UNIVERSITY OF OSLO
Department of Informatics

Latency reduction in distributed interactive applications by core-node selection and migration

Master Thesis

Geir Arveschoug Erikstad

Latency reduction in distributed interactive applications by core-node selection and migration

Geir Arveschoug Erikstad

Abstract

Massive multiplayer online games (MMOGs) have stringent latency requirements, and must support a large amount of concurrent players. To handle these conflicting requirements, it is common to divide the virtual world into smaller virtual regions. As MMOGs attract players from all over the world, it is plausible to disperse these regions on geographically distributed servers. Core-node selection algorithms can then be applied to locate an optimal server for placing a region or instance, based on client latencies.

In this thesis, we have designed and implemented a prototype of a middleware which supports core-node selection algorithms and migration. Using this middleware, we have built a test application which simulates a simple MMOG. This application has been tested on PlanetLab, with clients and servers distributed all over the world. We have also performed micro benchmarks on the middleware itself, to test how it performs.

Acknowledgements

I would like to thank my supervisors Paul B. Beskow, Pål Halvorsen, and Carsten Griwodz for their patience, guidance and valuable input during my work with this thesis. Without their help, this thesis would not have been possible.

I would also like to thank my girlfriend Carline for help and support during the process of writing this thesis.

Also thanks to all the guys at the Simula lab for creating an inspiring environment, and for helping keep a good morale, with both entertainment and helpful hints.

Finally, I would like to thank my family and my friends for supporting and encouraging me along the way.

Contents

Abstract **iii**

Acknowledgements **v**

1 Introduction **1**

1.1 Background and Motivation 1
1.2 Problem Definition 3
1.3 Main Contributions 4
1.4 Outline 5

2 Background **7**

2.1 Scenario: Massive Multiplayer Online Games 7
2.2 Migration 9
2.2.1 Virtual Machine Migration 10
2.2.2 Process Migration 11
2.2.3 Object Migration 13
2.2.4 Code Migration 14
2.2.5 Data Migration 15
2.2.6 Summary 16
2.3 Core-Node Selection Algorithms 16
2.3.1 Algorithms 17
2.3.2 Obtaining Link Latencies 20
2.3.3 Summary 21
2.4 Middleware 21
2.4.1 Remote Method Invocation and Name Service 23

2.5 Summary 24

3 Design **27**

3.1 Requirements 27
3.2 Serialization 28
3.3 Remote Method Invocation 29
3.4 Event distribution 30
3.5 Middleware control messages 31
3.6 Migration 31
3.7 Code and Library Versions 32
3.8 Name Service 33
3.8.1 Identification 33
3.8.2 Distribution 34
3.9 Core Selection Algorithm 35
3.10 Grouping of Objects 36
3.10.1 Overlay Topology 36
3.10.2 Latency Measurements 37
3.11 Internet Sockets 38

4 Middleware Implementation **39**

4.1 Middleware Main Components 39
4.2 Remote Method Invocation 43
4.2.1 Distributed Pointer 43
4.2.2 Name Service 45
4.2.3 Proxy Object 46
4.2.4 Connection Manager 47
4.2.5 Data Sender 48
4.2.6 Data and Connection Receiver 49
4.2.7 Skeleton 49
4.3 Events 50
4.4 Core Selection 50
4.4.1 Control Messages 51
4.4.2 Message Handler 52

4.4.3 Migration Service 53
4.4.4 Migration Group 53
4.4.5 Network Graph 54
4.4.6 Latency Measurement 55
4.4.7 The Core-Node Selection Algorithm 56
4.5 Migration of Objects and Groups of Objects 57
4.5.1 Migration of an Object 57
4.5.2 Migration of a Group of Objects 59
4.5.3 Changing the Migration Group of an Object 60
4.6 Summary 60

5 Test Application Implementation **61**
5.1 World 61
5.2 Avatar 64
5.3 View 64
5.4 Goldmine 65
5.5 Server 65
5.6 Client 66

6 Experiments **67**
6.1 Middleware Micro Benchmarks 67
6.1.1 Remote Method Invocation 67
6.1.2 Migration 70
6.1.3 RMI during migration 72
6.2 Test Application Experiments 74
6.2.1 PlanetLab 74
6.2.2 Experiment Set-Up 75
6.2.3 Run-Time Example 77
6.2.4 Results 78
6.3 Summary 82

7 Discussion **83**
7.1 Evaluation of the Experiments 83
7.2 Implications 84

7.3 Latency Estimation 85
7.4 Instances 86
7.5 Different core-node selection algorithms 87
7.6 Peer-to-Peer 87
7.7 Summary 88

8 Conclusion 89
8.1 Summary and Contributions 89
8.2 Future Work 90

A Source Code and Logs 91
A.1 Compiling 91

B Acronyms 93

List of Figures

1.1 Anarchy Online: Connection RTTs sorted by min RTT 3
1.2 Latency as physical distance 4

2.1 Network Graph 19
2.2 Middleware 22
2.3 Remote Method Invocation 23

4.1 Server/Proxy Middleware Overview 40
4.2 Client Middleware Overview 42
4.3 A remote method invocation example 44
4.4 Main Initialization Message 51
4.5 Example of a Network Overlay Graph 55
4.6 Migration Message with an Avatar Object 58

5.1 A World Object 62
5.2 Two World Objects - One Within the Other 63
5.3 The Avatars View 65

6.1 Average time of method invocation with standard deviation . . 68
6.2 Average time to migrate an single object of varying size with standard deviation 71
6.3 Average time to migrate groups of varying sizes, with objects of varying sizes, with standard deviation 72
6.4 Average time of RMI during migration with standard deviation 74
6.5 Geographical location of the PlanetLab nodes 76

6.6 Experiment one: First run - Aggregated mean RMI response time . . . 78
6.7 Experiment one: First run - RMI time per client with 1 server 79
6.8 Experiment one: First run - RMI time per client with 3 servers 80
6.9 Experiment one: First run - RMI time per client with 6 servers 80
6.10 Experiment one: Second run - Aggregated mean RMI response time . . . 81
6.11 Experiment two: Mean RMI time of sessions . . . 82

List of Tables

2.1 Results from running k-Median algorithm on graph V 20

6.1 RMI: Essential statistics in microseconds, 10000 runs 69

6.2 Migration of single object: Essential statistics in microseconds, 10000 runs 70

6.3 Migration of groups of objects: Essential statistics in milliseconds, 1000 runs 71

6.4 RMI during migration: Essential statistics in microseconds, with 10 integers in each object 73

Chapter 1

Introduction

1.1 Background and Motivation

Massive Multiplayer Online Games (MMOGs) have gained huge popularity during the last decade. There are now more than 16 million active subscribers to many different MMOGs [1]. The most popular of these games is by far World of Warcraft [2], a role playing game that had over 10 million subscribers in January 2008 [1]. The rest of the subscribers are divided between other MMOGs in several categories, like role playing games, sports games, shooters etc., with role playing games being the most popular.

In MMOGs players create one or more *avatars*. An avatar is a virtual representation of a character that the player controls in a virtual world. The virtual world is persistent and continuous, i.e., the world continues to exist after the player has logged out, and it is there when he logs back on. MMOGs can support thousands of concurrent players. Eve Online has had as many as 41,690 simultaneous clients connected at the same time [3].

MMOGs are interactive applications, i.e., the player interacts with the game world, and everything that exists in the virtual world, through his avatar. To provide players with a convincing gaming experience, the game world has to be consistent. Every player should have the same view of this world. This means in effect that if one avatar kills a monster, the monster should also appear dead to all the other avatars which are nearby. No one

else should be able to attack, or be attacked by the monster, after someone has killed it.

This real-time interaction the players have with the game environment, requires the servers to process all the events fast, and update all clients as fast as possible. Therefore, the virtual worlds in a MMOG are usually divided into regions. These virtual regions can be assigned to different servers inside a cluster or a server park, to even the load.

The response time (latency) a player experiences, has a direct impact on the game experience. In [4], it is shown that if the latency in an MMOG exceeds 500 ms, the game experience decreases drastically. When latency gets this high, it might for instance result in a player getting killed, because he did not have time to respond to an attack in time. Of course, this can be frustrating to players. It is therefore important to research ways of reducing latency in MMOGs.

Another problem is that the latency a player experiences is affected by the physical distance to the server. As MMOGs attract players from all over the world, it is impossible to provide the same latency to all players. If all clients connect to one server located in North America, as is the case in MMOGs such as Eve Online [5] and Anarchy Online [6], European players will experience a higher latency than American players because of the shorter physical distance to the server. In games like this, there will be different peak hours according to the players' geographical position. When it is evening in North America there will probably be more American than European players. During this period of day, the server placement in America is good for the majority of the players. However, when it turns to night in North America, there will probably be more European players than American. It would then be optimal to use a server based in Europe.

World of Warcraft [2] (WoW) has in a way taken the physical location into consideration. WoW has approximately three thousand players on each *realm* [7] (server cluster), and they have placed the realms close to the clients, i.e., they have servers both in North America, Europe, and Asia. The location of the server clusters is static.

In [9], packet traces from Anarchy Online have been analyzed. In fig-

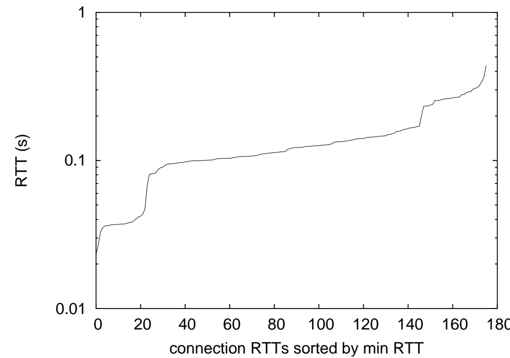

Figure 1.1: Anarchy Online: Connection RTTs sorted by min RTT - figure from [8]

ure 1.1, the minimum round-trip time for about an hour of game play for one game region is shown. Since the servers Anarchy Online uses are located in the US, this figure indicates that most of the players at this time are located in Europe, with some players connected from both the US and Asia. This figure shows that keeping a low latency to all players is a hard task, since latency is affected by physical distance.

1.2 Problem Definition

To lower the response time, one can support migration of objects. This means moving objects such as avatars and the virtual regions from one node to another. If one has a distributed set of servers to choose from around the world, one could move the avatars and the virtual regions closer to the physical center of players, i.e., reducing the aggregate latency of the interacting players.

Figure 1.2 shows how the latency maps to physical distance. In the figure, we see three players, located in Seattle, Mexico City, and Nuuk. With a server located in both Oslo and New York, we can clearly see that the server in New York would server these three players best.

To be able to find the most optimal server, one can use core-node selection algorithms. These algorithms use different heuristics to find nodes with different capabilities. There are core-node selection algorithms that search

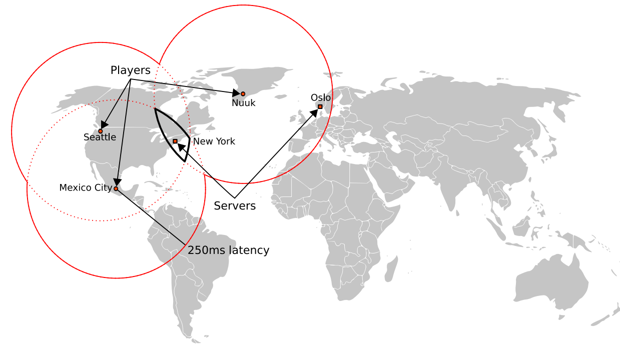

Figure 1.2: Latency as physical distance - figure from [8]

for one server among a set, which will yield the lowest mean latency for all clients.

In this thesis, we will make use of core-node selection algorithms and migration of objects. We want to test how these algorithms can be combined with migration to lower the aggregate latency of the players, and thus provide the players with a better game experience.

1.3 Main Contributions

We have seen that interactive distributed applications require low pair-wise latency between clients and the server to give the players a good gaming experience. We have observed that the physical distance between the clients and the server affects the latency between the client and the server. The goal of this thesis is therefore to try and lower the physical distance between the clients and the server.

To accomplish this goal, we have designed and implemented a middleware which tries to minimise pair-wise latency between the clients and the servers, by moving the servers closer to the players. To be able to move the server, we have found it necessary to look at different ways to migrate objects. We have identified data migration as a fast and good way to migrate objects in a distributed interactive application.

The middleware also needs a way to decide where to migrate the objects. We have found that core-node selection algorithms can be used to find servers

that deliver lower pair-wise latency to the clients, when such servers exist.

To be able to test our middleware, we have implemented a small test application which simulates a simplified MMOG. We have performed experiments with this application on a world-wide distributed set of machines called PlanetLab. Our results have shown that our middleware is capable of finding nodes that gives low aggregate latency for groups of clients, and migrate the game state to these.

The implementation of our middleware and the results from our experiments have been the base for a paper [10], which is currently pending review for the NetGames 2009 conference.

1.4 Outline

We will start this thesis by giving an overview of the requirements for our implementation and some related work, in chapter 2.

Chapters 3 and 4 deal with our middleware. In chapter 3, we will outline the basic requirements for such a middleware, and the design choices we faced on the way. Chapter 4 contains a thorough description of our implementation and how the main operations are processed.

In chapter 5 we will describe the test application we have implemented.

The results of testing our middleware with the test application will be discussed in chapter 6.

In chapter 7 we will discuss some of the implications and possibilities our middleware leads to.

Chapter 8 summarizes and concludes the work done during our work with this thesis.

Chapter 2

Background

In this chapter we will look at the background requirements for implementing a middleware such as described in the introduction. At first we will have a look at important properties of today's massive multiplayer online games, which are a background for our implementation. Then, we will discuss some basic operations a middleware should support: migration and core-node selection. (A thorough outline of the operations our middleware supports will be given in chapter 3.) We will discuss different kinds of migration, from moving entire operating systems down to migration of singular data, and give some examples of known implementations and uses. In much the same way, we will explain the use of core-node selection algorithms and list some of the existing ones. The remainder of this chapter is devoted to describing the use and basic implementation of a middleware in a distributed system. Here we will shortly explain some of the main elements of such a middleware.

2.1 Scenario: Massive Multiplayer Online Games

Massive Multiplayer Online Games (MMOGs) are internet games that create a persistent world that always exists online. The game can be played by enormous amounts of players logging on at any time. MMOGs both provide possibilities for, and depend on social interaction between players. Players present themselves in the fictional world using *avatars*, graphic representa-

tions of the player's alter ego. The games are event-driven and players are constantly being updated about the state of events in their vicinity. They can see each other's actions and respond to them, thus creating the game's development and output together.

There are several kinds of MMOGs. For instance First-Person Shooter (FPS), Real-Time Strategy (RTS), and Role Playing Game (RPG). However, nowadays all well-known MMOGs are basically MMORPGs (World of Warcraft, Age of Conan, and The Lord of the Rings Online to name a few). The common denominator for these games is that they all are social games, depending on social interaction between players.

MMOGs are extremely complex systems and are thus both time- and money-consuming projects. As an example, *Age of Conan*, released by Funcom in 2008, cost more than 100 million NOK [11]. The development of the game took over four years, starting in 2004. After release, still more money and man-hours are needed to develop the game further and respond to the active players' demands.

As game worlds are vast and filled with events and creatures, players do not need an overview over the entire world, a single player need not be updated about everything that is happening in the game world. The principle of *user perception* means that a player only needs to get information from the events around him. This means that once a part of world is out of sight, the player will not receive any updates from that part. This is also called area-of-interest management.

A game world is usually deployed on a centralized distributed system, such as a cluster or a grid. This system consists of nodes that are connected and can perform a job together, or a workload can be moved between nodes to increase speed. The game world is divided in virtual regions that can be statically or dynamically assigned to a server.

Large end-to-end latencies on the Internet make it hard to establish a constant game quality. As it is impossible to give a perfect, consistent view, it is the perceived game quality that counts. A way to improve game quality can be to reduce latency as much as possible.

2.2 Migration

Migration in a distributed system means that we can move an entity from one machine to another, while the system is running. Such an entity can take different forms. It could be a process, a piece of code, or an object. In this section we will describe a few implementations of migration of different entities. We will work our way down from large entities (virtual machine migration) to migration of processes, objects, code and data.

We will start by listing some of the motivations for implementing migration in different systems. Migration can be used to accomplish several goals. Let us have a look at some of the most important ones.

Dynamic Load Distribution

We can dynamically distribute the load among nodes. If a node in a network is overloaded, we could migrate some of the work over to another less loaded node.

Fault Resilience

When we get an indication that a node will go down, we can migrate the data from the faulty node, to another node.

Resource Locality

We can move a process closer to the resource it uses. This will give us less latency when we contact the resource. For instance, if a client needs to do many computations with data that is stored at a server, the client can send SQL code to the server. The server will then calculate the result, and send the result back. This leads to less data sent between the client and the server.

Moving the computations to the user's client will take away some load from the server. For instance html pages, JavaScript, and Java Applets use code migration to off-load the server, and do the computations at the client instead.

Resource Utilization

When computers are left on unused, we are wasting potential CPU cycles. With migration, we could move work out to computers which are idle. For instance, the SETI@home project [12] uses a screensaver to tell a central server when a computer is ready to accept a job. The server will then migrate a chunk of data for the screensaver to work with. This way we can utilize more machine power which would otherwise be wasted.

