Preface

This report is part of a project on the subject of distributed systems, on the seventh semester of Bachelor of Science, Software Engineering.

The project was completed at Aalborg University Esbjerg in the period 2nd September 2002 to 20th December 2002.

The report has been composed by Mikael Heinze, Lars Lodberg Hundebøl and Claus Hallas Nielsen. Supervisor was Uffe Kock Wiil.

Content of the CD:
- This project report in Microsoft Word, Office XP format and RTF format.
- A Win32 .NET executable version of the distributed multiplayer game.
- Source code for the distributed multiplayer game, DWorm
- Microsoft .NET framework runtime environment (inclusive service pack 2)

Mikael Heinze Lars Lodberg Hundebøl Claus Hallas Nielsen

Contents

1 Introduction

Computer games has become one the most popular entertainments of today. Many of these computer games are provided with the ability to function as a part of a multiplayer game in a distributed context, the distributed multiplayer games. Cheaper and cheaper hardware, both the computer parts itself but also the network parts, give more and more people the possibility for playing distributed multiplayer games at home or at the so called Internet cafes. The popularity of the distributed multiplayer games has also further increased in parallel with the increased popularity of the Internet. Since the breakthrough of the Internet in the mid 90's more and more ways to make use of it has been found. The gaming industry was not late to see the possibilities of the distributed multiplayer games. In the beginning it was a game like Doom [QUA03], typically played in a LAN (Local Area Network), involving two to four players. Today we have big online virtual worlds like EverQuest [EVE02], where players are paying money by the day to play. More and more people around the world have the possibility to connect to the Internet, relatively cheap, through faster and faster connections. Suddenly the possibility to play online has become possible for everybody.

Like any other distributed system, this kind of application has some requirements, although they might be a little different than business critical enterprise application.
This document describes a set of requirement, analysis, design, implementation, test and evaluation of a distributed multiplayer game in relation to issues from the area of distributed systems. Despite of the development process consisted of a number of iterations, where every iteration consist of an analysis, design, implementation and evaluation, the report presents the analysis part as a whole. The design, implementation, test and evaluation from each of the iterations are then written in separate passages. Having the analysis part presented in one part instead of in the actual iteration passage, are due to the overview. This characterizes the analysis part as being a part of the design document, because choices are taken continuously.

The final product is a distributed real time multiplayer game, based on the classical worm game. It is implemented using the .NET platform (C# programming language and a small part implemented as C++).

Throughout the report a distributed multiplayer game will be referred to as a distributed game, when it is assumed that a distributed game implicitly is a multiplayer game.

Source references are notated in brackets in the format with the authors three first letters in his/hers last name or an abbreviation of the web site name/headline followed by the year of publication, e.g. [MSD02] for a source published in 2002 by Microsoft Developer Network, Microsoft Corporation. See appendix A for source references.

Concerning project organisation, process model and risk analysis, see appendix B for further information.

For information about how the .NET runtime environment are installed and the distributed game, DWorm, is executed, see appendix C.

2 Problem definition

In this project, we are to examine what types of distributed games there exist. On basis on this examination a distributed game are defined. Like any other distributed system this kind of application must have some requirements, which need to be fulfilled. What are the main requirements for distributed games and what are the typical ways to implement distributed games? We will examine distributed games in context to some of the more traditional problems found in distributed system theory. This could be, but are not necessarily limited to, extended requirements like the ability to continued playing on server crash, server exit or network partitioning. It could also be things like location transparency and discovery service. We will examine why these abilities are rarely seen in games.

2.1 Limitations and decisions

Two perspectives on distributed games can be taken. The first perspective is distributed games seen from a LAN (Local Area Network) point of view. Second a distributed game can be seen from the point of view of the Internet. Sometimes taking these two different perspectives are the result of the great variation in the properties of the two types of networks. Both perspectives will be taken where it is necessary, when the theory not covers both perspectives at the same time. These are not the only two perspectives. Sometimes, when design decisions are taken, another choice of perspective may be taken, the perspective where a distributed game is seen from game company's point of view.

As a start the .NET platform is chosen as the implementing platform. The reason is the desire from the project group to learn the new technology and the new programming language C# (pronounced C sharp). This decision is taken, well aware that some performance problems can occur. Only if the analysis shows that .NET is completely useless, another platform will be taken into consideration.

Due to the limited time for the project, the project is limited to make use of only two protocol stacks, the TCP/IP- and UDP/IP-stack, due to the fact that these are the most common used protocol stacks today.

The game should be able to run both in window mode and full screen mode. Full screen mode is preferred in relation to performance, but for debugging purposes the ability to execute the game in a window is provided.

2.2 Purpose and goals

The problem definition in combination with the formal requirement to the project forms the purpose and goals of the project.

The purpose of the project is to provide the members of the project group with knowledge in theory and practice in designing distributed systems.

The goals of the project are:

- To examine distributed games and see how issues from distributed system relates to them.
- Partly develop a distributed game with focus on distributed system techniques.
- To analyze the .NET framework and evaluate the C# programming language as target platform for distributed games.
- Producing a report documenting all phases of the project.
- Group members have learned theory and practise used in distributed games and other distributed systems.

3 Choice of a distributed game type

In order to choose what type of distributed game to design and implement, the different types are presented.
Three types of distributed games exist. Real time distributed games with strong consistency, real time distributed games with weak consistency and tour based distributed games. Strong and weak consistency are not well defined, but refers to the importance of consistency.

Tour based distributed games are games where each player in turn makes one or more moves. Examples of this type of game are Chess and Backgammon. These games implicit has an absolute consistency.
Tour based distributed games can be both long and short termed.

Unlike tour based distributed games, where the others player waits for one player to make his move, the real time property of real time distributed games provides the ability for the player to make their moves simultaneously. Real time in relation to games does then not mean that the system is a real time system, where an operation must be carried out within a defined time constrain, but only that the game running always are processing in time and the participants simultaneously can make their moves.

Real time distributed games with strong consistency are games like Quake [QUA03], Doom [QUA03] and Counter Strike [COU01]. These first person shooter games require a near absolute consistency view of the game state. It is very important that when a player sees an opponent and then aims at the opponent and fires, then the opponent really have to be where he seems to be. The maximum number of participants is about 20, to keep the amount of communication at a low level. This type of games is considered to be short termed and normally no ability for saving the game state is provided.

Real time distributed games with weak consistency are games like Red Alert [WES02] and War Craft [WES02]. Often this type of games is strategy games where every player has 100 or more units to control. It is not absolutely necessary that every player sees the precise location of a unit. It is satisfying if the location is seen only as almost precisely. The maximum number of participants is about 8, but with 100 units associated with every participant, it potentially raises the amount of communication.
This type of games is considered to be long termed and the ability to save the game state is preferred.

The fact that tour based distributed games implicitly have absolute consistency makes them without interest. Depending on the choice is on distributed games with strong or weak consistency, the priority of the requirements may alter and the problems for the game as a distributed system may be different.
Among the two types of real time distributed games the type chosen for examination and development are real time distributed game with strong consistency.
It would be easier to design a game engine capable of handling a low number of participants in stead of hundred participants and at the same time having a game based on relatively simple rules. Distributed games having these properties are normally the real time distributed game with strong consistency.

The primary focus in this project is distributed systems and not game engines. Although the focus for this project is on distributed systems, we need some kind of game our game server will be running. Since the main focus on this project is on distributed systems our game will be rather simple, and it will not make use of any fancy graphics or have sound. Here follows a section describing the distributed game to analyze and develop.

3.1 Description of DWorm

The game developed will be a clone of the well known worm game. Worm is an old classical game. In the original single player version a player starts with a short worm. The goal is then to pick up objects and with each object the worm grows longer. As the worm grows longer you get less space on the screen to manoeuvre and risks to collide with you own tail increases. The more objects you have picked up, the longer your worm are and the more point you get.

In our distributed multiplayer version of the game each player will have a worm. The goal is then to survive as long as possible and not getting killed by hitting another worm or its own tail. It will be named DWorm, a contraction of Distributed Worm.

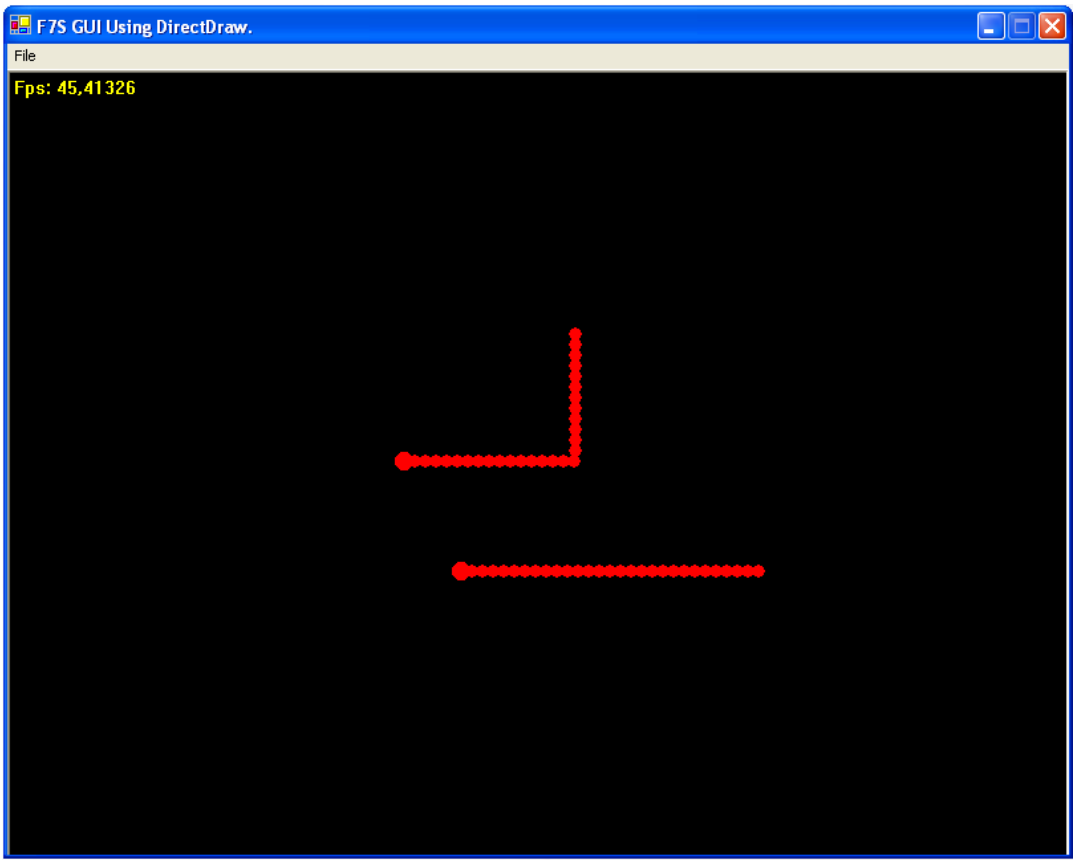

Figure 1, A screenshot of DWorm.

4 Requirements of DWorm

Regardless of the concrete type of a distributed system every distributed system contains some problems and issues according to the area of distributed systems. Some of these issues and problems are directly related to distributed games. The aspects of the defined distributed game DWorm in relation to issues from the area of distributed systems will form a set of requirements. Other aspects of distributed systems can also be taken into consideration. These problems and aspects together form a set of requirements the development of the game has to deal with.

4.1 Distribution challenges

In the following section some challenges in distributed systems will be covered. Not all challenges do necessarily need to be fulfilled for all distributed systems. Depending on requirements and type of application the importance of each of them may differ. In this section each of these classical distribution challenges will be covered and its relevance will be compared to our specific problem.

The distribution challenges presented are categorized as low and high priority. High priority means that it should have influence on the design. Low priority means that it should be a design issue if and only if it can be done without cost on high priority challenges and the project schedule allows it.

4.1.1 Heterogeneity

Heterogeneity refers to the ability to run applications on a collection of different platforms. Platform applies to things like networks, computer hardware, operating system and programming languages. For example does the Internet exist of a lot of different networks, but their differences are masked by the fact that they agree on a communication protocol or know how to convert between protocols.

For distributed games in general, network heterogeneity is of importance. If you want a game to run on the Internet, network heterogeneity has to be achieved. Network heterogeneity is partly implicitly achieved when the Internet makes use of the IP (Internet Protocol). In a LAN more protocol stacks can be used but only the TCP/IP and the UDP/IP will be used according to the project limitations.
Hardware and operating system heterogeneity is desirable but it is rarely achieved. This is mainly because today's gaming platforms are limited to very few, but also very different systems. In most cases the performance aspect and economic aspects denies companies the ability to make true multi platform products. The loss simply is not worth the gain.
Heterogeneity is low priority.

4.1.2 Openness

Openness refers to the ability to extend and re-implement a system in various ways. Openness requires that specifications and documentation are made available. This can either be done by publishing interfaces after implementation or agree on standardizations.

For commercial games openness is of minor importance or even undesirable. Games are often developed and maintained by a company which primary goal is to make money. Making a game where to much of the interface is open to the public, might make the game more vulnerable to cheats. The game company also looses control of the game, which can make it harder to make

updates and expansions. It is however often seen that source code has been released for old and obsolete games.
An open distributed system also makes it easier to gain knowledge about the functionality and lead to easier ways of cheating and exploitation of security holes. Beside, a framework design which makes a system extension easier normally costs on performance. Seen from a game company point of view easy control for updates should be possible, which is not the case if everyone can extend the distributed game. Then only the original core of the game can be updated. Openness is not a desired property of a distributed game.
Openness is different in the priority. When openness not is a desired property then it is high priority not to achieve openness.

4.1.3 Security

Security is important in a lot of distributed games today. Many games on the Internet are of a type where players has accounts they log into when they want to play. These accounts should be maintained in a way such that unauthorized users do not get access. Further more, owners that do have access should be limited from cheating and altering characters or whatever the account may hold. In games where people themselves create a server for a few hours, and invite friends to play, security might be of less importance.

Although distributed systems involves a lot of different users and computers, many of the information resources that are maintained in these kinds of systems are of a types owners does not necessarily want to make public to all people. Security is therefore of considerably importance. Typically security involves protection against unauthorized access, protection against alteration or corruption and protection against interference that can prevent access for authorized personal. Security is then high priority.

4.1.4 Scalability

A distributed system is described as scalable if it will remain effective when there is a significant increase in the number of resources and users. The most obvious characteristic is the ability to avoid performance loss, and preventing resources to run out when scaling a system. Another quality is the ability to keep the system at a reasonable financial cost when extending it.

Scalability is also a very desirable characteristic and the design and implementation should be made as scalable as possible. However the gameplay of many game types simply does not allow a dynamical amount of players, if it also has to be fun. Making a football game with more than twenty-two players would not necessary be a clever decision. In a lot of other game types gaming areas would also get too crowded, if players could keep joining. In other words, scalability beyond a certain point might not even be desirable for many games. The other way around it should be possible that an online world should be extended unlimited.
Scalability is high priority.

4.1.5 Failure handling

Any process, computer or network may fail. Therefore each component needs to be aware of possible ways in which the component it depends on may fail, and be designed to deal with each of those failures appropriately.

The importance of failure handling might depend on the kind of game service provided. In games, people themselves starts and play for a few hours, a server crash and other faults might be acceptable once in a while, although people would be fed up with it if it happens to often.
In online worlds maintained by companies failure handling is much more important. Such games should be online as often as possible. Another aspect is that faults should not be able to invalidate people's accounts or characters.
Failure handling is high priority.

4.1.6 Concurrency

Concurrency is that a shared resource remains valid even though multiple operations are running on that resource. An example could be that if money is transferred from one account to another, another operation may not get access to these accounts before the whole operation has been completed. The purpose of concurrency control is to gain consistency in a system.

A primary requirement for a real time distributed game is a strong consistency. That is, all players share a common view of the game area. If not a distributed game has a certain high degree of consistency it will not be worth playing. Concurrency is not critical but it is of importance. In a scenario where a player kills another player, then the two player's points and death/kill ratios should be consistent. It is not desirable to have a third player that interferes with this operation and also gets credited for killing the player or some of the players feels that they were cheated in another way.
Concurrency is high priority.

4.1.7 Transparency

Transparency is defined as the concealment from user and the programmer such that separate components in a distributed system are viewed as a whole rather then a collection of independent components. Transparency refers to aspects like [COU01]:

- *Access transparency* enables local and remote resources to be accessed using identical operations. Some games might be playable both as single and multiplayer. Local characters used in single player should be accessed in the same way as remotely stored characters used in multiplayer.
- *Location transparency* enables resources to be accessed without knowledge of their location. For a higher degree of user friendliness, some distributed games can find currently running games on the Internet or in a LAN without revealing where the games actually are running.
- *Concurrency transparency* enables several processes to operate concurrently using shared resources without interference between them. In real time distributed games concurrency is a demand otherwise the real time property disappear.
- *Replication transparency* enables multiple instances of resources to be used for increased reliability and performance without knowledge of the replicas by users or programmers. Users of an online gaming world should not be notified about what replication of an account is used.
- *Failure transparency* enables the concealment of faults, allowing users and programs to complete their tasks despite the failure of hardware or software components. Minor hardware faults for a game hosting service should not cause games to crash.
- *Mobility transparency* allows the movement of resources and clients within a system without effecting the operation of users or programs. In a gaming company hosting online games player accounts should be moveable without forcing users to log into the service in a different way.

- *Performance transparency* allows the system to be reconfigured to improve performance as loads vary. For many games performance should never suffer. A system should be build such that when the maximum amount of pressure is put on the system, it should still run close to maximum performance. A drop in response time from 100 milliseconds to 200 milliseconds can in some circumstances be unacceptable.
- *Scaling transparency* allows the system and applications to expand in scale without change to system structure or application algorithms. For online worlds scaling transparency is desirable, so it is possible to extend the world without interrupting already running games.

Transparencies main purpose is to make a system user friendly for users and developers. Like all computer systems games also has to be user friendly. Today more and more people with little computer knowledge are using games as a time killer for an hour or two. To make a successful game to those people it has to be easy to get up and running. Bothering them with too many details and technical problems are not acceptable.
Seen as a whole transparency is low priority but location, scaling and concurrency transparency is high priority.

4.2 Specification of requirements

Specifying the distribution challenges as high or low priority makes the requirements soft, which are requirements not directly measurable. Although some of the requirements are desired properties of DWorm, which can be measured; does it have the property or not?

The game should be designed to run both on a LAN as well as the Internet.

The DWorm type of game can reflect quick reactions and therefore requires low latencies in the network communication. The combination of low latency and absolute consistency is not an easy task to achieve. Low latency is defined as a communication time below 100 ms, but is acceptable when below 200 ms [QUA01]. In fast pacing real time 3D shooters like Quake [QUA03] more than a 100 ms communication time is not acceptable. In games with a slower gameplay, for example a strategy game like Command and Conquer [WES02], a greater package delay is acceptable. This is because absolute consistency does not necessary effect gameplay in these kinds of games.

DWorm should have a discovery service, for giving it a higher degree of user friendliness.

No known games have the ability to continue if the server crashes or the connection is lost. Some games can re-synchronize with the server if needed, e.g. Red Alert 2 [WES02]. Other games just continue without any opponents or the opponents become bots (game controlled players). It is possible to replicate data about a running game and then make it possible to continue the game from a certain state if the server crashes or the connection is lost.

The requirements are gathered up in the following table.

Distribution challenges	**Priority**
Heterogeneity	Low
Openness	High (for not to be achieved)
Security	High
Scalability	High
Failure handling	High
Concurrency	High
Transparency	Primarily low

Properties	
LAN	
Internet	
Low latency (< 200 ms)	
Discovery service	
Replication	

Table 1, Requirements overview

5 Analysis

The defined set of requirements for the DWorm will be used throughout the analysis in relation to distributed systems issues, for evaluation of the concrete issue according to DWorm and distributed games in general.
Throughout the analysis, design like choices concerning DWorm will be taken as a conclusion on every presented issue, a result of presenting the analysis as a whole, when its content is actually assembled from the iterations.

First the .NET platform is described and evaluated as platform for implementing and execution of distributed games and especially for DWorm.
Afterwards follow the analysis of the distributed systems architectures used in distributed games. No choice of architecture will be taken until the end of the analysis, because the choice of architecture has great impact on the different presented issues. In order to discuss and evaluate every issue the different architectures must be known for a greater understanding.
These issues are as follow: network communication, failure handling and availability, transaction and concurrency control, scalability, security, location transparency, synchronization and finally processes and threads.

5.1 The .NET platform

In the following we will discuss the features and possibilities of the .NET platform. We will give a short introduction to the internals of the .NET platform. Furthermore we will describe the distributed thinking of the .NET platform.
Finally we will conclude on how the .NET platform supports as an implementation platform for our type of project.

5.1.1 Introduction to the .NET platform

We will in this chapter describe the .NET platform architecture and define what the differences is from regular code written in e.g. C++ and code written using the .NET framework.
Reading the documentation from Microsoft Developer Network [MSD02] it is hard to figure out what the differences between the .NET platform and the .NET framework is. Therefore when we describe the .NET as a platform, we refer to it as the environment in which the applications reside. When we refer to the .NET as a framework, we refer to it as the programming framework with classes and methods. The components that make up the .NET platform are collectively called the .NET framework.

The .NET platform is a language-neutral environment for writing programs that can easily and securely interoperate. Rather than targeting a particular hardware/operating system combination, programs will instead target .NET platform, and will run wherever it is implemented. .NET is also the collective name given to various bits of software built upon the .NET platform. This can be products like Visual Studio .NET and Windows .NET Server.

