An Application Framework for the Distributed Simulation of Virtual Worlds by Spatial Decomposition

Peter Kulczycki

Blindwiesen 9
A-4040 Linz

August 31, 1997

Eidesstattliche Erklärung

Ich erkläre, dass ich die Diplomarbeit selbstständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht verwendet und mich auch sonst keiner unerlaubten Hilfe bedient habe.

Linz, im August 1997 Peter Kulczycki

Abstract

This thesis describes the design and the implementation of a distributed simulation of real worlds on a network of computers. The distributed simulation enables multiple users in distant locations to interact and to share experiences within the same virtual world. A key problem solved is the consistency of the different virtual world views under the constraint of limited network communication. Examples of applicability range from traffic and factory simulation systems to multi-user games and molecular dynamic simulations.

The virtual world we have modeled is a two-dimensional imitation of the real world. It is subdivided into an arbitrary number of regions, so-called cells, each under supervision of an independently and distantly running process. The virtual world is "populated" by virtual objects. Some of them are under control of human operators, others behave like autonomous agents, fulfilling some kind of task. In order to reduce communication between objects, several strategies based on the spatial property of the virtual world have been designed and implemented.

We have designed, implemented, and tested an object-oriented application framework, depicting the distributed simulation sketched above, plus a full functioning example based on this framework. Computers running different operating systems, e.g. Unix and Windows NT, may interoperate within one single simulation session.

Keywords: *Distributed Simulation, Distributed System, Multi-User Game, Spatial Decomposition, Virtual World.*

Kurzfassung

Diese Diplomarbeit beschreibt das Design und die Implementierung eines verteilten Systems (*distributed system*) zur Simulation (*distributed simulation*) von virtuellen Welten (*virtual world*). Das System läuft – physikalisch verteilt – auf mehreren Computern und ermöglicht mehreren Anwendern die gleichzeitige Interaktion in ein und derselben virtuellen Welt. Ein im Zusammenhang mit diesem System auftretendes Problem, nämlich die Gewährleistung der Konsistenz einer virtuellen Welt, wurde im Rahmen dieser Diplomarbeit gelöst. Dabei wurde besonders auf eine möglichst geringe Belastung des Netzwerkes geachtet. Der Bogen möglicher Anwendungen spannt sich von Verkehrs- und Fertigungssimulationen bis zu Spielen mit mehreren Teilnehmern (*multi-user game*) und zur Simulation des dynamischen Verhaltens von Molekülen.

Die von uns modellierte virtuelle Welt ist ein zweidimensionales Abbild der realen Welt und wird in eine beliebige Anzahl von Regionen unterteilt. Jede dieser Regionen wird von einem unabhängigen und entfernt ablaufenden Programm gesteuert. Eine virtuelle Welt wird von virtuellen Objekten "bevölkert". Einige dieser virtuellen Objekte können von Personen "manövriert" werden, während sich andere wie autonome Agenten verhalten, die in der Lage sind, beliebige Aufgabe zu erfüllen. Um den Kommunikationsaufwand zu senken, wurden einige Strategien, die die räumlichen Eigenschaften einer virtuellen Welt ausnutzen, ersonnen und implementiert (*spatial decomposition*).

Basierend auf dem verteilten Simulationssystem wurde ein objektorientiertes Application Framework und ein voll funktionsfähiges Beispielprogramm entwickelt, implementiert und getestet. Es ist möglich, verschiedene Betriebssysteme, wie z.B. Unix und Windows NT, innerhalb einer einzigen Simulation zusammenzuschalten.

Schlüsselwörter: *Distributed Simulation, Distributed System, Multi-User Game, Spatial Decomposition, Virtual World.*

To Edith, Thomas, and Franz

Contents

List of Figures

List of Tables

List of Examples

Acknowledgments

First of all, I want to thank *Prof. Bruno Buchberger* and *Prof. Witold Jacak* for their untiring efforts in setting up the Department of Software Engineering of the Upper Austrian Polytechnic University. Tribute has to be paid especially for their outstanding engagement while "bootstrapping" this institution. The Department of Software Engineering turned out to be an excellent opportunity for students to get perfectly trained for their future professional life. And it would *not* be that way without Prof. Buchberger and Prof. Jacak!

Then, I have to to thank *Dr. Wolfgang Schreiner*, my supervisor at RISC-Linz, for guiding me in superior style while I was working on this thesis. His great experience in addition to his endless patience in explaining complex technical problems characterizes him to be an excellent teacher. Whatever my problems were, he *immediately* lent me an ear conveying that I am really important to him.

Special thanks go to *Dr. Herwig Mayr*, who gave me the possibility to do my diploma thesis at RISC-Linz.

I am very grateful to *Mr. Gregory Curtis* and *Mr. Franz Lindenhofer* for proofreading this thesis. They increased the linguistic, the grammatical, and the stylistic quality of my English writings dramatically!

Finally, I would like to thank my wife *Edith* and my little son *Thomas*, my mother *Heidi*, and my grandfather *Friedrich* for generously supporting me during my entire studies. Many things would have been impossible without their help!

pwk

Chapter 1

Introduction

The idea behind distributed simulation systems is very simple: A simulated virtual world runs not only on one computer but on several, involving an arbitrary number of processes.

A number of obstacles have to be overcome in achieving this goal. Due to the fact that we want these systems to run over great distances, e.g. the Internet, we have to cope with a certain amount of *latency* in the delivery of, for instance, `Update` messages. Another difficulty ist the heterogeneity of the computer systems and therefore the heterogeneity of the software people use. This demands the system to be designed for *portability.*

The simplest approach, and the one that some multi-player computer games systems use, is merely to have each process in the simulation system broadcast the location and the state of each entity it maintains. These broadcasts are received by every process in the simulation system and are used to update their local copy of the entire virtual world.

This works acceptably on small, dedicated networks. However, there are a number of problems with it. The most important problem with broadcasting is the vast amount of bandwidth such a system consumes when there is a larger number of people participating the distributed simulation. So this method does not work well in larger systems. In other words: This simple approach does not scale up well. One has to invent and design a different system not susceptible to these kind of problems.

The first two chapters of this thesis give a comprehensive survey on the state-of-the-art in this area. Our contribution to this topic is a distributed simulation system that overcomes the difficulties mentioned above by having been designed very carefully.

Two major architectural issues enabled us to implement the intended distributed simulation system:

1. The entire virtual world to be simulated is spatially subdivided into separate regions and distributed among an arbitrary number of computers and processes. This approach avoids replicating the *whole* virtual world.
2. The overall design – from an operating system's and from a communication technique's perspective – corresponds to designs typical of peer-to-peer shaped systems. A non-centralized architecture prevents interconnected computers from getting somehow hierarchically arranged and, consequently, from turning into bottlenecks.

In order to allow users to interact with and to make experiences within *one* virtual world *simultaneously*, our distributed simulation system makes it possible to "populate" it by entities that correspond to a variety of things existing in real worlds. These *virtual objects* are vested with behavior, have visual shapes and physical extents, are equipped with characteristic properties, etc. Special virtual objects, so-called observers, allow users to get into and stay within a virtual scenario.

This thesis has been subdivided into five[1] chapters and one appendix. Two of the chapters are about the state-of-the-art in distributed computing. Four of the chapters (including the appendix) are about the distributed simulation system and the object-oriented distributed application framework we have created.

Chapter 2 gives a survey of the current state-of-the-art in distributed computing. It starts with definitions of essential terms, motivates the reasons for doing distributed computing, and finally illustrates several technical details.

Chapter 3 gives insight into the matters of distributed virtual worlds. It focuses on virtual worlds that can be subdivided spatially and that are populated by a variety of objects having individual behavior.

Chapter 4 describes how a distributed system for the simulation of a certain category of real worlds can be modeled and designed. The architecture of a virtual world that can be simulated by our system is explained from an application user's point of view.

Chapter 5 describes our distributed simulation system from a technical perspective. The overall technical design and the entire functionality our system offers is explained in detail.

[1]Not including Chapter 1 (Introduction) and Chapter 7 (Conclusions).

Chapter 6 shows the features and the use of our object-oriented distributed application framework built according to our distributed simulation system by means of providing some sort of tutorial.

Appendix A summarizes all class diagrams of all functional class groups of our distributed application framework including the application example.

Chapter 2

Distributed Systems

First, this chapter gives a definition of the term "system" together with a definition of the technical term "distributed system". Then, the reasons for the importance of distributed systems regarding technical and economical aspects is motivated. Third, the goals that distributed systems follow up and the design issues relevant for distributed systems are specified. Finally, some selected technical problems are itemized and discussed in detail.

2.1 Introduction

2.1.1 The Term "System"

The term "system" is of Greek and Latin origin and stands for "a group of related parts which work together forming a whole". As a consequence of this interpretation, it follows that interactions between the individual parts of a system are much more intensive than interactions between the system as a whole and the environment it is embedded (cf. [Mayr, 1997a]).

Internal interactions, i.e., interactions between parts of the system and *external* interactions, i.e., interactions between the system and its surroundings, are always performed in a structured way across well defined interfaces. It will turn out in this and in the following chapters of this thesis that interaction, or more specific the exchange of information between the various parts of a system, plays a central role in building up systems.

2.1.2 The Technical Term "Distributed System"

Various definitions of the technical term "distributed system" have been given in the literature. None of them is satisfactory and none of them in agreement with any of the others. We will comment on three different definitions of three different authors and give later on a personal one.

In [Tanenbaum, 1995], Andrew S. Tanenbaum defines a distributed system in a very concise way when he says that:

> A distributed system is a collection of independent computers that appear to the users of the system as a single computer.

There are two aspects in this definition. The *first* one deals with the hardware: The distributed system is made up of a collection of independent computers. The *second* one deals with the software: The users think of the distributed system to be one single computer.

A more precise definition of the technical term "distributed system" give George Coulouris et al. in [Coulouris *et al.*, 1994]:

> A distributed system consists of a collection of autonomous computers linked by a computer network and equipped with distributed system software. Distributed system software enables computers to coordinate their activities and to share the resources of the system – hardware, software, and data. Users of a well-designed distributed system should perceive a single, integrated computing facility even though it may be implemented by many computers in different locations.

This definition is more accurate than the former one, since it says something about how computers have to be connected. The definition of Coulouris et al. also introduces the term *resource*. Sharing resources is a main reason for building distributed systems.

The third definition of distributed systems we will mention is given by Michael D. Schroeder in [Schroeder, 1993]:

> A distributed system is several computers doing something together. Thus, a distributed system has three primary characteristics: multiple computers, interconnections, and shared state.

Now, we will give our personal definition of what we think that characterizes a distributed system:

> *A distributed system is an accumulation of heterogeneous processes working together by means of exchanging information and that appear to potential users as an integral whole.*

The attentive reader will have noticed the omission of the technical term "computer" and the emphasis on the technical term "process". In fact, this is the major difference that distinguishes our definition of a distributed system from the other three definitions given here. We think of a process to be the basic item to be distributed. Therefore, it is possible to run a complete distributed system on a single computer, although, for some reasons, it would not be wise to do so.

Processes are distributed among a more or less arbitrary number of available computers. An actual assignment of processes to computers depends on a variety of factors. Decisive factors can be the actual spatial distribution of resources, the number of available computers together with their load or even special cases like test environments. Nevertheless, in each case, the distributed system seen from the designers perspective stays the same.

2.1.3 Why Distributed Systems?

Just because it is possible to build distributed systems does not necessarily mean that it is a good idea to do so. But what are the advantages of distributed systems over centralized systems?

The real driving force behind the trend towards decentralization, i.e., distributed systems, is economics. The computing power of a CPU is proportional to the square of its prize. This is known as the Grosch's law (cf. [Tanenbaum, 1995]), and it is only true with mainframe technology. This law no longer holds with microprocessor technology. With this technology, one gets an enormous amount of computing power for a quite low price. Except in special cases, an economy of scale in computing no longer exists. This applies to both, computing and storage (cf. [Shuey *et al.*, 1997]). Thus, it is much more economical to set up a distributed system consisting of a large number of CPUs than to construct a centralized system consisting of one single mainframe computer.