System Administration

If we need to disconnect a node for maintenance, we can migrate all the running tasks to other nodes first. Then users will not be affected by the downtime.

Response Time Reduction

If we have more than one server, we can migrate objects to the server which is closest to the client, and thus reduce the clients response time.

2.2.1 Virtual Machine Migration

Virtual machines is a software implementation of a real machine, which can execute programs just like a real machine. The virtual machine acts as a layer between the machine and the running programs. This makes it possible to run the same programs on virtual machines one many different operating systems and hardware configurations.

In [13], a system that implements virtual machine migration is proposed. This system makes it possible to perform fast, transparent application migration. Neither the application or the clients communication with it can tell that the application been migrated.

In [14], another system which implements virtual machine migration is proposed. The migration happens so fast that the impact on the users is minimal.

2.2.2 Process Migration

A process is a key concept in operating systems [15]. It is an abstraction of a running program, consisting of a memory space, the executable program, the program data, its stack and a set of registers (program counter, stack pointer, and so on). When we want to migrate a process, we need to capture all of the data and state of the process. This is quite complex, as parts of the state depend on the operating system. The part of the state that are hard to capture could be open files, communication and kernel context.

The different kinds of process migration can mostly be summarized into the following steps [16].

1. A request for a process migration is given to a node. The node then accepts.

2. The node suspends the process, and declares it to be in a migrating state.

3. All the incoming communication to the process is queued, and forwarded when the migration is done.

4. The state of the process is extracted. This includes all the memory content, the registers, communication state (open files and message channels) and relevant kernel context.

5. A process instance is created at the destination node.

6. The state of the migration process is inserted into the process instance at destination node.

7. The communication with the original process has to be redirected to the new one. This can be done in different ways, and depends on the implementation that is used.

8. When enough state information has been transferred to it, the new instance is activated. When the complete state has been transfered from the old process, the old process can be deleted.

We will now have a short look at some systems that implement process migration.

DEMOS/MP

DEMOS/MP [17] was one of the first systems to implement fully transparent process migration. It is a message based operating system that uses links to send all messages between the different parts of the system. Since the system uses links, it avoids problems with open files and communication when we migrate a process. All the messages sent to the migrated process are forwarded by the kernel to the new location.

Sprite

Sprite [18] is a network operating system. It is intended to be used in a network of workstations. When one or more of the workstations are idle, Sprite will migrate processes from heavier loaded workstations to the idle ones. When the workstation is no longer idle, i.e., someone starts using it locally, the process is immediately returned to the original workstation. A process will always migrate from the workstation it was started on to an idle one, or back to the original workstation. The migration is fully transparent, i.e., the system hides from the user if a process is handled locally or at another workstation. The Sprite project has reported speedups for tasks such as compiling and simulation, in which programs exploit coarse-grained parallelism.

Mosix

Mosix [19] is a distributed operating system, designed to give the user a single UNIX-like environment on top of a cluster of nodes. The Mosix kernel consists of three layers. The lowest layer is a microkernel which deals with the hardware. The upper layer provides the standard UNIX system call interface. In the middle, there is a *linker* which connects the other layers. The upper and the lower layer can reside on different nodes.

Mosix implements a load balancing algorithm that moves processes from loaded nodes to other less loaded nodes. The algorithm used is a suboptimal algorithm, which means that it does not consider all information before it decides to migrate. This uses a lot less resources than an optimal algorithm.

2.2.3 Object Migration

Object migration is the act of moving an object from one node to another, during execution of a distributed program.

Object migration consists mainly of two parts: First we need to physically move one object from a node to another. This is usually done with serialization or marshalling of the object. Second, when the object has been moved, we need to tell the system where it can find the moved object, so it can continue to interact with it. This can be done with a *name service*, which is updated when objects are migrated.

Emerald

Emerald [20] is both an object oriented language and a run-time system (kernel).

Communication with objects happens through invocations. Emerald has two different invocations: The default is call-by-object-reference. This is a remote method invocation. The other invocation is call-by-move. This involves moving the object to the caller, so the remote method invocation is avoided. The programmer decides himself which invocation he wants to use.

An Emerald object consists of four parts: A unique identifier (network-wide). A representation, which is the data stored in the object. A set of operations, which is the procedures and methods the object provides. And it can optionally contain a process, which is executed in parallel with invocations of the object. An object with such an process is called active and an object without is called passive.

Both active and passive objects may be migrated. In addition to the Call-by-move invocation, objects can also be moved by a general construct.

This way we can move an object or a group of objects to a specified host, or to the host of a specified object.

Chorus/COOL

The Chorus Object Oriented Layer (COOL) [21] was developed in response to the difference between the operating system abstractions (processes) and the object oriented abstractions (C++ objects and similar). The aim was to develop a set of generic services, at the kernel level, to support multiple object oriented systems. These services should co-exist with the traditional operating system, so that the programmers might use existing data and services.

COOL supports a network transparent invocation model, in which objects are accessed via local identifiers. Object may therefore migrate freely, in a manner that is transparent to the programmer.

COOL consists of a number of layers built on top of the Chorus microkernel. In the system-space, the COOL-base layer provides support for object-oriented systems. Above this layer, in user-space, we have the Generic Run Time (GRT) interface. The GRT provides basic abstractions such as object management, invocations and activities. It is designed to support multiple programming languages, and is implemented as a library, which is linked with the applications.

The programmer only sees local memory pointers, and as such the remote access calls proceed via a locally residing interface object or proxy. These include code to either give remote objects access via Remote Method Invocation (RMI), or to map the object into local memory.

2.2.4 Code Migration

When we use process migration, we move both the code and the data of a process. When we use code migration, we move only the code.

Web Applications

Code migration is used heavily on the internet. Web programmers want to take load off the web servers. Therefore they make web sites that use Java applets, Flash or JavaScript. This kind of code is migrated to the user who browses their site, and then the code is executed at the users computer.

SQL

Another approach to code migration is a client-server system where the server has a large database, and the client wants to perform many calculations on the data. Then it can be useful to send code for performing all the calculations from the client to the server, do the calculations at the server, and return the result. This way we will use less bandwidth. The code for performing the calculations could in this case be SQL.

Windows Driver Migration in CUPS

Another reason to implement code migration can be flexibility. For instance the Common Unix Printing System (CUPS) printing daemon can be configured to dynamically send the drivers needed by a Windows machine to use a printer, when the Windows machine first connects to CUPS [22]. This way the person who owns the machine will not have to figure out himself how to get the right drivers, but it will function automatically when he connects for the first time.

2.2.5 Data Migration

If nodes share the same code base, an advanced form for object migration can be supported. When the code is shared, the node on the other end already "knows" what an object looks like. This means that the object does not have to be sent physically. We only need to migrate the data, which is used to re-create the object on another node. In [23] a middleware that supports migration by only sending the data in an object is proposed.

In this middleware, each object is kept track of with a distributed name service, and every new node connects to a known server and gets an updated name service table. The nodes all share the same code, so only the data of objects is serialized and sent. The middleware uses a type register to identify which type of objects which is migrated (since only the data is received).

Every object has to implement the abstract class *Object*. This gives every object an identifier which distinguishes it from all the other objects. Objects will also need to implement a *deflate()* function, which packs down the data from the objects when we migrate.

When an object is migrated it sends its new identifier back to the original node, which registers the new id as a care-of address of the objects in its name service. This way the objects can be reached after migration.

2.2.6 Summary

Different situations will ask for migration of more or less complex entities. Virtual migration, where an entire OS is moved, is easier than process migration, but it is a slow and heavy operation. Progress migration is the most complex operation to perform, since all data and state of a process must be moved in the right order.

Obviously, migration will be faster and easier when it comes to less complex entities as code or data. Code migrating works well on the internet, where programmers can use it to off-load their own servers, or make services easier available for clients. In systems where nodes share the same code base, the migration process could be speeded up by only migrating data. This could work well in distributed interactive applications, since they have a strict requirement for low pair-wise latency, which means that we want the migration to finish as fast as possible.

2.3 Core-Node Selection Algorithms

If we want to migrate objects, we need a way to find suitable destinations. To help find the nodes we should migrate to, we can utilize core-node selection

algorithms.

Core-node selection algorithms are algorithms that searches for nodes that provides certain attributes within a given set of nodes. Many different code-node selection algorithms have emerged, since there are different needs and requirements for the core-nodes. These requirements can be low latency to a group of clients, or it could be to find a core-node with low work load.

In [24] it is stated that a core-node algorithm is functional if it can identify one or more nodes in a network that yield a desired property. The different core-node selection algorithms use different heuristics for what they search for.

Distributed interactive applications have very strict requirements for latency. In this setting, we could benefit from using core-node selection algorithms to find well-suited nodes that could handle sub-groups of clients. These core-node selection algorithms have to search for nodes that yield low pair-wise latencies to the sub-group they should handle.

2.3.1 Algorithms

We will now have a look at the algorithms that fit best for selecting core-nodes in distributed interactive applications [24]. To understand these algorithms we will need to explain some of the terminology that is used in graph theory.

Terminology

A graph is denoted by $G = (V, E)$, where V is vertices (nodes in the network) and E is edges (links in the network). An edge $e \in E$ between $u, v \in V$ is denoted $e = (u, v)$. A graph can be directed or undirected. If G is directed, each edge has a direction. If G is undirected each edge is undirected, which is the same as a bidirectional graph.

Vertex degree is the number of edges connected to a vertex. It is written as $deg(v) = n$ where n is the number of edges connected to v.

A weighted graph is denoted $G = (V, E, c)$, where V is vertices, E is edges, and c is a edge cost function $c : E \to \mathbb{R}^+$. Each $e \in E$ has a

cost $c(E)$.

The distance between $u \in V$ and $v \in V$ is the minimum cost of all the paths that lead from u to v. It is denoted $dist(u, v)$.

The eccentricity of $u \in V$ is the maximum distance from u to all $v \in V$ $(dist(u, v))$. It is denoted $ecc(u) = maxdist(u, v) : v \in V$.

Pair-wise latency is the latency between one node and another. This is usually measured as the round trip time one package uses from one node to another, and back again.

The core-node set consists off all the nodes that potentially could be used as core-nodes.

The member-node set is the group of nodes that we want to find the best core-node for.

k-Center Vertex Core Node Selection Algorithm

The center vertex core-node selection algorithm (Algorithm 1) [24] finds the core-node which has the lowest maximum distance (eccentricity) to any other node in the member-node set (Z). This algorithm has time complexity of $O(n^2)$, when all the needed data is known at the node that does the calculations.

k-Median Vertex Core-Node Selection Algorithm

The median vertex core-node selection algorithm (Algorithm 2) [24] finds the core-node which has the lowest average pair-wise distances to the nodes in the member-node set. This algorithm has time complexity of $O(n^2)$, when all the needed data is known at the node that does the calculations.

To give an example of this algorithm, consider the graph in figure 2.1. Here we have six nodes, which makes the set V. Node one to four make set $Z \subset V$. This is the member node set. Node five and six make set $X \subset V$, which is the core-node set. We will then calculate the pair-wise latency from

Algorithm 1: k-center(G)

Input: A graph $G = (V, E, c)$. Sets $Z \subset V$ (the member node set) and $X \subset V$ (the core-node set).
Output: c, which is the optimal core-node

```
1 map<id, eccentricity> mapIdEcc;
2 foreach x ∈ X do
3     T = shortestPathTree(x, G);
4     v = maxEccentricityNode(T, Z);
5     ecc = getEccentricity(v, T, Z);
6     mapIdEcc.insert(x, ecc);
7 end
8 c = lowestEccentricity(mapIdEcc);
```

Algorithm 2: k-median(G)

Input: A graph $G = (V, E, c)$. Sets $Z \subset V$ and $X \subset V$.
Output: c, which is the optimal core-node

```
1 map<id, pair-wise> mapIdPairwise;
2 foreach x ∈ X do
3     pairwise = getPairwiseDistances(x, Z);
4     mapIdPairwise.insert(x, pairwise);
5 end
6 c = lowestPairwise(mapIdEcc);
```

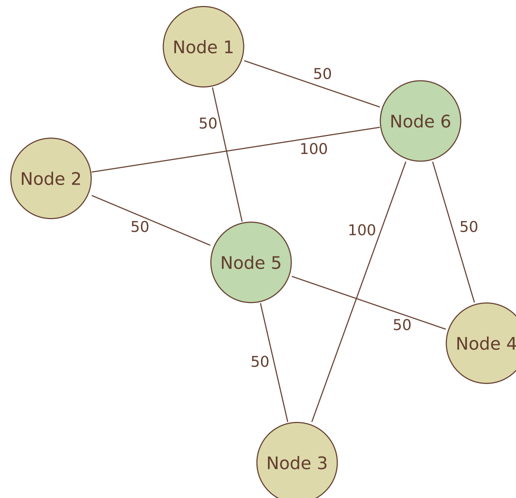

Figure 2.1: Network Graph

Node	Latency
Node 5	200
Node 6	300

Table 2.1: Results from running k-Median algorithm on graph V

each node in set X, to all the nodes in set V. This leads to the following results shown in table 2.1.

With $k = 1$, this core-node selection algorithm will result in node five.

2.3.2 Obtaining Link Latencies

To be able to perform core-node selection algorithms on a network, one needs to obtain the link latencies between the nodes one is interested in. There are two standard metrics for link latency:

Round-Trip Time (RTT) is the time it takes for one package to go from one node, to another, and back again. The RTT is usually used with reliable protocols like TCP. RTT can be measured from from a single node.

One-Way Transmit Time (OWT) is the time it takes for one package to go from one node to another. OWT is usually the metric used for unreliable protocols like UDP. To measure OWT, we need access to both the sending and the receiving node.

Latency measurement

There exist several tools to measure the latency on links on the Internet.

Ping measures the RTT of a link by sending a ICMP “echo request” packet to the target node. Then, it listens for a ICMP “echo response”, and measures the time since it sent the request. It will also record packet loss.

Traceroute sends out packages in batches of three. For each batch, it increases the *time-to-live* (TTL) value with one. The first batch have

a TTL value of one, the next a TTL value of two, and so on. For each node the packages reach, the TTL decreases by one, and when it reaches zero, it sends a package back to the node running traceroute. This way, one can measure the RTT to each node the packages goes through, on their way to the destination host.

Latency estimation

Using the abovementioned tools to get latencies between many nodes will create a lot of overhead. Since sending too many packages usually is not wanted, there are several tools which measure only some of the latencies, and estimate the rest. Most of these tools try to minimize the probing overhead.

For example, *Netvigator* [25] is a scalable network proximity and latency estimation tool. It uses information obtained from probing a small number of landmark nodes. When it probes the landmark nodes, it will also discover intermediate routers (termed milestones). Netvigator uses the distance information to the milestones to calculate the latencies between all the nodes.

2.3.3 Summary

In this section we have looked at two core-node selection algorithms that searches for nodes that yields low pair-wise latency to different groups of nodes. We have also shortly discussed different ways of obtaining the latency information one needs to perform core-node selection. In the next section we will look at how this functionality, and other functionality developers might need, can be provided in a more convenient way, with the help of a middleware.

2.4 Middleware

Building a distributed system is a complicated task. To help the developers concentrate on the application itself, and to hide away some of the complexity of the system, we often use a layer which sits between the operating system (OS) and the application.This layer is called *middleware* (see figure 2.2).

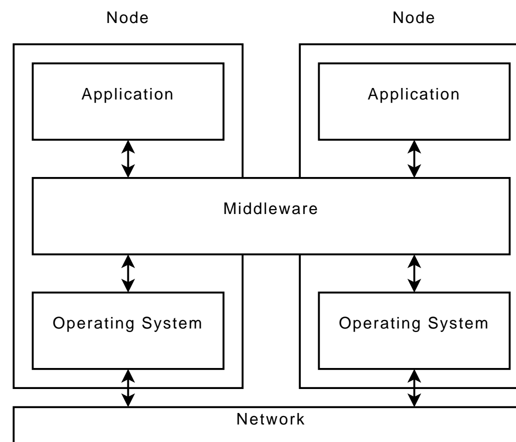

Figure 2.2: Middleware

In a distributed system, the middleware should ideally perform the following tasks:

- Make distribution transparent, i.e., hide the fact that objects reside on different machines.
- Hide the difference between multiple operating systems, hardware and communication protocols.
- Provide a simplified interface and functionality that programmers can work with.

In the following, we will focus on the first task, i.e., making distribution transparent. When all calls to objects are sent through the middleware, the programmer does not have to worry about an object being local or not. This behavior is called transparency.

In distributed systems, objects are distributed across a network that consists of multiple nodes. To be able to interact with objects on other nodes, middlewares created for distributed systems implement remote method invocation (RMI). RMI makes us able to call objects which reside on other nodes, as if they were local. To make this work, we need a way to keep track of objects in a distributed system. This can be done with a *name service*.