The .NET framework has two main parts:

- The Common Language Runtime (CLR).
- A hierarchical set of class libraries

The CLR is described as the execution engine of .NET. It provides the environment within which programs run.

5.1.1.1 Managed Code

Software written for the .NET platform is referred to as code written in managed code. Traditionally code is written in, and referred to as, unmanaged code. It is called managed because it runs under CLR.

One can compare the CLR to existing programming environment like the Java Virtual Machine (JVM), although unlike Java, CLR is not interpreted by the runtime environment. CLR uses a technique called Just-In-Time Compilation (or JIT Compilation). The CLR supplies memory management, type safety, code verifiability, thread-management and security. Existing Java Virtual Machines either stick with the JVM idea (they run the byte code in a virtualized machine), or for significantly better performance compile the byte code to native code, and run that. The .NET platform will only use the second of these methods. Whilst there is no reason that someone could not write a virtual machine this is not what its creators have in mind.

The managed executable consists of two elements; metadata and intermediate language. The intermediate language, or the CIL (Common Intermediate Language), is an abstracted high-level assembly language. The idea of the CIL is for it to be completely CPU independent, so it can translate for any CPU. The intermediate language is more high level than an assembly language.

For instance it includes instructions for creating a new instance of an object. The CLR translate these instructions into native machine language instruction at runtime using the JIT Compiler. One might say that the Common Intermediate Language describes the executable logic. The metadata on the other hand describes the class definition, method calls, parameter types and return values. Metadata is the binding rules for types found in external binary modules called managed assemblies.

The managed application is illustrated in Figure 2.

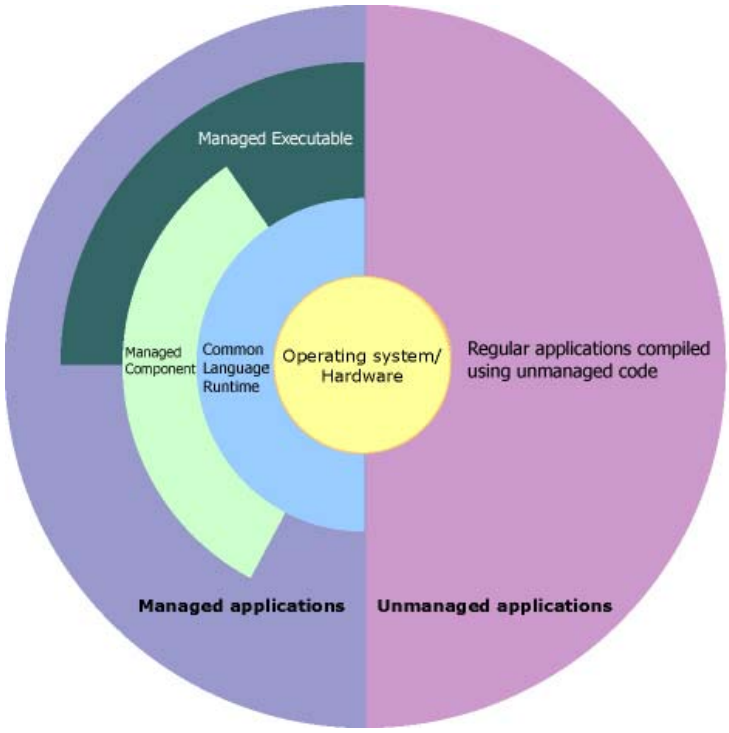

Figure 2, Managed application using the CLR

As the figure above illustrates it is also possible to create components which can run the managed environment.

The approach of managed code is certainly different from the regular unmanaged code. Unmanaged code, when compiled, typically includes the instructions of the program, but all definitions of classes and methods are lost at compilation time. Managed executables always exist of CIL and metadata, and therefore a definition of e.g. class definitions and method calls always exists.
The approach of managed code resembles the COM software components, which is discussed later on in the chapter describing distributed .NET.

When managed application is executed using the Common Runtime Language, there are certain checks the runtime environment makes. For example if the managed executable was ill-designed to break a security rule, then the CLR will catch this, and will refuse to execute the code. The same goes for if a data-type is attempted to coerce into an incompatible type (by typecasting) or if a reference to unassigned data is made.

Furthermore the CLR offers services like automatic thread management, security management and memory management. The memory management is managed, like JVM, using a garbage collector which automatically frees up memory when not referenced to.

The intermediate language is CPU independent. Once written and build a managed .NET application can execute on any operating system supporting the .NET platform. Although at this time of writing the only operating system supporting it are Windows. However on June 27th, 2001 Microsoft announced an agreement with Corel to develop a shared source implementation of a C# compiler and the .NET framework for the FreeBSD version of UNIX. A few weeks later, on July 10, 2001 Microsoft gave the go ahead to and open source version of .NET being planned by Ximian which are the developers of the popular GNOME user interface for Linux.

5.1.1.2 Safety

Another thing the CLR ensures is that programs do not attempt to do anything unsafe. The language most changed by this is C++. Regular C++ permits all sorts of tricks and mistakes. One can allocate a block of memory and then write too much data to that block, overwriting whatever happened to be lying beyond the allocated block of memory. Or one can pretend that one type is another type (deliberately or accidentally). And one can attempt to access any location in memory and modify its contents.
Whilst some of these can be conscious (deliberate cheating in a game) these tricks can also sometime be dangerous. The most famous of these being the buffer overflow. To counter these problems, the runtime simply states that you can not do that kind of thing, unless you have permission to do so. If you try to write past the end of a buffer, it will catch it and throw an exception. If you try to use types in a way that is not safe, it will catch it and throw an exception. One issue with this is that there are ways of generating code that is safe but which the runtime can not verify to be safe. To permit such code to be run, there is a privilege that can be assigned to users to permit them to run, for instance, unverified code from trusted sources.

5.1.2 Distributed .NET

One of the essential features of the .NET platform is the opportunity to build distributed systems. In this chapter the features of the .NET platform that makes it possible to build distributed systems and the predecessor technologies which was the evolutionary steps before the .NET platform, will be explained
But first a look at why the evolution of Microsoft's distributed component technologies is where it is at now.

5.1.2.1 A brief history

Since the Internet, especially the World Wide Web, went into the corporate mainstream market in the mid 1990's, an opportunity for distributed application using the Internet as a platform suddenly existed. The Internet platform was designed to provide a common information interface that is scalable and highly available.
In the resent years the Internet has become even more ubiquitous, with Internet access, incorporated in PDA's, sub-notebooks and almost any desktop computer around the world, the Internet has been even more widespread.
The .NET platform was designed to meet the challenges of a distributed environment like the Internet.
The .NET platform should make it possible to implement software solutions on a heterogeneous environment, because of the different platforms connected to the Internet.
It should also be possible to build software solutions that are scalable. An example on a large scalable network could be a banking network, where each branch has to be connected to the head office.
Because of the fast pace in which software is developed, the need for a platform to write applications that meets the demands for rapid application development and deployment exists.
The .NET platform meets all of the above demands.

5.1.2.2 The COM, DCOM and COM+ relation

To fully understand the possibilities of building distributed application using the .NET platform, it is necessary to look at the predecessors of the .NET platform. These are the COM, DCOM and COM+ technologies.

5.1.2.2.1 COM

Microsoft developed the COM (Component Object Model) technology as a way to make software components that were language independent and location transparent. COM is a specification that tells you to COM enable your software.
Because of a well documented specification, COM is also platform independent to a certain extend. It is of course possible to develop a COM software component to be used for the Windows platform, but there have also been examples of COM implementations for the UNIX and Linux platform.
At first glance COM is not easy to figure out and a lot of code has to bee written to make COM work. Although a programming environment, such as the Microsoft Visual Studio, makes it easier to create COM implementation using wizard tools.

5.1.2.2.2 DCOM

DCOM, which is short for Distributed Component Object Model, made it easier to build distributed software components with the use of Microsoft Transaction Server (MTS) which is a

distributed runtime environment. With MTS it is possible to add among others security, a larger scalability and distributed transactions, without writing any code to implement these features.

5.1.2.2.3 COM+

COM+ is often referred to as the next generation of COM. It is a run time environment like MTS with some additional features and is an integrated part of the Windows platform (was released with Windows 2000). Among the additional features (or services) are component load-balancing and publish and subscribe events.

5.1.2.2.4 Other technologies

The Windows Distributed interNet or Windows DNA is the next step for building scalable distributed applications using Windows and the Internet as a platform. COM was only built to work on an internal network of computers like the intranet. The main idea of Windows DNA is that is should be partitioned in tree tiers; the user interface (the Win32 API), the business logic (the software components) and data storage (the interaction with one or several databases). Windows DNA is meant as a conceptual construct for how to build large business application to work over the Internet.

So how does the .NET platform relate to these technologies? Well, one might say that .NET is the next stage in the evolution of COM.
As mentioned above, it is very difficult to develop COM software components. The .NET framework makes it easier to create software components with the use of managed code. The .NET framework has automated a lot of the hard work when building traditionally COM software components.
When building a software component using the .NET framework, one also builds a traditionally COM software component. It is, in fact, possible for a COM software component programmer to open a software component, build using the .NET framework, and it will look just like a regular COM software component.
It is also possible to migrate between .NET components and COM components. E.g. if a company has been using COM software components to build their business applications, it is still possible with the .NET framework to reuse the components build in traditionally COM

The COM+ environment provides component services which resides next, or is complementary, to the programming services provided by the .NET platform. Through the .NET framework it is possible to gain access to these COM+ component services.
In other words COM+ provides services for building, among others, scalable distributed applications and the .NET platform provides services that simplifies and enables rapider development.

5.1.2.3 Overview

The .NET platform builds further upon the constructional concepts of Windows DNA, which are the three fundamental tiers; presentation, business logic and data access and storage. It is possible to build highly available and scalable applications upon these three tiers using component based technology and the Windows platform with the .NET framework.
In other words a typically small distributed business application has a client which connects to a middle tier, an application server containing all of the business logic, which then connects to a database. In Figure 3 this is illustrated.

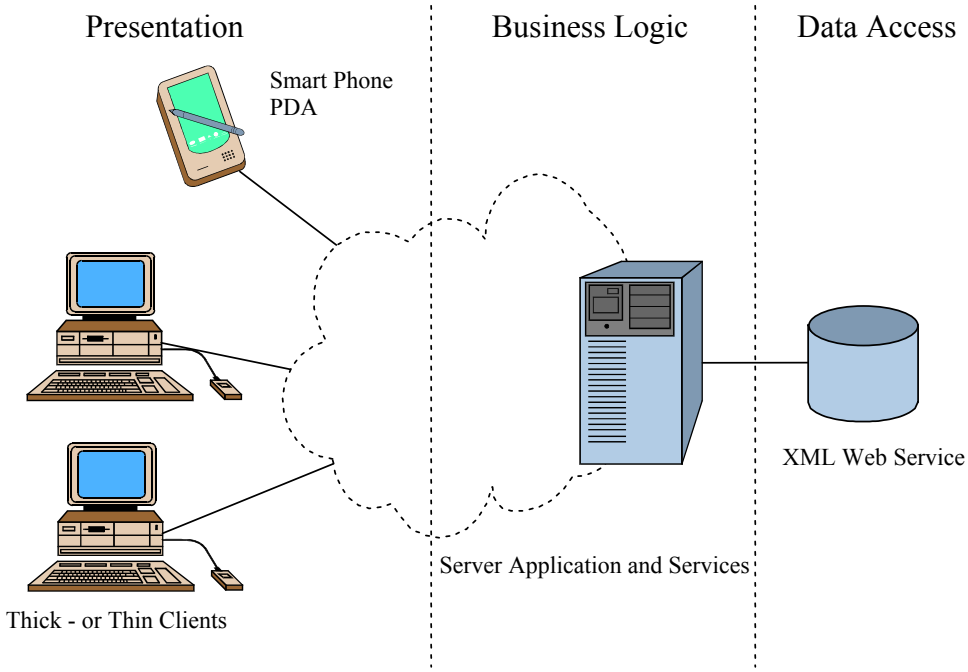

Figure 3, Distributed architecture design using the .NET platform

The presentation layer can either be implemented as a thick- or a thin client. The thick client can be implemented using the Windows Win32 API or indirectly through the .NET framework using the Windows Forms. A thin client could be implemented using a web browser. Just like Java this is also possible with the .NET framework.
The business logic layer can be described as the binding layer between the presentation layer and the data access layer. It is divided into application servers and services. On one hand it supports the clients with the advantages of e.g. COM+ services and security services using the .NET platform. On the other hand the business logic layer, in turn, interacts with the data access layer to access data storage.
Finally the data access layer provides services that support access to a database. This could for instance be stored as XML.
The .NET platform and the Windows operating system support elements within each layer. It is important to notice that this is only a concept by Microsoft on how they see larger distributed system being built. This model has its advantages of being very flexible. E.g. by using thin clients, the client interface is more portable. The illustration above describes a system with very different client platforms; or in other words a very heterogeneous environment.

The .NET framework makes it possible to build large distributed business applications. It is even possible to build application for a very heterogeneous system. Considering the gaming environment, one may ask: “is it really necessary to use a comprehensive framework like the .NET framework for a high-performance environment like the one used for games”. Well the answer to that question is yes and no.
A gaming environment on a heterogeneous system has newer really been implemented for the commercial gaming industry. But as computers connected to the Internet become more ubiquitous, why not develop a game to exist for a wide variety of platforms. Why not create a game that can be played on different platform like PDA’s, smart phones and desktops. Each time the player logs out of the game, the game states are saved on a storage device on the Internet. And each time the player logs into the game, the game is restored from a previous saved game.
One reason for why no commercial game ever has been implemented on different platforms, is that not many games can exist on different devices with different displays (a PDA has a much smaller display than e.g. a desktop computer).
Another, and much more obvious, reason is the different processor and memory capacity each device has. Due to this, the size of the game and graphics (like 3D) can not be the same; it has to be optimized for each device. Although, with the .NET framework, this task is much easier because of the common interface used.
For purposes of a multiplayer using a WAN like the Internet, the amount of data that can be send through a mobile device and the data send through a regular desktop connected to the Internet via a cable modem or router, are not the same.

As for now the .NET platform only is available for the Windows platform (Windows 98/ME/2000/XP and Windows CE .NET). But there is no telling what the future will bring us. As mentioned was the COM specification implemented on both Linux and UNIX machine, so why not also implement the .NET platform for these machines. As goes for the programming languages available for the .NET framework, it is possible to implement application in almost any programming language available. This makes it possible for everyone to program using the .NET framework. This also makes it possible for each programmer to choose his or her favourite programming language.

5.1.3 Relation to distributed games

Now looking at our specification of requirements, is the .NET platform functional as a programming platform for DWorm?
As described the .NET platform builds further upon the idea of Windows DNA, which provides us with the build-in-ability of constructing scalable applications with the use of the three-tier model. The high priority for developing a scalable distributed game is possible with the .NET platform.
Like any other application build for the Windows environment, it is possible to communicate using a LAN and through the Internet. The .NET framework makes it easier by providing encapsulated functionality (sockets) for this purpose. Furthermore the framework also offers functionality for supporting synchronization of threads and thereby meeting our goal of the high prioritized concurrency challenge.

However considering the gaming environment, is the .NET platform a suitable solution? Today's games typically use 3D graphics which has great performance demands. These demands are not met with the use of the .NET platform. We still have to remember that, although code written in managed code is an executable, it still uses JIT compilation techniques and has to run under the .NET platform, which never will be as fast as if code was compiled to assembler or machine language. On the other hand DWorm is not about building a fancy computer game, but instead to examine the possibilities which reside with in distributed games and a worm game can easily be implemented upon the .NET platform.

Our conclusion to whether the .NET platform can be used for our project, the answer is yes. The .NET platform can easily be used of a project of our magnitude and at the same time it supports our personal desire to learn a new programming environment.
Considering from a business perspective where games are build using advanced graphics, building a game with great performance demands, the .NET platform is properly not the best solution.

5.2 Architectures of distributed systems

In this chapter we will describe the three main network architectures in distributed systems; the client-server -, the peer-to-peer and the publish-subscribe architecture. We will first make a general description and then outline the pros and cons.
At the end of the architecture analysis a decision of which architecture to choose, will not be taken.

5.2.1 Client-Server

The Client-Server architecture is one the most widely employed network architectures within distributed systems. It is a simple structure with *n* amount of clients and at least one server. Figure 4 illustrates how each process, whether client or server, interacts with each other. Typically the client invokes the server with a request and the server then answers with a result.

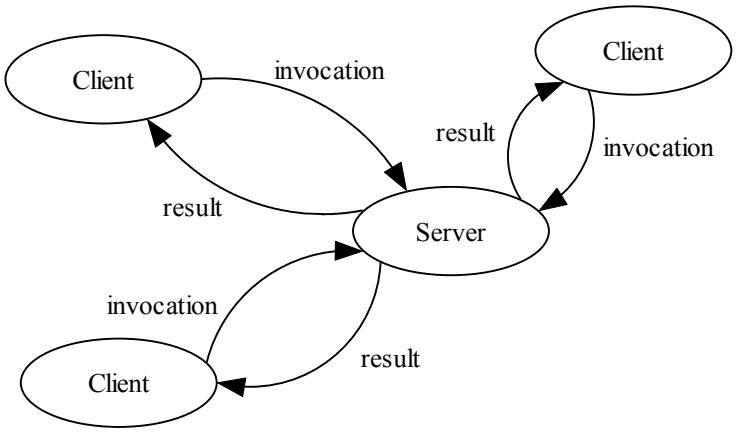

Figure 4, Client-Server Architecture

It is possible for the server in a specific client-server structure, itself to be a client in larger client-server architectures. E.g. we use a browser on the Internet to contact a search engine like google.com. Here we, as the user of the search engine, interact as a client of the server. The search engines job is to use our search criteria to find relevant web pages. To do just that the search engine uses web crawlers. Here the search engine now interacts as the client of the web crawlers which interact as the server.

In the gaming industry the client-server architecture is very popular. The main reason for this is that relevant data like where each player is located and how many points each player has is kept consistent. Only one copy of the data is stored and managed consistently by a single server.

Usually in a gaming environment the server collects commands send by the clients (such as buttons presses and movement commands) and then sends a new state back to the clients. The server is responsible for computing all or most game states and distributing them to its clients. After receiving the new state send by the server the client will then render the necessary graphics. The client thereby acts as a view into the server state. This type of communication resembles to a certain extend the thin client architecture.
The term thin client architecture is where the client is nothing more than an empty shell behind a software layer which provides a user interface. All computing takes place at the server.

Because of the necessary computation power involved with graphics, it is not possible to implement fully thin client architecture. A detailed discussion about thin and thick client will follow.
Alternatively a client can update its own position and states and send its new coordinates and states to the server. This makes a players own movement and behaviour immune to latency. However absolute consistency is harder to achieve since a players position on the server will always be behind the position on the players own machine.

5.2.1.1 Thick vs. thin client

One of the choices when designing a distributed system built upon the client server architecture is how much responsibility the client should have. This is referred to as the thickness of a client.

5.2.1.1.1 Thin client

In the optimal sense a thin client could be a browser or a "dumb" terminal window. In the optimal solution, there will be no additional software required. Moreover the thin client should not require a specific version or vendor of the client interface.

Advantages

- No download or installation of software.
- No installation conflicts, versioning or environment issues.
- Available from any PC on the corresponding net.

Disadvantages

- Low level of dynamic interaction.
- Must write to lowest (oldest) common client interface.
- Limited access services on client machine.

5.2.1.1.2 Thick client

The thick client is a system that is designed to have software loaded on the client. The typical representation is a windowed application on the desktop.

Advantages

- Rich screen functionality and high level of dynamic interaction.
- Increased security since persons without client might have a harder time to access system.
- Clients are able to access local resources.

Disadvantages

- Updates require users to download and install new software.
- Server may have to account of multiple versions of clients.
- Cannot access system from locations without client.

Not all clients can of course be classified so strictly. Some software may require a specific kind of software to be installed on the client. This software may however have so little functionality that updates are rarely or never necessary. Moreover many of today's services available through browser require the use to download and install additional software which sometimes may require an update.

5.2.1.1.3 Summary

In relation to many game types a thick client is requirement. There are simple games that are static enough such that many of the characteristic of a thin client can be achieved. However most games are of types where thick clients are a requirement. Games often have a high level of dynamic interaction and for performance requirements access to recourses on the client is a requirement. In relation to DWorm one of the goals is to make the client as thin as possible, if the client-server architecture is chosen. Making the client as thin as possible makes it possible to have as much of the game logic on the server and then make it easier to keep the game state consistent. Simple and small game extension will also be easier to implement. This does however not mean we will make a thin client since this will be impossible for the game type we will develop.

5.2.2 Peer-to-peer

Peer-to-peer is an architecture that let computers communicate with each other, without the need of a central machine with superior responsibility, as opposed to the client-server architecture. Instead each client maintains its own copy of its state, based on messages received from other clients (see Figure 5.)