But there are also various other reasons for building distributed systems. We will just mention the most important one: Some applications are inherently distributed. A supermarket chain may have many stores, each of which gets goods delivered locally, makes local sales, and makes local decisions. It therefore makes sense to keep track of the inventory at each store on a local computer rather than centrally at corporate headquarters. After all, most queries and updates will be done locally. Nevertheless, from time to time, the top management may want to find out how many goods it currently owns. One way to accomplish this goal is to make the complete system look like a single computer to the user, but implemented decentrally (cf. [Tanenbaum, 1995]).

There exist a lot of applications that are inherently distributed. Just to mention two more:

- computer supported cooperative work systems and
- computer supported cooperative game systems.

With both systems, an arbitrary number of people in different locations operate together on a single application in real time.

As it is true for the economical point of view, it is true for the technical point of view too: There are a lot of advantages of distributed systems over centralized systems. Table 2.1 mentions the most important advantages (cf. [Chow and Johnson, 1997], [Coulouris *et al.*, 1994], [Schroeder, 1993], [Shuey *et al.*, 1997], and [Tanenbaum, 1995]).

2.1.4 Goals and Design Issues

Although the designers of distributed systems must consider issues that are largely unrelated to the system's distribution – such as software engineering techniques, human interface design, algorithm design, etc. – we restrict our discussion to goals and design issues that arise specifically from the distributed nature of systems. However, the general goal for the design of a distributed system is not different from any other system. Both the users and the system designers are concerned with issues of the efficiency, flexibility, consistency, etc. of the system. But the underlying architecture of a distributed system raises the expectations and requirements of the system, and this causes the level of implementation complexity to consequently rise. These differences rest on the effect of distributed resources, communication overhead, and the higher likelihood of component failures in the system (cf. [Chow and Johnson, 1997], [Coulouris *et al.*, 1994], [Foster, 1994], [Sinha, 1997], and [Tanenbaum, 1995]). The major goals and design issues are summarized in the following list:

Item	Description
Economics	Microprocessors offer a better price to performance ratio than mainframes.
Inherent Distribution	Some applications involve spatially separated computers.
Reliability	If one computer crashes, the whole distributed system may still survive. This is assured, e.g., by means of hardware redundancy.
Resource Sharing	Resource sharing is a fundamental characteristics of distributed systems.
Scalability	Computing power can be added in small increments as well as data, functionality, and load can be distributed between servers.
Speed	A distributed system may have more total computing power at lower costs than a centralized system.

Table 2.1: Advantages of distributed systems over centralized systems.

Efficiency is harder to achieve in distributed systems than in centralized systems due to the effect of communication delays. Communication through message passing in spatially distributed environments incurs delays ranging from microseconds to milliseconds and seconds or even more. There are many reasons for this, and there is nothing we can do about it. With respect to system load distribution, problems such as bottlenecks or congestions of network traffic must be addressed. Computation speed and system throughput can be enhanced thorough distributed processing and load sharing, if the communication subsystem is carefully designed.

Flexibility From the user's point of view, flexibility includes the friendliness of the system and her/his freedom in using the system. Friendliness has a very broad meaning, but it primarily implies the ease of use of the system interface and the ability to relate the computation process in the user's problem space to the system. Freedom lets the user decide, how, where, and when to use the distributed system without unreasonable restrictions.

Consistency is harder to achieve in distributed systems than in centralized systems due to the lack of global information, potential replication and partitioning of data, the possibility of component failures, and the complexity of interaction among parts of the distributed system. All this contributes to the problem of inconsistency in the system. A system is consistent from the user's point of view, if there is uniformity in using the system and the system behavior is predictable. In addition, the system must be capable of maintaining its integrity with proper concurrency control mechanisms and failure handling and recovery procedures. Consistency control in data and possibly existing files or databases is a crucial issue in distributed systems.

Robustness is obviously a more serious problem in distributed systems than it is in centralized systems. Failures in communication links and processing nodes are more frequent than in centralized single computer systems. The question, what to do when some sort of failure occurs, arises immediately. Robustness with respect to fault tolerance means that the system capable of reinitializing itself to a state, where the integrity of the system is preserved with no degradation of its performance. Even without explicit failures, a system should be equipped with the ability to handle situations, such as changes in the topology, long message delays or the inability to locate a certain part of the distributed system. Robustness also addresses the aspect of security for users and the system itself.

Transparency is perhaps the most frequently used property in distinguishing a distributed system from a centralized system. It is a goal motivated by the desire to hide all irrelevant system-dependent details from the system's user. The reason for this can be found in the definition of the term "system" as it was given above. It is desired to create an illusion of the model the users expect to see. This principle has been practiced in the design of computer systems for a long time. But transparency is a more important issue in distributed systems due to the higher implementation complexities. There are several different aspects of transparency:

- access transparency,
- location transparency,
- migration transparency,
- concurrency transparency,
- replication transparency,
- parallelism transparency,
- failure transparency,
- performance transparency,
- size transparency, and
- revision transparency.

Table 2.2 shows some major kinds of transparencies.

Reliability One of the original goals of building distributed systems was to make them more reliable than single-processor systems. The idea is that if a computer goes down, some other computer can take over the job. In a widely quoted remark, Leslie Lamport once defined a distributed system to be "one on which I cannot get any work done because some machine I have never heard of has crashed" (cf. [Tanenbaum, 1995] and [Schroeder, 1993]). In theory, each computer participating a distributed system has to be capable of doing the job of any other computer in the system. The practice is that current distributed systems count on a number of specific servers being up. It is important with distributed systems to distinguish various aspects of reliability:

- availability,
- security, and
- fault tolerance.

Scalability Most current distributed systems are designed to work with a few hundred computers or processes. It is possible that future systems will be orders of magnitudes larger, and designs that work well for hundred machines will likely fail for a million machines. The central question is: Will the methods we are currently developing and using scale up to extremely large systems? Although very little is known about huge distributed systems, one guiding principle is clear: Avoid centralized components, tables, and algorithms. Having a single mail server to serve 50 million users would not be a good idea, even if it had enough computing power and storage capacity (cf. [Tanenbaum, 1995]). The following three concepts have to be avoided when designing large distributed systems:

- centralized components,
- centralized tables, and
- centralized algorithms.

Especially decentralized algorithms have to have the following characteristics, which distinguish them from centralized algorithms:

- No computer has complete information about the system state.
- Computers make decisions based only on local information.
- Failure of one computer does not sabotage the algorithm.
- There is no implicit assumption that a global clock exists.

Resource Sharing is the fundamental characteristics of distributed systems. It underpins all other characteristics mentioned here and it strongly affects the

Kind	**Meaning**
Location Transparency	The users cannot tell where resources are located.
Migration Transparency	Resources and data can move at will without changing their names.
Replication Transparency	The users cannot tell how many copies exist.
Concurrency Transparency	Multiple users can share resources automatically.
Parallelism Transparency	Activities can happen in parallel without users knowing.

Table 2.2: Kinds of transparency in distributed systems.

software architectures that are available in distributed systems. Resources may be items of data, software- or hardware components. Resources are distinguished from data items managed within individual processes by the need for several processes to share them. This leads to the need for resources to be managed externally. Current techniques manage resources by a service supplied by several cooperating processes (client/server model). Object-based systems may provide a more flexible model, but the best form of design for such a model remains the subject of further research.

2.2 Time and Coordination

The notion of time in the context of distributed systems introduces several problems, since, for example, simultaneity is a very tricky problem to be discovered. Two "forms" of time, i.e. physical time and logical time, have to be considered here.

2.2.1 Physical Time

Time is an important and interesting issue in distributed systems, for several reasons.

First, time is a quantity we often want to measure accurately. In order to know at what time of day a particular message occurred at a particular computer – for example for accountancy and bookkeeping purposes – it is necessary to synchronize the computer's clock with an authoritative and external source of time. This is known as *external* synchronization. Also, if computer clocks are synchronized with one another to a known degree of accuracy, then we can, within the bounds of this accuracy, measure the interval between two messages occurring at different computers, by appealing to their internal clocks. This is known as *internal* synchronization. Two or more computers that are internally synchronized are not necessarily externally synchronized, since they may drift collectively from external time.

Second, algorithms that depend upon clock synchronization have been developed for several problems concerning distributed systems. These include

- maintaining the consistency of distributed data,
- checking the authenticity of a request sent to a process, and
- eliminating the processing of duplicate updates.

2.2.2 Logical Time

Nearly all computers have a circuit for keeping track of time. This circuit is commonly called "clock" although the word "timer" would fit better. With a single computer and a single clock, it does not matter if this clock is off by a small amount. Since all processes on this computer use the same clock, they will still be internally consistent. As soon as multiple computers come into play, each with its own clock, the situation changes dramatically. Although the speed of the individual clocks stays fairly constant, it is impossible to guarantee that the clocks in different computers all run exactly the same speed. The resulting difference in time values is called *clock skew* (cf. [Tanenbaum, 1995]).

This brings up the question, whether it is possible to synchronize all the clocks to produce a single, unambiguous time standard. In his classic paper [Lamport, 1978], Leslie Lamport showed that clock synchronization is possible and presented an algorithm for achieving it. He extended his work in [Lamport, 1997]. Lamport pointed out that clock synchronization need not be absolute. If two processes do not interact, it is not necessary that their clocks stay synchronized. Because the lack of synchronization, it would not be observable for the processes that their clocks are out of order. Furthermore, Leslie Lamport emphasized that what usually matters is not that all processes agree on what time it is, but rather, that they agree on the order in which messages occur.

To synchronize logical clocks, Leslie Lamport defined a relation called *happens-before*. This relation is sometimes known as the *causal-ordering* relation or *potential-*

causal-ordering relation (cf. [Raynal and Singhal, 1995]). Intuitively, this relation is analogous to the *earlier-than* relation provided by physical time. More formally, we write

$$x \xrightarrow{p} y$$

if two messages x and y occurred at a single process p, and x occurs before y. Using this restricted order we can define the happens-before relation, denoted by $\rightarrow$ by the following axioms (where p is a process and m_1, m_2, and m_3 are messages):

$$\begin{array}{ll} \mathrm{hb}_1: & \exists p \, (m_1 \xrightarrow{p} m_2) \Rightarrow m_1 \rightarrow m_2. \\ \mathrm{hb}_2: & \forall m_1 \, (\mathrm{send}(m_1) \rightarrow \mathrm{receive}(m_1)). \\ \mathrm{hb}_3: & m_1 \rightarrow m_2 \wedge m_2 \rightarrow m_3 \Rightarrow m_1 \rightarrow m_3. \end{array}$$

Thus, if $x \rightarrow y$, we can find a series of messages $m_1, m_2, \ldots, m_n$ occurring at one or more processes such that $x = m_1, y = m_n$ and for $i = 1, 2, \ldots, n-1$, either hb_1 or hb_2 applies between m_i and m_{i+1}. That is, either they occur in succession at the same process, or there is a message m such that

$$\begin{array}{lcl} m_i & = & \mathrm{send} \quad (m), \\ m_{i+1} & = & \mathrm{receive}(m). \end{array}$$

The happens-before relation $\rightarrow$ is illustrated for the case of three processes p_1, p_2, and p_3 in Figure 2.1. It can be seen, that $a \rightarrow b$ holds, since the messages occur in this order at process p_1, and similarly $c \rightarrow d$ and $b \rightarrow c$ hold, since these messages are the sending and the reception of message m_1, and similarly $d \rightarrow f$. Combining these relations, we may also say that, for example, $a \rightarrow f$ holds. This is because of the transitivity law hb_3.