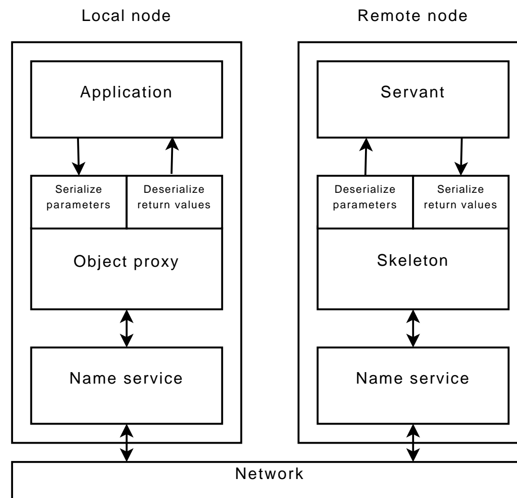

Figure 2.3: Remote Method Invocation

2.4.1 Remote Method Invocation and Name Service

A name service keeps track of all objects residing in a system. Usually one uses references to the local memory when pointing to an object. However, when a name service is used, references to objects are names that the name service can be queried with. It contains information about where we can find the objects we need, i.e., if they are local or remote; and if they are remote, on which machine they reside.

When an object is called, the middleware has to intercept the call and find out where the object resides. The middleware will check the name service, and get the host of the object. If the object resides on the same machine that called it, the middleware just forwards the call to the local object. If the objects resides on a remote node, the middleware creates a *proxy object* (called stub in Java RMI) which packs down the parameters and sends them to a *skeleton* on the remote node. The process of packing down data is called serializing, the un-packing is called deserializing. This process is shown in figure 2.3.

The skeleton deserializes the parameters and calls the object. It then serializes the return value from the object, and sends it back to the calling proxy object. The proxy object deserializes the return value and returns it

to the caller.

This is the basic outline of an RMI process. In the next chapter, we will discuss how we have chosen to implement the different parts in our middleware. In 4.2 we will see how a call is being processed through the middleware step by step.

2.5 Summary

In this chapter we discussed some of the background for our research. At first, we have seen motivations for implementing migration and core selection in a distributed system.

In massive multiplayer online games, a persistent game world is created. These games are characterized by social interaction, i.e., players work together and respond to events in the game world. It is important to reduce latency, so that all the players are constantly updated about the state of events in their vicinity.

Amongst other uses, migration can be used to distribute a workload evenly, to reduce latency, or to off-load busy servers. One can move entities such as data, code, objects, processes or entire operating systems to other nodes in the system. Migration of data is the fastest operation, but requires that all nodes share the same code. This kind of migration could be used in distributed interactive applications.

Core-node selection algorithms can be used to find the best nodes to migrate objects to. These kind of algorithms are designed to pick the optimal node(s) in a system to perform a certain task. To be able to use core-node selection algorithms on nodes in a network, the link latencies between the nodes have to be measured or estimated first.

Middleware can be designed to to reduce the complexity of distributed systems for programmers working on it. Such a middleware should provide programmers with a transparent, simplified interface that works on different operating systems and communication protocols, and hide the fact that objects reside on different machines. To do this, one needs a name service to keep track of all objects in the system. With this in place, remote method

invocation can be used to call objects that resides on other nodes, as if they were local.

In the next chapter, we will look more closely at the design of a middleware that can help reduce latency in a distributed interactive system like a MMOG.

Chapter 3

Design

In this chapter, we will describe the design choices we made for our middleware. This middleware has to be able to dynamically relocate objects to the best server, for a given set of servers. To find these servers, we want to use core-node selection algorithms built into the middleware logic. By the best server, we mean the server which will give all the connected clients the lowest possible pair-wise latency. To support relocation, our middleware will have to support migration. These requirements mean that we had to face some complex design choices.

We will start by listing the requirements for a middleware, if it is to be used in an MMOG. After that, we will go through different elements in the middleware and discuss design alternatives and explain our choices.

3.1 Requirements

MMOGs have very strict demands to the latency from the servers to the clients. We need to make sure the latency the users get is as low as possible. The middleware thus needs to relocate the objects the clients are communicating with, to servers that give the clients lower pair-wise latencies. It will therefore need to support relocation of objects.

Since we want the middleware to relocate objects by itself, it needs to be able to find a new server that serves the clients better than the current one.

We will use core-node selection algorithms to accomplish this.

To be able to find a better node for a object, the middleware needs to know which clients communicate with the object and are interacting with it. If the object depends on other objects, the middleware will also have to keep that in mind when making a decision to migrate it.

In short, our middleware will need to support the following:

Remote Method Invocation
: Objects can be called from both the same and other nodes that the object resides on.

Migration
: Objects can move freely from server to server.

Object Groups
: Objects that belong together should be grouped, and migrated together.

Core Node Selection Algorithms
: Find the best server for an object or a group of objects to migrate to.

Overlay topology
: Knowledge of the overlay topology is necessary to be able to perform core node selection.

Save dependency information between clients and objects
: The middleware needs to know which objects the different clients interact with. This information is needed to perform core-node selection.

In the following, we will go through a number of design choices we have made to meet these requirements, starting with serialization.

3.2 Serialization

As we stated in section 2.4, the middleware should ideally hide the difference between different machine architectures. Thus, our middleware's messages

need to be readable by all types of machines, independent of different hardware, software and OS configurations. This means that we must account for that machines can have different *endianness*. When a value that takes up more than one memory location has to be stored in the memory, there are two ways to do this. We can either store the most significant byte (MSB) at the highest memory address (little-endian), or we could store it with the MSB at the lowest address (big-endian). We will also need to agree to how to store the different data values that our programming language supports.

To accomplish this we have decided to use *eXternal Data Representation* (XDR) which is an IETF standard [26] from 1995. This standard documents an architecture independent way to store data. XDR is also a widely distributed and implemented standard, that is available on most systems. This makes it easier to support different operating systems, if we would want to support other systems at a later point.

When we follow this standard to write data to a buffer at one host, we can send it to another host, and reconstruct the exact same data. This will work even if the receiving host is of another architecture. In GNU/Linux, there is a library made by Sun Microsystems called *SunRPC* which supports writing to and reading from XDR buffers. Writing data to a buffer in an architecture independent way, is called *serializing* the data. Reading the data from the buffer is called *deserializing*.

3.3 Remote Method Invocation

We will implement RMI in our middleware much like the way we have described in section 2.4.1, with a local proxy object that takes care of serializing the parameters, redirecting the method call to the node that contains the original object, using the name service to locate it, and deserializing the return value.

The support for migration in our middleware complicates the RMI implementation, and makes it necessary to consider a few more scenarios than if we would have used RMI in a middleware without migration support. Since objects can relocate, we need to be able to handle a RMI request which comes

to a node after the object has migrated to another node.

One way to do this, is for the node that gets an RMI request for an object that has migrated, to send a message back to the caller telling where the object now resides. The caller will then have to send a new RMI message to the updated address. Another way is forwarding the RMI message to the new node which now contains the object. We have chosen to forward the messages, since we then end up with fewer messages sent.

When a node gets a forwarded RMI message, it is important that it also sends information about the new location of the object when it answers the original RMI, so we do not have to keep forwarding the RMI messages for the caller anymore.

A RMI call will also have to be a blocking call for the application, since the application cannot proceed with execution when it is waiting for a return value. This could lead to a complication, if two servers try to perform RMI on each other at the same time. If both servers block and wait for each other's answer at the same time, we have a *deadlock*. To avoid this deadlock situation, we have designed a variant of RMI which is nonblocking. We have called this *event* distribution and will explain this in the next section.

3.4 Event distribution

Event distribution works nearly identical with RMI, except for that it is not a blocking call. Since a call to an event is nonblocking, it can not be called to a method which gives a return value.

Many of the updates that a server need to give clients in an MMOG are small updates like the position of other avatars. These are updates that do not have a return value. If they are sent as an event, this makes the process faster.

3.5 Middleware control messages

Our middleware will also need to be able to communicate with the middleware on other nodes. Information like pair-wise latency, and lists of proxies will have to be shared among the nodes running our middleware. We could have implemented this information sharing as RMI calls between the middlewares, but since RMI in our middleware is designed to be a blocking call, this could potentially lead to deadlocks. We also want this information sharing to be as fast as possible.

Therefore we have decided to let our middlewares use specialized messages for all its communication with each other. This way sending a list of proxies to a client will be one message, and a RMI call from a client to a server will be another one. This takes away one level of indirection that we would have had if we used RMI for this communication.

3.6 Migration

We want our middleware to be able to relocate objects to other nodes. It should be possible to perform migration of objects during execution of an application. Our goal is to make this relocation as transparent to the user as possible. One of the ways we can accomplish this is by sending as little data as necessary when objects are migrated. We have decided to let all nodes share the same code and libraries. This way we only have to send the data values in a object when we need to migrate it. This gives us less overhead during migration, but it also means that the code base will have to be consistent on all nodes.

When we migrate a object, its data will need to be packed down (serialized) into the XDR buffer, which will make the object unreachable for a short while. We need to make sure that RMI calls and event updates for the object are not lost while it is in this migration state. We will accomplish this by buffering all the incoming messages for the object until migration is completed, and then send the messages to the new host.

It is also vital that migration of objects interacts with our name ser-

vice. When objects relocate, the name services on the sending and receiving node will have to be updated. We will discuss the name service further in section 3.8.

3.7 Code and Library Versions

To be able to perform RMI and migrate data from objects, we need to be sure that all servers and clients are running the same version of the application. Our middleware will not have support for migrating entire objects, i.e., it supports only migration of data in objects that share the same code base and interface. To make sure that migration is possible, we we need to check that the version of the application is the same on all nodes.

This can be done in two ways. Either all objects can keep track of the version number of the library they where created with. We will then have to append this number on every RMI and migration message. This way we can check that a RMI or migrating object is compatible with the libraries at the receiving node. The other alternative is to have a global version number on the entire application, and check it against the new nodes we connect to.

When a client connects to a MMOG, it will usually connect to a single server where it has to identify itself. Most MMOGs will then check which version of the application is being used, and ask to upgrade if it is not the latest. This makes sense because MMOGs never stop being developed and often add more content, which will have to be updated by all the clients.

If each object keeps its own version number, we could possibly upgrade the parts of the application that are not used for migration or RMI, without having to upgrade the library version of all the clients and the servers.

On the other hand, if we check version numbers when we connect to a new node, there will be less overhead when we perform RMIs and migration, since we do not have to send the version numbers as part of the messages. We can also remove the version check on the receiving nodes RMI and migration code.

We have chosen to check the version of our code libraries when we connect to a new node, since this alternative generate as little overhead as possible.

3.8 Name Service

In a distributed system, we need a name service to find out where objects reside. When we only work locally, we use a pointer to an object to get to the data and methods an object provides. This pointer points to the location in the memory where the object resides. But such pointers cannot point to another node's memory, so we need another way to figure out where objects reside.

The name service makes it possible to to keep track of objects regardless of where they reside. It is a table which maps an object identifier either to a local object, or to another node if the object is remote. To make this work, we need to be able to provide objects and nodes with a unique identifier.

3.8.1 Identification

Creating identifiers that are guaranteed to be unique is a hard task to perform on a computer. One way to do it, is to use certain system variables and combine those with a random number (for example, use the mac address of the node that creates an object, together with a time stamp and a random generated number.) But if many objects are created in a short period of time, we can not guarantee that two of them will not get the same time stamp and random number (though this is not very likely).

There is also a standard for creating unique identifiers for distributed systems defined in the *Open Software Foundations* OSF DCE 1.1 [27]. This standard describes how to create *universally unique identifiers* (UUIDs). A UUID is a 128 bit number. There are five versions of UUIDs, the difference being how the number is created. Some of the bits are reserved to describe which version of the standard the UUID is. Version 4 of the UUID standard uses a random number generator to create 122 of the 128 bits. If a cryptographically secure pseudorandom number generators is used, the chance of creating two equal UUIDs is as low as 4×10^{-16} after the creation of 2^{36} UUIDs.

We have chosen to use UUIDs since they are widely supported on different platforms, with existing libraries to create UUIDs in many programming

languages.

To uniquely identify nodes, we have chosen to use their IP addresses and listen ports. Since two applications can not use the same port at once, this makes for a unique identification. It will also make it possible to run two incarnations of an application at a node, and still be able to differentiate between them. This will also tell the middleware how to contact the node.

3.8.2 Distribution

In [28], Znati and Molka have analyzed different ways a name service can be implemented in a distributed system. We will here shortly summarize the different distribution schemes.

One way to implement the name service is as a central service at a defined server. In this case, every client connects to the central server when it needs to contact an object, and the name service tells it where the object is. The benefit of having just one name service, is that it is easy to keep the service consistent. The name service will always be up to date.

On the other hand this means that every client will need to contact the server with the name service, every time it needs an object. If there are many objects in the system, the name service server could easily become a bottleneck. Also, if there is an error with the name service server, the whole system could potentially stop working.

Another approach is to distribute the name service between every client and server in the system, so that they all have a local copy of the name service. This will remove the potential bottleneck from the system. But this solution as a whole is more complex, and it will be harder to always keep the name service in a consistent state. There are also different hybrid approaches to the name service, i.e., a combination of distributed and centralized implementation.

We have chosen to use the solution that works fastest, i.e., we have distributed the name service to all clients and the servers.

3.9 Core Selection Algorithm

One of the main objective of our middleware is to lower the pair-wise latency for the clients. We want to achieve this by using a core-node selection algorithm to find which proxies or server that is best at the moment. We also want the middleware to perform core-node selection dynamically.

We have chosen the *k-Median* algorithm described in section 2.3.1. This algorithm uses the pair-wise latency between the nodes we want to find a core-node for, and the potential core nodes, to find the core-node that yields the lowest total latency. In [24], Vik has performed simulations and experiments with many different core-node selection algorithms. k-Median is concluded to be a good algorithm for finding core-nodes that yield an overall low pair-wise latency in distributed interactive applications.

The algorithm needs to know about all potential servers and proxies, and it needs to know which clients we want to find the best server or proxy for. We need to decide if we want the core-node selection to be performed for all clients connected to a server, or if it should be performed for groups of objects.

To make this decision we need to consider our middleware's target applications, which are MMOGs. As we described in section 2.1, clients in an MMOGs communicate with the avatar object which resides in one virtual region. We want our middleware to support having more than one virtual region at a server, and, at the same time, be able to perform core-node selection on just one virtual region. This implies that we have to save information about which clients communicate with which objects. We will discuss how we can save this information in the next section.

For the core-node selection to happen dynamically we have to decide when to perform core-node selection. We could either perform core-node selection when certain conditions occur (i.e., when a new client connects, or when a new object is created for a client), or we could perform core-node selection at a given time interval.

Since it will take some time from the moment a client connects, until the information about network latencies is received, and since many clients can

connect and disconnect in a short period of time, we have decided to perform core-node selection at a given time interval, which can be adjusted by the programmer.

Another thing we need to consider is when the result from the core selection is good enough for the middleware to consider migrating objects to another server. If we have a short time interval between each invocation of the core-node selection, and we let the middleware migrate objects whenever it finds a better server, we might end up migrating objects back and forth between servers. The overhead of this operation might not make it worth it. To decide when we should initiate migration, we have decided to let the middleware have a threshold value. It will only perform migration when the results from the core-node selection are better than the threshold.

3.10 Grouping of Objects

Our middleware needs to support grouping of objects that depend on one another, and that should reside on the same server. To handle this information, we have designed object groups that save this information. Since these groups should be migrated together, this is also a good place to save information about which clients need the objects in the group.

The clients in our middleware will have to send information about which objects they need to be able to communicate with. This is gathered by the object groups at the server. With this information, the server can perform core-node selection and find the best node for a whole group of objects.

3.10.1 Overlay Topology

The core-node selection algorithm needs information about the pair-wise latency between clients and the server and proxies. This information should be accessible for the servers and proxies that want to perform core selection.

There are two alternatives as to where we can store this information. Either we can let all the proxies and the server save this information, or we could keep it at one central server. If the information is kept at a single

server, each of the other proxies and/or the server will have to query this node when they need to perform core-node selection.

We have decided to let all the proxies and the server have one network graph each. With this solution, all proxies and the server can perform core selection locally. This also means that the clients need to update all the proxies and the server with this latency information. It should also be noted that this information only need to exist on the server and the proxies; the clients do not need it.

3.10.2 Latency Measurements

The core-node selection algorithm depends on data about pair-wise latency between clients and the potential proxies and the server. This data can be collected in different ways.

As we discussed in section 2.3.2 there are different programs which can be used to measure the latency. If we want to support a large amount of clients, we should use some sort of latency estimation tool, since this will result in a lower number of packages sent on the connections. Since we are making a prototype, we have decided to use the ping tool which is standard program with most GNU/Linux distributions.

Every client will have to call ping and measure the pair-wise latency between itself and all the proxies and the server. Since the internet is an unstable network, this information will have to be updated once in a while. We have also decided to let ping send five packages to each server, and use the median value as the current latency information it sends to the server. This is because sometimes one single package can take a lot longer than the rest of the packages to go back and forth between the client and the server. We use the median, and not the mean, because sometimes we get one high round trip time value, and the mean would be skewed. Using the median with five packages makes us more resistant against getting a to high mean value.

3.11 Internet Sockets

Our middleware uses internet sockets to communicate with other nodes. We have decided to use TCP for communication, since we then do not have to think about retransmission of messages. To improve our middlewares performance, we have decides to set the sockets in a nonblocking mode.