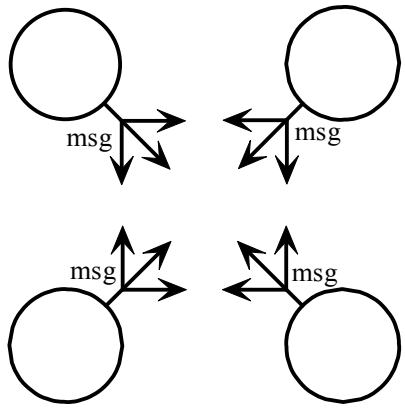

Figure 5, The peer-to-peer architecture

The primary advantages of the peer-to-peer architecture are that it avoids the dependence of a single machine and reduced message latency. Latency is the delay between sending and receiving a message between two processes. In client-server architecture, messages travel from clients to a potentially distant server. The server then updates each client's state and sends the result back. In a peer-to-peer architecture messages travel directly from one peer to another. It thereby avoids a possible bottleneck, which a central server, in the client-server architecture, sometimes tends to be. This can be extremely efficient on a small LAN where messages are sent through few active network components.

Performance in multicast peer-to-peer architecture on the Internet can however be a problem. All routers between two clients must be multicast[1] enabled. Although there are a large amount of multicast enabled routers on the Internet, it can not be guaranteed. What this means in practise is that, peer-to-peer over the Internet has to be done using unicast[2]. This unfortunately scales poorly with the number of peers. Another problem with peer-to-peer is to keep consistency. If messages are lost or arrive at different times, inconsistence between different peers might arise.

In the peer-to-peer architecture there is no central place of the game states. Instead each client has its own copy of the game, based on messages received from other clients. Clients multicast their own position and other game states to every client. Each client then computes its own game, and renders the game state.

[1] Multicast is communication to a group of processes which is already known.
[2] Unicast is communication to exactly one specific known process. We can say that unicast is a subset of multicast.

5.2.3 Publish-Subscribe

Publish-subscribe can be defined as *multiple objects in different locations that are notified of events taking place at an object* [COU01].
The *publisher* object makes events available for observation. Objects which want to receive any notification from these events, subscribe to them. These objects are also referred to as *subscriber* objects. The subscriber object can express an interest in types of events for which they want to receive notifications from.
To illustrate this we can consider a news service. The news service provides (or publishes) several categories of news (e.g. computer science, world news, local news etc.). Depending on interest, a user can subscribe to relevant news categories. Whenever news occurs for a specific category, these can be published to all users which have subscribed to that specific category. This example is illustrated in Figure 6.

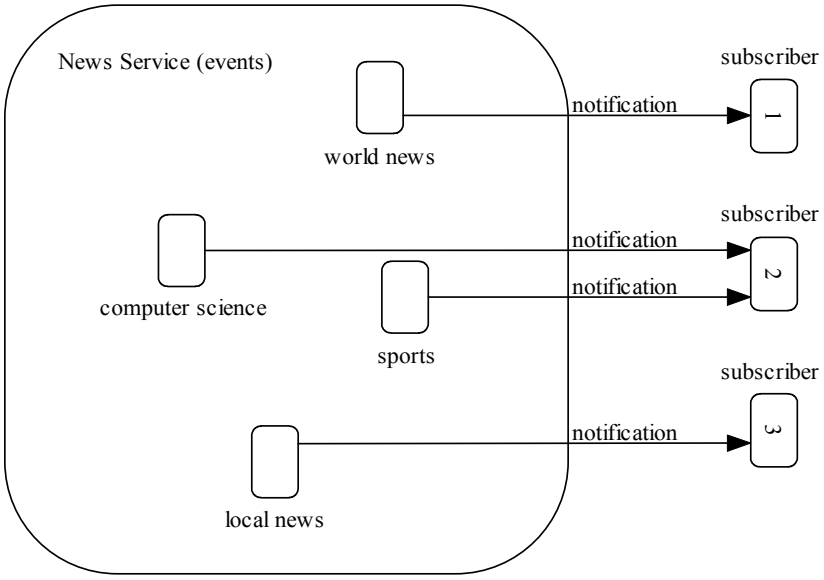

Figure 6, Example of Publish-Subscribe

In the figure above subscriber 2 subscribes to both the computer science and the sports events. Subscriber 1 and 3 subscribes to only a single event each.

The Publish-Subscribe architecture has the advantage of only delivering the information relevant for a certain subscriber. In the case of distributed games it is an improvement to only send relevant game state information to a subscriber (a player), than to send all the information for each player to all the players, which is often the case in a client-server - and peer-to-peer architecture. An example could be a player which is only interested in information about his or hers own audible and visible sight. The player will then subscribe to information at a certain coordinate in the virtual game world. In this approach the player is constantly changing its subscriptions.

5.2.4 Evaluation of architecture for distributed games

To summarize on each architecture, and how this relate to distributed games; we have outlined how they handle problems like consistency, scalability, robustness, latency, administration and cheating.

The information about problems relevant to publish-subscribe architecture and distributed games is gathered from the Mercury system [MER02]. The Mercury system is a science project which argues that publish-subscribe architecture could be used as the communication structure for a distributed game. To our knowledge the Mercury system is designed as a decentralized system, but never implemented.

	Client-Server	Peer-to-peer	Publish-Subscribe
Consistency	It is easier to achieve absolute consistency in client-server.	In peer-to-peer there are multiple copies of the game state. If messages are lost or arrive at different times, inconsistency arises.	As a decentralized system, consistency is hard to achieve.
Scalability	When the server reaches 100% CPU utilization, the server becomes a bottleneck. Scalability from that point is not possible, unless a cluster like server solution is used.	This architecture more easily allows for natural partitioning of game computation, as the number of client increases. 100% CPU utilization on one client in peer-to-peer has much less effect for other players.	The Mercury system provided a good scalability. This was due to the design of the publication and the subscription routing mechanism.
Robustness	Failure on the server will have effect on all the clients. (Single point of failure)	Failure or voluntarily exit on one machine has no effect on other.	As a decentralized system, failure on one node will not affect the whole system.
Latency	Roundtrip delay arises when a message needs to travel to a server, be computed, and then travel back to all clients.	Messages are sent directly from client to client, which in most cases are faster than a roundtrip delay in the client-server architecture.	Poor latency due to the high level of routing, matching and delivering a message to the correct subscriber.
Administration	As for a game company's point of view the client-server architecture makes it easier to distribute bug fixes, force user identification, track players and charge money accordingly. More control.	In the peer-to-peer architecture each player is more control. No user identification is required. Each player has to make sure to have the latest update for the game. Less control.	No information was available about how the mercury system handles the administration issue.
Cheating	With the use of a central server more strict control can be kept, and thereby makes it harder for deliberate cheating.	Since each client is responsible for rendering its own state, the peer-to-peer architecture is more vulnerable to deliberate cheating.	No information was available about how the mercury system handles the cheating issue.

Table 2, Architecture summary

High consistency is very important in cases where a player on one client clearly hits another player on another client, but the other player avoids the shot. Maybe at least one of them will afterwards feel cheated by the game.
The client-server architecture is more affected by a server crash than the peer-to-peer or the Mercury Publish-Subscribe architecture is. However there are ways to overcome this problem in the client-server architecture. This will however increase the complexity of the system.
The cheating aspects are not a high priority for our project, but it is considered high priority in the gaming industry. In the peer-to-peer architecture, as stated above, it is more vulnerable to deliberate cheating than the client-server architecture. Since each client is responsible for rendering its own state, based on other client's packages, it is possible to filter unwanted incoming packages. This could be packages containing info on a players own death. It will also be much easier to produce a package telling other players they have died, even though nobody really has killed them. In client-server some kind of direct manipulation of the server's game state has to be done if cheating has to be commenced.
The great advantage of publish-subscribe architecture is that a player only receives information relevant for that player. However if some sort of filtering mechanism is implemented at the server, which filters the correct information to each player, this feature can also be implemented in a client-server architecture. An example on this is Quakeforge [QUA02], which sends entity updates only for the audible or visible entities from a player's current location.

Today's games are either designed on top of client-server architecture, or less frequently on top of peer-to-peer architecture. To our knowledge no corporate game has yet been implemented upon publish-subscribe architecture. The reason for this is that the client-server architecture offers greater flexibility from a game company's point of view.
The client-server's greatest feature is how easy it is to keep a consistent state, because only a single entity (the server) has the only copy. Unlike the peer-to-peer architecture where every peer in the network has to have the same copy, and in times where messages are lost, it is difficult to remain a consistency between clients. A complex synchronization mechanism (or algorithm) is here needed to resolve any inconsistency that might arise.
The peer-to-peer architecture has the benefit of the latency aspect. The delay between sending a message to another process in the peer-to-peer architecture is far less than the delay in the client-server architecture. This is because a message is send directly to each peer in the peer-to-peer architecture, while in the client-server architecture the message is send indirectly through the server to all the clients (this is also referred to as a *round-trip*).
Typically the publish-subscribe architecture has the great feature of only publishing the relevant information to each subscriber, however the poor latency and not well documented use for distributed games (the Mercury system [MER02]) argues against this architecture. For example the mercury system is design as a decentralized system, but not implemented as one. There is too much uncertainty about the use of this architecture for distributed games.
It may be argued that if a server, in client-server architecture, simply sends out the game state without a request from the clients first, then it has some degree of equality to the publish-subscribe architecture. In a real time distributed game the game state constantly keeps changing and then it would be to performance consuming to let clients request game states from the server. The difference from a pure publish-subscribe architecture is that the publisher can have more different services to subscribe to and the subscriber can not send messages to the publisher in order to change its state. If the chosen architecture will be client-server with some degree of publish-subscribe property, it will still be referred to as client-server architecture.

The choice of architecture will be decided in the conclusion of the analysis document, when the other distributed systems issues has be discussed, because the choice of architecture has major impact on these issues and therefore needs to presented.

5.3 Network communication

From the start of the project it was decided to implement the network communication system using the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). Both protocol stacks are based upon the Internet Protocol (IP). A detailed description of the two protocol stacks, TCP/IP and UDP/IP, and their use in relation to C#, follows the brief introduction to the reference model for Open Systems Interconnection (OSI reference model).
Finally the usage of the network communication protocols in relation to DWorm will be described.

5.3.1 The OSI reference model

This paragraph briefly introduces the OSI reference model. The OSI reference model is a seven layered model (Figure 7), which purpose is to provide structured standards for protocol stacks.

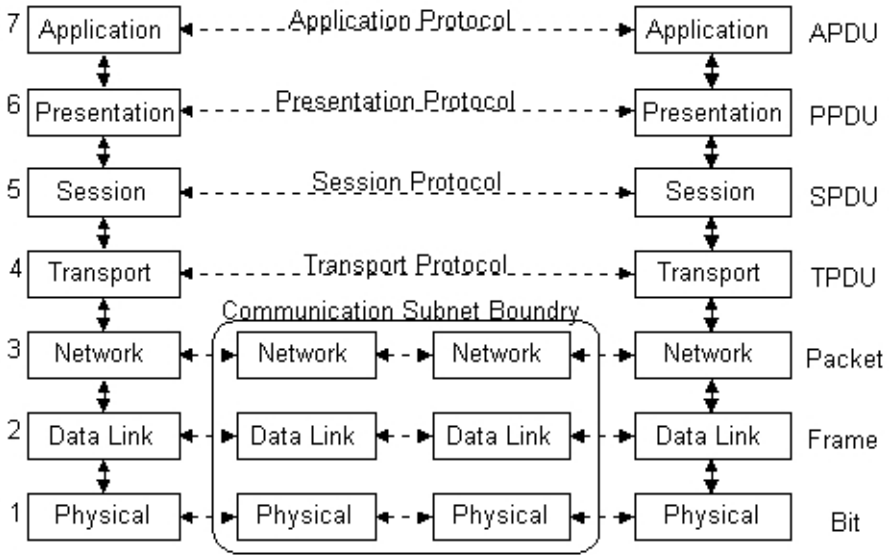

Figure 7, OSI reference model [RAD94]

Each layer from layer 4-7 uses a specific data unit when sending and receiving data, called PDU (Protocol Data Unit). Layer 1-3 uses bits, frames and packets respectively.

- Layer 7: application; interfaces to protocols that are designed for communication of specific application.
- Layer 6: presentation; defines how the data, to be sent through the network, are presented.
- Layer 5: session; for example automatic recovery in the communication.
- Layer 4: transport; lowest level for message handling, where messages are addressed to ports.
- Layer 3: network; transfer of packets between computers. In larger networks this requires route generations for passing through routers.
- Layer 2: data link; transmission of packets between network nodes directly connected by a physical link.
- Layer 1: physical: the hardware that makes up the network and carries the signals, both electrical, light and electro magnetic.

Developing protocols using the OSI reference model, makes it easy to replace or make changes to a layer.
The IP (Internet Protocol) belongs to the layer 3, where TCP (Transport Control Protocol) and UDP (User Datagram Protocol) belongs to layer 4.

Normally distributed systems are developed to represent layer 5, 6 and 7. The layers 5, 6 and 7 are not always designed as single layers but collapsed, as an integrated part of the system. The decision to have these layers collapsed is bound to the performance of the system. Saying that normally distributed systems are developed to represent layer 5, 6 and 7, then it is often layer 4 programming languages has an interface for. In practice programming platforms provides the ability to create sockets. When creating a socket in C# it is specified which type of communication the actual socket should use and which data type it will be able to send and receive. Sockets in C# are the interface to represent functionality in layer 3 and 4.

5.3.2 Communication types

The choice of communication are stated when a socket are created. Three general types of communications are used. Broadcast is when a computer wants to communicate with all other computers. So every single message transmitted from the computer can be received by every computer connected to the network. A subset of broadcast is multicast. In multicasting only a part of the computers in a network are to receive a message from a computer. Those computers able of receiving messages sent as multicast are considered as a multicast group. The multicast group can be either open or closed. In a closed multicast group the sender of the message have to be a part of the group. An open multicast group makes it possible for every computer in the network to send a message to the group. Finally messages are to be sent as unicast, a subset of multicast. In unicast a message only has a single recipient.

5.3.3 IP (Internet Protocol) – containing IP multicast

IP provides the ability for two computers to communicate with each other, identified by their IP-address. IP protocol stack is responsible for routing packets from the sender to the recipient(s). The routing is performed by computer and/or routers. In smaller networks no routing is required. It can not be guaranteed that packets is received in correct order, this job has to be done in higher layer than layer 3.

Multicasting using the IP protocol stack is called IP-multicasting. Sometimes it will not be possible to use IP-multicasting. IP-multicasting is often disabled between segments of larger networks, for examples the Internet. This strategy is due to keep the load of the network as low as possible. Sending a unicast message will only have one route through the network. A multicast message will travel all possible ways to reach as many possible recipients. A multicast message has a Time to Live (TTL) attribute, telling how many active network components as routers/computers a packet may pass before it is discarded.
Using multicast on the Internet is a theoretical possibility, but in practice many Internet Service Providers (ISP) has disabled this feature.

5.3.4 UDP (User Datagram Protocol)

UDP uses the IP protocol stack. UDP is an unreliable protocol with no guarantee of delivery. Because of the unreliability, UDP are good in situations where performance is more important than the delivery of the messages. For examples when transmitting video sequences. The loss of some packets is acceptable, as long it not is too much.

UDP provides the ability for process to process message communication. This is done by the use of defining what port to communicate through. In practice you specify the IP-address of the receiving computer and the receiving port, which is bound to a given process. But it can not be guaranteed that messages are received in correct order. UDP are used when multicasting in C#.

5.3.5 TCP (Transport Control Protocol)

TCP uses the IP protocol stack. Like the UDP the TCP have the ability for process to process message communication, by defining which ports to communicate through. TCP is used where reliable communication is required. TCP gives a guarantee of delivering the message. This guarantee are directly opposite to high performance, even when the message exchanging have no faults and no retransmission is required, because the TCP always makes sure that the recipient is probably connected to the network and the messages are correctly send and received. Another aspect of the reliability is that message is received in correct order.

5.3.6 Relation to distributed games

In distributed games multicasting can be used to send messages from server to all clients or between peers. But it may be concluded that multicasting cannot be used through the Internet and that it requires unicast instead. If DWorm should be able to work over both LAN and the Internet unicast should be used, but a higher performance version for LAN could be designed using multicasting, because most messages can be sent once to all recipients.

Distributed games, where network performance is high priority, should make use of UDP in order to keep network traffic at lowest possible level. But when initializing and ending a game, TCP should be used, because vital information about participant will be required to set up a game, information as IP-addresses, port number and alike.
DWorm requires high network performance with many messages. UDP will be most suitable for messages which contain information about game state. If DWorm should replicate data then TCP communication would be suitable, to be sure that a replica is correct.

5.4 Failure handling and availability

Two types of failures will be analyzed. Process crashes due to logic errors in the programming of the process or operating system failures or computer hardware breakdown. Second the failure in case of network partitioning.
Regardless of the type of failure, a high availability as possible is desired. High availability can be achieved using replication techniques. Replication is presented and discussed in relation to distributed games. In case of which clients are to choose a different server than the current, coordination and agreement techniques can be used; and therefore coordination and agreement techniques are described later on in this chapter.
Finally a conclusion on how failure handling can be used for the DWorm project will be presented.

5.4.1 Failures

Server, client and peer processes can fail. In the case that a server process fails in some way it would be nice if another server process could take over the control of the current game. In the case of a client or peer process fails then another client or peer should have the opportunity to join the game.

In case of network partitioning two perspectives may be considered, respectively from a server and from a client process perspective.
Seen from a client point of view three different types of network partitioning can happened.
A fail in the network can occur, but be masked by the nature of network routing when using the TCP/IP and UDP/IP protocol stacks, if the fail not has occurred in the single network line which connects the computer to the network. If the network fails in the single network connection which connects the computer to the network, then the connection has to be re-established before a client can participate in a game again. The fail can occur in such a way that more clients loses the connection to the server but still have connection to each other. Then these clients have the opportunity to continue the game, but it requires that a new server process can be started while the rest of the clients still communicate with the original server process.

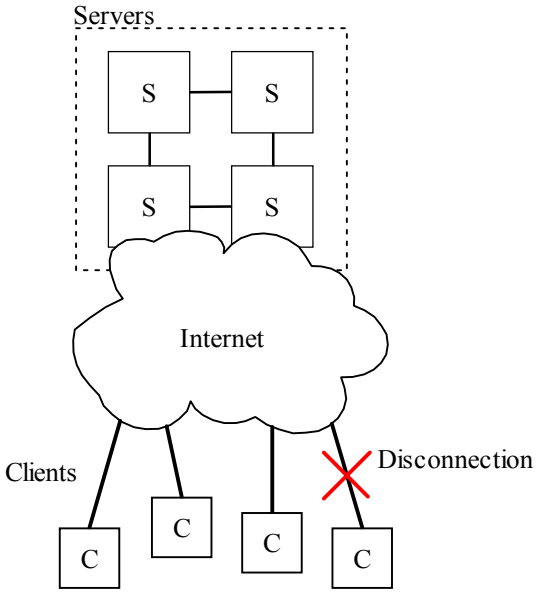

Figure 8, Client disconnected from the Internet.

In Figure 8 it is illustrated a client that is disconnected from the rest of the network. Here it is necessary to re-establish connection between client and the Internet before the client can participate in the game again.

From a server's perspective there are two different type of network partitioning. Again a fail can occur but can be masked. If the fail occurs in the single network line that connects the computer to the network then another server connected to the network can take over the control of the game, if the clients can agree.

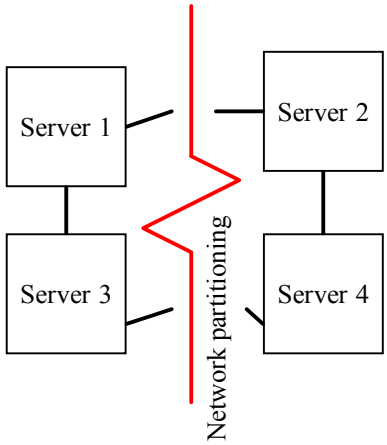

Figure 9, Network partitioning among servers.

Figure 9 illustrates how the network can be partitioned between four servers in a server group. In the above network it is not possible for server 1 and server 3 to see server 2 and server 4.

Despite of a process failure of network partitioning, a common way of determine one of these failures is the use of heartbeats. That is signalling to other processes that a process is still running and connected when no other process communication is exchanged. Then a process can be assumed crashed or disconnected due to heartbeat time out if the other processes in the distributed system can agree upon.

5.4.2 Replication

A method to achieve enhanced performance, high availability and fault tolerance is by the use of replication techniques. Replication can be defined as a way to maintain multiple copies of data at multiple computers.

An example of replication to be used as a performance enhancement tool is proxy servers. They caches copies of the web resources to avoid latency. Another example could be when several servers cache copies as a way to share the workload between them. This is also referred to as load balancing.
Usually when we want to use replication as a way to achieve high availability, we want to maintain data despite of a server failure or server disconnections. The idea is to reach response times that are so close to 100% as possible. An example of high availability could be a wireless network where it is not always certain if one is connected to the network or not. When working on a document on a remote server, it would be functional if it was possible to continue working on the document even if the connection is lost. When connection is re-established chances made to the document should be updated. This is also referred to as a disconnected operation.

A fault tolerance service can guarantee correct behaviour despite a certain amount and type of faults. For example when connection is re-established in the previous example, another user could made changes to the document on the remote server. There are now two different copies which conflict with one another when updates are made. A fault tolerance service can manage such operations.

In our case we would like to keep data consistent on one server. In case of a server crash, a secondary server has to take over the control of all the clients. If we want to continue with existing data stored at an earlier stage on the previously primary server, e.g. player information, replication can be useful. Replication can thereby provide high availability and fault tolerance. High availability is provided since it is possible to continue playing the game only with a minor delay (the time it takes to reconnect to a new server).
If we use some sort of fault tolerance service, we can guarantee that the data, which the secondary uses to generate an existing game and connect to existing clients, is correct data.