It can also be observed from Figure 2.1 that not all messages are related by the happens-before relation $\rightarrow$. For example, $a \not\rightarrow e$ and $e \not\rightarrow a$, since they occur at different processes, and there is no chain of messages intervening between them. We say that messages such as a and e that are not ordered by $\rightarrow$ are *concurrent*, and write this as

$$a \parallel e.$$

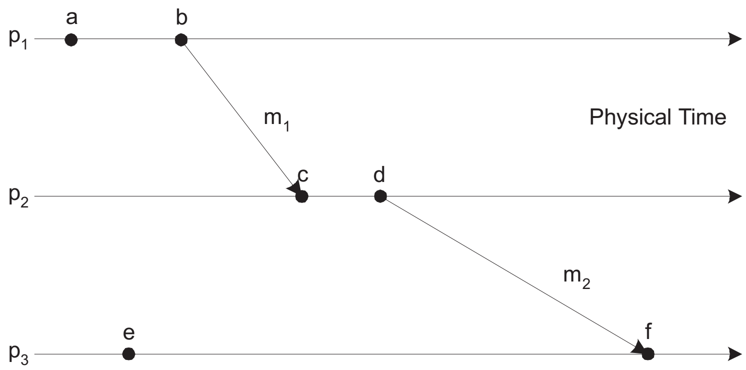

Figure 2.1: Messages occurring at three processes.

2.2.3 Logical Clocks

Lamport invented a simple mechanism by which the happens-before relation can be captured numerically, called a *logical clock*. A logical clock is a strict monotonically increasing software counter, whose value need not have any particular relationship to any existing physical clock. Each process p keeps its own logical clock C_p which is used by the process to timestamp its messages. We denote the timestamp of message m_1 at process p by

$$C_p(m_1)$$

and by $C(m_2)$ we denote the timestamp of message m_2 at whatever process it occurred. To capture the happens-before relation $\rightarrow$, processes update their logical clocks and transmit the values of their logical clocks in messages as the following three rules say:

1. C_p is incremented before each message is issued at process p: $C_p = C_p + 1$.
2. When a process p sends a message m, it piggybacks on m the value $t = C_p$.
3. On receiving (m, t), a process q computes $C_q = \max(C_q, t)$ and then applies rule 2 before timestamping the message receive(m).

Although we increment clocks by one, we could have chosen any positive value. It can easily be shown, by induction on the length of any sequence of messages relating two messages m_1 and m_2, that

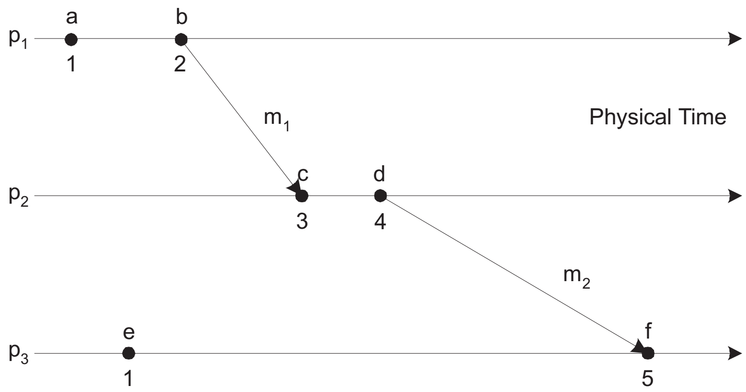

Figure 2.2: Logical timestamps for the messages shown in Figure 2.1.

$$m_1 \rightarrow m_2 \Rightarrow C(m_1) < C(m_2).$$

Note that the converse is not true. If $C(m_1) < C(m_2)$, then we cannot infer that $m_1 \rightarrow m_2$. In Figure 2.2 we illustrate the use of logical clocks for the example given in Figure 2.1. Each of the processes p_1, p_2, and p_3 has its logical clock initialized to 0. The clock values given are those immediately after the message to which they are adjacent. Note that, for example $C(b) > C(e)$ but $b \parallel e$.

2.2.4 Distributed Coordination

Distributed processes often need to coordinate their activities. For example, if a collection of processes share a resource or collection of resources managed by a process, then often mutual exclusion is required to prevent interference and ensure consistency when accessing resources (cf. [Coulouris *et al.*, 1994]).

A separate, generic mechanism for *distributed mutual exclusion* is thus required in certain cases – one that is independent of the particular resource management scheme in question. Distributed mutual exclusion involves a single process being given a privilege – the right to access shared resources – temporarily, before another process is granted it. In some other cases, however, the requirement is for a set of processes to choose one of their number to play a privileged, coordinating role over the long term. A method for choosing a process to play a particular role is called an *election algorithm*.

Distributed Mutual Exclusion

Our basic requirements for mutual exclusion concerning some resource or collection of resources are as follows:

Safety At most one process may execute in the critical section at a time.

Liveness A process requesting entry to the critical section is eventually granted it, as long as any process executing in the critical section eventually leaves it.

Ordering Entry to the critical section should be granted in happens-before order.

Liveness implies that the implementation is deadlock-free, and that starvation does not occur. A process may continue with other processing while waiting to be granted entry to a critical section. During this time it might send a message to another process, which consequently also tries to enter the critical section. *Ordering* specifies that the first process should be granted access before the second.

The simplest way to achieve mutual exclusion in distributed systems is to employ a process – a server – that grants permission to the critical section. For the sake of simplicity, we shall assume that there is only one critical section managed by the process. Figure 2.3 shows the use of this server. To enter a critical section, a process sends a request message to the server and awaits a reply from it. Conceptually, the reply constitutes a token signifying permission to enter the critical section. If no other process has the token at the time of the request, then the server replies immediately, granting the token. If the token is currently held by another process, then the server does not reply but queues the request. On exiting the critical section, a message is sent from the process to the server, giving back the token.

If the server's queue of waiting processes is not empty, then the server chooses the oldest entry in the queue, removes it and replies to the corresponding process. In Figure 2.3, we show a situation in which p_2's request has been appended to the queue, which already holds p_4's request. p_3 exits the critical section, and the server removes p_4's entry and grants permission to enter to p_4 by replying to it.

Unfortunately, there come several drawbacks with this approach. The server might become a performance bottleneck, since all entry requests and exit notifications are sent to it. Additionally, the server is a single point of failure. To overcome these disadvantages, several other algorithms have been invented (cf. [Agrawala and El Abbadi, 1991], [Chow and Johnson, 1997], [Coulouris *et al.*, 1994], [Raynal and Singhal, 1995], [Raynal, 1988], [Ricart and Agrawala, 1981], and [Tanenbaum, 1995]).

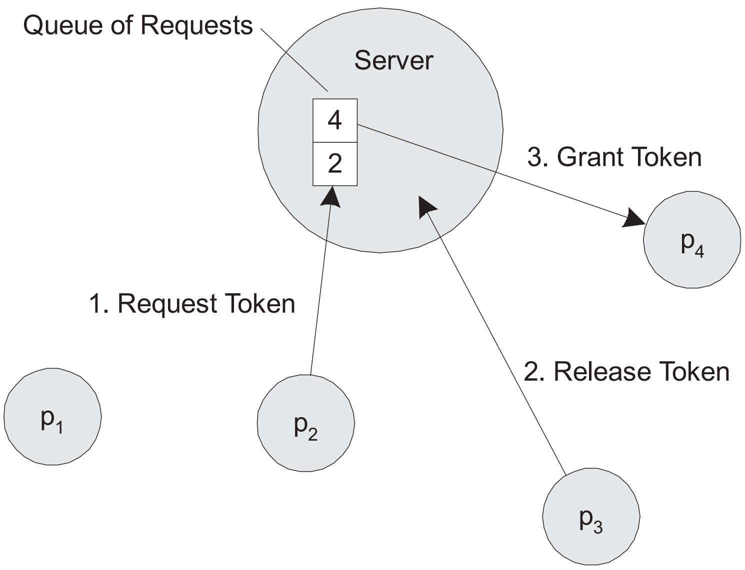

Figure 2.3: Server managing a mutual-exclusion token for a set of processes.

Elections

An election is a procedure carried out to choose a process from a group, for example to take over the role of a process that has failed. The main requirement is for the choice of elected process to be unique, even if several processes call elections concurrently.

As an example, we consider the *bully algorithm* described in [Garcia-Molina, 1982] and [Silberschatz and Galvin, 1994]. For this algorithm to work fine, we will assume that each process has an unique number within the entire distributed system.

When a process notices that the coordinator is no longer responding to requests, it initiates an election. A process p holds an election as follows:

1. Process p sends an `Election` message to all processes with a higher number.
2. If no one responds, p wins the election and becomes coordinator.
3. If one of the processes with a higher number answers, it takes over the election. p's job is done.

At any moment, a process can get an `Election` message from one of its lower-numbered colleagues. When such a message arrives, the receiver sends an `Ok` message back to the sender to indicate that he is alive and will take over. The receiver then holds an election, unless it is already holding one. Eventually, all processes give up but one, and that one is the new coordinator. It announces its victory by sending all processes a message telling them that from this moment on it will be the new coordinator. If a process that was previously down comes back up, it holds an election. If it happens to be the process with the highest number, it will win the election and take over the coordinator's job. Thus, the "biggest guy in town" always wins, hence the name "bully algorithm".

In Figure 2.4 we see an example of how the bully algorithm works. The distributed system consists of eight processes, numbered from 1 to 8. Previously process p_8 was the coordinator, but for some reason it has crashed. Process p_5 is the first one to notice this, so it sends `Election` messages to all processes with a higher number than it, namely to the process p_6, process p_7, and process p_8, as shown in Figure 2.4(a). Process p_6 and process p_7 both respond with an `Ok` message, as shown in Figure 2.4(b). As soon it gets the first of these responses, process p_5 knows that its work is done. It knows that one of the higher ordered processes will take over and become coordinator. In Figure 2.4(c), both, process p_6 and process p_7 hold elections, each one is sending messages to those processes higher numbered than itself. In Figure 2.4(d) process p_7 tells process p_6 that it will take over. At this point, process p_7 knows that process p_8 is dead and that it is the new coordinator. When it is ready to take over, process p_7 announces this by sending a `Coordinator` message to all running processes in the distributes system.

When process p_5 gets this message, it can now continue with the operation it was trying to do when it discovered that process p_8 was dead, but using process p_7 as the coordinator this time. If process p_8 ever restarts, it will just send a `Coordinator` message to all the others and bully them into submission.

2.3 Replication

Replication is the maintenance of on-line copies of data and other resources. It is a key to the effectiveness of distributed systems, in that it can provide enhanced performance, high availability and fault tolerance.

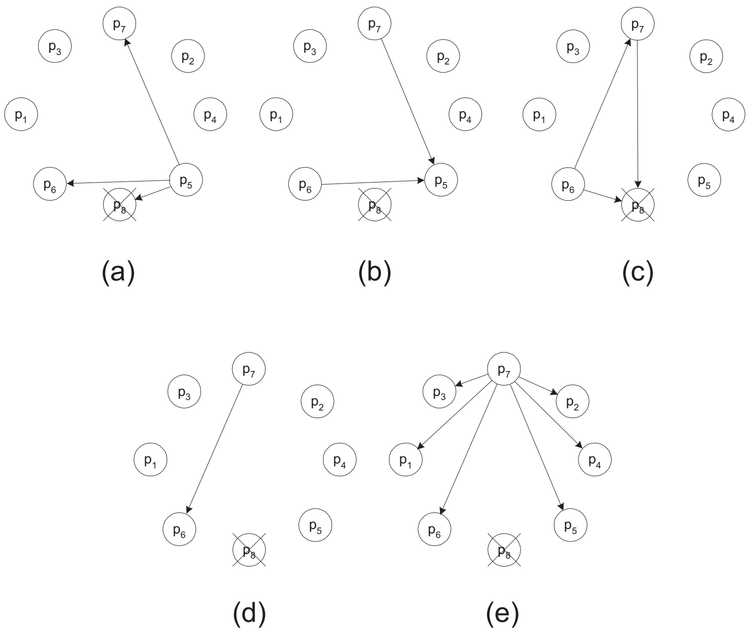

Figure 2.4: The bully election algorithm.