When data is sent over a normal blocking socket, it will not be possible to call send again before all the data in the first call has been sent. If we send data faster than the socket can handle, we will be required to wait for it to send all data. Even if there would be another socket that could accept data, we will have to wait. We will not be able to fully utilize the potential of the data sender, since it will stay blocked for large periods of time.

If we use nonblocking sockets instead, each call to write data to a socket returns at once, whether or not all data was sent. If not all data is sent, we will be informed about how much of the data still needs to be sent. We we can start to send data on another socket, and get back to the full socket later. This way the data throughput will be much better. A downside to using nonblocking sockets is that the code gets more complex, since we get a lot more state to keep track of.

Chapter 4

Middleware Implementation

We have implemented a prototype of the middleware which supports migration, remote method invocation, and core-node selection. Using this middleware we have implemented a simple interactive distributed application, which we have used for testing the described mechanics in a live system. In this chapter we will discuss the implementation of the middleware. In the next chapter we will go through our test application.

To explain the implementation details of our middleware, we will go through three examples on what the middleware does. First, we will go through the steps of a remote method invocation. Second, we will describe how core selection is performed. Last, we will describe how migration of objects and groups of objects work. As we go through these examples, we will explain how components of the middleware interact and are implemented, in the order we encounter them. But before we get to the examples, we will list the main parts of the middleware and explain them very shortly.

4.1 Middleware Main Components

Our middleware makes a distinction between three different kinds of nodes, these are *server*, *proxy*, and *client*. The server is the main connection point, and every other node starts by connecting to the server. The proxies are meant to off-load the main server. Objects will migrate between the server

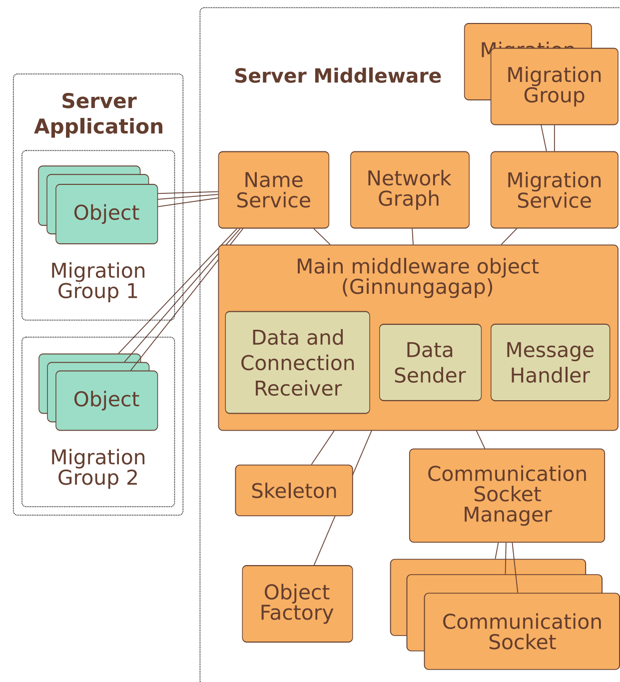

Figure 4.1: Server/Proxy Middleware Overview

and the proxies, based upon the latency information gathered from the clients.

There is only one server, but there can be an unlimited amount of proxies and clients, as long as the hardware of the server and the proxies can support it. The clients are the end users, and they connect to the server and all the proxies.

Proxies, clients, and the server all utilize the same code, with a few exceptions. The main class in our middleware is called `Ginnungagap`[1]. This class is implemented as a *singleton*, meaning that there can only be created one instance of it in a running application. There is a static method that returns a pointer to this one instance, and this serves as the *main entry point* to the middleware. Every call the application, or another part of the middleware needs to perform on the main object, goes through this pointer.

[1]A term from Old Norse mythology, Ginnungagap is what exists before the world is created; it is envisioned as an enormously wide gap, filled with energies but lacking form or order.

The Ginnungagap object has implemented five methods which run in infinite loops. Each methods runs in its own thread. These five methods are:

- `dataAndConnectionReceiver()`
  Receives messages from other nodes, and handles new incoming connections. Needed by all kinds of nodes.

- `dataSender()`
  Sends data to other nodes. Also needed by all kinds of nodes.

- `messageHandler()`
  Reads and handles the messages received by the data and connection receiver. This method is needed by the clients.

- `messageHandlerWithCoreSelection()`
  Similar to the message handler, except it also initiates the core-node selection algorithm at a given interval. This method is needed by the proxies and the server.

- `latencyMeasurer()`
  This method is important for the core-node selection to function. Needed by the clients to measure latency to the server and proxies.

The main object has pointers to all the other classes that the middleware needs. These classes are:

- `NameService`
  The name service keeps track of where objects reside. This object is used by all nodes. We will explain the name service in more detail in section 4.2.2.

- `CommunicationSocketManager`
  This class keeps tracks of all open connections to a node. It is also responsible for opening new connections if needed. All nodes need this object.

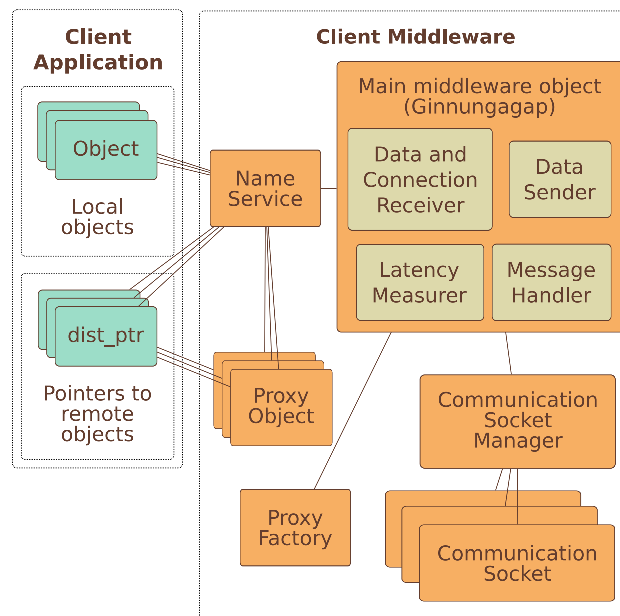

Figure 4.2: Client Middleware Overview

- `MigrationService`
  The migration service class handles migration of objects. This will be explained in section 4.4.3.

- `NetworkGraph`
  The network graph keeps track of the overlay network topology, and the latency information. This is also the class that performs the core-node selection algorithm. The network graph is further explained in section 4.4.5.

- `ObjectFactory`
  The object factory creates new, empty objects when the middleware gets an incoming migrating object. It determines the type of the object to be created dynamically. These objects are filled with the data from the migrating object.

- `ProxyObjectFactory`
  The proxy object factory creates new proxy objects when they are needed. Proxy objects are local placeholders for remote objects.

- `Skeleton`
  The skeleton handles RMI messages at the receiving end. We will explain how RMI and skeletons are implemented in section 4.2.

Figure 4.1 shows all the objects in the server and proxy version of the middleware. It also shows the threads that run in the main object. The version the client runs can be seen in figure 4.2.

4.2 Remote Method Invocation

Our middleware is designed to support RMI. An object can both be called from the node it resides on, or from a different node, using a proxy object. The middleware redirects the call to either a local or a proxy object, making the process transparent for the programmer.

RMI needs many different components to work. To explain our implementation, we will go though how a RMI is performed with our middleware. We will use an example object from the test application, which is described in chapter 5. Knowledge about the test application is not necessary to follow this example.

In figure 4.3 we see an overview of the RMI call we shall explain. A client calls the method `move(NORTH)` on an avatar object. The object resides on the server, and it has a *distributed pointer* pointing at it. We will start by explaining how this distributed pointer works. As can be seen on the figure, the RMI call starts at the dist_ptr box.

4.2.1 Distributed Pointer

Since we do not know where a object resides when we want to invoke one of its methods, we need a way for the middleware to intercept the call to the object. This is the indirection mechanism that we discussed in 3.3. In our middleware, this indirection mechanism is implemented as a *distributed pointer*.

Our distributed pointer is a *smart pointer* that can point to both local and remote objects. A smart pointer is a common programming idiom, that

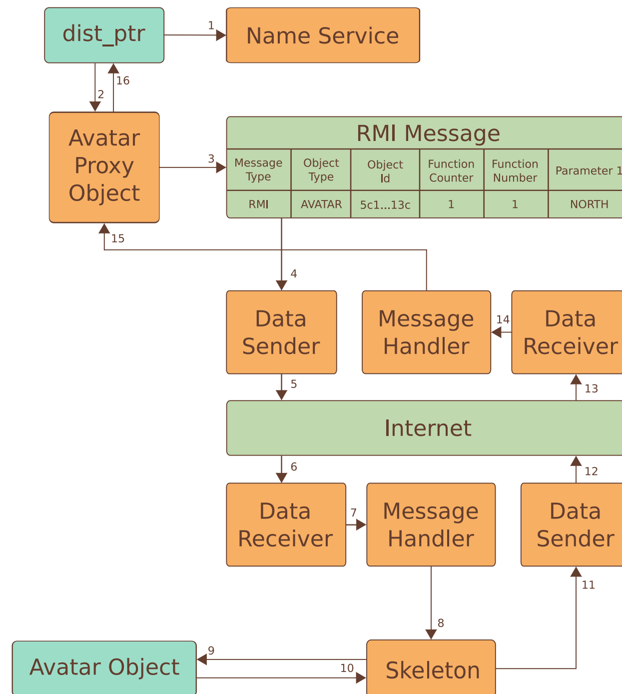

Figure 4.3: A remote method invocation example

is used often in C++. Instead of using a raw pointer, we hide the raw pointer inside an object. This makes it possible to extend the functionality of the pointer beyond that of a raw pointer. One of the common uses for this functionality is for the smart pointer to count references to an object (i.e., how many pointers that point to the object). When there are no pointers to the object left, the object can be deleted.

Our distributed pointer saves the object identifier to the object it points to. Upon every call to the object, it checks with the *name service* (see the next section) if the object is local or remote. If it is local, the smart pointer will get a pointer to the local object from the name service, and invoke the call directly on the local object.

If the object is remote, the distributed pointer will create a *proxy object* as a placeholder, and let it handle the invocation. The name service will hold on to the proxy object as long as there are one or more pointers mapping to the object identifier. If there are no more pointers, or if the object migrates and becomes local, the proxy object will be deleted.

A method for resetting the pointer is also included. This method makes the pointer point to the special value NULL (nothing). Then the pointer is cleared, the name service is called, and we check if any more distributed pointers point to this object identifier. If this was the last pointer, the name service will delete the proxy object.

We will now explain how the name service is implemented, before we have a look at the proxy object.

4.2.2 Name Service

Our name service makes it possible to find out where an object resides, and to get a pointer to that object, or a location if it is remote. It is shown in figure 4.3. It maps an *object identifier* to an *internet address*.

An object identifier is a unique 128 bit number, which identifies one, and only one, object. To create these numbers we use the *UUID library* that is a part of the *e2fsprogs* program [29]. This library creates *universally unique identifiers* (UUIDs). All objects that are registered in the name service will have a UUID created for them.

The object identifier is mapped to an object called `netaddr`, which contains the IP address, and the listen port of a node. Since it saves both the IP address and the listen port, it also tells the middleware how to create a connection to the node. These connections are initiated and managed by the *connection manager* which will be described in section 4.2.4.

The name service contains a list of all objects. When objects are local, the name service will also keep a pointer to the object. When objects are remote, the first distributed pointer which points to the object will create a proxy object. From that moment on, the name service will keep a pointer to the proxy object. This is to make sure that not more than one proxy object is created for each remote object. As we have explained in the previous section, the name service counts references to the proxy object, and deletes it when there are no more distributed pointers left.

To be able to save all the objects in a name service, we need a common pointer which can point to all the different object classes in the application.

In C++ we can accomplish this by making an *abstract base class*, which all classes will need to inherit. C++ then provides a method called *dynamic_cast* which we can use to convert a pointer type between a base class and a derived class. In our middleware, we have called this abstract base class `Object`, and all classes that should be able to migrate, or that should be callable with RMI, need to inherit this class.

In our example, we have a distributed pointer pointing to an Avatar object on the server. The client application calls the `move(NORTH)` method through the distributed pointer, which starts with checking the name service (1 in figure 4.3). The distributed pointer is told by the name service that the object it needs is remote. If this is the first call to the object, the distributed pointer orders a new proxy object of the correct type. The proxy object is created by a *proxy factory*, which is nearly identical to the object factory from section 4.5.1. The distributed pointer will then call the method from the proxy object.

4.2.3 Proxy Object

A proxy object is a local placeholder for a remote object. It is called from a distributed pointer, if the object the pointer refers to is not local. The proxy object contains all the methods which should be possible to call through RMI from the original object. This way, the middleware makes the RMI transparent to the programmer. This makes it possible to call a method with the same name in the original object or the proxy object. The call is then redirected by the distributed pointer.

All the methods in the proxy object are numbered, so that we can tell which method to call, by just adding an integer to the message buffer. The proxy object will also have a private integer which identifies the type of object. The receiving node will need to know this, since all objects are saved as a common object pointer in the name service.

When a method in a proxy object is called, the proxy object serializes all the parameters the method is called with, and creates an `RMI` message with the parameters. An example of an RMI message can be seen in figure 4.3 at

step 3. This message also contains info about which type of object we are contacting, and which method we want to call in this object. The receiving node needs this information to call the correct object. This message is sent to the node which is registered in the name service as the host of the object we want to call.

If the object has moved from the node the RMI message has been sent to, the receiving node will create an `FORWARDRMI` (Forward Remote Method Invocation) message, and sent it to the new host of the object. The `FORWARDRMI` message is identical to an RMI message, except that the address of the original caller have been appended, so the new host will know which node sent the RMI. (This process is recursive, if the object has migrated from the host which receives the `FORWARDRMI` message, it will be forwarded again.)

When the object resides on the node the RMI or `FORWARDRMI` message is sent to, the *skeleton* on the receiving node will handle the message, and send back an RMI acknowledgment message with the return value. The proxy object will return this to the client that called the method. If the message was a `FORWARDRMI` message, the acknowledgement message will also contain information that tells the caller to update its name service with the new host.

It is important to note that the proxy object will block when it is waiting for an answer from the skeleton at the other node. This might lead to a deadlock, a situation described in section 3.4. To avoid this, the objects on the server application communicate with each other through *events*, which are further explained in section 4.3.

When a proxy object is constructed, it will send a message to the migration group which contains the original object, and tell it that a certain client needs the object. When a proxy object is deleted, the destructor sends a message to the migration group, to tell that the client does not need the object anymore.

4.2.4 Connection Manager

We have implemented a connection manager which handles all connections to a node. It makes sure that no messages are sent anywhere before we have

validated that the connection is OK. It keeps connections open, and opens new ones when the middleware needs them.

When a new connection is opened, the connection manager sends one initialization message. This message contains information about the code version it is running, and the network address the connection should be associated with. The network address is needed because the outgoing connection will use another port than the port which is used for listening and identification. All messages that the middleware wants to send over the new connection are buffered until the connection is validated.

The node that receives the new connection, reads the initialization message and checks if it does not already have a connection with this node. It also checks if the code version is the same. If both of these are OK, an acknowledgement message will be sent back to the node that tried to open the new connection. Then all the buffered messages will be sent over the new connection. If the code versions differ, the application will terminate on the node that tried to open the connection. The user will receive a message saying that the code version on his machine is too old.

4.2.5 Data Sender

In our example visualized in figure 4.3, we see that the proxy object sends an RMI message (step 3) using the data sender (step 4). The data sender is one of the required threads for the middleware to function properly. Our middleware uses internet sockets to communicate with other nodes, and the data sender is responsible for sending messages over these sockets. When it wants to send data, the data sender calls the `send()` method on a socket.

Each socket is wrapped inside a *communication socket* class. (Except for the *listen socket* which has its own wrapper.) When the middleware wants to send a message to another node, the message will be put inside the communication socket, and the data sender will be notified that this socket has data to be sent.

As explained in section 3.11 the sockets are set in nonblocking mode to be able to send large amounts of data in an effective way. Each call to write

data to a socket returns at once, whether or not all data was sent. If not all data is sent, we will be informed about how much of the data still needs to be sent.

The fact that all calls return at once could result in the call going in a loop when all of the sockets are full. The data sender will try to write to them all, using too much resources. To make sure this does not happen, we use the *epoll system call*. Epoll is an improved version of select, that scales better with many connections. With epoll, we can add all the sockets we want to write data on to an epoll list. We can then call epoll on that list, and the thread will be blocked until one of the sockets is ready to write data.

4.2.6 Data and Connection Receiver

On the receiving side of the connection, the data and connection receiver thread works in much the same way as the data sender. This thread listens on all the sockets for incoming data. In addition it listens on a special socket called the *listen socket* for new connections.

Just like the data sender, this thread has an epoll list over all the sockets that are active. It calls epoll on the list and blocks until there is incoming data on one of the sockets. These sockets are also in nonblocking mode. Until the message is complete, the state about the data read is stored in the communication socket objects. When the data and connection receiver has read an entire message, it puts it in the message queue for the message handler (step 7 in figure 4.3). The message handler will read the message as described in section 4.4.2.