We will now consider the basic replication architecture to get greater understanding for designing our own replication architecture. The basic architecture (Figure 10) consists of *n* amount of client, at least one front end and a replication service.

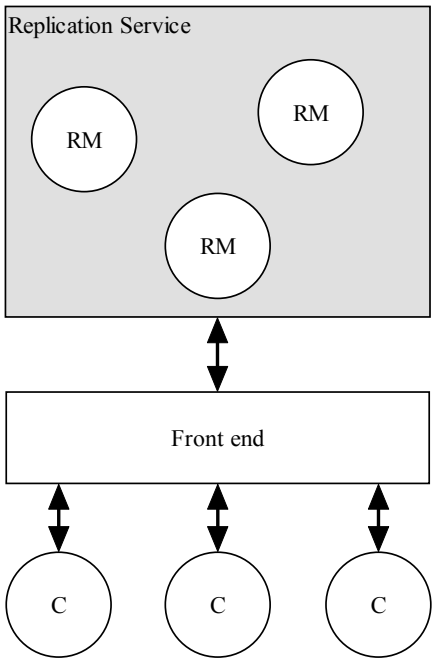

Figure 10, Basic replication architecture

Figure 10 illustrates the basic replication architecture with three clients (*C*), one front end and a replication service with three replication managers (*RM*). The replica managers are components which job is to contain replicas on a given computer and perform operations on them directly. These replica managers provide a service to the clients through the front end. The front end can be implemented directly in client's address space or it may be a separate process.
There are five phases involved with a single request by a client. These are

1. Request

The front end makes a request to one or more replication managers either by multicasting or by communication via a single replication manager which then communicate with another replication manager and so on and so forth.

2. Coordination
   The replication managers prepare for execution of the request and decide whether or not to execute the request at all.
3. Execution
   The replication managers execute the request.
4. Agreement
   Consensus is met on how to handle the request. What effects will the request have.
5. Response
   The replication managers respond to the front end.

Each phase can vary accordingly whether it is a fault tolerant system or high availability system.

The replication of data is something that should not be visible from the client perspective; it has to be replication transparent, because a system that provides a fault tolerant service behaves different from a system that provides support for example disconnected operations.

Definition of replication transparency:

> *Replication transparency enables multiple instances of resources to be used to increase reliability and performance without knowledge of the replicas by users or application programmers.* [COU01]

The knowledge of a replication service or more than one copy of a file exists the client should not be aware of.

It is important to notice that the use of replication is often used with persistent data. Replication is typically implemented as an integrated part in database systems like Microsoft SQL Server Version 7.
Storing data persistent is a heavier task than storing data in memory. If we want to use replication as a way to store the state of the game this is not something which effectively can happened all the time, if this has to happen persistently.
Replication techniques can instead be used e.g. if player information has to be stored on several servers. One might consider a game where the player has a game character which has some characteristics which change each time the player is logged on to the system. In this case replication could be considered as a useful technique. Since the player information is replication in such a way that several servers have access to the data, it is possible, in case of a server crash, to access the player information through another server. Changes to the player information could be saved at a checkpoint (time period) or each time it is needed.
Considering a system where it is not necessary, in case of a server crash, to continue from the same position known before a server crash, replication could also be used for saving the state of the game. A game state could be saved at a checkpoint (like described with the player information). When continuing after a server crash on a new server, the game continues from the previously saved state at a checkpoint.

5.4.3 Coordination and agreement

In the case of which one client wants to leave the primary server and switch to a secondary server (e.g. if a client cannot connect to the primary server), one might imagine all the clients have to meet a consensus for which server they are going to proceed the game on. If the client is the only one who cannot connect to the primary server, and ten other clients have no problem connecting to the primary server, there is no need for the client with the lost connection to change to the secondary server. This could just mean that the client has a bad connection.

Roughly speaking the consensus problem can be described as

> *...processes to agree on a value after one or more of the processes has proposed what that value should be.* [COU01]

To relate this to the above example, we can say that the client (or process) which have trouble connecting to the primary server, make a proposition to the other clients to connect to the secondary server (a change in value). The other clients (or processes) agree on to be connected to the primary server, since only one of the clients has trouble connecting to the primary server. In this case the proposition made by the client is turned down.

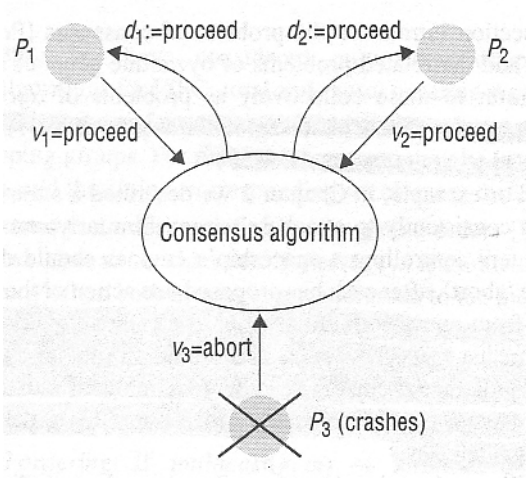

Figure 11, Consensus for three processes [COU01]

Figure 11 illustrates a consensus algorithm where each process starts in an undecided state. Process P_1 and P_2 propose (votes) to proceed (v_1, v_2) whereas process P_3 proposes to abort (v_3) and then crashes. The remaining to processes each decides to proceed (d_1, d_2).

Consensus in the first example is met when the largest amount of processes agree on a value. We can call this function for *majority*(v_1, v_2, ... , v_N); where v_i is the proposed value. The majority function returns the value that occurs most often or returns the value $\perp \notin D$ if no majority exists ($\perp$ is a valid value representing false and D is the set of all processes).

The consensus algorithm has some conditions which should be held for each execution. These are termination (eventually each process set its decision value), agreement (all processes makes the same decision) and integrity (if the processes all proposes the same value then any process in the decided state has that value).

If we consider a system where no process can fail, the termination condition in our example is guaranteed with some sort of reliable communication like the TCP protocol. Likewise the agreement and integrity condition is guaranteed using the majority function.

Because DWorm is designed as a real time distributed game with strong consistency which is primarily short term, it is not as important to continue a game from a previous state after a crash. Since replication techniques used to guarantee to continue from a previously saved game is a very heavy task, is not worth it doing, since it only is a short term game.
In a short term game, it does not really matter if a player is disconnected from a specific game, as long the player can re-connect to the server again. That is why a server is to reach as close 100% availability, rather than the game states. In long term games (like MMOG's), however, a high availability should be pursuit both for game states and for game servers.
The failure handling designed for DWorm, should make it possible to continue playing after a possible disconnection of the client, when connection is re-established. Furthermore it should be possible to reconnect to another game server, if one of the game servers crashes.

5.5 Transactions and concurrency control

Following we will describe transaction and concurrency control in general and how this relate to distributed games. Finally we will conclude on how we can gain an optimized distributed game system by the use of transactions and concurrency control.

5.5.1 Transaction

A way to ensure that an operation either is carried out or not carried out at all is by the use of transactions. In other words a transaction should either guarantee that the transaction is carried out and stored permanent on disk or if a crash occurs, then a transaction should guarantee that all effects are erased. Transactions relate to concurrency control techniques, in that one can say that transactions are units of concurrency control, which will be discussed later on in this chapter.

A transaction typically consist of several operations, for example a banking transaction where a client *a* wants to translate €100 to another client *b*. This is illustrated in Table 3.

Transaction T:
a.withdraw(100);
b.deposit(100);

Table 3, Example of a banking transaction

A transaction could be characterized as a way to:

- *Ensure that all of the objects managed by the server remain in a consistent state whenever they are accessed by multiple transactions and in the presence of server crashes* [COU01].
- *Deal with crash failures of processes and omission failures in communication, but not any type of arbitrary behaviour* [COU01].

Properties of transaction can be remembered by the mnemonic ACID; which is short for Atomicity, Consistency, Isolation and Durability.
Atomicity is for that a transaction is an atomic unit. Transaction is either carried out or not all. When a transaction is carried out it must change an object from one consistent state to another. Transactions should never be seen by another transaction while a transaction is running. Only when a successful transaction has occurred (this is also referred to as a committed transactions), other transactions is able to see the changes made by the transactions. The durability property is that changes that are made by a transaction must be saved on a permanent storage. It is important that this data will survive even if the server process crashes.

5.5.2 Concurrency control

We can define concurrency control in relation to transaction as:

> *The aim for any server that supports transactions is to maximize concurrency. Therefore transactions are allowed to execute concurrently if they would have the same effect as a serial execution – that is they are serially equivalent or serializable* [COU01].

In other words we can say that concurrency control is needed to prevent concurrent transactions from interfering with each other, as if it had the same effect if the transactions was executed serial.

To achieve concurrency one can use techniques like locking and time stamping. Locking is a way to lock objects to ensure synchronisation of access to concurrent transactions. A timestamp is issued to each transaction when it enters the system (e.g. each transaction is giving an increasing sequence of integers). A timestamp protocol will then manage the concurrent executions in a predetermined serial execution. In case of two transactions conflict, the one with the lowest timestamp will get precedence.

5.5.3 In relations to distributed games

So how does transactions and concurrency control relate to distributed games? In the following we will describe two scenarios where transactions and concurrency control could be relevant for distributed games. The scenarios are:

- A game character transfer from one server to another.
- Exchange of objects in role-playing games like Diablo.

In MMOG's (massively multiplayer online games) like Sony Online's EverQuest [EVE02] when joining as a member to this virtual world, one is instantly associated with a specific game server. Each time a player is logging on to the EverQuest virtual world, it will always be the same game server. According to Sony Online this is due to *maintain game balance, prevent fraudulent activity, and to eliminate the heavy burden and cost it would put on the customer service department.*
Now when a friend, to an already established player, connects to EverQuest for the first time, it is not certain that this will be the same game server as his friends. For this reason Sony Online has a service which enables a player to transfer, for minor fee ($50-$300), his or hers game character from one game server to another.
Now we do not now this for certain, if Sony Online uses transaction techniques to handle such a transaction of character between game servers, however this would likely be the case.
Transaction could be used to guarantee that a character would be transferred safely from one server to another. It would prevent, in case of a crash that a character all of a sudden would exist on both game servers, if the character is first is copied to the other server and then a crash occurs, the character is not deleted from the first server, and thereby letting a players character reside on both servers.

Consider a scenario where two characters are to exchange items between each other. This is something which occurs often in the role-playing game Diablo [BLI02]. Here we have an interaction between two players and a need for concurrency control thereby exist.
When a player wants to trade objects (e.g. weapons) with another player in the Diablo virtual world, each player has to agree on the transaction of objects. Imagine the following example in Table 4.

Transaction A	*Transaction B*
a.moveobject(armour); b.receiveobject(armour);	b.moveobject(sword); **CRASH** *a.receiveobject(sword);*

Table 4, Example of two transactions in Diablo

The above example illustrates what could happen if no concurrency control between the two transactions exists. Transaction A manage to move *a*'s object and transfer it to *b* before the crash. Transaction B only manages to move its object before the crash, but never translates its object to *a*.
In this case transaction and concurrency control could be ideal for such a transfer of objects between players.

We have tried to illustrate above what transactions and concurrency control can be used for in relation to distributed games. However since we do not have any persistent data, nor do we have objects which each player has to transfer between each other, we do not really have any reason to use transactions and concurrency control.

5.6 Scalability

Consider a game where *n* amount of players can connect to a close to infinite virtual world. This would be the ideal environment for a distributed game, but physically impossible.
In the following we will examine the theory of scalability, how this theory is related to distributed games and conclude on which scalable issues we have to consider for our specific game type. Finally we will conclude on how the scalability issue will affect distributed games in general and our DWorm project.

Distributed games can vary very differently in scale. From distributed games like Westwood Studios' Red Alert II [WES02] with a maximum of 8 players at a time to other distributed games like Sony Online's EverQuest [EVE02] which promise up to 2,000+ active players at a time, different scalability challenges reside when building a distributed game.

Scalability can be defined as the following:

> *A system is described as scalable if it will remain effective when there is a significant increase in the number of resources and the number of users.* [COU01]

Related to distributed games, a system is considered scalable if it will remain effective even if an increased amount of players is added to the game within the games natural boundaries (e.g. 2,000+ players in EverQuest).

The four challenges presented in the design of a scalable distributed system are

1. Controlling the cost of physical resources
   The quantity of physical resources are required to support *n* users should be at most O(*n*) [COU01]. E.g. one server can support 20 players, if an equally server is added, then the system should support 40 players.
2. Controlling the performance loss
   An increase in structure size (whether hierarchic or linear) will result in a loss in performance. The time taken to access data should be no more than O(log(*n*)); where *n* is the size of data [COU01].
3. Preventing software resources running out
   Prediction of the demand the system will be put through in the future.
4. Avoiding performance bottlenecks
   If only one entrance to shared resources exists, this can quickly become a bottleneck.

The system and application software should not ideally need to change when the system increases in scale, but this is difficult to achieve. Techniques that are used to achieve a more scalable system are the use of replicated data, caching and multiple servers (handling of commonly or several similar performed tasks). Further more the scaling of the system should be transparent.

Scaling transparency:

> *Scaling transparency is allowing the system and applications to expand in scale without change to the system structure or the application algorithm.*

The essential in building a fully scalable distributed game would be as mentioned in the introduction to scalability, to build *a game where n amount of players can connect to a close to*

infinite virtual world. Considering the challenges of building a fully scalable system, this would be impossible. It is e.g. impossible to predict the demand on a system where it is possible for *n* amount of players to connect. Although impossible, many software companies have tried to create a gaming environment similar to this. These are the massively multiplayer online games (or MMOG) available through the Internet. We have already mentioned EverQuest which is properly one of the most popular MMOG's. EverQuest has the capacity to host 100,000+ simultaneous players all at once.

Many factors make one wonder how it is possible for EverQuest to host over a 100,000 players simultaneously, when EverQuest is a highly graphical game played over the Internet.
First and foremost it is important to notice that EverQuest is divided into smaller *zones*. This means it is not possible to interact with every single player, but only with players within a specific zone. A zone could e.g. be a forest, a village, a jungle etc. Within a zone players can interact with players which are in their view of sight. This limits each player to only receive information about other players, which reside with in each player's sight.
However the real backbone of MMOG games, like EverQuest, is the server platform. Although it was not possible to collect exact information about the server architecture used by EverQuest (Sony Online), typically a distributed game system like the one Sony Online provides are build upon several parallel servers which can manage 2,000+ players. Each server then manages a small part of the gaming world.
Considering the challenges of building a scalable system, the above mentioned with several linear servers, it is harder to control the loss of performance since the system consist of several servers. In a system like this, it is also hard to predict the future use of the system.

We have found a similar system from Rebel Arts [REB02] which provides a similar functionality. Rebel Arts have developed a highly scalable and fault tolerant network application called VAST (short for Versatile Accelerated Server Technology). The idea of the VAST system is to remove the requirement for building highly scalable server platforms by providing an off-the-shelf product which provides such functionality. Rebel Arts claims the VAST system it is possible to host 100,000+ on a single server. This eliminates the parallel server architecture and there by reduces the control of performance lost. The controlling of the cost of physical resources is also much easier since the VAST system is connected in a cluster. If it is necessary to host more players, another node is added to the cluster.

There is no question about it, building a highly scalable distributed game like EverQuest is certainly a cumbersome task. However, since our system is designed for real time distributed games with strong consistency and the amount of participants around 20, the challenges of building the system scalable is not that hard.
When designing a game server supporting 20 players, we have to make sure to deliver a communication platform to support this amount of players. A single server can easily manage 20 players.
Now considering the same game from a business point of view, it would be ideal that more than 20 players at a time can play this game. This means that several games should be available for use. If one game reaches the maximum of 20 players, another game should be available. A game which is scalable from this point of view is not concerned on how many active players a single game should contain, but instead how many players the game servers hosting the game should contain. It is not as much the software perspective but instead the hardware perspective where

scalability is an issue (as for MMOG where both the software and hardware is an important perspective to consider).
Therefore when considering scalable hardware solutions, we should be concerned on the cost of expanding the Internet connections as for the cost of adding new game servers to the system.

5.7 Security

Security in relation to distributed games is primarily about cheaters and how to avoid it. Cheaters are to be found every where. Distributed games are not an exception when it comes to cheaters. Cheating in distributed games can be divided in two kinds.
It must be expected that many crackers will attempt to find a way to cheat in a game as soon as the game is released. To avoid crackers taking advantages of a weakness in a distributed game, as low openness as possible should be obtained.
Secondly, distributed games where it is possible to play in online worlds, a log in procedure to an account are required, due to keep different players characters kept available only to authorized persons.

The last aspect of security is attempts to bring down a running game service, like an online world, by for example a denial of service attack. This type of attack is specific for servers in client-server architectures. The idea of bringing down one peer is not the same as bringing down a server, where many users can be disturbed once. In most case it is hard to avoid these kinds of attacks, because most servers, like web server are designed to process as many requests as possible, but the fixed degree of scalability for a distributed game server set a natural limit of the number of requests. Still the network connection could be blocked due to a lot of traffic.

The aspect concerning security in games is that the game logic should not float on the network. By letting network packages containing information like shooting and coordinates, you are making the game vulnerable to cheating by package manipulation. In context to security and game, such information should be eliminated as much as possible.

The way to reach high security in a distributed game is to keep the system as less open as possible.

5.8 Location transparency

For providing the system with a certain degree of location transparency, lookup of currently running games can be done using a discovery service. The purpose of the service is to make it possible to find and eventually join games on the network without any knowledge to the host IP-address.
The approaches to the location service are from the point of view of a LAN and the Internet respectively. The discovery service can be based on multicasting. By reference to the earlier discussion using IP-multicast over the Internet, a like discussion may be taken here. The ability to communicate using multicast, can not be guaranteed in a network containing routers, if the routers do not support multicasting. The discovery service can then take advantage of an already existing name service, the DNS (Domain Name System).

5.8.1 The LAN approach

Imagine a scenario with 20 machines, connected through a LAN network, capable of executing our game. Two individual games are up and running. One of the machines wishes to find and join a game. The first solution would be to find out what IP-addresses all the machines have and then sequentially ask each and every one: “do you have a game I can join?” But how is it possible to find all IP-addresses?
The other solution could be to let either the servers hosting the games (client-server architecture) or let every participant in a game (peer-to-peer architecture), participate in an open multicast group using unreliable multicasting. A machine who wants to find available games on the network, multicasts a message querying for possible games to join, hosts or peers answers using TCP communication.

5.8.2 The Internet approach

If a game has to execute using the Internet, then an IP-multicast solution can not be guaranteed to function properly. An alternative solution is to let a central server(s) manage information about all the actual running server processes. If a client process wishes to join a game then it queries the central server about a list of server process. The location of the central server can be hidden using the DNS service. Then the central server can be moved and the DNS updated, without anyone noticing it.

5.8.3 Relation to distributed games

In order to achieve the desired location transparency, for the users of DWorm or other distributed games, a discovery service should be developed. It is possible to make DWorm and other distributed games location transparent.

5.9 Synchronization

For real time games with strong consistency requirements, the most important factor of all is the agreement of the game state. If this can not be achieved all other problems gets irrelevant. As a basic problem in distributed systems, the agreement of a consistent view is known as synchronization. In this section we will cover some of the basics of synchronization and then conclude on how this relates to distributed games.

5.9.1 Physical clocks

Computer clocks, like any others, are not in perfect agreement. They have different clock drift, which means they count time a little different. This gives a difference between clock readings which is called skew. Although this difference between two clocks might be extremely small, the difference will over time grow no matter how precisely the two clocks was initiated.
Some times it is necessary to know what time of day events occur in a distributed system. For example, we require computers around the world to timestamp electronic money transaction. This makes it necessary to synchronize clocks with a common source, such that different processes on different machines agree on the time. This form of synchronization of clocks is called physical synchronization.
Unfortunately this process is not as simple as it sounds. Message transmissions vary, and different processes may be competing for synchronization. This has the unfortunate effect that completely physical synchronization is not possible.

5.9.1.1 Synchronization in synchronous system

In a synchronous distributed system, bounds are known for the drift rate of clocks, for the maximum message transmission delay, and for the time it takes to execute each step of a process. Unfortunately we only know the maximum message transmission delay and the actual transmission delay is subject to variation. In general processes may be competing for machine and network recourses. Nonetheless, there are always a minimum transmission time.
In a synchronous distributed system there by definition is a lower bound and upper bound for the time it takes to transmit a message. Let this message transmission uncertainty be u, then $u = (transmit\ max - transmit\ min)$. This means that in a synchronous distributed system, the optimum synchronization is achieved if all clients sets the time to $t + (transmit\ max + transmit\ min)/2$ where t is a time sent from a central point to all clients. But even with this algorithm absolute agreement of the time is not obtained.

5.9.1.2 Synchronization in asynchronous system

Most distributed systems found in practice are asynchronous. This means that upper bounds for message transmission are not known. This is particularly the case for the Internet. For an asynchronous system we can therefore only say that the message transmission uncertainty u is $u = (transmit\ min + x)$, where $x \geq 0$. The value for x is not known, although a distribution values may be measurable for certain networks.