2.3.1 Motivation

Replication is used widely. For example, the Domain Name Service (DNS) maintains copies of name-to-address mappings for computers and other resources, and is relied on for day-to-day access to services across the Internet. The caching of files and other data from servers in client computers is another form of replication, since the data held in caches and at servers are replicas of one another. The motivations for replication are as follows:

Performance Enhancement The caching of data at clients is by now familiar as a means of performance enhancement, but replication can be applied equally at the level of servers to enhance service response times. Data that is shared between a large client community should not be held at a single server, since this computer will act as a bottleneck that slow down responses and has a limited throughput capacity in terms of requests processed per second. It is preferable to distribute copies of the data to several servers, and to arrange that each server provides the data to a smaller community of users which are close to it in network terms.

Availability Enhancement If service data are replicate at two or more failure-independent servers running equivalent software and reachable by independent communication links, then client software can in principle access an alternative server should the default server fail or become unreachable. That is, the percentage of time during which the *service* is available can be enhanced by replicating servers. If n servers each having an independent probability p of failing or becoming unreachable, then the availability a of data item stored at each of these servers is

$$a = 1 - p^n.$$

An important difference between caching systems and server replication is that caches do not necessarily hold collections of data items such as files in their entirety. So caching does not necessarily enhance availability.

Fault Tolerance If each of a collection of servers process every request from a client in parallel, then it is possible to provide guarantees of correct request processing, even though one or more of the servers should fail. This is a stronger statement than that of high availability, because it can include real-time guarantees, and also guarantees against arbitrary, so-called *byzantine*[1] (cf. [Shuey *et al.*, 1997]), failures rather than simple, so-called *fail-stop*, failures in which a component stops after changing to a state in which its failure can be detected.

A chief requirement when data or other resources are replicated is the *replication transparency*. That is, clients should not normally be aware that multiple *physical* copies of data exist. As far as clients are concerned, data are organized as individual *logical* data items (or objects), and they name only one item in each case when they request an operation to be performed.

We will not go into more details concerning replication transparency and proceed with the aspects of consistency regarding distributed systems together with replication of objects.

2.3.2 Consistency

In addition to the transparency requirement mentioned above, there is another general requirement for replicated data – one which can vary in strength between applications – the demand for consistency. Consider that clients make requests, and that requests involve updates as well as reads upon a set of logical data items. It

[1] Named for the Byzantine Generals problem.

is normally not acceptable for different clients to obtain different results when they make the same requests affecting the same logical data items. At least, this is not acceptable if the results lead to detectable and significant inconsistencies between different applications or users, or if the results break constraint within a single application. There mainly exist two different models of replication schemes concerning consistency:

Asynchronous Model This simple model of replica management is one in which all client requests are processed by the local replica server. The local replica servers communicate updates to all other servers managing replicas of the same data items. In this asynchronous model each server allows a client operation to return as soon as it has been performed locally, and communicates updates to its peers whenever it is convenient, or perhaps periodically. Servers process updates as they arrive.

Totally Synchronous Model In this model, all update requests are *totally ordered*. That is, requests are processed at all replicas in the same order. Only after the single current update request has been processed by all servers holding replicas of the affected data item, control is passed back to the client that requested the update. Then the next request, from whatever client it came, is handled.

In general, most replication schemes fall between the simple asynchronous and the totally synchronous models just described. A degree of replica consistency beyond that obtained in the asynchronous model is often required, but the response time of the totally synchronous model is sometimes inadequate. Various techniques have been designed to achieve satisfactory trade-offs between consistency, availability, and response time. The most important among these are two basic approaches:

Quorum-based Schemes Some schemes minimize communication costs and enable a degree of concurrent processing by allowing some but not all managers of replicas to be contacted before a read or update request is allowed to complete.

Causality There are designs which achieve a degree of concurrent request processing by allowing some requests to be processed in different orders at different replicas – in so far as this cannot produce inconsistencies. An important ordering of this type is *causal ordering* (cf. [Coulouris *et al.*, 1994]).

An important general mechanism for keeping replicas consistent is to multicast updates to all processes that manage physical replicas. That is, updates are sent in messages which are delivered to precisely the set of processes that manage replicas of the corresponding data item. *Process groups* are logical destinations for multicast messages, whose membership can be kept transparent to the message sender. The ISIS system (cf. [Birman, 1993]) provides process groups and ordered multicast on top of existing operating systems, particularly Unix.

2.3.3 The Basic Architectural Model

We now give a basic architectural model for the management of replicated data. The model involves replicas of data items held by distinct *replica managers*, which are processes that contain the replicas and perform operations upon them directly. Note that we avoid the term "replica server" and use "replica manager" instead, because this general model may be applied to an application rather than a service implementation, and client processes can in that case act as replica managers.

Typically, each replica manager maintains a physical copy of every logical data item, but there is no reason why data items should not be maintained by different sets of replica managers. For example, some data items might be needed mostly by clients on one Local Area Network (LAN) rather than another, and so there is little to be gained by replicating them at managers on the other LAN.

The general model is shown in Figure 2.5. Clients each make a series of requests. Each request in general involves a combination of reads of logical data items and updates to logical data items. Requests that involve no updates are called *read-only* requests. Requests that update at least one data item are called *update* requests. Each client's requests are first handled by a component called a *front end*. The role of the front end is at least to communicate by message passing with one or more of the replica managers, rather than forcing the clients to do this themselves explicitly. The front end is the vehicle for implementing replication transparency. A front end may be implemented as a user package executed in each client, or it can be a separate process. The choice involves a trade-off between, on the one hand, the greater efficiency of a local procedure call (in the contrary to a remote procedure call) over communicating with another process; and, on the other hand, possible advantages to share information such as server addresses in the front end on behalf of all clients at the same LAN.

In different models, the front end communicates per operation either with only a single replica manager, with several replica managers, or with all replica managers. Furthermore, the front end may collate the results of requests performed at more than one replica manager. For example, if a fault tolerant service is provided, the front end may be used to request values from all replica managers, and to select

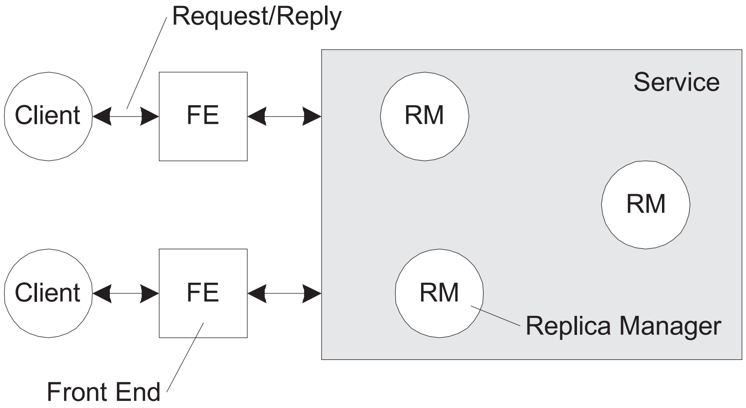

Figure 2.5: A basic architectural model for the management of replicated data.

values returned by a majority to be passed back to the client, in case one or more of them is faulty.

If clients and replica managers are separate processes, the model approximates what is called the *gossip architecture* for highly available data (cf. [Coulouris *et al.*, 1994] and [Ladin *et al.*, 1992]). This name was chosen because the replica managers exchange “gossip” messages periodically in order to convey the updates (the “news”) they have each received (see Figure 2.6).

In the gossip architecture, the front end normally communicates with an individual replica manager for each operation. Alternatively, it can communicate directly with more than one replica manager per operation. The gossip architecture does not define whether front ends are separate to client processes.

In the *primary update* model (see Figure 2.7) of data replication for enhanced availability, all front ends communicate with the “primary” server when updating a particular data item.

The primary server propagates the updates to the other servers, called “slaves”. If the primary server fails, one of the slaves can be promoted to act as the primary server.

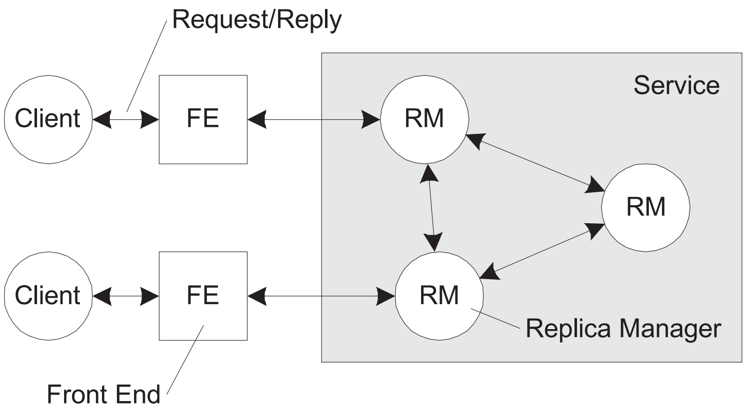

Figure 2.6: The gossip model for the management of replicated data.

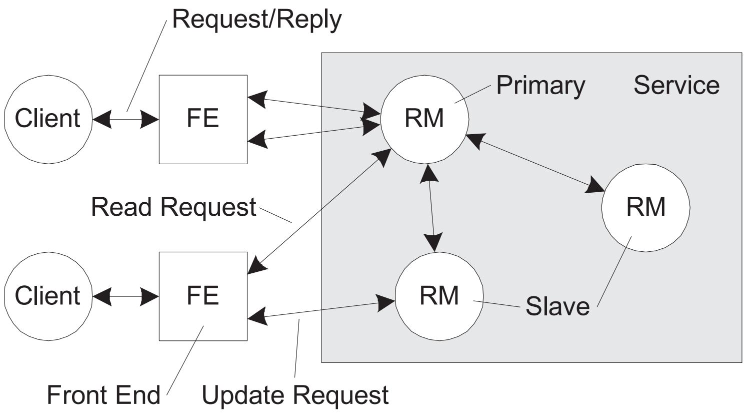

Figure 2.7: The primary copy model for the management of replicated data.

Chapter 3

Distributed Virtual Worlds

This chapter gives insight into the matters of distributed virtual worlds. It focuses on kinds of virtual worlds that can be modeled by taking advantage of the spatial properties of real worlds. The behavior of objects inside such a virtual world completes it.

3.1 Introduction

In the contrary to the previous chapter, this chapter does not give an explicit and detailed definition of the technical terms of the special field it is about. It simply relies on the intuitive idea the reader has of virtual worlds. We refer to [Roehl, 1995a] for an overview in distributed virtual world systems, and to [Hubbold *et al.*, 1997] for a description of the various design issues.

In addition, this chapter considers only *objective* virtual worlds (cf. [Greenhalgh, 1996b]), that is, worlds in which the same information is in principal available to all participants. This is the method adopted in almost all current virtual world systems and is in accordance with people's experiences in everyday life. However, subjectivity may be of importance in some situations.

Normally, a virtual world is composed of a number of identifiable objects or artifacts. These typically have properties like

- a location,
- a visual appearance,

- an audible character.

They also may have other characteristics, e.g.,

- video image or
- information for tactile feedback (cf. [Mayr, 1997b] and [Wolf, 1996]).

The object and each of its characteristics are represented and encoded as values of some appropriate and agreed types. Some of this information reflects persistent state while other types of information represent transient events. An object may be decomposed in a number of information sources, one for its location, another for its appearance, and so on, i.e., applying functional scoping. Each information source has a double character:

- viewed at any instant in time it has some current state, and
- viewed over a period of time it is seen to transmit transient messages.

Some transient messages describe changes in the persistent state and so are reflected by the state, while others have no persistent reflection.

It is characteristic of objective virtual world systems, and also one of the most significant characteristics of computers in general, that any number of replicas can be made of this information and be distributed to other processes. In virtual world terms, each object is expected to have many *simultaneous* observers, and each observer to observe many objects simultaneously.

3.2 The Spatial Model

With respect to various technical and physical restrictions, it is not practicable to design and implement *distributed* virtual worlds by just "modeling it as a whole and distributing it among as many computers as currently available". In turn to be operatable, some strategies based on several characteristic properties of the world to be modeled have to be developed and applied.

