(), dynamic_cast<Object*>( type ) );

In file: "main.cpp" add the following lines:

#include "example.h"
```

Appendix B

Source Code and Documentation

The source code and documentation for the implementation can be found on the compact disc accompanying the printed version of this thesis, and is otherwise available for download as a tar archive at `http://paul.beskow.no/master/paulbb.middleware-cd.content.tar`.

B.1 Content

The compact disc, or archive, consists of four directories.

1. *documentation*
   This directory contains the documentation generated from the comments in the source code, and contains two sub directories, one for the *latex* version of the documentation and one for the *html* version.

2. *generator*
   This directory contains the code generator.

3. *pygccxml-0.8.5*
   This directory contains the modified version of the pygccxml module.

4. *source*
   This directory contains the source code for the middleware.

B.2 The Generator

B.2.1 Installation

Before the generator script can be run it is necessary to install the *pygccxml-0.8.5* module. This is accomplished by running the following command from the root of the *pygccxml-0.8.5* directory:

```
python setup.py install
```

This module relies on the GCC-XML program, which can be obtained from: `http://www.gccxml.org/HTML/Download.html`.

Note: It is important that the development version from the CVS repository is used. This is because the available *latest release* does not support the functionality we require.

B.2.2 Usage

Once the module has been installed the script *parser.py* from the *generator* directory can be run as follows:

```
python parser.py example.h
```

Where the file *example.h* is substituted with the name of the header file you wish to parse. See section 4.7 for more discussion on the use of the parser.

B.3 The Middleware

B.3.1 Installation

The code is written in C++ and the makefile is written for use with GCC. We have not tested the build process with other compilers. In order for the code to compile, the cryptography library *beecrypt* needs to be installed, this can be obtained from `http://sourceforge.net/projects/beecrypt`. Once this is in place the following command can be run in the directory *source/Debug*:

```
make
```

B.3.2 Usage

There are three possible ways of running the program, as the primary server:

```
./middleware server <port> 0
```

as the secondary server:

```
./middleware server <port> 1
```

or as a client:

```
./middleware client <host> <port>
```

For a detailed example of its use, see section 4.6.

Bibliography

[1] Tom Beigbeder, Rory Coughlan, Corey Lusher, John Plunkett, Emmanuel Agu, and Mark Claypool. The effects of loss and latency on user performance in unreal tournament 2003. In *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 144–151, New York, NY, USA, 2004. ACM Press.

[2] Lothar Pantel and Lars C. Wolf. On the impact of delay on real-time multiplayer games. In *NOSSDAV '02: Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pages 23–29, New York, NY, USA, 2002. ACM Press.

[3] The Entertainment Software Association. ESAs 2006 essential facts about the computer and video game industry. `http://www.theesa.com/`, jan 2007.

[4] B. S. Woodcock. An analysis of mmog subscription growth. `http://www.mmogchart.com`, jan 2007.

[5] C.J. Bonk, V.P. Dennen, and INDIANA UNIV AT BLOOMINGTON. *Massive Multiplayer Online Gaming: A Research Framework for Military Training and Education*. Defense Technical Information Center, 2005.

[6] J. Preston, L. Booth, and J. Chastine. Improving Learning and Creating Community in Online Courses via MMOG Technology. *Proceedings of the 35 thSIGCSE Technical Symposium (Norfolk, VA, March 2004), ACM Press, pending acceptance*.

[7] Funcom. Anarchy Online. `http://www.anarchy-online.com/`, *January*, 2007.

[8] Linden Lab. Second Life. `http://secondlife.com/`, *January*, 2007.

[9] B. De Vleeschauwer, B. Van Den Bossche, T. Verdickt, F. De Turck, B. Dhoedt, and P. Demeester. Dynamic microcell assignment for massively multiplayer online gaming. *Proceedings of NetGames' 05, Hawthorne, NY, USA*, pages 1–7, October 2005.

[10] T.N.B. Duong and S. Zhou. A dynamic load sharing algorithm for massively multiplayer online games. *ICON' 03, Sydney, Australia*, pages 131–136, October 2003.

[11] Rajesh Krishna Balan, Maria Ebling, Paul Castro, and Archan Misra. Matrix: Adaptive middleware for distributed multiplayer games. In Gustavo Alonso, editor, *Middleware*, volume 3790 of *Lecture Notes in Computer Science*, pages 390–400. Springer, 2005.

[12] A. Bharambe, J. Pang, and S. Seshan. Colyseus: A Distributed Architecture for Online Multiplayer Games. *Proceedings of NSDI' 06, San Jose, USA*, pages 155–168, May 2006.

[13] TIOBE Software. Programming Community Index. *`http://www.tiobe.com/tpci.htm`, April*, 2007.

[14] Sun Microsystems Inc. Remote procedure call protocol specification. In *Networking on the Sun Workstation*. February 1986. Published as rfc1057.

[15] W. Rosenberry, D. Kenney, and G. Fisher. *OSF Distributed Computing Environment: Understanding DCE*. O'Reilly, Sebastopol, 1993.

[16] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93, 2002.

[17] Michi Henning. A new approach to object-oriented middleware. *IEEE Internet Computing*, 8(1):66–75, 2004.

[18] S. Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2):46–55, 1997.

[19] N. Brown and C. Kindel. Distributed Component Object Model Protocol–DCOM/1.0. *Online, November*, 1998.

[20] The Open Group. COMsource. `http://www.opengroup.org/comsource/`, *April*, 2007.

[21] The Open Group. `http://www.opengroup.org/`, *April*, 2007.

[22] K. Arnold, J. Gosling, and D. Holmes. *Java (TM) Programming Language, The*. Addison-Wesley Professional, 2005.

[23] A. Wollrath, R. Riggs, and J. Waldo. A Distributed Object Model for the Java System. *Computing Systems*, 9(4):265–290, 1996.

[24] T. B. Znati and J. Molka. A simulation based analysis of naming schemes for distributed systems. In *Proceedings of the 25th Annual Simulation Symposium*, pages 42–53, Los Alamitos, CA, USA, April 1992.

[25] J.H. Howard, Carnegie-Mellon University, and Information Technology Center. *An Overview of the Andrew File System*. Carnegie Mellon University, Information Technology Center, 1988.

[26] S. Saroiu, K.P. Gummadi, and S.D. Gribble. Measuring and analyzing the characteristics of Napster and Gnutella hosts. *Multimedia Systems*, 9(2):170–184, 2003.

[27] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of SIGCOMM' 01, San Diego, CA, USA*, pages 149–160, aug 2001.

[28] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, apr 2001.

[29] B. Goodheart and J. Cox. *The magic garden explained: the internals of UNIX System V Release 4: an open systems design*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1994.

[30] J. Postel. Domain Name System Structure and Delegation, 1994.

[31] Object Management Group. `http://www.omg.org/`, *April*, 2007.

[32] Object Management Group. Catalog of OMG Specifications. `http://www.omg.org/technology/documents/spec_catalog.htm`, *April*, 2007.

[33] C. E. Perkins. Mobile IP. *Communications Magazine, IEEE*, 35(5):84–99, 1997.

[34] C. Perkins. RFC 2002: IP mobility support, October 1996.

[35] Raj Srinivasan. XDR: External data representation standard. RFC 1832, August 1995.

[36] Sun Microsystems, Inc. XDR: External data representation standard. RFC 1014, June 1987.

[37] D. van Heesch. Doxygen Manual, 2003.

[38] Bjarne Stroustrup. *The C++ Programming Language: Third Edition*. Addison-Wesley Publishing Co., Reading, Mass., 1997.

[39] Michael Donahoo. *TCP/IP Sockets in C: Practical Guide for Programmers*. Morgan Kaufmann Publishers, pub-MORGAN-KAUFMANN:adr, 2003.

[40] J. Postel. RFC–793 Transmission Datagram Protocol. *Information Sciences Institute, USC, CA, Sept*, 1981.

[41] K. Egevang and P. Francis. RFC1631: The IP Network Address Translator (NAT). *Internet RFCs*, 1994.

[42] Christoph Ludwig Schuba. On the modeling, design, and implementation of firewall technology, January 01 1997.

[43] Boost.org. Boost C++ Libraries - Serialization. `http://www.boost.org/libs/serialization/doc/index.html`, *April*, 2007.

[44] D. Winer et al. XML-RPC Specification. *Jun*, 15:7, 1999.

[45] Sun Microsystems. ONC+ Developer's Guide. `http://docs.sun.com/app/docs/doc/816-1435`, *April*, 2007.

[46] Roman Yakovenko. pygccxml. `http://www.language-binding.net/pygccxml/pygccxml.html`, *April*, 2007.

[47] B. King. GCC-XML the xml output extension to gcc. *Undated. Online:* `http://www.gccxml.org/HTML/Index.html`.

[48] GCC, The Gnu Compiler Collection. `http://gcc.gnu.org/`, *January*, 2007.

[49] Blizzard. World of Warcraft. `http://www.worldofwarcraft.com/`, *January*, 2007.

[50] CCP. EVE Online. `http://www.eve-online.com/`, *January*, 2007.

[51] Carsten Griwodz and Pål Halvorsen. The fun of using TCP for an MMORPG. *Proceedings of NOSSDAV' 06, Newport, RI, USA*, pages 1–7, may 2006.

[52] Abdennour El Rhalibi and Madjid Merabti. Agents-based modeling for a peer-to-peer mmog architecture. *Comput. Entertain.*, 3(2):3–3, 2005.

[53] J.E. White. Mobile agents. *Software agents table of contents*, pages 437–472, 1997.

[54] W.M. Farmer, J.D. Guttman, and V. Swarup. Security for Mobile Agents: Issues and Requirements. *Proceedings of the 19th National Information Systems Security Conference*, 2:591–597, 1996.

[55] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. *Proceedings of SOSP'01, Lake Louise, Alberta, Canada*, pages 188–201, oct 2001.

[56] Dennis Geels. Data Replication in OceanStore. Technical Report UCB//CSD-02-1217, Computer Science Division, U. C. Berkeley, nov 2002.

[57] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In Henry G. Baker, editor, *Proc. Int. Workshop on Memory Management (IWMM)*, volume 986, pages 211–249, Kinross, Scotland (UK), September 1995.

[58] N. C. Juul and E. Jul. Comprehensive and robust garbage collection in a distributed system. In *Proc. Int. Workshop on Memory Management*, number 637, pages 103–115, Saint-Malo (France), 1992. Springer-Verlag.

[59] Jose M. Piquer. Indirect distributed garbage collection: handling object migration. *ACM Trans. Program. Lang. Syst.*, 18(5):615–647, 1996.

[60] P. Beskow, P. Halvorsen, and C. Griwodz. Latency Reduction in Massively Multiplayer Online Games by Partial Migration of Game State. *To appear in the Proceedings of the 2nd International Conference on Internet Technology and Applications (ITA' 07), Wrexham, North Wales, UK*, September 2007.