5.9.2 Logical time and logical clock

On a single machine, events can be ordered uniquely by times shown on the local clock. Even in multi threaded applications or different processes events can be uniquely ordered by the clock. However as already mentioned in the previous section we cannot synchronize clocks perfectly across a distributed system, we can therefore not in general use physical time to find out the order

of events occurring. However instead of keep track of events by the physical time when they occur, events can be ordered by two obvious points [COU01].

- If two events occur in the same process, then they occurred in the order in which the process observes them.
- Whenever a message is sent between processes, the event of sending the message occurs before the event of receiving the message.

These two relationships are known as the happened-before relation or casual ordering.
With these two simple rules in mind a simple numerically ordering of events can be obtained. This numerical ordering is called a logical clock, and has no relation to the actual physical time.

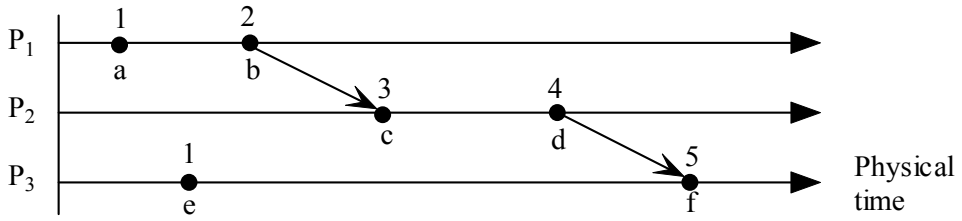

Figure 12, Time stamped events occurring in three processes

In Figure 12 three different processes are illustrated. Each process keeps track of its own events and corresponding time stamps. When messages are sent across the network their timestamp are sent with the message. When a message is received by a process this processes timestamp counter are increased to the timestamp in the received message. Finally the event of receiving a message further increases the time stamp counter in that process. By this simple mechanism ordering of events is possible. It should be noted that if two happened-before relation holds between two events, then the first might or might not have trigger the second. Another thing to note is that in two separate processes there might be events identical timestamps, this is the case for event *a* and *b* in the above example. In this case the actual ordering is not known, but it is guaranteed that both of the two events are independent of the other.

5.9.3 Relation to distributed games

In a classic client server game with only one server the synchronization problem is almost none existing, even though it is so important. This is mainly because the states are only present on the server machine, and each object on the server is actually controlled by the server. When a client do something it is not at that point changing a state in the actual game, it is only requesting a state change which the server can then process at a later point. If two players take action independently involving the same object the messages are handled in the order they are received by the server. This is because games should run with a speed where it is not possible to se who actually requested the state change first, and for performance reason nothing is done to check it. The synchronization is in other words implicit achieved and the goal is simply to distribute the server state as fast as possible, such that all clients match the server state as close as possible. You can say that all the clients' state is simply a delayed view of a server state. Another aspect of synchronization is that all clients display the same game state at the same time. However in a game where display updates easily gets above 30 per second this is very hard to achieve. The central point of control only makes sure the order of event is synchronized, not that all client states are in total synchronization with each other, this is however not that big an issue as long the clients are within or below 100 to 200 milliseconds of synchronization. In most circumstances physical time stamping is not an issue either. There might be places where it is a requirement but

it is not a normal requirement. The real problems with synchronization in games are if game states are present more than one place. This could be if more than one server is used to achieve scalability or fault handling, or if a peer-to-peer architecture is used.
Like in the client server architecture with central control these architectures also suffers with the problem of participants displaying the same picture at the same time, but as mentioned this is not that big an issue. Moreover each participant is responsible for the game state. This gives even more complicated problems where different participants can disagree on the order of events, and inconsistency arises.
Consider a first person shooter in a peer-to-peer where two different players independently shoots the same third player. The third player dies, but who gets the kill? Both of the shoot events are happening at the same time independently of each other so logical time ordering is not a possibility. Because of the transmission delay different participants gets the network packages in a different order and some clients decides to give player one the kill, other participants decides to give player two the kill. Now inconsistency has arisen. The solution here is the use of physical clock, but as mentioned 100% correct ordering of event using this method can not be guaranteed. The problem of achieving absolutely consistency in games with such high performance requirements unfortunately showed to be such a cumbersome task that it would be a whole project in it self. Therefore no solution to the problem will be presented here.

5.10 Processes and threads

Distributed games, like any other distributed system, has to consist of at least two processes. A discussion about how many processes a distributed system should consist of and how many threads every process should spawn are taken here. C# has the ability to spawn new processes from an already running process and to use thread programming.

5.10.1 Processes

The distributed game has to be divided into a number of processes, regardless of the choice of architecture, client-server or peer-to-peer. In the nature of distributed systems the system has to be executed on different processes, which eventually are on different computers connected through a network. In a distributed system based on peer-to-peer architecture, normally every peer is represented as a process, where every process is running on one specific computer. In a distributed system based on client-server architecture, normally the server is a process running on one computer and the clients are processes running on different computers. This gives a number of processes and computers equal to the amount of clients plus the server. But a computer can have more processes from a distributed system running. For example if a computer runs both the server process and one of more client processes. Another example is one or more client processes running on one computer.

If a distributed game is based on client-server architecture it must be decided how many processes the game must consist of. If a computer acts only as a client then only one process has to exist on the computer. This is the normal scenario for a real time distributed game, because only one person can interact with the computer at a time. If a computer acts only as a server then only one process has to exist on the computer. But a computer can both act as server and client. Then it has to be decided whether a single process should be both server and client or should be divided into two processes.

A single process as server makes it easier to dedicate a computer as server. It would be a waste of processor time, if a process was forced to run both client and server unaffected that no persons would interact with client part of the process. But on the other side it will be more complicated for users playing private games to explicitly start a server process.

The distributed game will have a process for every client or peer. It should be possible to start server processes in two ways. First way is to start a server process directly, so a computer can acts only as a server. Second a client should be able to create a server process, for users playing private games. The reason for not to combine the client and server in one process is that the client is wanted as thin as possible. In case of peer-to-peer architecture only one process per peer is possible.

5.10.2 Threads

The work that has be done in a process can either be done sequentially or in lightweight processes know as threads. Using threads can follow three models, thread pr. request, thread pr. connection or thread pr. object.

Sequentially work, means that all work the process must carry out happens in a specific order and is repeated until the process is aborted. For example if the server in a distributed game interacts with two clients, then communication and data processing with the first client is performed, then

communication and data processing with the second client is performed. The problem with the sequentially model is that the process are blocked for a long time (seen from a computers point of view), while communication happens between the server and clients, because network I/O-operations belongs to the slowest operation in a computer system and becomes a bottle neck. This brings down the overall performance of the process. If it is crucial for a process to continue, because data from a specific server or client process is needed, then the sequential model can be taken into consideration. Servers using the sequentially model are called iterative servers. Iterative servers are used when a client sends single requests to which the server in short time can send back an answer, for example daytime-servers. An advantage is that no need for synchronization is needed. In a peer-to-peer architecture the problem is the same as for the server in client-server architecture. The peer has to wait for the communication to all other peers before it can proceed.

When using threads in games, where performance is critical, three conditions must be taken into consideration.

- The amount of threads and their lifetime, in relation to memory consumption.
- How to synchronize the thread, if they have access to common data areas.
- How scheduling is performed and what the priority setting of the threads is, so no threads suffer to starvation.

Thread pr. request creates a thread every time a request/response is received. This gives a great overhead in creation and destruction of the threads and can not be recommended in a system where many request/responses exist. The advantage is the possibility to design different type of threads addressed to a certain type of request, where complex operation may be carried out. In a distributed game where the communication is simple and vast the system would suffer from the overhead in creation and destruction of the threads.

Thread pr. object, can be used in distributed system based on remote object invocations. In distributed games the use of remote object invocations would lower the performance of the system. A lot of processor time would be spend on marshalling data in the communication process and make the overall system performance too slow. Besides, a distributed game normally uses simple data types.

Thread pr. connection associates a thread with every connection and closes the thread when the communication process has ended. In a distributed system with a lot of request/responses and a requirement of low response time, it is an advantage that both the client and server process do not need to worry about creation and closing connections, but just uses the same connection every time. Memory and processor consumption will be low as long the amount of clients connected to a server or the amount of peers are low.

5.10.3 Relation to distributed games

Thread pr. request will only be suitable in tour-based games where information about game state only will be send when a move is made. Otherwise a distributed game will suffer for creation and closing of the threads every time a connection is created. The same for work carried out sequentially, where the sequence of who will move is known in advance.

The preferred model, in a real time distributed games like DWorm, will be thread pr. connection. Because in a synchronous client-server system, the client can not wait for a key press to happened and only then receive game states. This also requires reliable communication, because the distributed game otherwise could happened to block if some of the communication where lost. The communication has to be asynchronous. Using threads makes this possible.
Using threads provides a distributed game with a higher degree of scalability, because a thread can be responsible for creating new thread as clients or peers joins a game. Threads also make the concurrent nature of real time distributed games possible.

5.11 Analysis conclusion

The .NET framework was analyzed as target platform for distributed games and DWorm especially. The .NET framework can be used as target platform, but the special enterprise application features it provides do not make sense to take advantage of in relation to distributed games. The .NET platform can fulfil the requirement that DWorm should run on both a LAN and the Internet. Besides the .NET platform can provide heterogeneity.

Throughout the analysis, issues from the area of distributed systems were presented. On basis of this presentation a choice of architecture for the distributed game DWorm is taken. The conclusion on these issues will briefly be summarised.

Unreliable communication should be used unless it is communication concerning information vital, e.g. exchange of IP-addresses, to a game. The use of unreliable communication is due to keep network traffic at lowest possible level and gain the highest level of performance.

Distributed games and DWorm should be location transparent for a higher degree of user friendliness. In order to provide DWorm with location transparency two types of a discovery service was described. One for use on LAN where multicasting is possible and one for the Internet, which make use of the already existing DNS service.

A requirement for replication was defined, but through analysis it turned out that replication is not an issue for short termed real time distributed games. Replication is normally a solution to high availability. In case of short termed real time distributed games, cost benefit relation is too low. The probability of a situation where a replica is needed is too small compared to the work doing the replication. Maybe in long term real time or tour-based distributed games it would be an issue. Here a backup to replication could be done at specific checkpoints. When it comes to accounts in an online world replication should be a requirement, in order to preserve that account information does not get lost.

Scalability for distributed games can be divided into two categories for the client-server architecture, scalability for an instance of a game and scalability for game servers. It is often desirable to set a maximum limit of participants in a single game, but still maintain possibilities to host more games as a whole.
Scalability, when using the peer-to-peer architecture, avoids the single point of failure that a server makes. In relation to performance for the peers, the performance drops rapidly proportional with the number of peers because the communication connections rise dramatically. Seen from the users of a distributed game and a game company's point of view it is more desirable to let a central game server be connected through an Internet connection with great bandwidth, instead on having to rely on computers with big expensive Internet connections at home.

Using peer-to-peer architecture is more vulnerable to cheating and cracking. It is harder to keep central control of player accounts and more game state information has to be sent through the network, when the peers has to agree on the actual game state. Manipulating the communication between a server and client will not change to the game state, only the rendering on the client.

The client-server architecture will be the architecture DWorm will be design upon.

The conclusion on the reason why game companies primarily only uses the client-server architecture is the same for choosing it for DWorm. Designing a distributed game with as thin clients as possible makes it easier to let the client processes work with an astonishing graphical presentation of the game state, because all the game logic are handled by the server. Having all the game logic in the server makes it easier to keep to game state consistent. In a peer-to-peer distributed game it is a hard task to synchronize the different peer's clocks, so an ordering of events can be done. One could say that synchronization in a short term real time distributed game could take place when initializing the game. Because of the short term property, it is limited how large a skew between the peers clock can reach. But if peers disagree on the game state then consensus should be obtained. Using centralized client-server architecture the game state is only represented one place, at the server. Having a central server also provides central control of the game, for examples for log into accounts and updates. At the same time the server also becomes the single point of failure.

6 Design, implementation and test

In the following section we will describe and document the design, implementation and test of a distributed game, DWorm. The goal here is not to make a visually astonishing game with a great gameplay. The goal here is to try implementing a real time game with a strong requirement for consistency, such we get a more in depth insight into the techniques used for these kinds of applications. The game has been developed using an iterative approach. Below the main goal for each of the iterations is listed.

- **Iteration 1:** A single player version of the game.
- **Iteration 2:** Connect clients to a server and start sending messages to the server.
- **Iteration 3:** Creating multiplayer game playable on local area network using multicast.
- **Iteration 4:** Making the game playable on the Internet.
- **Iteration 5:** This iteration is a short and theoretical iteration describing various issues that could be cleaned up or changed in the application.
- **Iteration 6:** Create a game service environment with the ability to host games in a group of servers.

6.1 Iteration 1, Single player (client)

The purpose in this iteration is to get started on the product and get the basics implemented. Although the idea is to make a distributed game, we need to have other game related problems solved before we can begin the implementation of network communication etc. Below the goals for this iteration is listed.

Main goal

- A single player version of the game.

Sub goals

- Get started on and get a first hand impression of C# and the .NET framework. Learn the basics syntax and learn how projects are structured.
- Set up a DirectDraw window.
- Render and move DirectDraw primitives.
- Create a game engine that processes user input and render graphics at optimum performance while still making sure movements for game objects are identical on different machines.
- Implement a worm that represents a player in the game.

6.1.1 DirectDraw

Besides the minor challenge of getting started with C# and .NET our first real challenge is to set up a way to render high performance graphics through the DirectDraw API. DirectDraw is a technique where images are processed and drawn directly in the video card. Instead of creating a standard windows form with a canvas to draw on, you have a coordinate system in the video card where an image is build and displayed. The advantage of DirectDraw compared to a standard windows form canvas, is a tremendous performance boost.
DirectDraw can be either window mode or full screen mode. In full screen mode DirectDraw and the application that created the window has exclusive privilege to render visible graphics. Full screen mode is the widely used graphical mode for games, since it gives the best performance. Games are however also the one application type that uses this mode since you lose the ability to have more than one visible application/window opened at a time. For that reason it was a requirement to be able to run in windowed mode because debugging would be near impossible in a full screen application.
The exact technique behind the two modes is a little different. Basically there are two memory areas in the video card. They are known as the front buffer and back buffer. The front buffer is the one that is visible on the screen. The back buffer is where an image is build before it is displayed. In each game cycle an image are build in the back buffer. When an image has been build in the back buffer it is displayed in the front buffer and the game takes a new game cycle. In full screen mode the displaying of a new image is done by making a pointer switch between the front buffer and back buffer. In windowed mode the back buffer is copied to the front buffer. This is because in window mode the position of the front buffer might change, therefore it is necessary to specify the position for the rectangle representing the front buffer. In window mode it is also constantly checked if some areas of the DirectDraw surface are hidden behind another window. If this is the case, the hidden area should not be painted in the front buffer.
The reason for having a front buffer and a back buffer is to avoid what is known as tearing. Tearing is a flickering effect that can be seen when an image is drawn while watching it. No image can be drawn in an instance. Therefore it is built in the background and first displayed

when it is done. The previous picture is then shown while a new picture is drawn in the background.

6.1.2 Game engine

The way many games are running is a little different then many other applications. In many business applications, the program waits for input from different input devices. This input is then processed accordingly. If no inputs are given to the program, it will wait passively and do nothing. This is also the case for some game types. A computer chess game has much of the same characteristics. However a real time game does not wait for input to process. The reason it is called real time is because the game state are constantly changing in real time whether it gets any input or not. This sets some different requirements to how the main user interface is designed and implemented.
The goals in these kinds of games are to take full advantage of the processor and video card speed without blocking the user interface for input. You also have to make sure that the game will run at the same speed on fast and slow machines. A faster machine should however take advantage of a smoother gameplay and more frames per second.

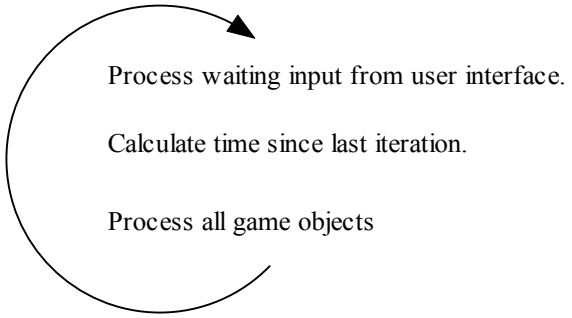

Figure 13, Flow of the game loop

Above in Figure 13 the flow of the game is pictured. First all waiting input from the user interface is processed. If this was not done the game would not respond to any input at all. Then the time since last iteration is calculated in milliseconds. Finally the calculated time is parsed to all the game objects, which are then processed. All objects that need to be processed in each game loop inherit from a common interface. This interface has the following method *DoRenderObject(float deltaTime)*. If necessary a game object can then multiply an operation like a move with *deltaTime*. This ensures that the speed of that object will be identical on different machines.
Another thing to handle in a game application is key presses. To get smooth gameplay it is not enough just to process keys when they are pressed. For example in a word processing application the effect of holding down a key can be seen very clearly. In short intervals a key press event is sent to the application. This roughly generates 10 key presses per second. However in a game running with 100 frames per second this would give a very stutter gameplay, since the object only moves every one out of tenth frames. It is therefore not acceptable just to move when a key press event is generated. Instead this is handled by having an array of booleans. On a key down event the corresponding place in the array are set to true, on key up event it is set to false. A game object that relays on keyboard input can then in each game cycle read whether a specific key is pressed or not.

6.1.3 Class diagram and description

Below the class diagram for the game in this iteration is pictured.

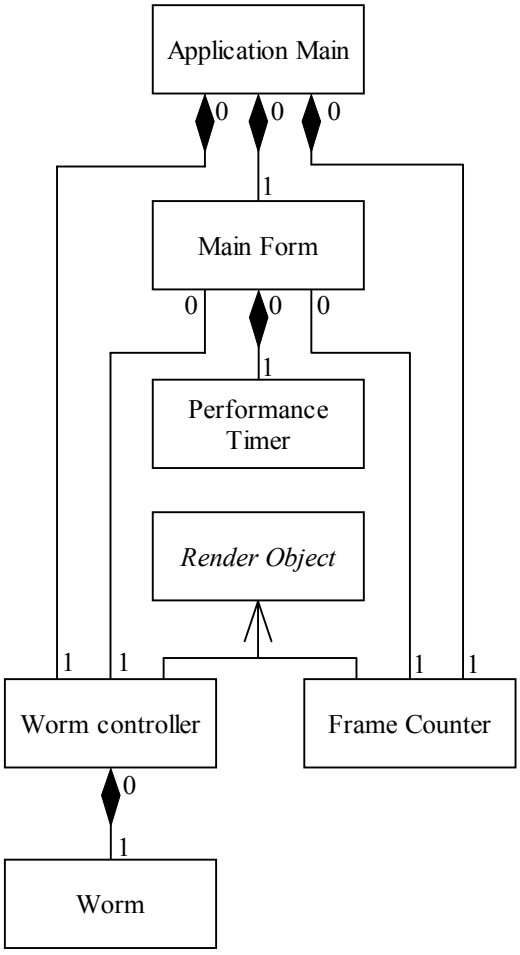

Figure 14, Class diagram for first iteration.

Application main

This class represents the entrance point for the application. It only has one static method which is the main method. This method instantiates the user interface and the objects our game will have. In this case it is an object to that controls and draw a worm and a frame counter.

Main form

This is the user interface in the application. It can either be created in full screen or windowed mode. From this class a reference to a drawing surface and a reference to an array indicating what keys are pressed can be obtained. Objects implementing the Render Object interface can be added and removed. The main form will then in each game loop call *DoRenderObject(float deltaTime)* on them.

Performance timer

This is a high resolution timer. A high resolution timer is a timer running with a very high precision. On a standard computer the system clock runs with a resolution 1000 tics per second. This class encapsulate functionality for a very high resolution timer. The exact resolution is processor speed dependent. Not all computers do however support this timer in which this class

will use the standard system clock. The timer class is used to time how long it takes to run a game loop. This value is then parsed to all render objects.

Render object
Render objects are an interface forcing inherited classes to implement the method *DoRenderObject(float deltaTime)*. These objects can then be added to the main form. It is the main form that is responsible for running the game loop and process user input. The main form will then in each game loop call *DoRenderObject(float deltaTime)* on each game object added to the main form. If a render object needs to get info on key presses, a reference to an array indicating what keys are pressed, can be obtained from the main form. If a render object is a graphical object the main forms back buffer draw surface can also be obtained. The goal with this approach was to make the user interface independent of the actual game. Instead render objects can be added and removed from the game without messing with the actual user interface.

Worm controller
The worm controller's job is to control the game worm on behalf of keyboard input, and draw a graphical representation of the worm. Application main are responsible for obtaining the keyboard array and graphical render surface and pass to the worm controller in its constructor.

Frame counter
The frame counter calculates the amount of game loops per second. This number is then displayed on the user interface. Application main are responsible for passing the graphical render surface in the frame counters constructor.

Worm
A worm represents the object in the game which the player can control. A worm has methods to change its direction and move it. It also has methods to check for collision with other worms. Its position is obtained by an array of coordinates that represent its current position.

6.1.4 Conclusion

In this iteration we got started and got the first feeling of C# and the .NET framework. First impression of this language was that it was useful in context to our problem. DirectDraw was implemented using DirectDraw version 7 and the application supports both full screen and windowed mode. The user interface is very responsive while the application is running with 100% CPU usage. The game is running smooth on an average machine for today's standard. On different machines it also seems that the speed of the worm is identical. The goal of creating a single player version of the game was obtained.