4.2.7 Skeleton

The skeleton resides on the node which contains the original avatar object in our RMI example. It is responsible for deserializing the data in the message sent by the proxy object and call the correct method in the original object, as shown in step 9 in figure 4.3. The skeleton will start by reading the object UUID from the message. With the UUID, it can get a pointer to the object from the name service. Then it reads an integer from the message. After the

integer has been checked against a table with all possible object types, the skeleton changes the object pointer to the correct pointer type.

Another integer is read from the message, which tells the skeleton which method should be called in the object. Then it deserializes the parameters for the method call, and performs the call. The return value is then serialized. The skeleton then creates a `RMIACKRETVAL` (RMI ACKnowledgement and RETurn VALue) message, and sends it back to the calling node. There, the proxy object will deserialize the return value, and return it to the caller.

4.3 Events

One of the important parts of our middleware has not been described during this example. These elements are called *events*. Events are nearly identical to RMI. The difference is that events can not have return values, and that the call to an event does not block and wait for a return message. Since they do not block, there is no risk of getting in a deadlock, which makes it safe for servers and proxies to send events to each other.

The programmer decides if a method should be an event or an RMI. If the object is local, there is no difference as to how the call is handled. If it is remote, the proxy object creates an `EVENT` instead of a `RMI` message. The event is then sent to the node that is registered to have the object.

If the object resides at the node the event is sent to, it is handled by the skeleton just like an RMI message, except that the skeleton does not make an acknowledgement message. But if the object has migrated to another node, the event will automatically be forwarded to the new node, and a name service update will be sent to the sender. This is a difference from RMI, where the sender would have to resend the RMI message to the new host.

4.4 Core Selection

In this example, we will show the process of core-node selection and automatic migration of an object group. As we go through the process, we will discuss

Message Type	Node Type
MAININIT	CLIENT

Figure 4.4: Main Initialization Message

the different parts of the middleware when we encounter them on the way. We start by looking at the *messages* that the middleware uses to communicate with the middleware on another node, and the message handler method.

4.4.1 Control Messages

Our middleware communicates with the middleware on other nodes by sending control messages. Whenever the middleware on one node needs to contact the middleware on another node, it will construct a message with the wanted arguments and send it to the other node. A message can for instance be a remote method invocation, or a migrating object.

As explained in section 3.2 we have chosen to use XDR for our communication. We have written two classes called `XdrSendBuffer` and `XdrReceiveBuffer` which work as wrappers around the Sun RPC code, and make us able to write to, and read from XDR buffers in a convenient way.

The messages in our system start with one integer that tells what kind of message it is. We use an enumeration (which maps integer values to names) with the different message types, to make this readable in our code. An example of a message is the main initialization message, which can be seen in figure 4.4. This message is sent by clients and proxies to the main server when they connect for the first time.

Here we see that the buffer contains an integer `MAININIT` at the start, and then it contains another integer to tell if the connection node is a client or a proxy. These messages are read by the message handler.

4.4.2 Message Handler

The message handler is implemented as a method that runs in an infinite loop and reads messages that are being put in a *message queue* by the data and connection receiver (which will be described in our example of a remote method invocation). The message queue is implemented as a FIFO queue from the C++ standard library, which can hold XDR buffers. All threads in the middleware are implemented as POSIX threads. This gives us the possibility to use POSIX thread synchronization primitives. The message queue is locked with a *mutex lock*. This makes sure that only one thread tries to manipulate the message queue at any given time. This is necessary to make sure that the queue does not get corrupted.

Since the message handler goes in an infinite loop, we must make sure that the thread blocks (sleeps) when there is nothing to do. We do not want too use CPU cycles to check if the queue is empty over and over again in the infinite loop. To avoid this, we use a *condition variable* for the queue. This blocks the message handler thread when the queue is empty, and makes it wait for the condition variable. The data and connection receiver will signal the condition variable when it has added one or multiple new messages to the queue. The message handler will then wake up and start handling the messages one at a time. It will pop one message from the queue, and set a pointer called `currentMessage_` to point to the new message.

When the message handler receives a message, it will first read an integer from the start of the message, and see what type of message it is. Then it calls the method that corresponds with the message type, and sends the rest of the message as a parameter. This method will then read the rest of the message, and perform the necessary tasks. All of the different messages have corresponding method in the message handler.

The message handler comes in two versions. One that performs the tasks described above in this section, and one version that is extended with core-node selection. The first version is used in clients, while the latter is used in servers and proxies.

The server and proxy version of the message handler uses a timed wait

for the condition variable. This means that it waits for a new message for max 60 seconds, and even if there are no new messages, it will wake up from its sleep after that. If the timer expires when the thread is awake, the new call to wait for the condition variable will return at once, and tell that the timer has expired.

When the timer expires, the message handler will call a method in the *migration service* object called `doCoreSelectionOnAllGRoups()`.

4.4.3 Migration Service

The migration service object is responsible for all the *migration groups* (which will be described in 4.4.4). It maintains a reference to all the migration groups that reside on the node. The migration service also contains the methods that initiate migration of an object. Exactly how one object is migrated from one node to another will be explained in 4.5.1.

When the message handler calls the `doCoreSelectionOnAllGroups()` method, the migration service will go though the list of migration groups, and call `doCoreSelection()` on each of the groups. The `doCoreSelection()` is a method in the migration group object.

4.4.4 Migration Group

Migration groups are used to make sure that objects that depend on each other are kept together on one node. They are also used to store information about dependencies between objects and clients that use them. All objects that are to be migrated dynamically based on core-node selection results, need to be part of a group. This is because the core-node selection algorithm needs information about which clients need an object or a group of objects, to be able to find the best location for that object or group. Migration groups with only one object can exist.

It is the responsibility of the programmer who uses the middleware to make sure that all objects that are to be part of the dynamic migration, are included in a group. This does not happen automatically. The programmer will also have to register the group with the migration service when is it is

ready to be potentially migrated. (For instance, one needs to add all objects before one registers the group with the migration service.)

The data about dependencies between objects in the group, and clients that need them, are sent from the clients to the host with the migration group. When a client creates a new *proxy object* (as described in section 4.2.3) for one of the objects in the group, the proxy object will send a *need message* to the host. The need message tells the host that the client now has a distributed pointer to the object, and should be included as a dependency when we perform core-node selection. The message handler will read the message, and add the information to the migration group.

When the migration service calls the `doCoreSelection()` method in the migration group, the migration group will, if the group is active, make a list of all the clients that need one or more of the objects in the group, and of the clients that have an object that one or more objects in the group need. This list is sent to the *network graph object*. The migration group will return the current best fitted node, and the gain (these two terms will be explained in section 4.4.7) from moving the group there. Before we explain the core-node selection algorithm, we need to look at the network graph object.

It is possible that the migration group is not active. This happens with groups that are in a migrating state. It also happens when objects move from one group to another. When a group is waiting for a object, it will stay deactivated until the objects get to the same node as the group. We will explain how objects can change groups in section 4.5.3

4.4.5 Network Graph

The network graph object contains information about the network layout. We use the Boost::Graph [30] library to save this information. An example of a graph is shown in figure 4.5. In the graph we do not make a distinction about application type. The server, proxies and clients are all saved as nodes. But we only have links from each of the clients to each of the proxies, and from each of the clients to the server. We do not need information about links between the server and the proxies. The network graph object keeps

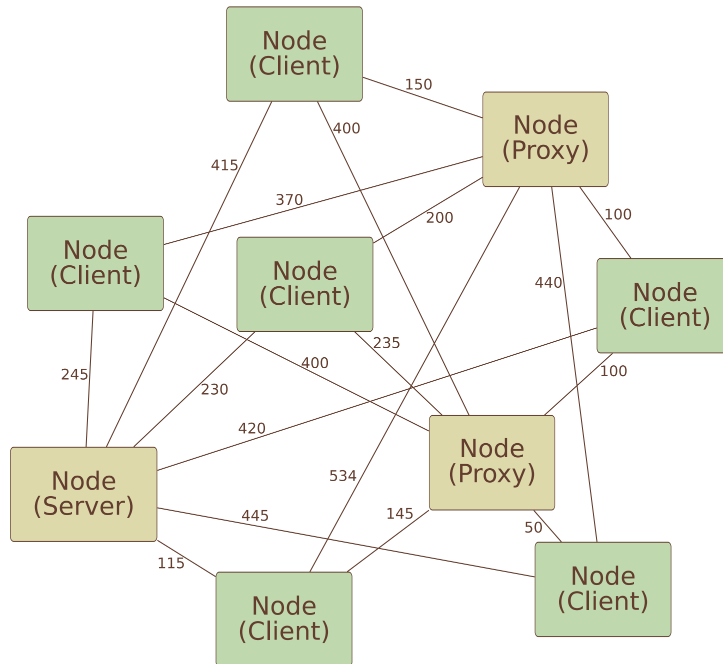

Figure 4.5: Example of a Network Overlay Graph

one list with all the clients, and one list with all the proxies and the server.

The graph is weighted, the weight being the latency between nodes. This information is updated in an adjustable interval by the clients. Each of the clients runs a thread called *latency measurer*, that gathers the information, and sends it to the server and the proxies. We will describe this thread in detail in section 4.4.6.

The network graph object contains information about all clients, server and proxies. When a migration group provides it with a list of clients that need an optimal proxy or server, it uses the information it already possesses to perform a core-node selection. We will explain this algorithm in section 4.4.7. First we explain how the network graph is filled with information about latency and nodes.

4.4.6 Latency Measurement

When a new client connects to the main server, the server will send this client information about all proxies that are connected to the server. This

way, each client gets a list of all connected proxies, and the server. When a new proxy connects to the server, the server will inform all clients about this. This way the client will always have an updated list of the nodes it has to measure the latency to.

The client measures the latency to all proxies and the server at an adjustable time interval. We have set this interval to 5 minutes by default. (Since the latency seems to be pretty stable in our experiments.) The latency measurer is implemented as an infinite loop which runs in its own thread. The loop will sleep and block between each time it measures the latency.

To measure the latency, we use the external program ping as mentioned in section 3.10.2). We have created a python script which reads an IP address from the standard input. Then it calls ping with this IP address, and a parameter that tells ping to send 5 echo request packages to the IP address. The script will then parse the output from ping, and find the median round trip latency for the given host. The result is then written to standard output, which is read by the latency measurer thread.

When the latency measurer thread has collected the median round trip latency to all proxies and the server, a specific message containing this information will be constructed. This message will then be sent to all the proxies and the server. The thread will then call sleep for 5 minutes, before it starts from the top again.

4.4.7 The Core-Node Selection Algorithm

The core-node selection algorithm uses the information from the migration group object about which clients to search for a better server location for. It uses the lists of proxies and servers for potential new location for the migration group. The algorithm used to find the best new server, is *k-Median Vertex Core Node Selection Algorithm* with $k = 1$, explained in detail section 2.3.1. Basically, the algorithm takes a list of client nodes as input, and gives the current best server or proxy as the return value.

The algorithm has been altered to also return the total gain of migrating

from the current node to the resulting best node. The gain is calculated as the average improvement in round trip latency for each of the clients. A threshold has been set in the Ginnungagap object. The gain has to be greater then this threshold for the migration group to decide to migrate. If this happens, the migration group calls the `migrateGroup()` method in itself, and the group will be migrated to the new node. We will explain how a group is migrated in section 4.5.2, but first we will explain how a single object is migrated.

4.5 Migration of Objects and Groups of Objects

Now, we will show how migration of a single object, and migration of an object group is performed. We will also see that it is possible to change the group an object belongs to, during execution. First, we will look at how a single object is migrated.

4.5.1 Migration of an Object

Migration is designed to be initiated by the middleware, as a part of the core-node selection mechanism. But it is also possible to initiate the migration call from an application, which is built with the middleware, if this is necessary.

The migration service object contains a `migrateObject()` method. The first thing this method does, is to change the object state from normal to migrating. When this is done, the object becomes unreachable for RMI or local calls, until the migration is done. This is done to make sure that the object does not change after it has been *serialized*, which is the next step in the migration event chain. If the node gets any RMI messages or event messages while the object is in migrating state, the message is saved in the migration service. When we get a confirmation from the receiving node that the object has been reconstructed there, all the buffered RMI messages and events will be forwarded to the new node.

Message Type	Object Type	Object Id	dist_ptr<World>		dist_ptr<View>	
MIGOBJ	AVATAR	5c1...13c	Uuid	NetAddr	Uuid	NetAddr

Figure 4.6: Migration Message with an Avatar Object

To serialize the objects means to read all the data from the object into a XdrSendBuffer (see section 4.4.1). An example of a XdrSendBuffer containing the data from an Avatar object (which is explained in section 5.2) can be seen in figure 4.6. Since all nodes share the same code base, this is enough data to reconstruct an exact copy of the object at the receiving node.

All objects that should be possible to migrate need to implement a special method called *deflate*. This method creates a migration message, with all the current data values in the object. The method is then called by the migration service when migration is initiated.

The migration message is sent to the node we want to migrate the object to. There the message handler will get the message, and read an integer to acknowledge it as a migrating object. The message handler will then call the `handleIncomingMigration()` method, and let it take care of the rest.

The `handleIncomingMigration()` method reads another integer from the XDR buffer. This integer tells the middleware what kind of object is being migrated. With this integer and the rest of the buffer, the *object factory* object is called.

The object factory is a special kind of object, whose only job is to create new objects of different kinds. The integer which describes the object type is checked against a table, and the object factory will call the constructor of the needed object type. The constructor is called with the XDR buffer as a parameter. All objects that need to support migration, must implement a constructor that takes an XDR buffer. This constructor reads the value of the different data the object contains, and saves them in the new object. We now have an exact copy of the object, on the new node.

When the new object is finished, the object factory returns a pointer to the `handleIncomingMigation()` method. This method will then update the

local name service (explained in section 4.2.2). If the node got the object from another proxy, it will send a message to the server, which updates the servers name service too. (If the object came from the server, or is sent to the server, this is not necessary.)

Last, a `MIGACK` (MIGration ACKnowledgement) message is created and sent to the first node. This message only contains the *object identifier* (described in 4.2.2). When the node that migrated the object gets the `MIGACK` message, it updates its name service, and forwards all the buffed `RMI` messages and `EVENT` messages it got during the migration. Finally, the original object is deleted.

4.5.2 Migration of a Group of Objects

Migration of an entire migration group is initiated when the core-node selection call in the migration group returns with another node, and the gain is higher than the threshold. The migration group will call the `migrateGroup()` method in itself, and it will initiate the migration of the entire group.

When group migration is initiated, the group is first deactivated. (We do not want to do another core-node selection while we already have decided to migrate.) Then all data about the group, i.e., data about member objects and about the clients that need this group, is serialized in a message with the header `MIGGRP` (MIGration GRouP). All objects are serialized, just like when a single object is migrated, but instead of sending them one and one, they are all appended to the `MIGGRP` message. This message is then sent to the destination node.

When the `MIGGRP` message is read on the destination node, it will deserialize the data about the migration group, and re-create the migration group there. Then each of the objects is deserialized and created. As soon as all objects are created, the receiving node will create a migration group acknowledgement message, and send it to the sending node. The sending node will then update the name service, and forward all the buffered RMI and EVENT messages, before it deletes the old group.

4.5.3 Changing the Migration Group of an Object

The migration group has also implemented a method that makes it possible to move an object to another group. This method is designed to be called from the application. We need this in the test application when we move an avatar from one world to another.

All the information about the object, including information concerning the clients that need it, will be transfered. If the new group resides on the same node, this is a simple operation, but if the new group resides on another node, the object itself will have to be migrated to the new node. While the object is migrated, both groups are deactivated.

4.6 Summary

In this chapter, we have shown how our middleware is implemented, using three examples to guide us through all different components. We have seen an example of remote method invocation using our middleware. We have also explained how core-node selection is performed and how migration of objects and groups of objects is done. The next chapter will be devoted to the implementation of our test application, which is built using our middleware.

Chapter 5

Test Application Implementation

Our test application simulates an extremely simplified MMOG. It consists of just a few different objects, and an automated client with some artificial intelligence. We use this application to simulate a MMOG and run experiments on Planet Lab. We will first describe the different classes that the application is built with, then we will take a look at the server and the client part of the application.

5.1 World

The world class contains a two-dimensional vector of structs that contain information about each *cell* in the world. A cell can contain four different things: an avatar, a goldmine, a sub-world, or nothing. These categories will be explained below.

The world class has a print method, which will create outputs like in figure 5.1, to show the status of the world. In this example, an `A` means that that the cell is occupied by an avatar, a `G` is a goldmine, and the `.` is an empty cell. Each of the two cells with an avatar has a pointer to that respective avatar, and the cell that contains a goldmine has a pointer to the goldmine. All of these are distributed pointers.