In our thesis, we focus on the *spatial* properties of real worlds and examine how these properties allow virtual worlds to be modeled in a suitable way.

3.2.1 Motivation

In an ideal virtual world, with an unlimited network bandwidth and with unlimited computational resources, a process and its possible human possessor would have simultaneous access to all available information inside the entire virtual world, and be able to organize that information to extract all relevant material as far as this process is concerned. So, each object of a virtual world, obviously including any observers of that virtual world, would be fully aware of every other object in all its forms.

However, everyday world is somewhat less than ideal in this respect. Computers and their users must, of necessity, lower their demands and aspirations. The greatest single challenge that arises from that is of consumers using the available finite resources in the best, i.e., most efficient, way: finding the subset of the available information which is most relevant and useful. To achieve this, the concept of *scoping* (cf. [Greenhalgh, 1996b]) has been introduced. The scope of a consumer of information describes a logical boundary around the information they wish to obtain and which they believe to be the most desirable.

3.2.2 Spatial Scoping

The nature of virtual environments analogous to everyday world we use to live in, introduces three factors which can be exploited in the selection of spatial scopes for information and interaction purposes:

- Rendering and perceptual limitations, e.g. the use of perspective projections, result in more distant objects being seen with less detail, and the field of view is typically limited.
- There exists some analogy with the distance attenuation in the everyday world. Electromagnetic radiation, e.g. light- and radio waves, as well as sound propagating in air drop off distance as $1/distance^2$.
- Interference between objects, e.g. inside a closed box, may cut off the objects inside the box from the objects outside.

These factors tend to make perceptual and interactional spaces relatively compact, structured, and well defined, and allow for the progressive abstraction – at decreasing levels of detail – of more distant objects. The first two factors mentioned above suggest scopes based on the *distance* from an observer, possibly with some consideration of the direction in which the observer is currently facing.

The third factor introduces a direct dependence on the current contents of the virtual environment within this larger scope. Here, the actual position of an observer determines the complexity of interaction with its surroundings. Current systems, which use such spatial scoping include:

- MASSIVE (cf. [Greenhalgh and Benford, 1995a], [Greenhalgh and Benford, 1995b], and [Greenhalgh, 1996a]), in which objects' auras define spherical scopes,
- AVIARY (cf. [Snowdon and West, 1994]), in which the visual renderer scopes its interaction with the virtual world through a projection of the clipping volume in the virtual space, and
- NPSNET (cf. [Macedonia *et al.*, 1995]), which is developing a system with approximately circular scopes based on the real-world characteristics of military engagement.

When considering a user perceiving and interacting with a virtual world, the ideal use of spatial scoping would exclude, i.e., ignore, objects which could not be perceived or influenced in any case. Furthermore, it would view objects at reduced qualities, e.g. lower levels of detail, only when such reduction was indistinguishable because of limitations of the renderer or the user's perceptual system. In practice, this is no *sufficiently* small scope, and allocation of resources must be restricted to the more significant subsets of this *ideal scope*.

To implement spatial scoping, the system must translate an observer's ideal scope of interaction into the corresponding set of objects, and make use of appropriate communication facilities to monitor those objects as illustrated in Figure 3.1.

3.2.3 Focus, Nimbus, and Awareness

The use of spatial scoping facilitates scaling to many users by limiting the number of object interactions that must be considered. For sufficiently large virtual worlds, the number of interactions that must be considered will be governed by the extent of the object's spatial scopes and by the population density of the space. As a consequence, the world can be extended infinitely at the edges of the population, without increasing the local complexity for individual objects being closer to the center of the world.

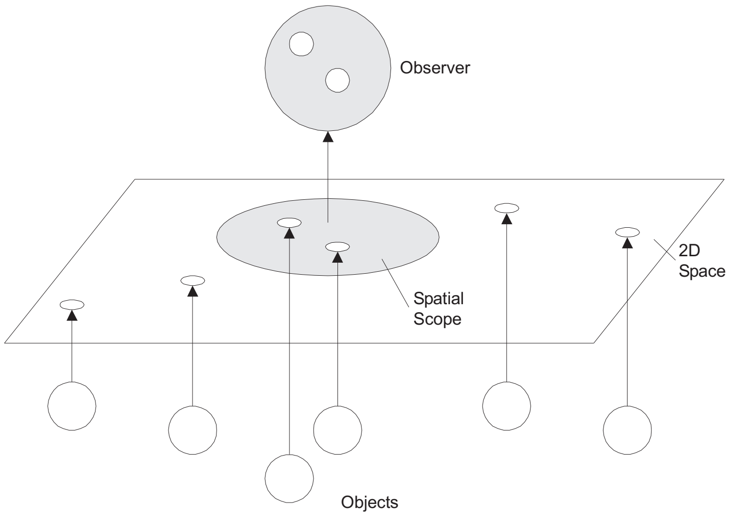

Figure 3.1: Illustration of spatial scope in two-dimensional virtual worlds.

A process – or more generally a dynamic group of processes – monitors all of the objects' spatial scopes and notifies objects when interaction with other objects becomes possible. These *spatial-scope-monitoring processes* may also notify objects when normal interaction ceases. They act as spatial traders, making object addresses available to other objects based on spatial criteria.

The main concept in controlling interaction between objects is *awareness* (cf. [Benford *et al.*, 1994]). An object's awareness of another object quantifies the subjective relevance of the other object. Awareness may correspond to the volume of an audio channel or the level of detail of a graphical rendering. In general, more attention, i.e., more network bandwidth and computational power, will be devoted to objects with higher awareness values. Both objects want to affect their levels of mutual awareness – the observing object wants to focus its attention in particular areas or on particular objects, while the observed object wants to control its "visibility" so that objects in some areas are more aware of it than those in other areas. Here, the term "visibility" not only applies to the visual medium, that is, visibility may be applied to other media, for instance to a medium transporting audio data, too.

The concept of *focus* describes the observers level of attention, while the concept of *nimbus* describes the observed object's manifestation of observability. The observer's awareness of the observed object is then a combination of the observed object's nimbus and the observer's focus. See [Greenhalgh, 1994] for a mathematical model for generating focus, nimbus, and awareness.

The spatial model of interaction with respect to crowded collaborative virtual environments is extended in [Benford *et al.*, 1994], [Benford and Greenhalgh, 1997] and [Benford *et al.*, 1997].

3.3 Behavior

The technical term "behavior" plays a major role in the context of virtual worlds. Assuming that an object has a collection of attributes, behavior is the ever-changing state of all those attributes. For example, an object may have the attribute "location"; as it moves around, its location changes. The way in which location changes over time is its behavior with respect to its location. A more detailed description of behavior can be found in [Roehl, 1995b].

3.3.1 Determinism and Non-Determinism

Behavior divides objects into two categories: those whose behavior is *deterministic* and those whose behavior is *non-deterministic*. Describing an object's behavior as deterministic simply means that its state is a function only of time, i.e., we can determine the complete state of the object at any moment. Objects with non-deterministic behavior are by nature unpredictable. Certainly objects controlled by users, i.e., observers, fall into this category, since users do things for their own internal reasons, and it is impossible to predict what a human being will do at some arbitrary time in the future.

However, it is not only human beings who exhibit non-deterministic behavior. Any object that has a randomly behaving mechanism inside will be similarly unpredictable.

Each category (deterministic or non-deterministic) can be further subdivided. Objects with deterministic behavior can be *static* or *animated*, and objects with non-deterministic behavior can be *newtonian* or *intelligent*:

Static Objects A static object's behavior is obviously deterministic. We know that its state will never change, and therefore its state at any given time is known. Any kind of permanent structure, like buildings, falls into this category. Since a building may change as a result of human actions, a simulation which allows this to happen would not treat buildings as static objects.

Animated Objects Unlike static objects, animated objects do change their states over time. However, these changes are easily predictable, since they are functions only of time and possibly of a set of predefined parameters. The hands of a clock and the moon in its orbit are examples for animated objects.

Newtonian Objects respond to changes in their environment. However, they do so in a very straightforward and generic way. One can pick them up, put them down or throw them and they will react according to the laws of whatever virtual physics their designers implemented. Newtonian objects react upon and respond to stimuli. Chairs and chess pieces are examples for newtonian objects.

Intelligent Objects appear to have free will. They have specific goals, complex behavior and are inherently unpredictable. Obviously, objects controlled by human operators fall into this category.

3.3.2 Levels of Behavior

As described earlier, every object has attributes, e.g. location, orientation, shape, etc. and behaviors alter those attributes over time. Just as it was helpful to divide objects into different categories, it is useful to divide behavior into different *levels*. For example, consider a virtual squirrel (cf. [Roehl, 1995b]). It might be capable of several "high level" behaviors, such as foraging for food, evading a predator, or finding a mate. At any given time, it is carrying out exactly one of those activities. It would not be eating while simultaneously fleeing a predator.

At the very highest level of "virtual intelligence" lies the process of selecting one of those behaviors. This selection may be based on any number of factors. For example, the decision whether to look for food or find a mate may be based on the current "hunger" value of the squirrel. The decision to fight or flee when encountering a predator might be based on the squirrel's speed, strength, distance, or exhaustion.

In [Roehl, 1995b], Bernie Roehl suggested to introduce four levels of behavior. He defined these four levels as:

Level 0 Direct modification of an object's attributes, e.g. "set location to point p".

Level 1 Change in an object's attribute over time, e.g. "scamper forwards 30 cm/s".

Level 2 Series of calls to level 1 behaviors to perform some task, e.g. "forage for food".

Level 3 Top-level decision making, e.g. "decide whether to forage, flee, or find a mate".

The level 3 behavior has the job of selecting a level 2 behavior. It sets priorities and does "executive level" decision-making. The level 2 behaviors decompose a task into simpler level 1 actions, which in turn update the actual entity state.

We do *not* agree with the kind of classification of behavior as it is described in [Roehl, 1995b]. It is just mentioned to complete the survey of the state-of-the-art in distributed virtual worlds.

Chapter 4

A New System for Distributed Virtual Worlds

In this chapter, we describe how a distributed system for the simulation of a certain category of real worlds can be modeled and designed – emphasis is put on worlds that can be decomposed by taking advantage of their spatial properties.

4.1 Introduction

Since this chapter compares the related terms "real world" and "virtual world", we will give an explanation of how these terms are used throughout the following.

Real World The term "real world" is used in order to designate a part of the world we *usually* live in and which is planned to be the subject of a simulation process. A part of the world might be, for instance, a classroom, a town, or a factory, or less common, the whole universe, a deep-sea scene, or a piece of iron on a nuclear, i.e., atomic, level (cf. [Lomdahl and Beazley, 1993]). Even "real worlds" existing only in imagination are conceivable here.

Virtual World The term "virtual world" designates the imitation of a certain real world as it is modeled and represented inside some computer system running a simulation of it. A virtual world in general depicts those properties and items of a real world that the virtual world's creator or designer is interested in. Briefly: Modeling an extract of the real world and adapting it to the needs of a specific distributed simulation system results in a virtual world.

4.2 Goals and Design Issues

The major objective of this thesis, seen from a potential user's perspective, was to develop a distributed simulation system – more precisely an object-oriented application framework for simulating virtual worlds – that allows a distributed simulation of a specific category of real worlds to be performed. This category of real worlds is distinguished from other categories in that these real worlds are decomposable by means of exploiting spatial properties.

In contrast to some existing categories of distributed simulation systems like multi-user games, which simply multiply an entire virtual world in order to distribute it, it was demanded that our distributed simulation system distributes *one sole* virtual world among an arbitrary number of processes and computers. This approach requires a fundamentally different design than the trivial approach does. Especially the meaning of the term "time" in addition to the problem of keeping the whole virtual world consistent distinguishes our design from trivial design. Whereas trivial systems like multi-user games stated before simply propagate state changes in one particular copy of the virtual world to other copies, a very different method has to be chosen in order to synchronize more complicated systems.