6.2 Iteration 2, Client and server

In this iteration the goal is to give clients the ability to find a server, connect to the server and start sending messages to the server.

Main goal

- Connect clients to a server and start sending messages to the server.

Sub goals

- Implement network communication.
- Implement a discovery service.
- Ability to send key presses from a client to a server.

6.2.1 Discovery service

Two models of the discovery service were proposed. The multicast model for use where routers support multicast and a model with a central server based on the existing Internet service DNS. Only the multicast model is implemented due to the time limit of the project.

The multicast model was considered implemented in two ways. Both solutions make use of threads.
First solution was to let a thread in the server process run a multicast sockets which listen for incoming multicast messages. Then the clients have to multicast out, when they wishes to join the game and the servers answers. All the time the thread in the server process listens for incoming messages (a blocking call) it takes up processor time every time the thread are scheduled. Although it is a small amount of processor time, it is after all an amount of time.
Second solution is to let the clients listen for multicast messages. In this solution a thread in the server process instead multicasts out a message with a given interval. In the actual implementation the interval are per 3000 ms. This is done by letting the thread enter a sleep state. Three seconds is not a long time seen from a human being, but a processor can perform a lot of work in three seconds.
Performance in the server process is of great importance, therefore the second solution was chosen. The message, the thread in the server process multicasts, is an array of bytes of the length four containing the IP-address of the computer that runs the actual server process.

6.2.2 Communication creator

The communication creator is a data structure used to control communication and network connections. It furthermore provides some methods for converting between IP-addresses and byte arrays and the other way around. The communication creator are developed as a separate component and it is used by both the server processes and the client processes.

One could say that the communication creator carries out the trivial work in creating and closing the different network connections. It contains a number of methods which returns different types of sockets. Three overall types of sockets can be returned. The multicast socket in four variants; a sender and receiver for the discovery service, a sender and receiver for the graphic data. TCP sockets in two variants; a server socket and a client socket. UDP sockets in two variants; a server socket and a client socket. The data type, all the sockets uses, are byte arrays which is the most simple data type a socket in the .NET platform can make use of.

The reason for collecting all the network connection control in one data structure is the possibility for easy changing and extension of the control. Which ports to use for the different types of communication and all information about IP-addresses for multicast is located in the communicator creator. By having a total central control over IP-addresses and ports minimizes the risk of logic errors in the program where a port would be explicitly stated.

6.2.3 Sending key presses

During game run after the initial communication has been carried out the only messages the client need to send to the server are key presses. The main problem here is that you need fast performance and some degree of reliability. It is not desirable that a key press is reflected one second too late, and moreover it is not desirable to lose key presses so they will not be reflected at all.

One way is to use a reliable protocol and then only send a message when a key is pressed or released. When as an example pressing the upwards arrow key, a message indicating the key has been pressed will be send to the server. The server will keep this key in a pressed state until a key up message for this key is received. In normal circumstances, when holding a key down on a keyboard, it will keep generating key down events. The following key down events can be filtered on the client to save network traffic. Since a key state is only sent when they are changed a reliable protocol is needed for this approach. Using an unreliable protocol could cause key state changes to be lost, causing, as an example, a player to keep moving although a key has been released. The big drawback with this approach is the increased time overhead reliability needs, and that following messages can not be delivered before presiding messages. On a small LAN network this should not cause a problem. On a WAN like the Internet where collision possible is more frequent, and messages have an increased tendency to be delayed, can however give less desirable results. If one message gets delayed all the following will be delayed as well. The option is then to either complete all requested key presses delayed, or simply discard all of them.

Another approach, and the one we actually decided to implement, is to send all key states in a well specified interval. Using an unreliable protocol, keys that are currently pressed, are sent with a specified rate. If key up and key left are pressed for a second, all messages sent in this interval will indicating these keys to be down. The key in this approach is to find a balance between sending key states often enough to avoid the feeling of sluggish control and not burden the network with to much traffic. Depending on the game type, and overall requirements of the application, the send rate is typically between 10 and 30 milliseconds. This equals that approximately between 33 and 100 messages are sent per second. Because this method is using an unreliable protocol, we do not have any guaranty if a message will reach its destination, or they will be received in a correct order. However the delay of one message will not block the following messages. If a player presses forward but the first package indicating that is lost. The following package will still indicate a forward move. This might actually be faster than waiting for a reliable protocol to guarantee a delivery. If it is undesirable to receive packages in random order, it is possible to keep track of packages by giving them a number, and just take advantage of the fastest delivered package. Late arriving packages can then be discarded without any trouble. To keep the network package as small as possible one bit can be used per key. This gives the ability to have eight keys per byte in the byte array send across the network. As said the drawback is the increased network traffic this method requires, it is however faster than using a reliable protocol. But the actual required network traffic required can be minimized to a bare minimum. Let’s say that a game requires the use of 16 different keys and we sent them with a 20 millisecond interval,

then we are sending 16 bit 50 times per second. That gives a total bandwidth consumption of 100 byte per second which is next to nothing.

6.2.4 Class diagram and description

In this iteration the server and the client are developed as two separate programs. First the client is described, afterwards the server is described. In this iteration the client are still responsible for render and controlling a worm, the main difference is that the client will connect to a server to which key presses can be send. The server then simply displays these key presses. The server does in other words not have a real functionality at the end of this iteration.

6.2.4.1 Client

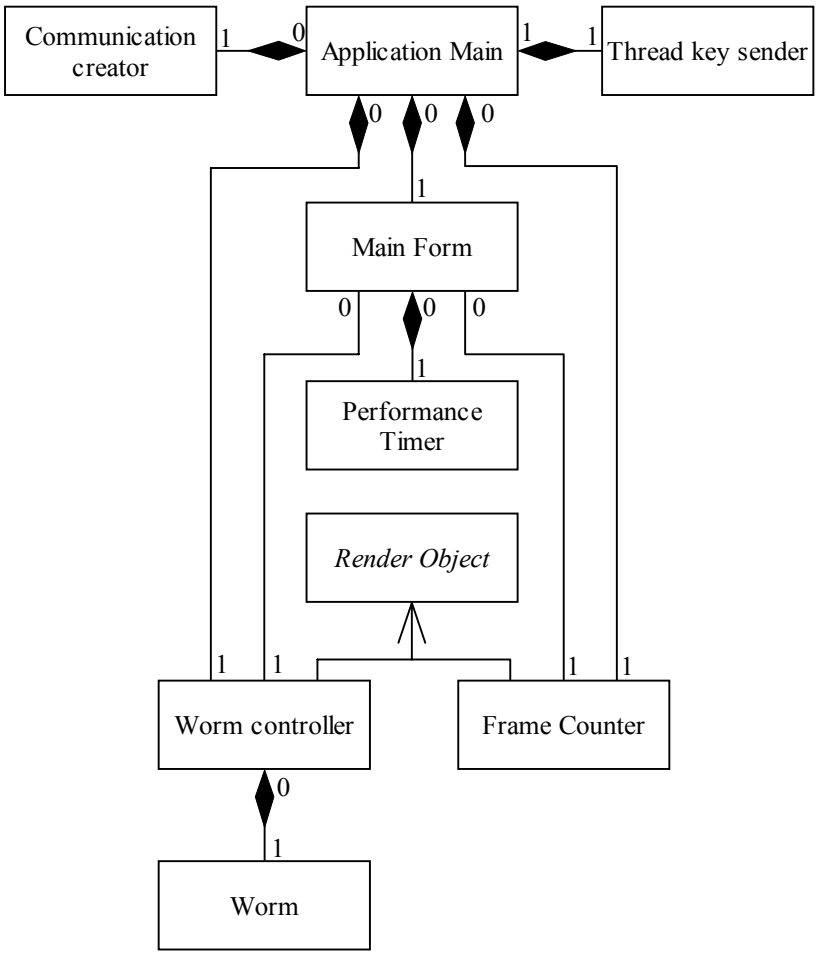

Figure 15, Client class diagram.

None of the classes from the previous iteration has changed in this one. Only the main method now initializes some more. However two classes have been added, the communication creator and the key sender.

Communication creator

This class is responsible for creating different kinds of communication sockets. It also have methods for listening for a server multicast message. If a server is found the server IP are obtained and used.

Thread key sender
This is a class running as a separate thread. It is responsible of sending key states to the server. The sender socket is parsed to it by the Communication creator. A reference to an array indicating what keys are pressed is obtained by the main form. Key presses are not sent in each game loop, this would simply flood the network with to much traffic on a fast machine. Therefore is this class not inherited from Render Object. Instead key presses are sent in short intervals. The main form does however updates the key press array in each game loop.

6.2.4.2 Server

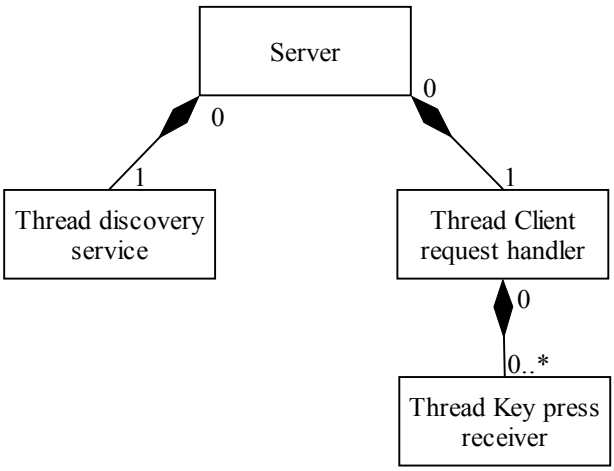

Figure 16, Server class diagram.

Server
This class is the main entrance point of the server. Its only method is the static main method. It spawns all threads in the server and then goes in a state where it waits until it should close. It then kills all running threads and shuts down.

Thread discovery service
This class is a thread that, in intervals, multicast a message with the servers IP address. This message can then be picked up by clients such that if a server is running on a LAN it can be found without knowledge of its IP.

Thread client request handler
This class runs as a thread and listens for clients contacting the server. This class is then responsible for creating the necessary threads for each client.

Thread key press receiver
This is a class created for each client contacting the server. The class is created as a thread. In its current implementation it is responsible for showing what keys the clients are pressing.

6.2.5 Conclusion

We got the communication to work and were able to have a server reading what keys different clients were pressing. The discovery service was also working properly on a LAN. When a client is started it is able to automatically find and connect to a server. However on the Internet the discovery service is not working. This indicates that it was not possible to find a route between the test machines with multicast enabled routers.

6.3 Iteration 3, Multiplayer

In this iteration the goal is to make a working multiplayer game. Clients should be able to connect to a server and control their corresponding worm. The server should be able to multicast information needed by the clients such that the game state can be rendered on all clients.

Main goal

- Creating multiplayer game playable on local area network using multicast.

Sub goals

- Move game logic from client to server.
- Control a worm on a server on behalf of client messages.
- Multicast server state from server to clients.

6.3.1 Server to client messages

Besides sending key presses to the server, the client's job during game run is to act as a view port into the server state and render a graphical representation of an actual scene. Again the main problem here is to achieve fast performance. From the individual players point of view the feeling of fast reflection of key presses, is only achieved if the roundtrip for the client to server key press message and the following server to client screen update message is small. The goal her is to get below 100 milliseconds, but less then 200 is moderately acceptable, from that point delays starts to be critical and noticeable. Player positions on the screen also have to match player positions on the server as close as possible. Having scenarios where two players have a collision on the server, but where the screen updates are so slow that both the players still sees a scene with no collision is not desirable. But if the key press roundtrip delay is achieved this goal will be trivial achieved as well.

The frequency of the screen update messages will decide the actual frame rate on the clients. Although a client might render 100 frames pr. second, you do not get 100 different frames if only 20 screen update messages are received in a second. The problem here is to achieve fast performance and sending messages with a frequency that gives a smooth gameplay, without putting to much pressure on the network. Reliability here is of less importance. If messages are lost we will simply get a little less smooth animation on the clients. This will not cause a problem. If 100 screen messages are sent per second we can loose half of them and still achieve an acceptable frame rate of 50 considered that the 50 lost messages are an evenly spread of the 100 messages. Again like when sending keyboard presses, the send rate is typically between 10 and 30 milliseconds, depending on game type and requirements.

It was a design criteria to design our client as thin as possible. It does however need to have knowledge of the objects in the game. In theory a pixel by pixel description of the scene could be sent. This is unfortunately much to slow and is not possible for games which do require real time performance and high frame rates. Not only would it require and extreme amount of network traffic just to sent a 16 bit 640x480 picture description, but it will also be way to heavy to decode and encode on the client and server, and it would not take advantage of today's 3D accelerated hardware. Instead the whole level and all game objects are known on both server side and client side. This limits the amount of information to things like coordinates, direction, speed, specific states or other things that is changing during game run. But the graphic engine and how objects are drawn resides on the clients.

All game objects existing in the game needs to have a data structure containing all the changeable information for that object. This data structure must always have the same size, or in other words it should not be possible to use dynamic array or alike in this structure. This is because the client need to know the exact size of the data it receives, and because we need to know the sizes of objects when we serialize them to byte arrays. During game run this structure will constantly get updated by the game logic as the objects gets moved around. When a package is sent from the server to the clients, these data structures are serialized into a byte array for all game objects. The clients then deserializes this array back into game objects and draws the scene. Serializations are done simply by taking a game objects information structure and make a memory copy to a byte array. All game objects byte arrays is then assembled in one big array and sent. Figure 17 illustrates the principle of serializing a game objects into a network packet.

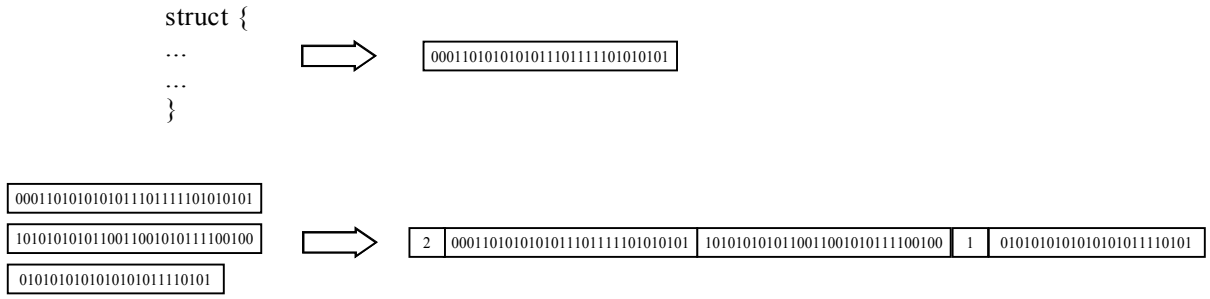

Figure 17, Serialization of game objects to a network package.

To make decoding simpler and the network packet smaller, client and server knows the order of different objects in the array. A block containing a group of the same object type is always preceded by a number indicating how many objects of this type the byte array contains. When a client decodes the array illustrated in Figure 17 it first reads that this array contains 2 objects of type one, and then decodes them into structures of this object type to override the old values. Afterwards it knows one object of type two needs to be decoded. Again decoding is done simply by making a memory copy of the array into a structure of this object. Finally when decoding is done for all objects, their states will match the state on the server and they are rendered.

6.3.1.1 Text message approach

Instead of sending game object structures across the network, text messages could be sent instead. For different objects information like position, rotation and other relevant info would simply be sent as plain text. Each position in the byte array sent on the network would then hold one character which corresponds to exactly one byte. The advantages compared to the selected approach can however be hard to find.

Network packages would in most, if not all circumstances increase. First of all some kind of character would be needed to separate tokens. Secondly numbers will in most cases take up more space stored as strings. As and example an 32 bit integer will always take up four byte stored as a number, but as string it can in worst case take up to ten characters which equals ten bytes. Moreover helper tokens might be used to separate different sub string blocks in the final string. Taking all these things into consideration a total package size increase of 100 byte is not unrealistic. If such packages are sent 50 times per second the total bandwidth usage, increases

roughly by 5 kilobyte per second. In other words, sending network packages as plain text requires more overhead data than affordable.

Another problem with plain text is that this approach is actually more difficult to implement. Make functionality to encode these packages from the corresponding game objects and then again decode and update objects is not as easy as just make a plain memory copy from and to structures. Specific code to parse information for each object would have to be written. Updating or changing game objects with new state variable would also require more work. Code is simply harder to maintain since a changes in game objects also requires changes in the code that parses the network messages and updates the game objects.

6.3.1.2 C# serializable approach

In C# it is possible to mark classes or just attributes in a class as serializable. From the programmer's point of view, the process of making a class or some of its attributes serializable, is just to mark the class or attributes with the serializable marker. Then it is possible to make an instance of a class, or some of its attributes, persistent or sending and receiving through a network.
By the use of serialization the information about worm instances easily can be sent through the network. The advantage is that objects can easily be made serializable. The drawback in relation to distributed games, with high performance requirements, is bound to system performance, because sending and receiving serialized object through network connection requires use of streams. Streams require more steps of wrapping which takes more resources than explicitly making the serialization of required data. By explicitly state which data to send and receive, and explicitly decide how to serialize it, one can make the process more efficient, because sending metadata about data can be spared. Beside, the exact amount of data send through the network can be selected.

6.3.2 Class diagram and description

In this iteration the worm and the controlling of objects are moved from the client to the server. The client are sending key presses to the server, the server are then running and controlling the game. Messages describing the state of game objects are multicasted from the server.

6.3.2.1 Client

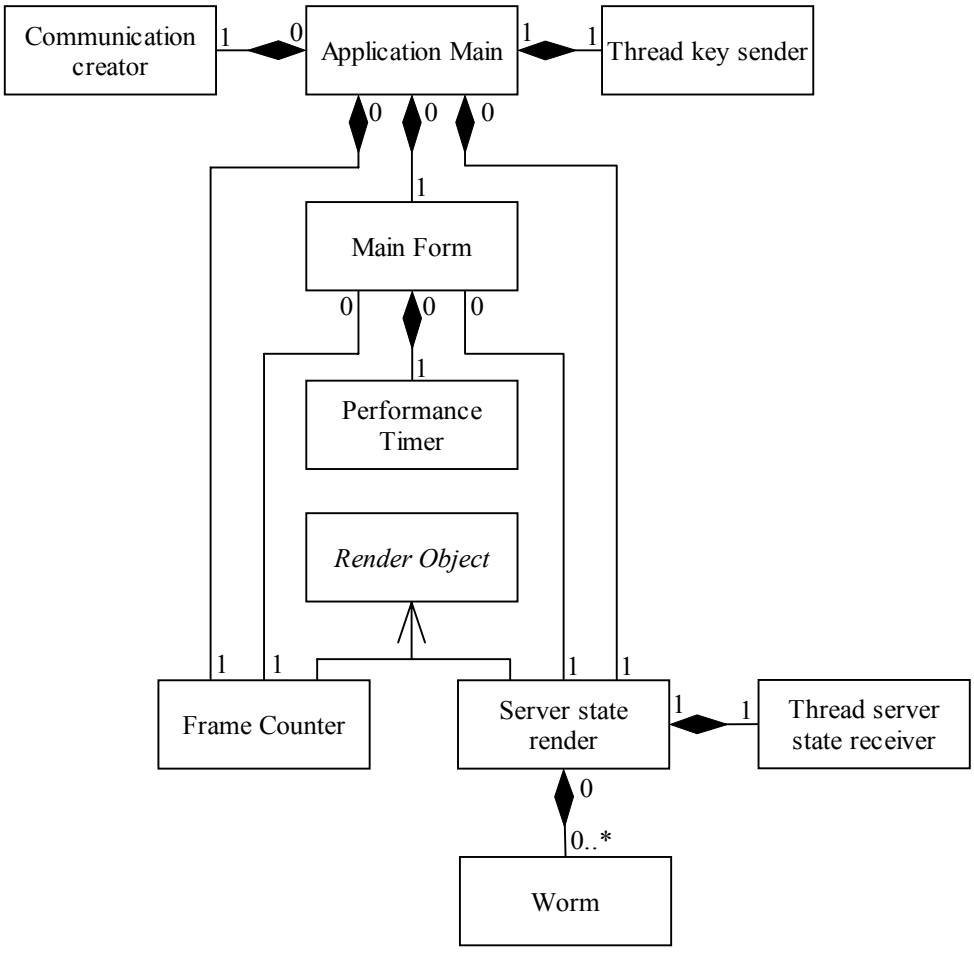

Figure 18, Client class diagram.

The actual changes in the client from the previous iteration are that the worm controller has been changed to a server state render class. This now receives the server state from Thread server state receiver thread.

Thread server state receiver
This class is running as a thread. It is responsible for receiving server state description packages. When a package is received it is returned to the server state render class.

Server state render
This class is responsible for decoding server state packages, updating the states of all the worms, and render the graphics accordingly. This class will in most cases render the same graphic more than once simply because updating packages will not be received from the Thread server state receiver often enough. However this model with the receiver running as a thread, the engine will not wait in a blocking state and wait for a new package to be received.