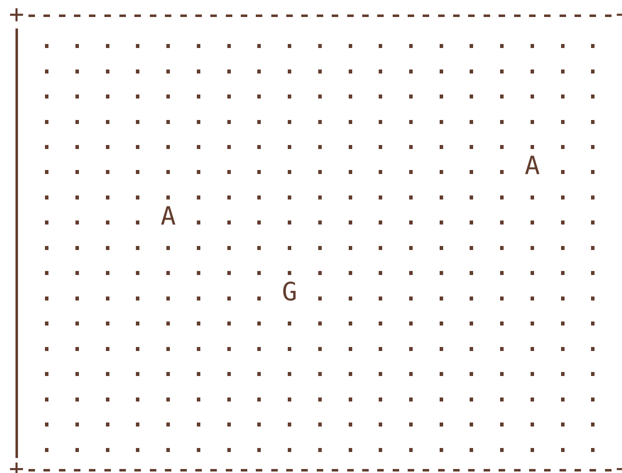

Figure 5.1: A World Object

As we discussed in chapter 2, players in a MMOG are unable to see the entire game world. This characteristic is implemented in the test application as the *view size*, which corresponds with the amount of the world each of the avatars can see. If the view size is seven, the avatar can see 7×7 cells of the world, with itself in the middle.

It is also possible to nest worlds within another world. The reason for doing this, is that if we have small areas that we predict will become crowded (i.e., an area with a goldmine), we want to be able to perform core selection on this group, and possibly migrate it. An example can be seen in figure 5.2. The lower world is a part of the world on top.

To make sure that all objects in the world are kept on the same server, each world will create a migration group and add all objects that reside in the world to that group. When we have a world within another world, we will get two migration groups.

As we can see from the figure, when we have two worlds next to each other or one world within another one, each of the worlds has a buffer zone outside its edge. The buffer zone is set to be half of the view size. The `,` means a buffered cell, which is empty. `a` means a buffered cell with an avatar, and a `g` means a buffered goldmine.

The buffer zones are designed to be able to send updates to avatars inside

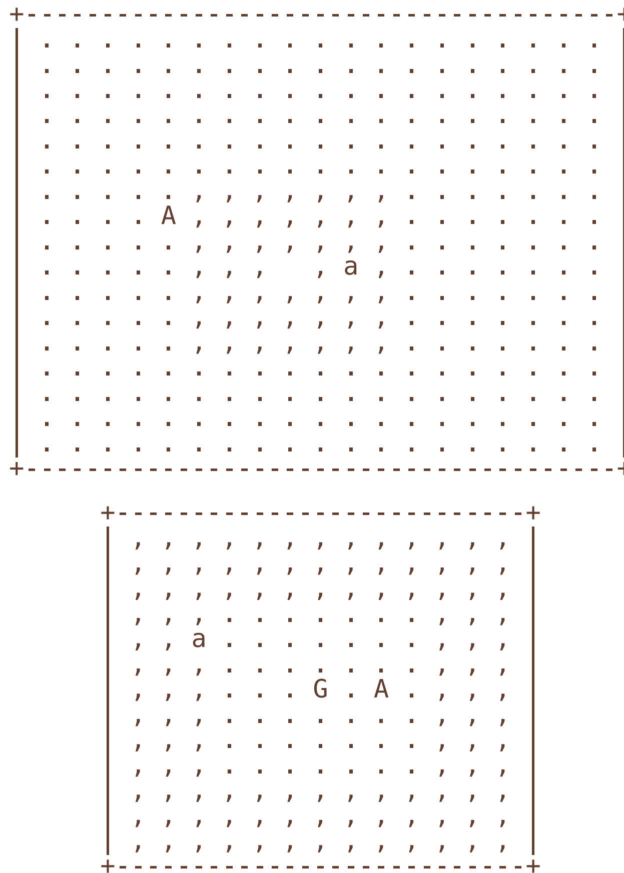

Figure 5.2: Two World Objects - One Within the Other

one of the worlds, without having to perform RMI on on the other world. (As we recall from section 4.2.3, a server should not perform RMI on another server, since this might lead to a deadlock.) When an avatar moves close to the edge of a world, this world will send events to the neighbouring world which has buffered that part. When an avatar moves from one world to another, it will also change migration group to the group of the new world.

The main world object is the world that a client has to call to get an avatar. This one we have given the object id 1. The client registers in the name service that the object with id 1 resides at the main server. This way, all new clients will always be able to contact the main world, since the main server will have the new address if it has migrated.

5.2 Avatar

Clients can call the method `getNewAvatar()` from the main world to create a new avatar object. The main world will then create an avatar object and put it in one of the empty cells. A client can move the avatar object around in the world, calling the `move()` method upon it.

The avatar is included in the same object group as the main world when it is created. If an avatar moves over to another world, it will change migration group.

5.3 View

Each client has a view over the world, which is what their avatar can see. Views belong to a client, and do therefore not support migration. The server decides how big the view should be. An example of a view is seen in figure 5.3. With a view size of seven, this is what the avatar to the right in figure 5.2 would see.

The world object will update a client's view if a client moves its avatar, or if another avatar moves inside the area-of-sight. When updating the view, it uses events instead of RMI. The view class also contains methods that the

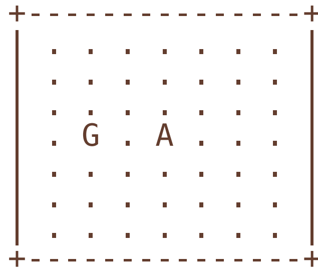

Figure 5.3: The Avatars View

client can call to see if the cell it wants to move to is free, or to figure out if there are any goldmines within the area-of-sight. If there are any visible goldmines, the view class tells the client in which direction it has to move.

5.4 Goldmine

Goldmines can be created on a free space on a world. They simulate points of interest which can be found in any MMOG. The goldmines are just static objects, with no special functions. They are migrated along with the world they reside in.

5.5 Server

The server will create worlds for the avatars to populate. It will make sure that the main world has the object identifier 1. The clients will then be able to get a pointer to the main world object. The world a server creates will typically look like the one in figure 5.2, with a small world containing a goldmine inside the main world.

If any of the world objects are local, it server will write the status of the world to the screen every other second. This way, we can see what is going on, and if one of the worlds is migrated, we will see that it moves to another server.

5.6 Client

When a client starts the application, it will map the object identifier 1 to the main server. Then the client creates a distributed pointer to the world with object identifier 1. It will then ask the world about the view size, and create a view object of the correct size. It will send a pointer to the view as a parameter, and call the `createNewAvatar()` method in the main world. This way, the clients gets a pointer to a new avatar. The pointer to the world is then cleared, so the client is not registered as a dependency of the main world when core-node selection is performed.

Our client has implemented some simple artificial intelligence. It will pick a general direction by random, and start to move in that direction, one move each second. For each move, the client has a 5% chance to change the general direction. If the way is blocked by another avatar or the end of the world, it will also pick a new general direction. If a goldmine comes within area of sight, it will move next to the goldmine, and stand there for 20-30 seconds, before it will start moving again.

The client will start out looking for goldmines. When it has found one, it will no longer search for them. But for each move it makes after it has stopped looking, there is a 3% chance that it will start to look for a goldmine again.

We have also implemented a manual client where a player can control the avatar with the keyboard, and see what is around the avatar on the screen. We used this to test that the implementation of walking from a world to another was correctly implemented.

Chapter 6

Experiments

In this chapter, we will go though the experiments we have performed with our middleware and test application. The experiments are divided into two categories. First, we have performed micro benchmarks on the middleware itself to see how it performs. Second, we have performed large-scale experiments on *PlanetLab*, a large network of machines spread out over the world, with our test application. We will go though the micro benchmarks first, and after that we will see how the middleware performs on many nodes.

6.1 Middleware Micro Benchmarks

To test how the basic components of the middleware performs, we have performed micro benchmarks of our RMI and migration implementation. The micro benchmarks were performed using the local network. The machines run Ubuntu 9.04, and have the same hardware configuration, with an Intel Core 2 Duo E6750 2.66GHz CPU and 2 GB of RAM. We are first going to see how RMI performs. Then, we will look at the results for migration.

6.1.1 Remote Method Invocation

To test RMI, we have created a simple test class, which implements nine methods with different return values and parameters. First, we have tested the methods by calling them directly from a program, to get the execution

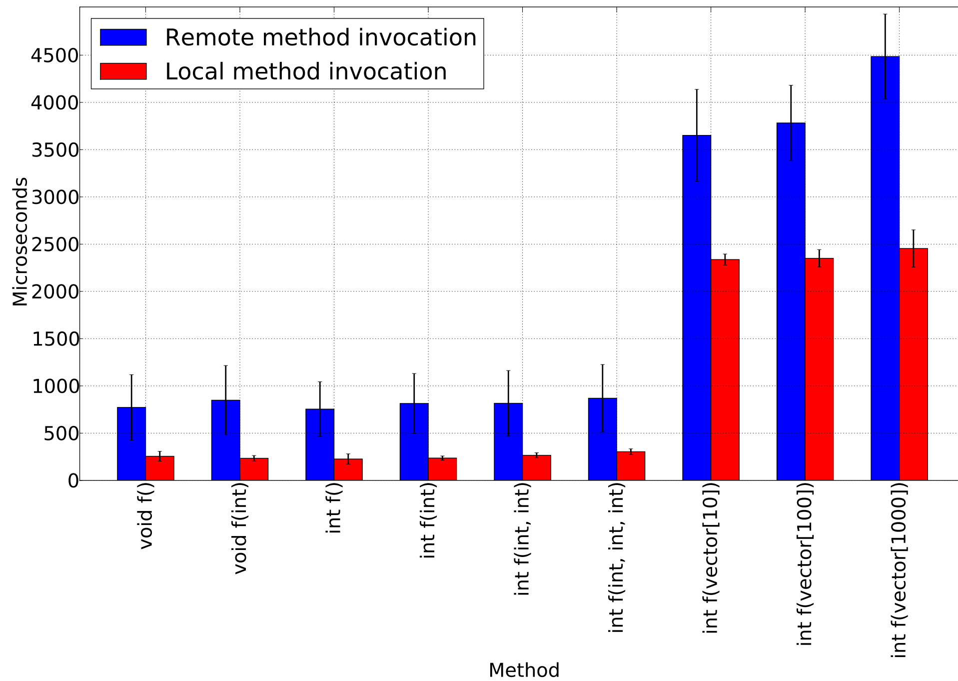

Figure 6.1: Average time of method invocation with standard deviation

times without the middleware. Then, we have tested them with our middleware, both using RMI, and calling the methods locally though the middleware. The RMI is performed between two machines on a local network. Each method has been invoked 10000 times.

In table 6.1 the different methods and their test results are shown. Figure 6.1 show the average and the standard deviation of the different methods. The results shows that the RMI introduces an overhead, which is excepted. This overhead comes partially from the need to serialize and deserialize the parameters and return value, and from the latency introduced by the network. With the methods that have an vector as a parameter, we can clearly see that the amount of data sent have a great impact on the RMI overhead. Performing a local method invocation through the middleware adds no noticeable overhead.

The standard deviation is relatively low for all of the local method invocations. It is larger for the remote invocations, and this is due to more randomness when data has to be sent over the network.

void f()	Local	266.50	262	26.11	1266	262
void f()	Middleware Local	265.65	263	22.11	1913	262
void f()	Remote	817.70	811	94.16	4061	735
Mean difference between local and remote		551.20				
void f(int)	Local	265.10	262	23.51	1344	262
void f(int)	Middleware Local	266.69	263	20.16	1288	262
void f(int)	Remote	818.50	813	75.23	2815	725
Mean difference between local and remote		553.40				
int f()	Local	266.04	262	30.02	1955	262
int f()	Middleware Local	265.73	263	19.84	1219	262
int f()	Remote	818.33	814	72.46	4708	732
Mean difference between local and remote		552.29				
int f(int)	Local	266.61	262	24.56	1305	262
int f(int)	Middleware Local	265.37	263	20.58	1412	262
int f(int)	Remote	820.50	815	68.52	2458	730
Mean difference between local and remote		553.88				
int f(int, int)	Local	264.80	262	21.71	1896	262
int f(int, int)	Middleware Local	265.56	263	17.59	1225	262
int f(int, int)	Remote	819.55	815	70.18	4036	723
Mean difference between local and remote		554.75				
int f(int, int, int)	Local	267.23	266	34.80	2495	262
int f(int, int, int)	Middleware Local	266.94	263	18.34	393	262
int f(int, int, int)	Remote	820.79	817	76.77	4579	690
Mean difference between local and remote		553.56				
int f(int_vec[10])	Local	265.90	262	29.37	1928	262
int f(int_vec[10])	Middleware Local	267.07	263	17.31	440	262
int f(int_vec[10])	Remote	835.80	830	66.34	3210	738
Mean difference between local and remote		569.90				
int f(int_vec[100])	Local	265.41	262	24.20	1948	262
int f(int_vec[100])	Middleware Local	266.85	265	17.73	1279	262
int f(int_vec[100])	Remote	991.62	987	70.25	2901	898
Mean difference between local and remote		726.20				
int f(int_vec[1000])	Local	267.11	262	30.34	2062	262
int f(int_vec[1000])	Middleware Local	264.71	263	12.17	389	262
int f(int_vec[1000])	Remote	1674.60	1667	94.23	5336	1595
Mean difference between local and remote		1407.49				

Table 6.1: RMI: Essential statistics in microseconds, 10000 runs

# Ints	Mean	Median	Std	Max	Min
1	494	485	135	5348	396
10	517	511	102	4700	398
100	660	652	95	5384	575
1000	1315	1313	76	4292	1236

Table 6.2: Migration of single object: Essential statistics in microseconds, 10000 runs

In conclusion, we see that there is overhead associated with RMI, compared to local method invocation. This overhead is to be expected, since we have to communicate over the network. It is well within reasonable limits, and small enough to not affect gaming experience in games created with our middleware.

6.1.2 Migration

To measure the time migration takes, we have created a test class which holds a vector of integers. When we create an instance of the object, we can decide the length of the vector, so we can easily create objects with varying size of data. We have tested our migration facility with both single objects, and groups of objects. We will first look at migration of a single object.

Single objects

To test migration of single objects we have created a server and a proxy application, which sends objects from the server to the proxy, and writes the times in microseconds to a log file. The server creates an object, and migrates it to the proxy. This is repeated 10,000 times. Between each sent object, the server sleeps for 30,000 microseconds, to avoid flooding the network. The time is measured from when first an object is ordered migrated, until the proxy has sent back a `MIGACK` message, the name service has been updated, and the old object deleted.

In figure 6.2, the results of migrating a single object are shown. It is shown that the time gradually increases with the size of the object. This happens because the amount of transmitted data increases with the size of the objects. Table 6.2 shows more detailed statistics.

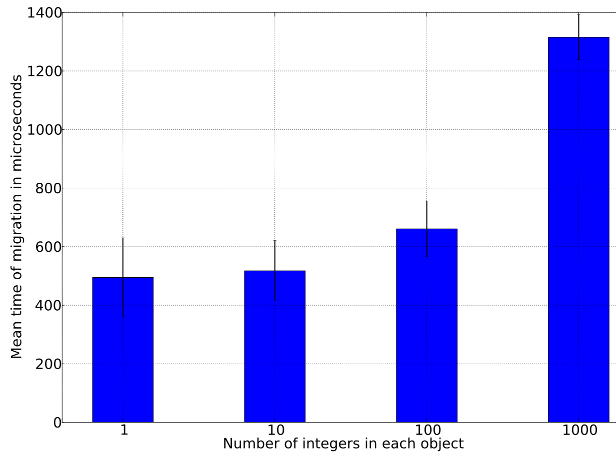

Figure 6.2: Average time to migrate an single object of varying size with standard deviation

# Ints	# of objects	Mean	Median	Std	Max	Min
1	10	1	1	0.06	2	1
1	100	4.57	5	0.55	12	4
1	1000	60.94	61	2.22	66	51
10	10	1.01	1	0.14	5	1
10	100	5.02	5	0.15	7	5
10	1000	65.42	66	2.26	71	56
100	10	2.01	2	0.12	4	2
100	100	9.02	9	0.18	11	8
100	1000	109.62	110	2.69	120	96

Table 6.3: Migration of groups of objects: Essential statistics in milliseconds, 1000 runs

Object groups

To test migration of groups, we have created groups of varying size, and migrated them the same way as we did with single objects. Both the size of the groups, and the size of each object in the group have been varied. Between each group that is being sent, the waiting time for the server has been increased to 0.5 seconds, to compensate for the larger amount of data transmitted.

In figure 6.3 the results for the migration tests with object groups are shown. Table 6.3 summarizes the essential statistics from the test. Here,

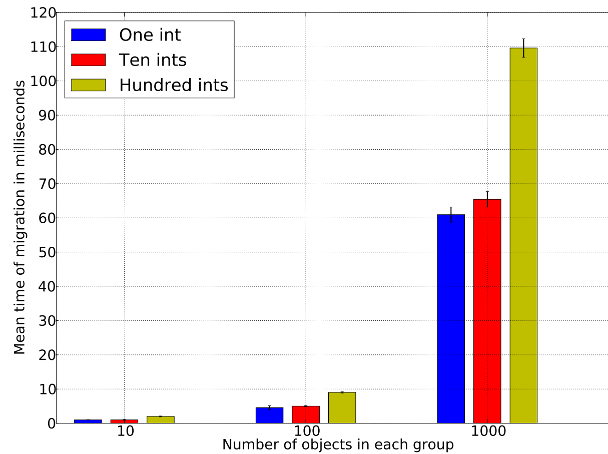

Figure 6.3: Average time to migrate groups of varying sizes, with objects of varying sizes, with standard deviation

much of the same pattern as migration of single objects can be seen. The time to migrate a group depends both of number of objects in a group, and of the size of each object.