4.2.1 Requirements

Prior to the design and the implementation of our distributed simulation system, the basic requirements this system has to meet in order to function as intended have to be defined. This includes, among others, the following four requirements:

One World It has to be possible for an arbitrary number of users being physically distant from each other – eventually spread over the entire globe – to interact simultaneously within *one* virtual world. It is not convenient to have an arbitrary number of identical replicas of a virtual word, which are synchronized via broadcasting state changes. There *has* to be only one virtual world. This demand requires

- the virtual world to be distributed among processes as well as among computers. If there is just one virtual world, i.e., there exist no replicas of the whole virtual world, it is implied that this virtual world has to be somehow subdivided. How this has to happen is a major problem solved in this thesis.
- some parts of the virtual world to be replicated. Proceeding on the assumption that several users stay within the same region of the virtual

world, and assuming that these users work at distant computers, it has to follow that these particular parts of the virtual world have to be replicated.

- the virtual world to be kept in a consistent distributed state – especially if the preceding two requirements are taken into account.

Openness It has to be possible for users to join the virtual world at any time and from any place they want to. Additionally, leaving the virtual world at any time has to be possible, too. This requires

- some parts of the virtual world to be replicated among processes as well as among computers. The argumentation for justifying this demand is analogous to the preceding item.
- some parts of the virtual world to be migrated from one process to another process as well as from one computer to another computer. The need for this requirement is obvious if one considers that a new user, who currently entered a virtual world, has to be supplied with that part of the virtual world in which he is now in.

Liveliness It has to be possible to "populate" a virtual world with an arbitrary number of virtual objects. It has to be practicable to vest these virtual objects with certain properties and with certain behavior. It is demanded to have particularly equipped objects capable of being representatives for users in order to allow them to "live" within a virtual world. These "special objects" – later on we will call them *observers* – are a vehicle for allowing users to "slip" into a virtual world.

In general, virtual objects populating a virtual world have to be subdivided into the following three categories, each of which having distinct behavior and properties:

Obstacles and Artifacts are permanent structures which are not the subject of being changed during the lifetime of a distributed simulation. They represent things like buildings, tables, or rocks existing in a real world and correspond to *static objects* as described in Section 3.3.

Autonomous Agents are structures that move around in a virtual world and represent things of the real world like robots, asteroids, or molecules. This category of virtual objects corresponds to the *animated-objects* category and to the *newtonian-objects* category as described in Section 3.3.

Observers are more complicated virtual objects inhabiting a virtual world. They are representations of users inside a virtual world. Observers supply associated users with individual views of a virtual world – they lend their "eyes" to the user. Additionally, observers allow users to "slip" into a virtual world by means of allowing users to interact with other virtual objects that populate a virtual world. Thus, they lend their "bodies" to users participating in a distributed simulation, too. This category of virtual objects corresponds to the *intelligent-objects* category as described in Section 3.3.

Interoperability The ability to run under the control of multiple operating systems together with the ability to run multiple window systems during a *single* simulation session has to be achieved. This demand requires our distributed simulation system

- to be implemented on a variety of different operating systems. Since several parts of a distributed simulation system may run under different operating systems simultaneously, it is obvious that different operating systems exchange information among each another. A mechanism for enabling this to happen smoothly has to be designed and implemented.
- to be capable of handling a variety of different window systems. Several strategies ensuring this kind of interoperability have to be designed and implemented.

4.2.2 Design Issues

Since the distributed simulation system we have developed within the framework of this thesis meets the definition of what a distributed system is, it has to agree with the basic design issues mentioned in Section 4.2. Our distributed simulation system has to fulfill several of the issues itemized there:

Flexibility Our distributed simulation system must be designed to accept users at any time and at any location without any restrictions. Besides this, it has to be easy to use and map the users' problem space, i.e., the real world, to a set of processes and a set of computers in an adequate and transparent way.

Consistency Although our distributed simulation system has to lack of globally available information, has to utilize the technique of replication, and has to subdivide the subject it is responsible for, it has to be strictly consistent. There has to be uniformity in using the system, no matter when and where a user attends it. Moreover, it has to be guaranteed that distant users make exactly the same experiences within a virtual world, no matter how long they have been connected to the system, and regardless whether a user's computer is idle or not.

Location Transparency [1] Our distributed simulation system has to hide several influences that arise, if users attend the system from within different locations, i.e., computers. In addition, it has to be transparent to users, which process is currently running on which computer. It is the job of the system that processes have to be launched or terminated in response of user actions; no user has to be bothered with that.

Migration Transparency Users of our distributed simulation system have to be unaware of possible migrations of parts of a virtual world. If a user changes her/his current view of the virtual world she/he is in, this action might involve a lot of migrations of parts of the virtual world from one process to another process – or from one replica to another replica. These migrations must be transparent to users.

Replication Transparency With an increasing number of users using our distributed simulation system simultaneously, the number of replicas of certain regions of a virtual world increases. The process of generating new replicas has to be transparent to these users. The same has to be true if users disconnect from our distributed simulation system. The number of existing replicas might be reduced. This process has to be transparent to the users of our distributed simulation system too.

Size Transparency From the technical perspective, the extent of a virtual world has to be transparent. Users will be aware of the extent of a virtual world only by moving around and thus exploring it.

Scalability The major issue in achieving scalability is to avoid centralized structures in general. Our distributed simulation system must not have any centralized tables and algorithms. This includes that

- no process is allowed to have the complete information about the system state,
- all processes have to make decisions based solely on locally available information, and
- no assumptions have to be made about the existence of a global clock.

[1] A fact, a property, etc. being *transparent* (to users) means that its existence is concealed and will therefore not be noticed.

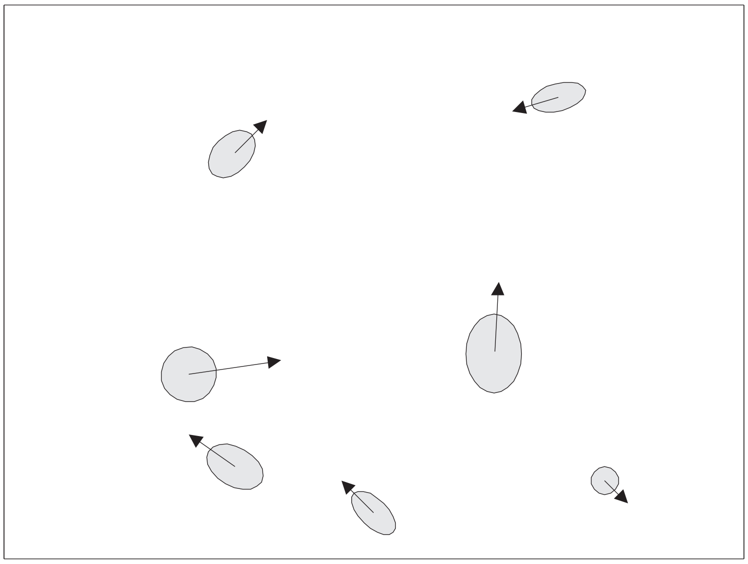

Figure 4.1: Example of a virtual world populated by virtual objects.

4.3 Our Virtual World

In the preceding section, we gave a survey on the goals and basic design issues our distributed simulation system has to meet. We mentioned that real worlds we are focusing on have the intrinsic property of being subdividable spatially. Moreover, it was mentioned that a virtual world to be simulated is populated by virtual objects. Figure 4.1 gives an example of how such a virtual world populated by an arbitrary number of virtual objects might look like.

4.3.1 Spatial Decomposition

We expect a real world – which is planned to be simulated – to be spatially subdividable. Motivated by the theoretical issues mentioned in Section 3.2, subdividing a virtual world is not only caused by distribution, it is also a concession to the limited network bandwidth predominating especially the Internet. But what does "spatially decomposable" mean with respect to our distributed simulation system? In Figure 4.2, the subdivision of the virtual world shown in Figure 4.1 is demonstrated.

A virtual world is subdivided into an arbitrary number of quadratic-shaped and equal-sized regions of finite spatial extension. In the diction of our distributed

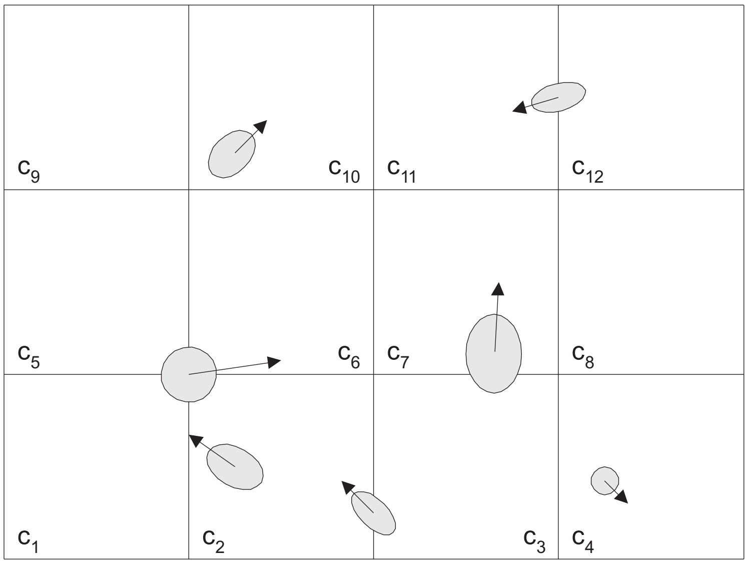

Figure 4.2: Example of a subdivided virtual world.

simulation system, these regions are called *cells*. In order to form a suitable virtual world, these cells must not overlap. In addition, the entire finite set of cells has to result in the whole real world to be simulated. In Figure 4.2, the virtual world in Figure 4.1 is subdivided into twelve cells, numbered from 1 to 12, which are arranged in three rows each of which consists of four cells. Larger virtual worlds might be composed of larger cells or of a larger number of small cells. It is not necessary that cells have to be combined in a rectangular arrangement in order to form entire virtual worlds.

The demand for a finite number of cells prevents the virtual world from getting infinite in its extensions, thus keeping it manageable for our distributed simulation system.

The job of subdividing entire virtual worlds into a predetermined number of non-overlapping cells is done automatically upon startup by our distributed simulation system. Users are unaware of this process.

4.3.2 Distribution and Replication

After subdividing a virtual world into an arbitrary number of cells, they must be distributed among processes running on several computers. It is the job of our

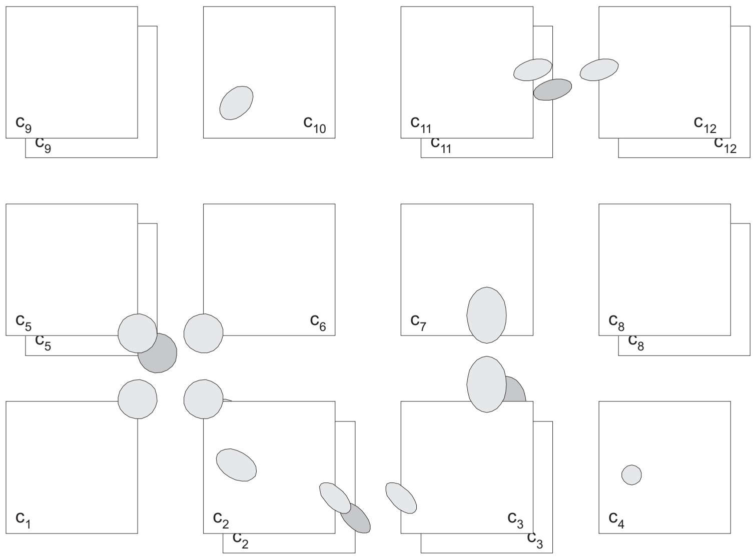

Figure 4.3: Example of a distributed and replicated virtual world.

distributed simulation system to do the distribution. This process is transparent to any user of the system. Major problems solved by our distributed simulation system regarding this process are:

- the multiplication of cells, i.e., the multiplication of the virtual objects populating individual cells,
- the launching of processes each of which is responsible for a specific cell and its replicas on a predefined set of computers, and
- the migration of virtual objects from one process to another process.