6.3.2.2 Server

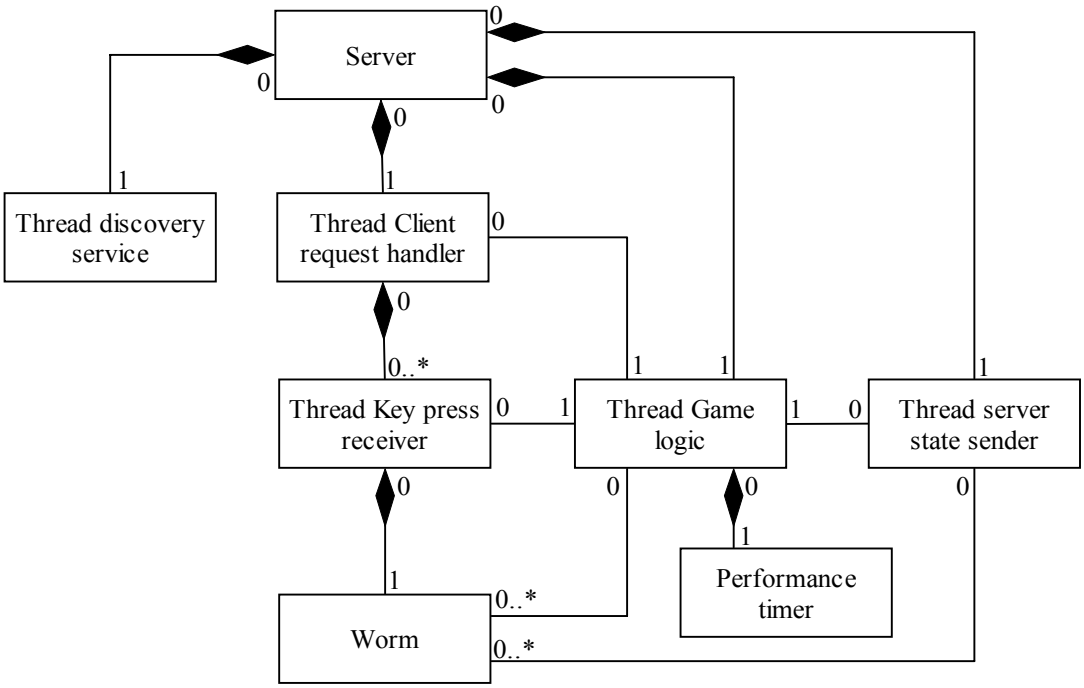

Figure 19, Server class diagram.

Most of the server has been changed from previous iteration. The server class is still responsible for creating and shutting down most threads and classes. The discovery service is the only class that have not changed at all. Rest of the classes will be described here.

Thread client request handler
This class is running as a thread. It is still responsible for receiving requests from clients that wishes to join and leave the game. When a client wishes to join the game, a key press receiver thread is created and a reference to the game logic is parsed to this key receiver thread.

Thread key press receiver
The key press receiver creates a new worm which is added to the game logic. When a client is leaving the game the corresponding key receiver thread is killed and the worm is moved from the game logic. This thread keeps a reference to the corresponding worm such that its direction can be changed on behalf of client messages.

Worm
This is the object representing a player in the game. At a later point making all game objects inherit from a common interface, like the render object on the client side, would be convenient. However in the current implementation a worm is the only game object.

Thread game logic
This is the thread responsible of moving the worm and checking for collisions. The performance timer is the same class as previously described in iteration one and which also resides on the client. One of its purposes is to make sure that an object is moving at the same speed on different machines.

Thread server state sender
This thread is responsible for reading the state of all objects in the game logic. This thread then compose the network package and multicast it through the network.

6.3.3 Conclusion

DWorm was implemented as a running multiplayer game. However it is only running on a LAN. This is because the server states are being multicasted and all routers between client and server has to be multicast enabled. This is unfortunately not the case for most routers on the Internet. The game is actually running very smooth and is very playable, especially on a LAN which has high bandwidth and low latency.

6.4 Iteration 4, Internet

The purpose of this iteration is to make the game playable on the Internet. This involves the ability to connect directly to a server without rely on the discovery service to find a server for you. Secondly the server state has to be unicasted to all clients.

Main goal

- Making the game playable on the Internet.

Sub goals

- Give clients the ability to connect directly to a server.
- Sending server state to clients using unicast instead of multicast.

6.4.1 Client

No new classes have been added to the client since last iteration. The Thread server state receiver now uses a unicast receiver socket instead of a multicast receiver socket for incoming server state packages.
When launched, the client now shows a dialog box, where a user can enter a server IP, if no IP address are entered the discovery service will search for a server to join. The dialog box is shown in Figure 20.

Figure 20, Dialog box displayed when the client are launched.

The overall class diagram is however still the same as the one in iteration 3.

6.4.2 Server

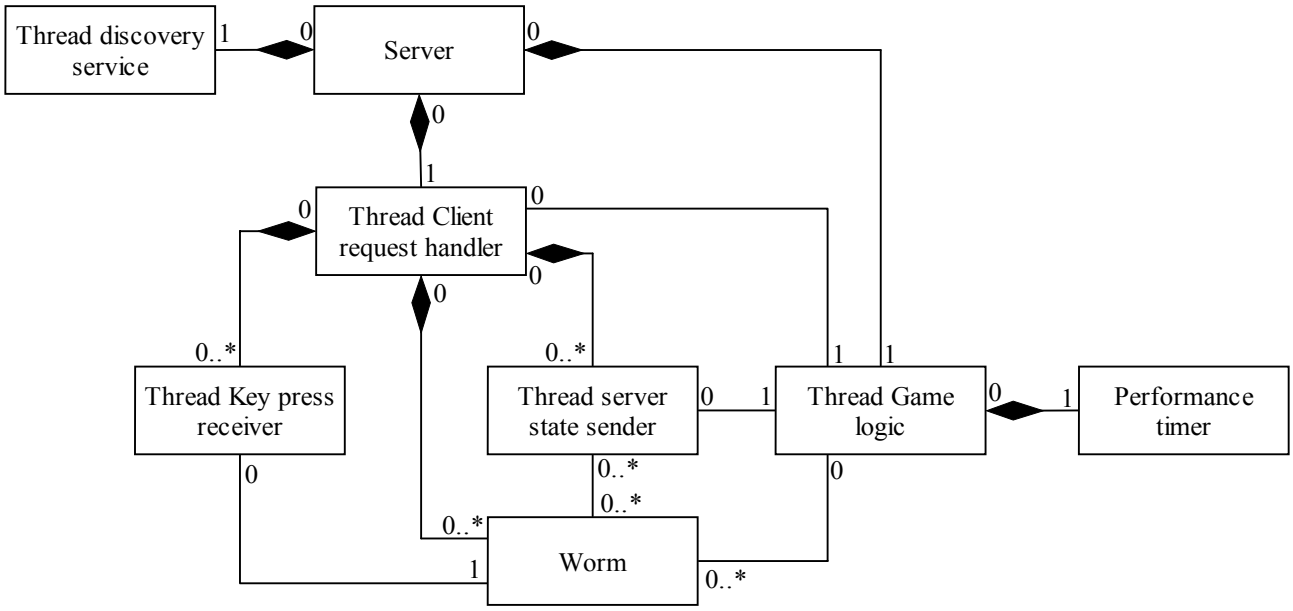

Figure 21, Server class diagram.

The primary changes on the server side in this implementation are that the server now has a server state sender thread for each client. The classes with changes since last iteration will be described here.

Thread client request handler
This thread or class are still responsible for handling clients connecting to - and logging of the server. Its main difference is that this thread now creates both a client receiver thread and a server state sender thread for each client. Moreover it now creates a worm for a player and parses a reference to the key press receiver and to the game logic.

Thread server state sender
A thread is now created for each client, where it unicasts the server state to a specific client.

6.4.3 Testing on the Internet

In the first implementation of the network communication the primary goal was to make it work. We wanted the ability to connect several clients to a server and play DWorm. Optimization was not the primary concern, before we had the ability to se how well the game performed on a network.

In the first implementation the client was sending key press messages every 20 millisecond. The actual package was a byte array of size four. Only the four arrow keys to move around were needed. Each byte could either be zero or one depending on whether the corresponding key was pressed or not.

The real concern was the server to client message. They are considerably larger than just the key presses. Moreover the server has to unicast these packages to all clients. These messages are sent

every 20 millisecond. Below is the struct that are serialized to byte array for each snake. This struct is then deserialized back into a worm on the client and then rendered.

```
struct Worm
  {
        public int          Color;
        public int          HeadSize;
        public float        HeadX;
        public float        HeadY;
        public int          TailSizes;
        public int          TailLength;
        public float[]      TailX = new float[29];
        public float[]      TailY = new float[29];
  }
```

On LAN, the size of these packages showed no problem. DWorm was running smooth. When testing on the Internet it unfortunately turned out to use too much bandwidth. The server simply was not able to transmit such a package for just two players every 20 millisecond through the outgoing 128 k/bit connection, we tested it on. We could then either accept that a server simply does need a network connection with a bigger bandwidth, or try to lower bandwidth consumption. The main problem for us here is that our worm game object is pretty big since coordinates are sent for each tail region. Having a game object with only coordinates, rotation, and some Boolean indicating status, would lower the network traffic quite a lot but also make the client thicker.

6.4.4 Conclusion

The game now runs on the Internet. However bandwidth consumption has increased dramatically and the server now has to handle a grater amount of threads. The sender and receiver thread for each client are the main reason for the increase of threads on the server. They are however put in a sleep state after each run, which limits the overall CPU usage used to context switching such that the problem is not that big. The bandwidth consumption on the server is however so big that a low end Internet connection can become a serious bottleneck.

6.5 Iteration 5, Optimization

At this point an iteration involving various optimization and changes to the application would be appropriate. This could involve things like:

- Making the server in such a way that all objects in the game are inherited by a common class or interface. The server could then be redesigned in such a way that it would make use of polymorphism. This would make the worm class be a part of a bigger class hierarchy and making other objects easier to add.
- The key press messages send from client to server can be made smaller. In the current implementation each key uses one byte in the byte array. This could be optimized such that each key only uses one bit. This gives the ability to have eight keys per byte.
- The snake could be changed in such a way that it moves in steps. This would make the worm movement less smooth. The advantage of this approach is that the last bodypoint in each move is simply moved up as the front body point. This would save a lot of traffic since coordinates for each body point did not have to be in each package. Instead the coordinates for the head are only sent. The client can then by it self remove the last body point up as the head. All the middle body points are then not moved at all. This kind of move is also the way most worm games are implemented. We did however prioritise a smooth movement higher and did not foresee the consequences of the bigger network package.
- Some of the data types in the package sent from server to clients should be made smaller. This could be done by changing some of the data types in the network package. Some of the integers can safely be converted to bytes or shorts. Further more the tail coordinates can be rounded on the server side. This gives the ability to use a 16 bit short data type instead the 32 bit float used at the moment.
- If a client crashes or in other way leaves a game in the “wrong” way a server should timeout the player and remove the worm from the game. This could be done by having heartbeats between a server and its clients.
- In the current implementation each server state sender thread creates its own packages and sends. At a later point this can be optimized such that a network package is only created once. Each thread then simply send the latest build package. This will limit CPU time usage.
- Making check on the server to see when maximum number of clients has been reached. If the maximum is reached, new clients should not be allowed in the game. At the current implementation clients can keep joining until the game gets too crowded or until the server crashes.

However because of the limited project time left, and that these problems are not directly related to distributed systems we have chosen not to implement these changes. Instead we are moving directly to the next iteration.

6.6 Iteration 6, Group of servers

In this iteration the goal is to make the game run in an online server group. The game should be able to run in an environment where more servers are hosting games. Users should then be able to connect to this game service and obtain a list of games hosted. Furthermore the clients should be able to join the ones that are not currently at the maximum amount of players.

Main goal

- Create a game service environment with the ability to host games in a group of servers.

Sub goals

- Ability to host a great amount of games.
- Obtain the ability to take advantage of network load balancing.
- Obtain greater availability such that server crashes do not paralyze the whole game service.

6.6.1 Description of game service

Consider an environment where an amount of games are hosted. This could be three computers each hosting 5 games on different ports. Each game can handle maybe four players. This gives a total player capacity of 60 players. Clients or users of this game service could then contact this environment and be presented with a list displaying these fifteen games and the amount of players in each. The player could then select an appropriate game to join.
It is important to notice that games are not dynamically created and destroyed. The service has an amount of games running which can than be joined and left as desired. This does mean that a scenario with games running without players would be likely.
This system does not support a complete dynamically scalable environment. First of all, each game has a maximum number of players allowed. Further more the service are running a pre defined amount of games. It should however be scalable in such a way that if popularity increases hosting of more games should be easily allowed.
If a game suddenly crashes, the service should in a limited amount of time discover that this game is not available, and the game should be removed from the list presented to players joining.
Games in progress and players connected to the game that crashed will be disconnected and the game will be lost. The players will however end back at the list presenting the remaining games still running. But since the game type is short termed games crash and loss of them is not that big an issue.
This service will not give failover handling or ability to keep playing the same game on a crash. It will however be a model where crash of a machine or a game will *only* set a player back to a game selection screen. The service as a whole should still be running.

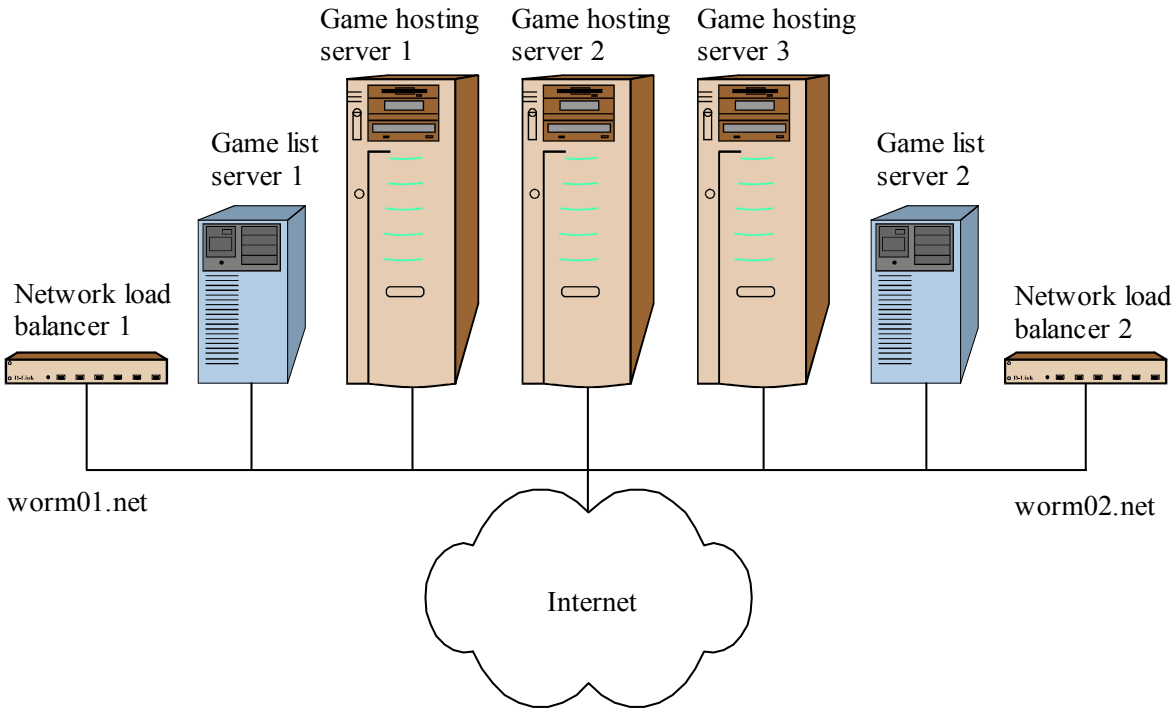

Figure 22, Example on how a game service could be build.

Figure 22 illustrates how a game service could be hosted. In this example the service can be contacted by two addresses. One of two network load balancers is then contacted. All they do is redirect a contact to a game list server. When a client has contact to one of these servers a player should be presented with a list of games currently running on the game hosting servers.
The primary job for a network load balancer is to keep track of what game list servers are available, and then redirect all contacts to one of those servers. This can be done by various techniques like round robin or fewest connection, etc. The advantages of this approach are that the network load balancer has very little responsibility, and on the servers behind with a greater responsibility, traffic can be more evenly spread. If one of these server machine crashes the network load balancer should automatically stop redirecting traffic to this server. Further more hardware implemented load balancers can give a higher degree of stability and availability. Unwanted traffic can often be filtered more effectively and auto reset on faults is often a possibility. Total stability of a load balancer can of course never be guaranteed, that is why the above example has two entrance points. Again depending of needs this can be scaled accordingly. Although the network load balancers are the entrance point for the service, the other machines should also have a public IP address on the Internet and a direct connection to a game list server would the also be possible.
The responsibility of the game list servers is to keep track of games running on the hosting servers, how many players are in each of the games, and other info that might be relevant. More than one of these servers are desired, this avoids that the crash of one server paralyzes the whole game service. Players can then select a game, be redirected to the game and start playing.
Servers and games can crash. Players connected to games or machines that are crashing should re-contact to an entrance point of the game service. The load balancer should then make sure a client is reconnected to a game list server that is still running. The game list server should then shortly afterwards notice that some games have disappeared and updates its list accordingly. The whole system should also be designed in such a way that server machines and games can easily be

removed and added. On server crash, all it should take to be running again is a restart of the game or the whole machine.

6.6.2 Client

No new classes have been added to the client in this iteration, but some basic functionality has been changed.
The dialog box when starting the client has been changed a little in this iteration. Previously the dialog box contained an edit box where an IP address for a game server could be entered. If this box was empty the discovery service would search for a game server. This edit field is still there but now it should contain an IP address of a machine running a game list server, a list of game servers can then be obtained. An appropriate game server can be selected and joined. Figure 23 shows this dialog box.

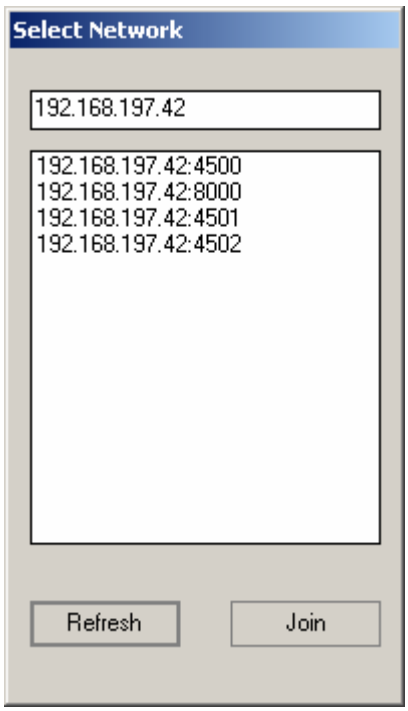

Figure 23, Dialog box displayed when the client are launched.

Moreover the discovery service has been removed. This was done because the game service solution developed in this iteration is very minded against the Internet, and the discovery service really does not have any functionality unless it is running on a LAN.

6.6.3 Server

Our server system is in this iteration now built of two servers. This is as mentioned a game list server and a game hosting server.

6.6.3.1 Game hosting server

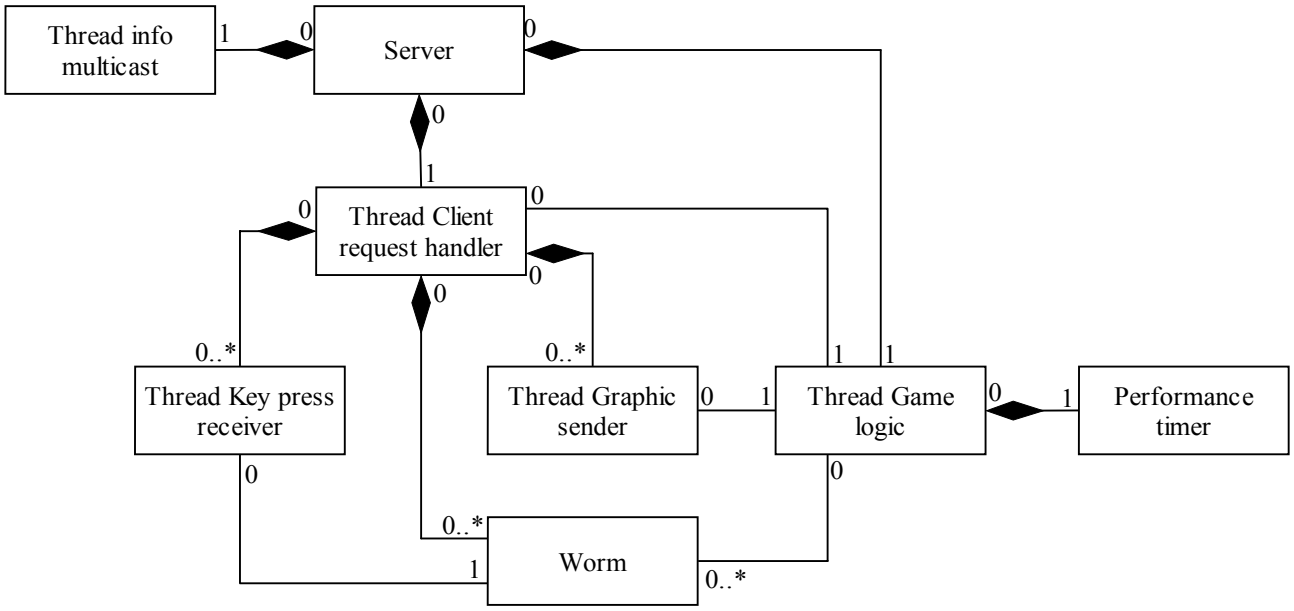

Figure 24, Game hosting server.

The program known as the game server in previous iteration has not changed much. The thread discovery service has been replaced by thread info multicast. When starting a server you now also have to enter the port the server runs on, this is done such that a single machine can host multiple game servers.