6.1.3 RMI during migration

The overhead the migration inflicts on RMI have been tested. When RMI is performed on an object that migrates, three tings can happen. First, the `RMI` message can reach the node the object migrates away from, after the object has been migrated to a new host. Second, the `RMI` message can arrive while the object is in migrating state. Last, the host which migrates the object to another node, can send a name service update to the node performing RMI before the RMI is initiated.

In the first case, the `NetAddr` of the node performing RMI, will be embedded in the `RMI` message, and an `FORWARDRMI` message is created, and forwarded to the node which now contains the object. The `RMIACKRETVAL` message will then be sent from the node with the object directly to the calling node, with the new location embedded in the message.

In the second case, a `FORWARDRMI` message is created from the `RMI` mes-

# of objects	Type of RMI	Mean	Median	Std	Max	Min	# RMIs
100	All	1126.56	916	1844.06	108452	591	7904
100	Buffered/chained	3984.18	3743	1845.24	10754	1043	118
100	Regular	1083.25	914	1809.66	108452	591	7786
1000	All	7626.15	926	17157.12	107591	613	7843
1000	Buffered/chained	46377.29	48466	16023.42	107591	1115	1143
1000	Regular	1015.32	904	950.67	38673	613	6700

Table 6.4: RMI during migration: Essential statistics in microseconds, with 10 integers in each object

sage, and the message is buffered until the node have gotten a `MIGACK` or `MIGGRPACK` message. Then the message is forwarded as in the previous case. This is the case which takes the longest time to resolve.

In the last case, the migration will be completely transparent to the node performing RMI, since the location is updated before the RMI is initiated. The `RMI` message will be sent to the correct node, without going though the old one.

To be able to test how much time the RMI takes when the first or second case occur, we set up a server, a proxy, and a client. We created an object group on the server for the first test run, with a hundred objects, each containing ten integers as data. For the second run, the object group size was changed to a thousand, while the object size remained the same.

The client was programmed to perform an RMI on the object, and write the time it took to a log file. The clients waits 100,000 microseconds between each of the RMIs. The server was programmed to check each 800,000 microseconds if the object group is local, and if it is, migrate it to the proxy. The proxy performs the same test as the server, but the interval between each check is decreased to 700,000 microseconds. The difference in time between each RMI, and the check if the group should be migrated is to make sure that we will hit all of the different cases explained above. Both the server and the proxy are programmed to migrate the object group to the other node 2,000 times each, which means that the object group has migrated 4,000 times during the test.

In figure 6.4 the results of the tests are shown. Since the clients logs when its name service is updated, we are able to see when we hit the first or the

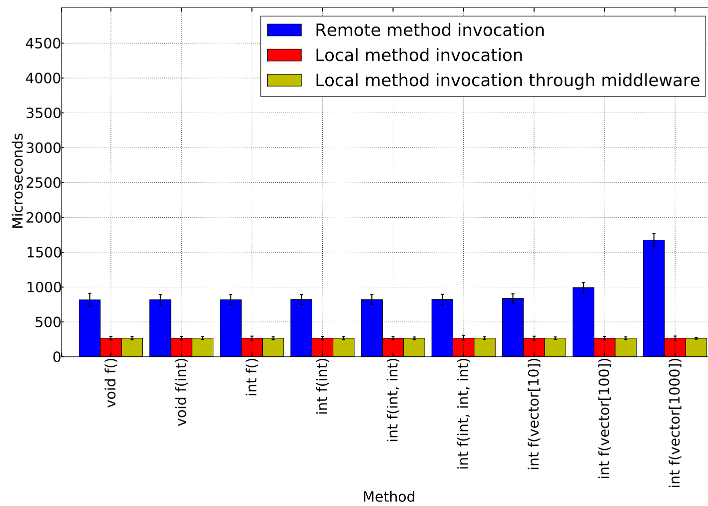

Figure 6.4: Average time of RMI during migration with standard deviation

second case explained above. But since both of them look the same to the clients, we can not distinct between the first and the second case. Table 6.4 shows more detailed statistics about the two test runs.

We see that the RMI takes a considerably longer time to complete if it is performed just when the migration takes place. This time is also greatly affected by the data size of a group. It is reasonable to assume that this overhead can be reduced by forwarding the `FORWARDRMI` messages once they arrive at the node with an object in migrating state. Because of limited time, we have not been able to test this while working on this master thesis.

6.2 Test Application Experiments

We have performed large scale experiments with our test application on PlanetLab. The purpose of these experiments is to simulate how a real MMOG would perform using our middleware.

6.2.1 PlanetLab

PlanetLab [31] is a collection of internet connected machines distributed all over the globe. There are approximately 1,000 machines connected, most

of them hosted by research institutions. On PlanetLab, an application gets allocated a *slice*. A slice is a set of allocated resources distributed across nodes on PlanetLab. A user can decide which of the nodes should be included in the slice. A virtual server is created on every node that is added to the slice. The resources a node allocates to a single slice is called a *sliver*. Many slivers can run on a single node at the same time.

Our experiments were performed on approximately 125 nodes. The number varies a little during our experiments, since PlanetLab nodes can be unstable. Sometimes nodes are restarted, and sometimes they become unreachable from other nodes. Since all the resources on a single node are shared between many users, the nodes can also become unresponsive, or just very slow, when the virtualization decides to give another sliver more resources. Still, PlanetLab gives us one of the most realistic environments to test our application on.

6.2.2 Experiment Set-Up

The nodes we use in our experiments have been chosen with help of the CoMon [32] monitoring infrastructure for PlanetLab. We used CoMon to get a list of nodes that are alive, and with no known problems, stripped down to only one node at each site. This resulted in a list of approximately 250 nodes. (If we allowed for more than one node at each site, we got around 500 nodes. This tells a lot about the unstableness of PlanetLab: approximately half of the nodes were not alive, or had reported problems.) From this list, we randomly picked 130 nodes, which can be seen in figure 6.5.

We performed two different kinds of experiments. The first PlanetLab experiment was performed two times. The first time, we selected six nodes with as low load as we could find. The second time, we selected six nodes randomly. The six nodes were set to be the server and the proxies for this each experiment. For each set of six nodes, we performed the test three times, using one, three, and six servers: The first time we only have one server, the next time two proxies are added to this same server. The last time we add three more proxies, still including the proxies and the server from the second

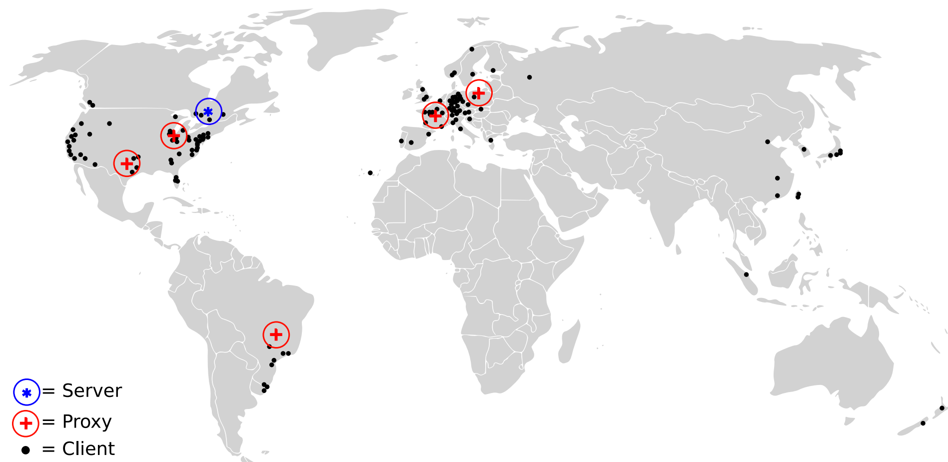

Figure 6.5: Geographical location of the PlanetLab nodes

test.

Our server test application is set up to create a single world object. The world has a size of 40×40 cells, and a view size of 11. The clients are programmed to move 600 times each, and sleep for half a second between the moves. After the 600 moves, the clients disconnects. This makes each test last for a little more than 5 minutes, plus the RMI response times for each of the 600 moves.

The server is started first. If any proxies are included in the test, they are started right after the server. Then we start all the clients at the same time, and let the test run until all of the clients have disconnected.

We want migration to be triggered easily, to be able to study the results of it. Since our experiments only take a short amount of time, we have set the core-node selection algorithm interval to one minute. This way we can be sure that the algorithm will be performed at least four times during one test run, resulting in potential migration. The threshold for migration was set very low: two milliseconds per client.

We programmed the clients to log how long each RMI `move()` took, so this information could be parsed after the experiments. The server and proxies have logged the results of each of the core-node selection algorithms, and it was noted when and where objects were migrated. We will now go through a run-time example, and explain what happens when we run the

test application on PlanetLab before we show and analyse the results.

In the second experiment we performed on PlanetLab, we wanted to simulate a small region from a MMOG, so we changed the clients, so they connects and disconnects for a random time. This is to simulate how player enters and leaved virtual regions. The random time a client was connected followed an exponential distribution with a scale of 60 seconds. We also changed how often servers performs core-node selection to 30 seconds. This way, we could test how capable the middleware is to support groups that constantly changes, and see if we could get a low aggregate latency for all the clients.

During our experiments, it became clear to us that there were some errors in the implementation of how objects changes groups. This led us to the decision to not perform tests with more than one world, as there would have to be invested a to large amount of work to correct some of the bugs in the implementation. We also believe that the experiments we have performed tests the parts of our middleware that are important. This also led us to remove the goldmines, since their only goal was to attract avatars into the smaller region of the world.

6.2.3 Run-Time Example

The first thing to be started is the server. The server will then create a world object, with the object identifier 1. Then, the proxies are started, and connect to the server. A list of proxies is updated at the server, and at each of the proxies.

After this, the clients are started. We use a *parallel ssh* program to be able to start all clients at once. Each of the clients will register the world object in the name service, with the address of the main server, and the object identifier 1. The clients will also connect to the main server, and be registered in the network graph. Each of the clients will then get a list of all proxies from the server, and they will start to ping them all. When the clients have gathered information about the pair-wise latency to all of the proxies and the server, they will send this information to the server and the proxies. This way, the network graphs will be updated on both server and

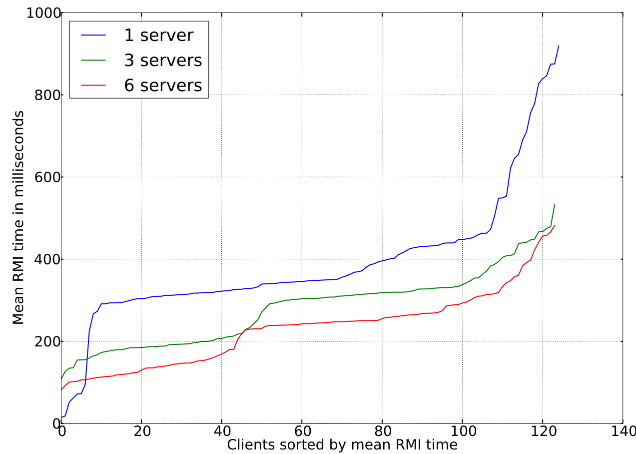

Figure 6.6: Experiment one: First run - Aggregated mean RMI response time

proxies. The latency measurement has its own thread, so it is performed in the background.

When the clients have connected to the main server, they will perform an RMI on the world object, and request a new avatar. A distributed pointer to this avatar will be returned to each of the corresponding clients. The clients will then use this distributed pointer to call the RMI move on each of their avatars, once every half second, until they have moved 600 times.

The server will perform core-node selection on the world and all the avatars within it. Each minute, it checks if the clients that have an avatar will earn more than two milliseconds each in round trip time if the world is moved to another proxy. If the group is migrated to any of the proxies, this proxy will start performing core-node selection each minute, in the same way as the main server.

6.2.4 Results

In figure 6.6 we see the results of our first experiment. The figure shows the aggregated mean RMI response time for each number of proxies. For each run with more proxies, we see that the aggregate mean RMI time for the

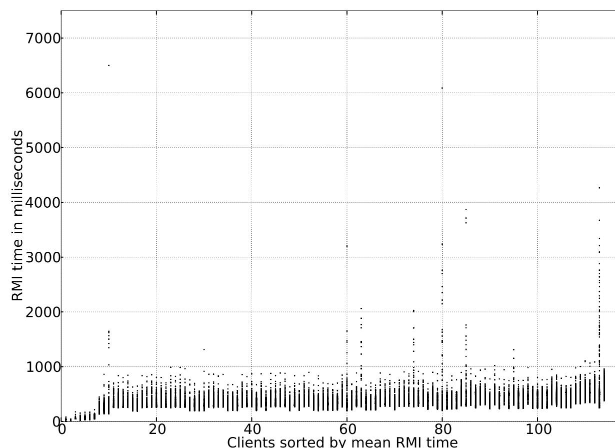

Figure 6.7: Experiment one: First run - RMI time per client with 1 server

clients gets lower. This indicates that the core-node selection algorithm have found a better located node, and migrated the virtual region there, for each of the runs.

In figure 6.7, 6.8, and 6.9, scatter plots for each of the three runs is shown. Here we see that when we add some proxies, and the middleware starts to migrate object groups, we get some RMIs with much higher invocation time. We can assume that this happens due to the way we have implemented RMI during migration.

We can also see that we have some outliers, and some clients that have much more unstable RMI times than the other clients. Since the internet is an unstable network, the outliers can be due to the path between a client and the server being disrupted while the package is on the way. When this happens, the TCP protocol will retransmit the package after a given time interval. This could lead to the long RMI times.

Since all the applications on PlanetLab runs inside a virtual machine, it could also happen that the slice running our application is unscheduled. If one of the clients is unscheduled during the RMI, it would also lead to a larger RMI time.

Figure 6.10 shows the second run of the first experiment. Here the test

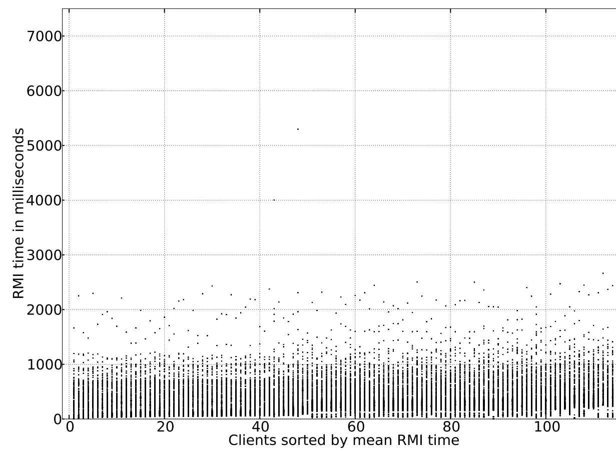

Figure 6.8: Experiment one: First run - RMI time per client with 3 servers

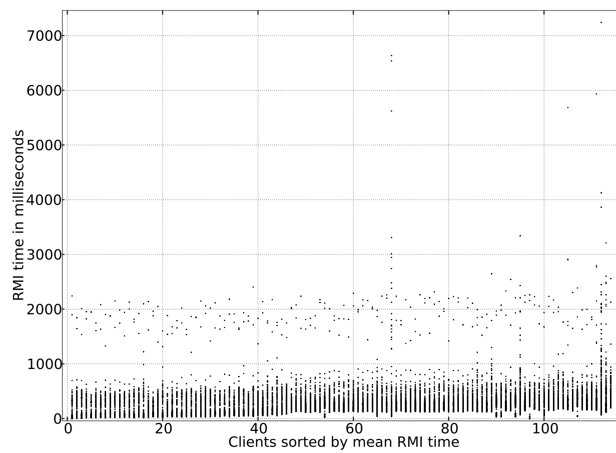

Figure 6.9: Experiment one: First run - RMI time per client with 6 servers

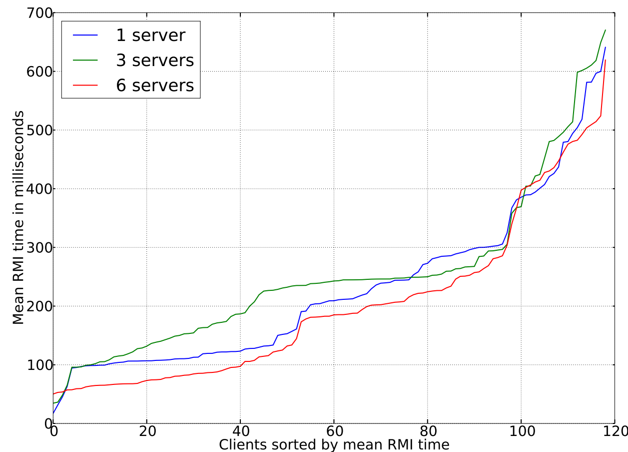

Figure 6.10: Experiment one: Second run - Aggregated mean RMI response time

with three servers have a higher mean RMI time than both the run with one and the run with six servers. The difference is though so small, that it can be explained with the unstableness of the internet, and the PlanetLab nodes. The test with six server performs better than both the others.