Figure 4.3 shows an example of a distributed and replicated virtual world. Each of the twelve cells together with its replicas is under supervision of an independent process. In this example, there exist replicas of the cells c_2, c_3, c_5, c_8, c_{11}, and c_{12}. It can be seen that there exist several replicas of individual virtual objects too. Replication is not only performed, if a new user joins the system, it is also performed when virtual objects cross cell boundaries.

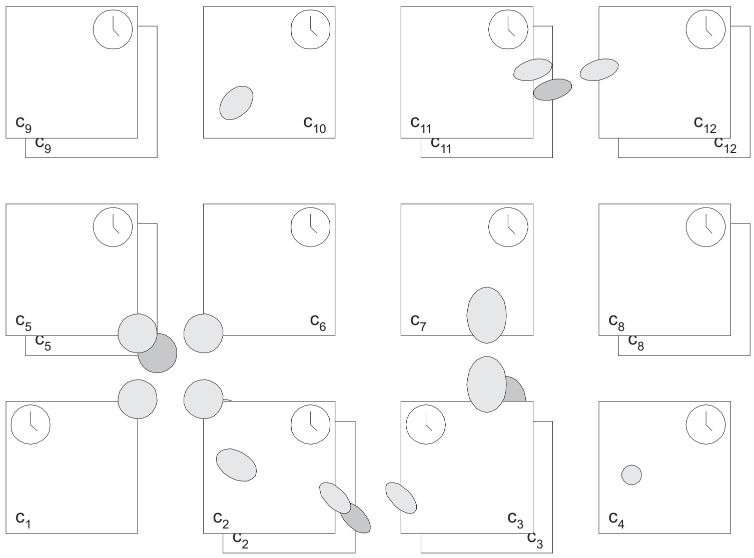

Figure 4.4: Example of a virtual world supplied with decentralized time.

4.3.3 Logical Time and Consistency

A major problem solved within the design of our distributed simulation system is the distributed consistency of a virtual world. This required us to give the term “time” a particular meaning within our system.

Consistency of a virtual world means that each cell together with any existing replicas of this particular cell has to be consistent. As long as there is no virtual object crossing the boundaries of a cell, no further problems will arise.

One can imagine each cell to be equipped with its own logical clock. This clock is used to supply the cell with logical time. Additionally, this clock is also responsible to time any existing replicas of the particular cell. As a consequence, since one clock feeds a cell including the cell's replicas, the cell and its copies stay synchronous. In other words: Everything that happens *deterministic* in one cell will happen deterministic in all replicas of that cell. See Figure 4.4 for an illustration.

Emphasis is put onto the word “deterministic”, since the existence of virtual objects behaving in a non-deterministic way, e.g. *observers* will undergo the systems consistency. Virtual objects that behave in a non-deterministic way have to follow a specific method in changing their state: They have to announce their approaching change of state and notify all existing replicas of this fact.

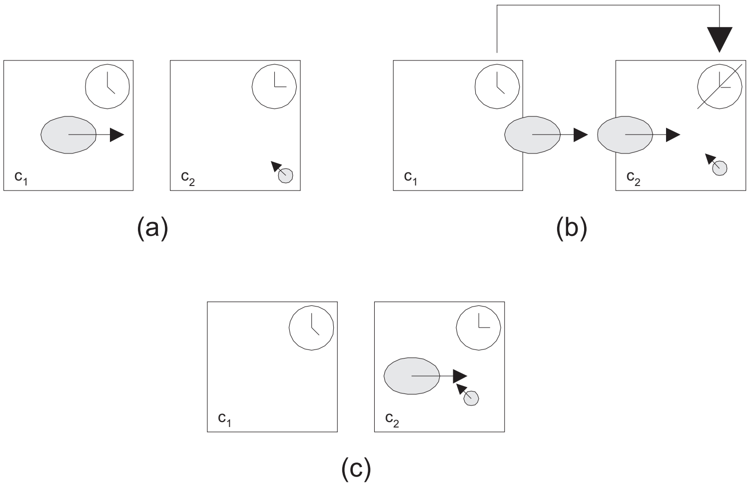

Figure 4.5: The migration of a virtual object.

4.4 The Population of Our Virtual World

A virtual world is populated by virtual objects. Section 4.2.1 itemizes the properties and the behavior a virtual object may have.

4.4.1 Migration

Some problems arise when a virtual object is going to cross cell boundaries, i.e., migrating from one cell to another cell. Since this object – for the time of its migration – belongs to more than one cell, it is no longer sufficient to synchronize one cell together with its replicas, as described before, now it is necessary to synchronize all concerned cells and their replicas. The demand for synchronizing more than one cell results in a union of all affected logical clocks.

In Figure 4.5(a), a virtual object located in cell c_1 is going to cross cell boundaries and will shortly afterwards enter cell c_2. Cell c_2 contains a virtual object, too, which is on collision course towards the virtual object in cell c_1. Both virtual objects are not aware of each other, since they are located in different cells. There is no need for any synchronization, the two logical clocks are still working independently from each other.

In Figure 4.5(b), the situation is changing, since the virtual object in cell c_1 is migrating to cell c_2. Our distributed simulation system is now in charge of handling the following three actions:

- The virtual objects populating cell c_1 have to be replicated in cell c_2.
- The virtual objects populating cell c_2 have to be replicated in cell c_1.
- The logical clocks of the two affected cells have to be synchronized.

These actions not only affect the two cells, they also affect any existing replicas of these two cells. The necessity of synchronizing the logical clocks can be found in virtual objects like that in cell c_2. Since it is possible that two virtual objects collide during the migration of one – or of both – of them, the affected cells have to be linked together. This linkage has to be performed by means of cell replication and by means of clock synchronization. If the two virtual objects collide, it will happen in both cells *simultaneously* and *synchronously*. As a result, the entire virtual world stays consistent.

After the virtual object of cell c_1 has migrated completely, the linkage of the two affected cells together with their logical clocks can be abolished. From this moment the cells again know nothing from each other (see Figure 4.5(c)). The whole procedure is repeated any time a virtual object crosses cell boundaries.

4.4.2 Spatial Scope

The version of spatial scoping we designed and implemented in our distributed simulation system works by means of grouping cells to *clusters*. By default, this *area-of-interest* – as we call our version of spatial scopes – of a virtual object, especially observers, consists of the cell the virtual object currently resides in. But it is possible to extent a virtual object's area-of-interest to a cluster of cells consisting of the cell in which the virtual object resides plus the eight surrounding cells. This is illustrated in Figure 4.6.

The method of achieving a synchronized cell cluster, i.e., a synchronized area-of-interest, works in the same way as the migration of virtual objects: The logical clocks of several participating cells are synchronized.

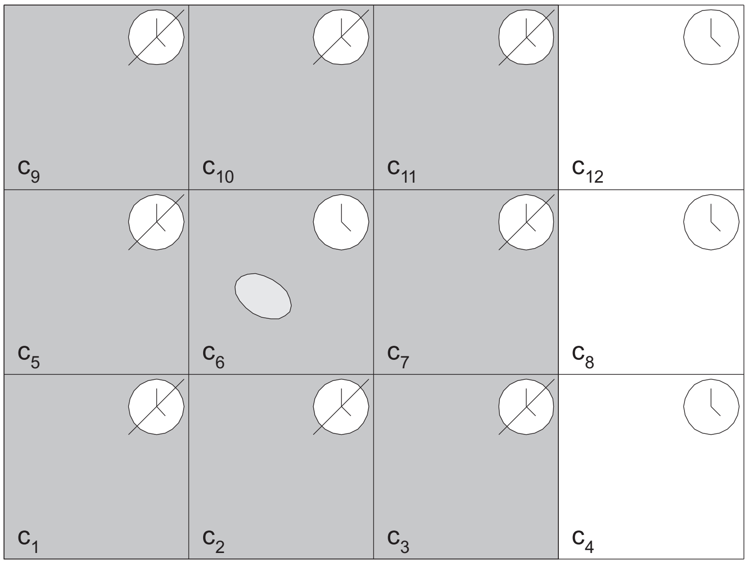

Figure 4.6: The area-of-interest of a virtual object.

4.5 Restrictions

Two peculiarities arise from our approach to handle logical time and logical clocks on a per-cell basis, restricting the number of applications that can be built using our distributed simulation system as a conceptual basis:

Expensive Migrations Since a lot of actions – many of them require virtual objects to be transmitted across some network from one process on one computer to another process on another computer – have to be performed in order to allow virtual objects to move from one cell to another, the creator of a virtual world should consider this to happen very seldom. In other words: Our distributed simulation system is inadequate for the simulation of real worlds, where cell crossings happen very often.

An example for such an unsuitable situation is the simulation of the traffic between two towns: Imagine two towns connected via a freeway with a lot of traffic on it. And imagine the virtual world's creator puts one town into one cell and the other town into another cell. This arrangement implies a tremendous amount of boundary crossings. Each crossing will entail both towns to be replicated, i.e., transmitted over some network connections.

Relativistic Effects Since cells are normally totally isolated from each other, it is possible for their logical clocks to drift apart for a noticeable amount resulting in a quite considerable clock skew. If a virtual object now migrates from a cell with a logical clock that is fast to a cell with a logical clock that loses, it appears to the population in the "slow" cell as if the migrating virtual object came from the future. Vice versa, the migrating virtual object seems to be pushed into the past.

Chapter 5

Design and Functionality of Our Distributed Simulation

This chapter describes our distributed simulation system from a technical perspective. The overall technical design and the entire functionality our system offers is explained in detail.

5.1 Introduction

In the preceding chapter we gave insight into the general ideas upon which our entire distributed simulation system is founded. In this chapter it is our intention to introduce the fundamental concepts of our distributed simulation system without bothering the reader with object-oriented design concepts or technical details. Readers already equipped with the basic knowledge presented in Chapter 4 and interested in hearing more about our distributed simulation system will be told, how, from a technical point of view, the concepts presented in the preceding chapter have been realized.

The terminology used throughout this chapter originates from terms like “cell” and “virtual world” to “process”, “communication technique”, and “message”. This reflects the changing of paradigms from conceptual issues to design and technical issues; a deeper understanding of the design and the functionality of our distributed simulation system is imparted here.

5.2 Design

The building blocks of our distributed simulation system – from an operating system's point of view – are a heterogeneous set of processes. There are three types of processes working together by means of information interchange:

- *Cell Manger*,
- *Replica Manager*,
- *Simulation Manager*.

Terminology: Prior to going into medias res, we want to comment on some terminology: A viewer's perspective determines what a "building block", process or manager, is called. From an operating system's point of view, these entities are called *processes*. From a system designer's perspective, these entities are called *managers*. Anyway, the names for these entities are normally used interchangeably. They usually denote the same thing. Exceptions will be pointed out if necessary.

The totality of these three types of processes[1] make up a distributed simulation according to our distributed simulation system. The functionality of these three processes corresponds to some major issues mentioned in Chapter 4, e.g. cells and replication. They master this duty just by interchanging information, more precisely by the interchange of messages and by the transmission of virtual objects.

A major design goal in the development of our distributed simulation system was to avoid extensive process hierarchies like those inherent in pure Client/Server architectures (see Figure 5.1(a)). Moreover, we intended to develop a system in conformance with common peer-to-peer models (see Figure 5.1(b)). The reason for this design precondition can be found

- in the handling of logical time together with the demand for saving network bandwidth, i.e. to limit the use of messages, for synchronizing the entire virtual world and
- in the attempt to gain a higher system's stability in the case of network failures.

As far as our distributed simulation system is concerned, we took the best of both worlds and built a system that benefits from both models.

[1] Two of them – replica manager and simulation manager – may have several instances running simultaneously.