Thread info multicast
This is a thread that in intervals multicast its own server IP and server port on the network. This information can then be picked up by a game list server. A game list server can then maintain a list of running game servers on the network.

6.6.3.2 Game list server

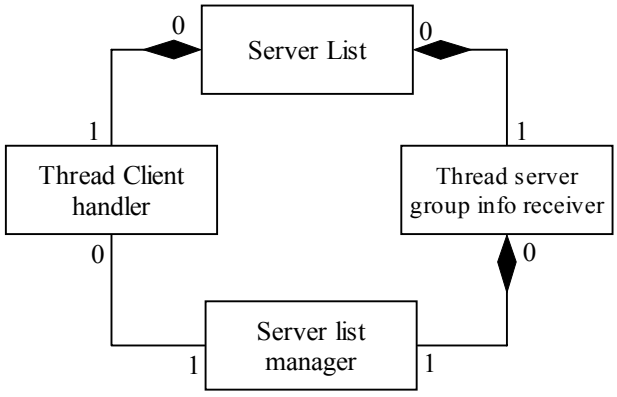

Figure 25, Game list server.

This is a server that receives multicast messages from running game servers on the network. A client then connects to this server and selects a game to join.

Server list
This is the entrance point for the application and only has one method which is the Main method for this application.

Thread server group info receiver
This class runs as a thread. It is responsible for receiving multicasted messages from running game servers. When a message is received it is added to the server list manager.

Thread client handler
This thread accepts requests from clients that wishes to obtain a list of running game servers. This list is then obtained from the Server list manager and sent to the client.

Server list manager
This class is responsible for maintain a list of game servers on the network. Servers can be added to this list, and a timestamp for when it was added is maintained. If a server already exists in the list, its timestamp for when it was added is updated. In intervals all servers timestamp is controlled. If a timestamp reach a certain age, the server is removed from the server list.

6.6.4 Communication flow

The communication flow of DWorm is illustrated below.

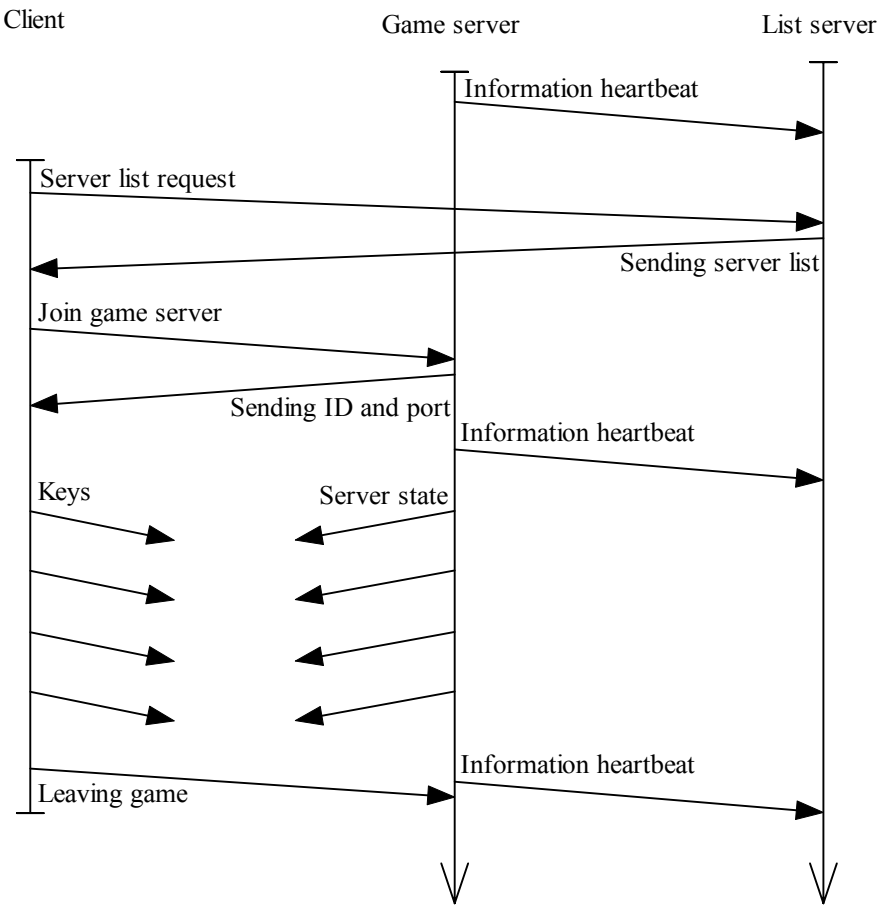

Figure 26, Message flow in the game system.

When the game list server is launched it listens for incoming traffic. Heartbeats from game server are used to maintaining a list of servers. On a server list request from a clients a list is returned to the corresponding client. The client can then send a join request to a game server. Afterwards the game server retunes the player ID and the port to which key presses should be sent. At that point the server sends it server state in intervals to the client and the client sends keys to the server in intervals. When the client leaves the game a message indicating that is sent to the server and the player is removed from the game.

6.6.5 Conclusion

In this iteration we focused on making a highly available and scalable game service. This service was developed and is working. We are able to contact a game list server and obtaining a list of running games. An appropriate game can then be selected and joined. More game servers can be hosted on the same machine, and on a multicast enabled network, multiple game list servers are able to maintain lists of games of all the games running on the network. Game and list servers can be dynamically added and removed. Testing against a network load balancer switch was not possible.

6.7 Design, implementation and test conclusion

We implemented a game working on both LAN and the Internet. Although it was not visually astonishing it was working and performing well. The objective in this project was distributed systems. Design of gameplay was low priority and therefore the gameplay was not fully implemented. This means that the game did not have a scoring system, and on collision worms do not die and re-spawn. It does however still get you visual feedback on collision between worms. The requirements for high performance were never the less still present, and failure to fulfil this requirement would still be very noticeable. Finally a scalable game service was developed. This service featured scalability and availability. Besides the goals for the project like learning .NET and implement a distributed game, some issues concerning games in general was also learned. This involves issues in running an application never just passively waiting for input, and make sure such an application behaved identical on different machines.

The iterative approach did not give us any difficult problems. Functionality was added step by step and problems were eliminated on discovery. The risk of having an unforeseen problem changing a planned design was eliminated. The only noticeable issue with this approach was that the code and implementation in the end became a little unstructured. However a few days of code cleanup could fix this.

C# as implementation language and .NET as target platform turned out to be useful as implementation language. We did however run into some annoying issues concerning memory and pointer operation. The most noticeable of these examples was a problem with coping structures into byte arrays and back. This was however still possible by the use of unmanaged code within an otherwise managed application. By using .NET and an object oriented implementation language we may also have lost some degree of performance compared to C or C++. This was however never an issue for the application, and the use of hardware accelerated graphic was still possible using DirectDraw.

.NET does ship with some easy to use functionality that can shorten development time for large distributed enterprise applications. None of this functionality was however used in the development of DWorm. We simply did not find any use of functionality like thread pooling, transaction handling.

7 Evaluation

A part of this project was to examine and learn the .NET framework and C#. The section presents the project group members experience and ideas about these topics. Also presented are different improvements techniques for distributed games and the pervasive gaming perspective.

7.1 .NET

Upon building a game, one of the choices is to select the appropriate programming language and platform. In the gaming industry high performance, nice graphic and a good gameplay is some of the most important factors. Good gameplay is individual between people and is not directly related to language and environment. However to achieve performance and nice graphics you are in some way depending on specific programming languages and API's. On the PC today, the dominating gaming platform is Windows, and to push an application on this platform to the limit, games of today are often either developed using C or C++. Those languages have a good balance between being a high level language and good performance. High performance graphics are rendered using one of the following methods. DirectDraw is 2D rendering supported by almost every video card. The two other techniques are Direct3D and OpenGL. Both of these techniques require a video card that supports 3D hardware acceleration. Render and controlling graphics gets a little more complicated than with DirectDraw since these methods requires the programmer to set up a 3D world.

One of the secondary purposes of this project was to get insight into the .NET platform. Although we never really have to choose a language, one of the things that could have forced us to choose another programming environment than .NET, was if we did not have the ability to render high performance graphics. The graphic did not necessary needed to be nice, but it had to be fast. Drawing on a canvas with 15 frames per second on today's high end machines was not acceptable for us and in games in general. Fortunately it turned out such that C# and .NET was useful. DirectDraw and Direct3D graphics could be implemented using Visual Basic libraries shipping with DirectX 7. An OpenGL wrapper for C# has also already been developed.

.NET in relation to game performance is a little hard to evaluate on since we do not have the ability to compare to equal products developed and compiled for .NET and for Windows and x86 directly. JIT compilation as .NET uses can provide good results once a program is up and running, but can be very slow at load time (as it has to both load and compile the program), and can not necessarily perform some of the more time expensive optimizations that optimizing compilers perform. Further more .NET does run in and environment that constantly manages and checks the program, (if invalid memory being referenced, running garbage collection, etc.) further more pointer operations and other operation directly involving memory is not possible. All these things can however still be disabled or allowed as long code is directly marked as unsafe. This can give the extra performance and abilities required for games, many of the advantages of .NET will however then be lost.
Again looking back in history to the time when Windows95 was first introduced the preferred gaming platform was DOS. Back then people was also saying that Windows did not give the performance required for games. Never the less no games are more developed to DOS. The advantages of the Windows platform over the DOS platform were so big that the small loss of performance became irrelevant. Further more as CPU speed and memory capacity increases, the percentage used to other things not directly involving the game also gets smaller. In the later stages of this project Microsoft released a beta version of DirectX 9. This version should support

.NET platform directly. How exactly this will influence on games we do not know. But it could be that Microsoft once again is trying to move games to a new platform, and this time .NET.

.NET is Microsoft biggest investment ever and it looks like their way of trying to find a way to standardize the computer platform. For Microsoft to succeed in this, game developers has to start target this platform as well. If they will only the future can tell.

7.1.1 C#

C# is an object oriented language. Using an object oriented language makes it easier to develop applications which can be altered and extended due to the component structure. Earlier applications implemented with an object oriented were heavier to execute compared to a corresponding application implemented with a procedural language. Because of the efficient computers of today this is not a problem anymore. The bottleneck today is the network bandwidth and video cards.
Because C# is a language designed for the .NET framework, explicit memory controls is not an option. C# suffers from the missing functionality of direct memory operations, when implementing games.

7.2 Improvement techniques

To help keep the latency and bandwidth consumption down, increase overall performance, and enhance overall gameplay experience, some techniques can be used.

One of the advantages of having server architecture is that the server can dynamically filter information before sending it to the clients. The server only needs to send a client information, for objects that are within a players view. Similarly, sound can be filtered if it is to far away. The algorithm to compute this can be pretty complex depending on the game; it can however decrease bandwidth consumption pretty much.

A rather simple strategy to cover up for latency is to let players have “sluggish” action. In strategy and role-playing games, reactions are not a factor. Even games that feel like action games may mask latencies by implementing vague or sluggish action. Games can fake real-time interaction by requiring a second to sluggishly swing an axe, or by using weapons or spells where the exact time or direction of the action is unimportant.

7.2.1.1 Dynamic send and receive rate

To increase performance most of the messages between machines is sent unreliable. Reliable data does take precedence over unreliable data. Reliability is achieved using TCP/IP “tree way handshake”. This protocol can however seriously decrease performance, therefore only data that is absolute necessary is sent using this protocol. The rest is sent unreliable using the UDP protocol. This protocol does not guarantee that a package reach its destination. Depending on the game, handling of lost packages might or might not be necessary.

Using unreliable communication for sending and receiving messages between client and server, it can not be guaranteed that messages reach their destination. But using reliable communication can lower performance to much and can therefore not be a solution. A way to make unreliable communication observable, in the meaning of keeping track of how large the packet loss is, is to number the packets continuous. This feature will lower the performance of the system as a consequence of more operations to perform. It can be in a part of the system, which is not directly

in relation to the network communication. This is desirable because the network communication already is a bottleneck.
Adding this feature to DWorm makes it possible to dynamically lower or raise the send and receive rate, making the network utilization effective as possible.

7.3 Pervasive gaming

One of the new trends when it comes to distributed systems are pervasive computing. Today we live in a world where universal presence of mobile computing, computer networks and wireless communication gets more important in everyday life. Pervasive computing can be summarized as computing power freed from the desktop - embedded in wireless handheld devices, telephone and home appliances. Although the future is hard to predict the picture of today is that this kind of mobile computing will grow rapidly over the next years.
Today's companies are also starting to realise the importance of games on these handheld devices. Companies are beginning to develop games to PDA's and most mobile phone is sold with a few games incorporated. Still these games have been rather simple single player games without any possibility for distributed gaming. That does not mean distributed gaming on handheld devices has yet to be invented. The first SMS (Short Message Service) games involving many people have already been created, although they, because of the limitations with SMS, have been rather simple.
This picture is however beginning to change. SUN Microsystems [SUN02] has released Java2 micro edition. This is a special edition of Java targeted against small handheld devices. Over the last year devices taking advantage of this technology has also started to bee available for the average consumer. Looking at the future it seems that it's just a matter of time before the first real distributed games will be playable on both PC and different handheld devices.
The idea with pervasive computing is that services are available all the time. This is unfortunately a very difficult task to achieve for some game types. Many games of today pushes the hardware to the limits, and it does not seems that these small handheld devices will catch up anytime soon, and when they do the PC game technology will have evolved even further. If, or rather when, pervasive distributed gaming gets its breakthrough, it will most likely be of the turn based type where the hardware platform is of less importance. Another aspect is the fact that a lot of pervasive computing devices simply are not suited for gaming. Nobody is really expecting the ability to play a game on a refrigerator.

A future scenario could be distributed game played by passengers in car or trains, where laptop computer are connected through wireless networks. Vehicles who constantly and fast moves around. The challenge here is to predict the route of the car or train. The distributed game then has to prepare the network connections to come, so the connection does not get disrupted. This could for example be done because a train's route is known because it is bound to the railway line. A car normally follows a road. Maybe a cars route are planned by the use of GPS.

8 Conclusion

Prioritizations of the distribution challenges in developing of distributed system were made. The requirements to DWorm and requirements to this type of game in general, were set up. On basis of analysis of different distributed game types, DWorm was defined. The other game types where analysed and put in relation to distributed system theory. Through analysis, design and implementation DWorm was developed. It was however not a full implementation.

The analysis of the .NET framework showed that it can be used as target platform for distributed games. No specific features in the framework are especially suited for distributed games. C# as language for implementation of distributed games was successful.

The project consisted of six iterations, where five iterations were completed. The iteration was left out because it did not relate directly to distributed systems.

Each member of the project group has gained knowledge about the theory and practice of building a distributed game. Specific solutions to problems in relation to distributed games were evaluated mainly on our own judgement and knowledge, since in depth documentation on distributed game solutions could not be found.

Appendix A – References

World Wide Web

[BLI02]
Blizzard
Blizzard Entertainment
http://www.blizzard.com/

[COM97]
Components Online "The Revolution of Computer Software"
Components Online
http://www.components-online.com/

[COU01]
Counter Strike
http://www.counter-strike.net

[DEV01]
.NET Framework Training Modules
DevHood
Jason Clark
http://www.devhood.com/

[EVE02]
EverQuest
Sony Online
http://www.everquest.com/

[MER02]
Mercury: A Scalable Publish-Subscribe System for Internet Games
School of Computer Science, Carnegie Mellon University
Ashwin R. Bharambe, Sanja Rao & Srinivasan Seshan
http://www.acm.org

[MSD02]
Microsoft Developer Network
Microsoft Corporation
http://msdn.microsoft.com/

[QUA01]
A Distributed Multiplayer Game Server System
Electrical Engineering and Computer Science Department, University of Michigan
Eric Cronin, Burton Filstrup and Anthony Kurc
http://warriors.eecs.umich.edu/games/papers/quakefinal.pdf

[QUA02]
Quakeforge

The Quakeforge Project
http://www.quakeforge.net/

[QUA03]
Quake
ID Software
http://www.idsoftware.com

[RAD94]
RAD Data communications – The OSI Reference Model
RAD Data communications
http://www.rad.com/

[REB02]
Rebel Arts
Rebel Arts LLC
http://www.rebelarts.com/

[SUN02]
The Source for Java Technology
SUN Microsystems
http://www.java.sun.com/

[WES02]
Red Alert II
Westwood Studios
http://westwood.ea.com/

Books

[COU01]
Distributed Systems – Concepts and Design (3rd edition)
Addison-Wesley 2001
George Coulouris, Jean Dollimore & Time Kindberg
ISBN: 0201-61918-0

[GOR00]
The COM and COM+ Programming Primer
Prentice Hall PTR 2000
Alan Gordon

[GUN02]
C# - Introduktion for programmører
Ingeniøren bøger 2002
Eric Gunnerson
ISBN: 87-571-2392-6
(First published as: ”A programmers introduction to C#, second edition” by Apress)

[JEN00]

Data kommunikation
Teknisk forlag 2000
Stig Jensen
ISBN: 87-571-2092-7

[MUN96]
Strategisk projektledelse
Marko 1996
Andreas Munk-Madsen
ISBN: 87-7751-115-8

[SCH02]
C#: The Complete Reference
McGraw-Hill/Osborne 2002
Herbert Schildt
ISBN: 0-07-213485-2

Appendix B – Organisation, process model, risk analysis and time schedule

Organisation

The project group consist of three persons. Taken into consideration the size of the group and experience from earlier projects, there will be no formal project manager and no further organisation of the group members. All decisions are therefore taken by consensus.

Process model

The model used in the project is an iterative experimental model. Initially a set of requirements are defined. Based on these requirements an iterative sequence is repeated as long as the time scheduled for the project allows it. Every sequence starts with (re-)planning and (re-)scheduling the actual iteration and the following iterations.

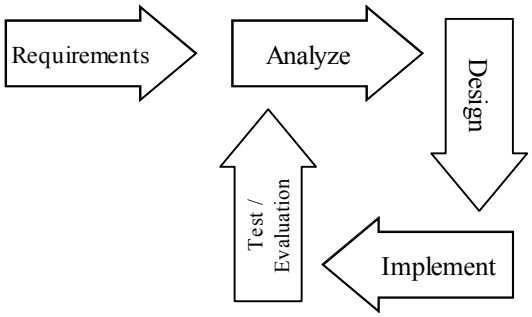

Figure 27, Project model

Risk analysis

Following risks was known at start of the project and planned to be managed as described. Risks are categorized as low and high. Management both goes as avoidance and handling if a risk comes true.

- Long term illness (low)
  - Avoidance can not been planned.
  - Handling: Fewer iterations may be completed.

- Supervisor is not present (low)
  - Avoidance and handling: find out which other teachers can be helpful.

- Missing knowledge about issues in the project, which can make to project impossible to complete (high)
  - Avoidance can not been planned.
  - Handling: project theme and/or areas may be altered.

Appendix C – User manual for DWorm

DWorm consist of three executable files. All of them need to be running to be able to play the game. It is the DWorm version for the Internet, which uses unicast and not multicast.

First of all to be able to run the application the .NET framework needs to be installed on the computer. The .NET framework for Windows and .NET service pack 2 can be found on the CD coming with the project. Launching a .NET executable without .NET will simply give you an error message.

Although the game is a distributed multiplayer game it can be launched and tested on a single machine. To get the program up and running start by launching the *ServerListManager.exe.* Only one instance of this program can be launched on a machine. This will launch a program that maintains a list of running game servers in the network.

Afterwards at least one game server has to be launched. Find the *Server.exe* and launce it. The server will ask you to type what port it should run on. Just follow the instruction and enter a port number. The server list managers, then within a short time, discover that a game server has been started. More game servers can be launched on the same machine as long two servers do not run on the same port.

Finally to play the game the client needs to be launched. Find the *F7S_GUI.exe* and launch it. The following dialog box should show up.

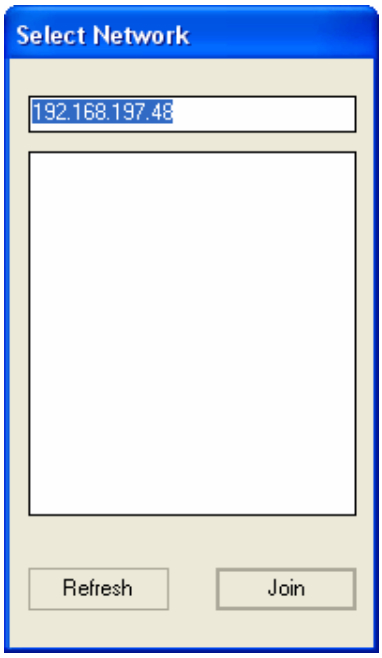

Figure 28, Game server list dialog box.

In the top edit field an IP address of a machine running a game list server should be entered. At launch it should contain the IP address of the local machine. If the game list server is running on the local machine press the *Refresh* button. If the game list server is running somewhere else enter the IP address of the corresponding machine and press *Refresh*. A list of running games should then be obtained from the game list server. Select a game and press the *Join*. A message box will

ask you if the game should run in full screen or windowed mod. Select the desired mode. If windowed mode is selected another message box will be displayed. It will ask you if the graphical back buffer should be placed in the video card or in the system memory. Placing it in the video card, gives a much higher performance, than placing it in the system memory. However a few video cards do not work probably with the back buffer in video memory. Select the desired mode.

The game should now be running. Exit the game by pressing escape. Afterward the servers can be shut down by pressing enter.