The results from our second experiment is shown in figure 6.11. In this experiment, clients disconnects and reconnects at random. In total the clients have connected approximately 1230 times. We logged both how long time the RMIs took, and how many migrations that happens. The migrations were also timed. The object group have migrated a total of 23 times. The mean migration time was 813 ms, excluding one exception, which was 28868 ms. We can clearly see the spikes this one migration lead to, in figure 6.11. We have not seen such large spikes in our earlier tests, but this test ran for a much longer time.

A possible explanation for this spike could be that the sending or the receiving node was unscheduled during the migration. The link between the two nodes can also have been disrupted for a short while. We tested the connection after the experiment with ping between the two nodes, but it did not reveal any problems latency wise, but ping is not affected by the virtualization on the nodes in the same way as our application.

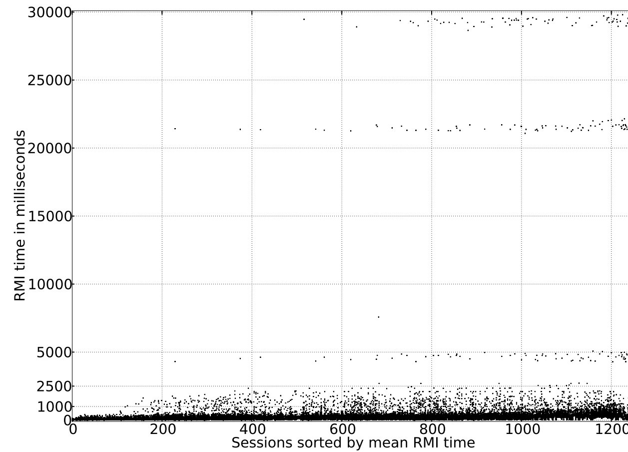

Figure 6.11: Experiment two: Mean RMI time of sessions

It should also be noted that most of the RMIs in all of our experiments is under the 500 ms limit, which in [4] is found to be the limit where the game experience in a role-playing game starts to decrease.

6.3 Summary

In this chapter we have looked at the experiments we have performed with our middleware and test application. We have shown that our middleware performs good enough to support MMOGs with micro benchmarks. We have also tested the middleware with our test application on PlanetLab, and we have seen that the middleware is capable of finding better placement for objects among a given set of servers, if this exist. We have also seen that the way our middleware handles RMIs during migration in not optimal, and that this functionality can be optimized.

Overall we can conclude that our middleware performs the tasks it was designed to, with an acceptable performance. In the next chapter we will discuss the results further, and we will also look at some other possibilities that our middleware provides.

Chapter 7

Discussion

In the previous chapters we have discussed the design (chapter 3) and the implementation (chapter 4) of our middleware. We have also discussed the test application (chapter 5). Since the rationale of our design and implementation have been discussed in the previous chapters, we will in this chapter discuss our experiment results. We will also have a look at some of the implications of a distributed middleware, and at some of the possibilities it provides.

7.1 Evaluation of the Experiments

The experiments we have performed with our middleware and test application have showed that our middleware is able to deliver low enough RMI times to be usable in MMOGs. The middleware is clearly capable of finding better servers for groups of clients.

The experiments have also showed that the handling of RMI during migration could be optimized. We do get one RMI which takes to long, and would lead to a "lag-spike" (client unable to respond to the user while waiting for a return value from the server) in a game, if the `RMI` message arrives at the server during a migration. It is reasonable to assume that we could avoid this if we forwarded the `RMI` message at once, instead of waiting for the acknowledgement message for the migration.

Another thing which can be seen from the experiments on PlanetLab, is

that some clients get a very long response time on one of their RMI requests. This happens most likely due to the TCP package get lost on the way, and have to be retransmitted, maybe more than once. These cases are shown as the outliers in the scatter plots of the RMIs in chapter 6. This can be handled better, by our middleware, or by our application, or by both.

For example, we could use events instead of RMI, and just send more of them from the clients to the servers. If the server confirms each of the events, one could embed data from all previous unconfirmed events, whenever one performs a new one. If the server receives the same information twice, it would just discard it the second time.

The middleware could also keep a list of the RMIs it waits for an answer for, and retransmit each of the messages if an answer have not arrived in a given time interval. This way, if a message uses a long time to arrive at the server, it would be resent, and hopefully it would be sent by another path than the previous message.

7.2 Implications

Our middleware does not have implemented functionality which handles partial failures, i.e., nodes that crash or become unavailable. Implementation of this functionality goes outside the scope of this thesis, but we will discuss some viable solutions.

Both clients and servers can become unavailable. A client going down is not critical to the system, but some kind of clean-up method would have to be implemented. In effect, the avatar(s) the client controlled have to be saved and removed from the game world. Thus, the client can continue from where it got disconnected, the next time it connects. It is possible to implement a clean-up method, which the middleware calls each time a connection is registered broken.

It is much more critical if one of the servers crashes during execution. All the objects the server controls, will then become unavailable, and the state they are in will be lost. To counter this, we could implement replication of objects at one or more servers. These objects would have to be updated

whenever the state of the objects that are replicated is changed. This would lead to more overhead, but it would be possible to recover from a disconnected node, by updating the name services to point to the replicated objects.

Another issue is how to perform garbage collection. When objects are no longer needed, they should be deleted, so that the memory can be utilized by other objects. Finding unneeded objects, and deleting them is referred to as garbage collection. On a local system, one could implement reference counting to all objects. This means that one count each pointer that points to a specific object. When this counter reached zero, the object becomes unreachable, and one can safely delete it, since no one uses it.

In a distributed system, this becomes a much harder task, since nodes which uses objects can become unavailable. In such a case, it it hard to decide if an object can be deleted, since there is a possibility that the client can come back online, and might try to use the object again. A number of viable solutions to this problem have been proposed in [33–35]

7.3 Latency Estimation

In our implementation, we have decided to use the external program ping for gathering pair-wise latency information. Since our goal was to test the middleware and test application with around 130 nodes, this is a good way of getting exact numbers for the latency, without to much overhead. However, when we increase the number of nodes, and/or the number of servers/proxies, the overhead of performing ping between each client and all servers/proxies could soon be larger than feasible.

To counter this, latency estimation can be used instead of latency measurements. In [24], Vik has performed experiments with both Netvigator [25] and Vivaldi [36], to see if they provide accurate enough information to use with core-node selection algorithms. Vik concluded that both Netvigator and Vivaldi gives good enough estimates to select the best, or close to the best, core-node, with Netvigator performing better than Vivaldi.

Using latency estimation instead of measurement can be a good way to lower the overhead which are introduced by the support for core-node

selection algorithms in out middleware.

7.4 Instances

Many of the MMOGs implement a concept called *instances*. An instance is a virtual world which is disconnected from the rest of the world. It is usually a dungeon or another indoor area. The instance is created for a group of players, and only exists for that group. The main goal for instances is to give the same content to many groups of players at the same time, and give all players equal possibility to play all the content of the game, i.e., all players can try to kill the mightiest dragon, it will not die and be unreachable for all but the first player.

Players enters instances with the intent to finish the goal of the instance together. (That can be to kill a special monster, or to kill all monsters, get a special item, etc.) Because of this, there is little to zero variation in the player group which is inside an instance. This makes instances ideal for core-node selection. It is possible to find the best server location for a given group of players, and chose to create the instance at that server. There will be no need to perform core-node selection again for that given group of players, since the group will be the same, as long as the payers play through the instance. Some players might get disconnected because of errors in the internet, but this will not effect the remaining group.

With core-node selection, one would be able to give the group of players the best possible pair-wise latency to the server, for the whole instance run. Another type of instances that have become popular is different kinds of *battle* instances. In these instances, players fight against other players. For these fights to be fair, core-node selection could be used to ensure that the players have approximately the same pair-wise latency to the server. This can be achieved with looking at other core-node selection algorithms, and we will discuss some of the others in the following section.

7.5 Different core-node selection algorithms

There are many different core-node selection algorithms. In these thesis, we have tested the k-Median algorithm. This algorithm finds the server with the lowest total pair-wise latencies to all clients. Since we use the total pair-wise latency, we could end up with selecting a server that gives nearly all of the players the lowest possible pair-wise latency, and one player a much higher latency.

The k-center algorithm addresses this problem, by selecting the core-node that have the lowest eccentricity, i.e., the lowest possible pair-wise latency to the node with the highest pair-wise latency. In the battle instances mentioned in section7.4, this algorithm could be used to give the most fair fight for all players.

Another factor our algorithm does not consider, is the load of the servers. It is a possibility to let the load of each server be a parameter of the core-node selection algorithm. This would be interesting, especially if one is to use our middleware in a peer-to-peer setting (as discussed in section 7.6).

7.6 Peer-to-Peer

Our test application is developed with a server/client model in mind, since this is the model used almost all of todays online games. Our middleware would also be possible to use in a peer-to-peer setting. One could program each client to also act as a potential proxy, and then the core-node selection algorithms would have lots of more nodes to choose from.

There are a few complications with this, 1) there will be a great overhead on the network if all clients ping each other, 2) there will be much duplication of information, if each clients should have overlay network information, 3) partial failures will be an even larger problem, since clients tend to be more unstable than the servers in a server park.

The first and last issue have been discussed earlier in this chapter. The second issue can be addressed by selecting one or more servers where this information is kept. This server could then be contacted with RMI from

all nodes that needs to perform core-node selection, and the server could perform the algorithm and return the answer.

7.7 Summary

In this chapter we have discussed the results from our experiments in greater detail. We have also looked at some of the possibilities created by our middleware, and some of the implications a distributed systems have.

In the next chapter, we will summarize the work we have performed, and the knowledge we have gained during our work with the thesis.

Chapter 8

Conclusion

During the course of this thesis, we have designed, implemented and experimented with a middleware that helps lowering latency for interactive distributed applications. We will in this chapter summarize our work, and the most important contributions. We will also present some issues we consider worthwhile to pursue.

8.1 Summary and Contributions

We have seen that interactive distributed applications require low pair-wise latency between clients and the server to give the players a good gaming experience. We have observed that the physical distance between the clients and the server affects the latency between the client and the server. The goal of this thesis is therefore to try and lower the physical distance between the clients and the server.

To accomplish this goal, we have designed and implemented a middleware which tries to minimise pair-wise latency between the clients and the servers, by moving the servers closer to center of the players. To be able to move the server, we have found it necessary to look at different ways to migrate objects. We have identified data migration as a fast and effective way to migrate objects in a distributed interactive application.

The middleware also needs a way to decide where to migrate the objects.

We have looked at two different core-node selection algorithms. We found that the k-median core-node selection algorithms can be used to find servers from a set of servers, that deliver lower pair-wise latency to the clients, when such servers exist.

We have tested our middlewares performance with micro benchmarks, and found that the overhead introduced by RMI and migration is low enough for use in MMOGs. We have also implemented a small test application which simulates a simplified MMOG. We have performed experiments with this application on a world-wide distributed set of machines called PlanetLab. Our results have shown that our middleware is capable of finding nodes that gives low aggregate latency for groups of clients, and migrate the game state to these.

The results from our experiments have also been the base for a paper [10], which is currently pending review for the NetGames 2009 conference.

8.2 Future Work

In chapter 7 we looked at some of the shortcomings of our middleware, and some of the viable solutions for these. We will now have a look at some of the other work which could be done to enhance our middleware.

Our middleware comes without a code generator. This means that the programmers using our middleware will have to manually create the proxy objects, and the migration methods (deflate, and the migration constructor). Even though we have created helper methods for this, it is still tedious work, which can be automated. A code generator would also help speed up the development time, when using our middleware.

It is also possible to create bindings for higher level languages like Python, which could be both convenient, and time saving.

Adding support for another OS (for instance Windows), could help make this middleware more attractive for developers.

Appendix A

Source Code and Logs

The source code for the prototype middleware and the test application can either be found on the CD which accompanies this thesis, or it could be checked out from its git repository with the following command:

```
git clone http://baldr.no/ginnungagap/ginnungagap.git
```

The logs from our experiments can also be found on the CD.

A.1 Compiling

To compile the code, the following programs with developers libraries are needed:

- `e2fsprogs`
- `glibc`
- `Boost::Graph`
- `Cmake`

The code have only been tested on GNU/Linux. To compile the code, just run the script `compile.sh` which are located in the root directory.

Appendix B

Acronyms

MMOG Massive Multiplayer Online Game

RMI Remote Method Invocation

RTS Real-Time Strategy

FPS First-Person Shooter

RPG Role Playing Game

OS Operating System

GRT Generic Run Time

CUPS Common Unix Printing System

RTT Round-Trip Time

OWT One-Way Transmit Time

Bibliography

[1] Mmogchart.com. http://www.mmogchart.com/, August 2009.

[2] World of warcraft. http://www.worldofwarcraft.com, August 2009.

[3] Eve online: Breaks the wall yet again! http://www.eveonline.com/ingameboard.asp?a=topic&threadID=1024582&page=2, August 2009.

[4] Mark Claypool and Kajal Claypool. Latency and player actions in online games. *Commun. ACM*, 49(11):40–45, 2006.

[5] Eve online. http://www.eve-online.com, August 2009.

[6] Anarchy online. http://www.anarchy-online.com, August 2009.

[7] World of warcraft: Realms faq. http://www.worldofwarcraft.com/info/faq/realms.html, August 2009.

[8] Paul B. Beskow, Knut-Helge Vik, Pål Halvorsen, and Carsten Griwodz. Latency reduction by dynamic core selection and partial migration of game state. In *NetGames '08: Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, pages 79–84, New York, NY, USA, 2008. ACM.

[9] Carsten Griwodz and Pål Halvorsen. The fun of using tcp for an mmorpg. In *NOSSDAV '06: Proceedings of the 2006 international workshop on Network and operating systems support for digital audio and video*, pages 1–7, New York, NY, USA, 2006. ACM.

[10] Paul B. Beskow, Geir Arveschoug Erikstad, Pål Halvorsen, and Carsten Griwodz. Evaluating ginnungagap: a middleware for migration of partial game-state utilizing core-selection for latency reduction. In *Pending paper: NetGames '09: Proceedings of the 8th ACM SIGCOMM Workshop on Network and System Support for Games*, 2009.

[11] `http://http://www.aftenposten.no/kul_und/article2439793.ece`, August 2009.

[12] Seti@home. `http://setiathome.ssl.berkeley.edu/`, August 2009.

[13] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 25–25, Berkeley, CA, USA, 2005. USENIX Association.

[14] Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, and Chen Yu. Live migration of virtual machine based on full system trace and replay. In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 101–110, New York, NY, USA, 2009. ACM.

[15] Andrew S. Tanenbaum. *Modern operating systems*. Prentice-Hall, second edition, 2001.

[16] Dejan S. Milojíčić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *CSURV: Computing Surveys*, 32, 2000.

[17] Michael L. Powell and Barton P. Miller. Process migration in demos/mp. In *SOSP '83: Proceedings of the ninth ACM symposium on Operating systems principles*, pages 110–119. ACM, 1983.

[18] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N Nelson, and Brent B Welch. The sprite network operating system. Technical report, Berkeley, CA, USA, 1987.

[19] Amnon Barak, Shai Guday, and Richard G. Wheeler. The MOSIX distributed operating system: Load balancing for UNIX. *Lecture Notes in Computer Science*, 672, 1993.

[20] E. Jul and B. Steensgaard. Implementation of distributed objects in emerald. *Object Orientation in Operating Systems, 1991. Proceedings., 1991 International Workshop on*, pages 130–132, Oct 1991.

[21] Rodger Lea, Christian Jacquemot, and Eric Pillevesse. Cool: system support for distributed programming. *Communications of the ACM*, 36(9):37–46, 1993.

[22] Push windows printer drivers with cups. `http://www.enterprisenetworkingplanet.com/netsysm/article.php/3621876`, December 2008.

[23] Paul Beskow. Migration of objects in a middleware for distributed real-time interactive applications. Master's thesis, The University Of Oslo, 2007.

[24] Knut-Helge Vik. *Group Communication Techniques in Overlay Networks*. PhD thesis, The University Of Oslo, 2008.

[25] Puneet Sharma, Zhichen Xu, Sujata Banerjee, and Sung-Ju Lee. Estimating network proximity and latency. *SIGCOMM Comput. Commun. Rev.*, 36(3):39–50, 2006.

[26] Xdr: External data representation standard. `http://www.rfc-editor.org/rfc/rfc4506.txt`, August 2009.

[27] Cae specification: Dce 1.1: Remote procedure call. Technical Report C706, The Open Group, 1997.

[28] 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, 1992.

[29] Ext2/3/4 filesystem utilities. `http://e2fsprogs.sourceforge.net`, August 2009.

[30] The boost graph library. `http://www.boost.org/doc/libs/release/libs/graph`, August 2009.

[31] Planetlab. `http://www.planet-lab.org`, August 2009.

[32] Comon. `http://comon.cs.princeton.edu/`, August 2009.

[33] 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.

[34] 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.

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

[36] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: a decentralized network coordinate system. *SIGCOMM Comput. Commun. Rev.*, 34(4):15–26, 2004.