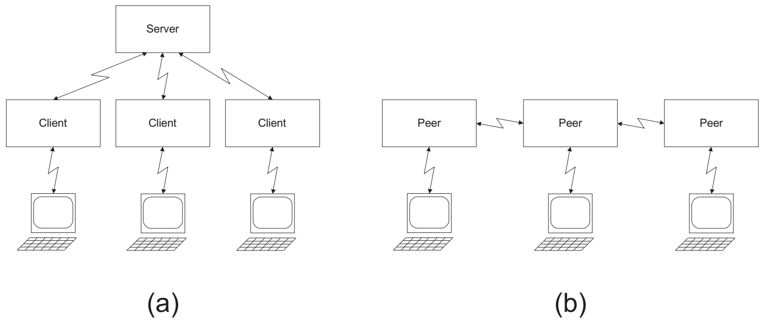

Figure 5.1: Client/Server and peer-to-peer model.

The approximation to peer-to-peer systems is reflected by the subdivision of a virtual world into cells. This subdivision results in an arbitrary number of independent units having equal rights; there is no hierarchy among them. This independence from any higher instance is maintained throughout an entire simulation session. But there are two exceptions: The first one results from a simplification of our distributed simulation system, the second one is a concession to the system's throughput.

Migration If virtual objects cross cell boundaries, it is necessary to unite the affected cells, as it is explained in the preceding chapter. In order to do that, this procedure is delegated to a "coordinator". This coordinator is not elected. There is, for the sake of simplicity, a "default coordinator" which takes over the process of unifying cells. As a consequence it turns out that this single coordinator has to be placed in a higher level of hierarchy, thus our peer-to-peer shaped distributed simulation system briefly turns into some kind of Client/Server, i.e. hierarchical, model.

Replication Whereas the previous item dodges the demanded peer-to-peer model temporarily and only under certain conditions, this item breaks with the peer-to-peer philosophy very fundamentally – for a good reason. Replication is used in our distributed simulation system to allow multiple users to enter the same region of a virtual world simultaneously (see Chapter 4). In order to keep several existing replicas of a single cell synchronous, a "global" logical clock has to be implemented. This can be achieved via two different approaches:

1. **Central Approach** The central approach – the one we have chosen – appoints a central instance which is responsible for synchronizing its assigned cell replicas. One can imagine this instance as some kind of "global time coordinator". The benefit of this solution is that it does not stress the available network bandwidth too much. The disadvantage

of this technique is that the coordinator faces the danger of becoming a bottleneck and represents a single point of failure.

2. **De-central Approach** A suggestion for a de-central approach, which is motivated by preventing drawbacks like those previously mentioned, is made in [Lynch, 1996]. An algorithm named *Replicated State Machine*, implemented in each participating process, synchronizes all pendants via the broadcast of messages from one participant to all the others. Since these broadcasts are emitted by each process, the network is under heavy stress.

So, a designer of a distributed system like our distributed simulation system has to decide whether to implement the centralized method – for the sake of simplicity and limited network bandwidth – or to implement the de-centralized method – for the sake of a greater fault tolerance.

To sum up, it can be said that our distributed simulation system conforms, if one focuses on the higher-level design, to the peer-to-peer model as long as there is no cell clustering necessary. If cell clustering becomes necessary – for the reasons of virtual object migrations – our distributed simulation system briefly turns into somewhat of a hierarchical Client/Server system.

5.3 Functionality

The functionality section grants insight into the behavior of the three kinds of managers – cell, replica, and simulation manager. The sum of all functionalities of all managers together with the dynamic behavior described in Section 5.6 results in an extensive specification of our distributed simulation system.

5.3.1 Cell Manager

The cell manager is the "master coordinator" in our distributed simulation system, chiefly responsible for starting up a simulation session smoothly (see Figure 5.2). Although, according to this diagram, the cell manager seems to take up a very central position during a simulation session, its (dynamic) presence is restricted to the period of booting. The only exception during which our distributed simulation system gets centralized, is the period of a virtual object's migration from one cell to another cell (see Section 5.2).

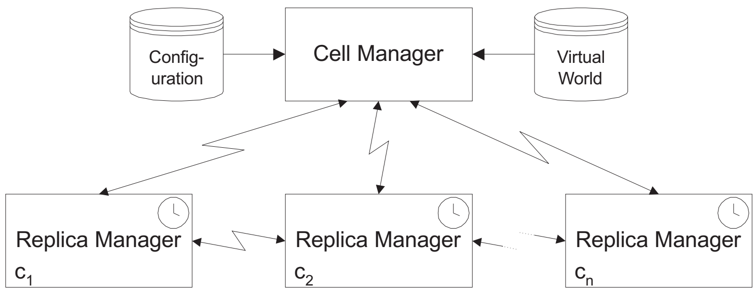

Figure 5.2: Block diagram of the cell manager.

The cell manager is mainly in charge of carrying out six different jobs:

Load Virtual Worlds The cell manager is responsible for loading virtual worlds from a *world database* into memory. Prior to the execution of the cell manager, the virtual world to be loaded has to be designed, implemented and stored by a *creator*. The virtual world is stored in a database by a mechanism called *passivation* and consequently read from a database by a mechanism called *activation*. The possibility to be *serialized* (passivated and activated) is a fundamental property of all objects[2] existing in our distributed simulation system.

Load Configuration Settings The cell manager is responsible for loading configuration settings from a *configuration database* into memory. Again, this is done via activation. Mainly, these configuration settings include information like computer names and, subsequently, network IP addresses. This information is used later on by the cell manager to determine the targets of physical process distribution.

Decompose Virtual Worlds The cell manager is in charge of the subdivision of a virtual world previously loaded from a database. The decomposition happens according to the spatial properties of a virtual world and results in a predetermined number of cells. Additionally, the cells are prepared for being distributed across some network.

Launch Replica Managers The cell manager is responsible for launching distant running processes (replica managers). The number of replica managers launched corresponds to the number of cells a virtual world consists of. This is the consequence of the fact that each cell is under supervision of a replica

[2]Here, the term "object" denotes a programming construct.

manager. Several items have to be taken into account prior to the distribution of a (large) number of processes over some network:

- The set of available computers taking part in our distributed simulation system has to be defined. This information will be extracted from the configuration settings.
- The quality and the capacity of the network connections to the various computers has to be estimated.
- The current – and, if possible, the future – load of computers eligible for running additional processes has to be determined.

For a survey of algorithms capable of handling the last two items which fall into the class of *load distributing and load balancing algorithms*, see [Bubendorfer, 1996], [Severance and Enbody, 1996], and [Sinha, 1997]. The approach we have chosen within our distributed simulation system is a static form of load distribution that is carried out according to the experiences of a person, e.g. a network administrator. Once the processes are distributed, this arrangement is not changed.

Initially Distribute Cells The cell manager is in charge of transmitting cells to the previously distributed replica mangers. Transmitting a cell means to transmit all virtual objects within that cell. Transmission of objects in general over some network connections is again performed by the serialization mechanism.

Group Cells If a virtual object is going to cross cell boundaries, i.e., migrating from one cell to another, it is necessary to group all affected cells in order to maintain consistency (see Chapter 4). Since we did not implement an election algorithm that temporarily determines a coordinator capable of doing this, we decided to instruct a default coordinator to do this kind of groupings. So, the cell manager became an instance of the default coordinator and represents the "everlasting election winner". We refer to the *Migration* paragraph in Section 5.2.

As mentioned, the cell manager is the central process in our distributed simulation system. But having a central process would severely violate the peer-to-peer properties we want to achieve for our distributed simulation system. If one looks at the previously listed items, he/she will discover that all items, except the last one, apply only during the system's startup. Thus, the cell manager "centralizes" our distributed simulation system only during the bootstrap phase. After that phase, centralization only occurs in the case of cell clustering.

5.3.2 Replica Manager

The replica manager is a mechanism for cells[3] to be replicated, since each cell together with any existing replica(s) is under supervision of a replica manager. Replication is used in our distributed simulation system to allow multiple users to coexist simultaneously within a single cell. Since users may be physically distant from each, some mechanism to replicate cells is obligatory. A replica manager supervises two slightly different kinds of simulation managers:

- Simulation managers representing cells without a user interface – so-called *Cell Prototypes*. A cell prototype is the prototypic model of any possibly existing replica of itself.
- Simulation managers representing cells equipped with a user interface – our implementation currently supports either a Unix-X11 or a Win-32 graphical user interface (GUI). These cells are replicas of the prototypic cell models.

Figure 5.3 shows a block diagram of a replica manager having one cell prototype[4] and two replicas of the prototypic cell attached to it. A replica manager not only allows cells to be replicated, it also exists for another essential reason: It supplies attached cells with logical time. This leads to the list of functionalities the replica manager offers:

Launch Simulation Managers The replica manager is responsible for launching a distantly running processes, namely the simulation manager that is a representation of a cell prototype. The same considerations as mentioned in Section 5.3.1 have to be taken into account prior to the launch of a processes in a network of computers.

Transmit Cells The replica manager is in charge of transmitting a cell to a previously launched simulation manger. Transmitting a cell means to transmit all virtual objects within that cell. Transmission of objects in general over some network connections is performed by the serialization mechanism (see Section 5.3.1).

Time Simulation Managers As a major assignment a replica manager has to supply all attached simulation managers with logical time. This has to be done in order to keep a cell and all existing replicas of that cell synchronous, i.e.,

[3]Here, the terms "cell" and "simulation manager" are used synonymously.

[4]Each replica manager has exactly one cell prototype.

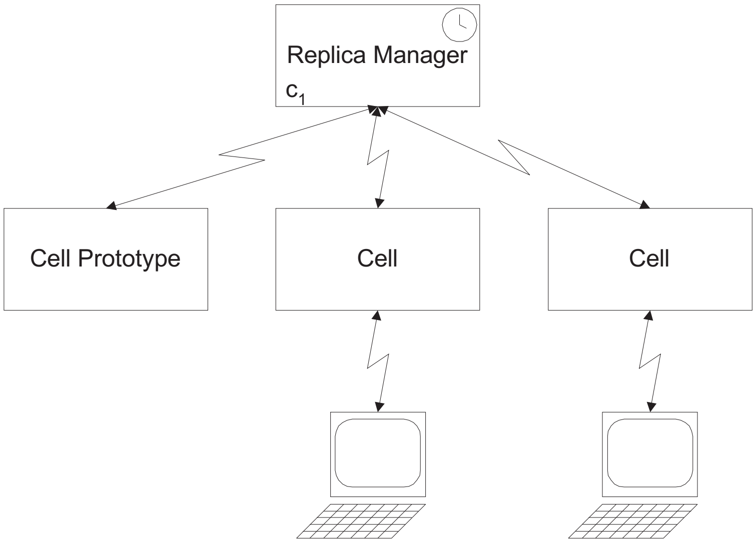

Figure 5.3: Block diagram of the replica manager.

guaranteeing the synchronizity of cells, and thus guaranteeing the consistency of a virtual word.

Add Users A replica manager is responsible for integrating new users into a running simulation session. The reception of an additional user effects the entire distributed simulation system. A lot of actions (see Section 5.6.5) have to be performed, always under the premise of keeping the whole virtual world in a consistent state.

Representing the "middle classes" in the hierarchy of managers, the replica managers – as a totality in our distributed simulation system – work together in a way that has more of a peer-to-peer system than of a traditional Client/Server system. If the cell manager is supposed a special kind of replica manager[5] that coordinates replica managers if necessary, our distributed simulation system then represents a quite mature, peer-to-peer shaped distributed system.

5.3.3 Simulation Manager

From a conceptional perspective there is no difference between a cell and a simulation manager. But from a technical point of view we have to distinguish: A cell is a set

[5] We remind the reader of the previously introduced term "default coordinator".

of virtual objects staying in a certain region of a virtual world, whereas a simulation manager is a process that manages all the virtual objects belonging to a cell – for example:

- it “feeds” virtual objects with time from the replica manager,
- it checks for collisions, and
- it forwards messages, events, and commands to addressed virtual objects.

Figure 5.4 shows a block diagram of a simulation manager. As already mentioned in Section 5.3.2, there are two types of simulation managers:

- simulation managers having no user interface and
- simulation managers equipped with a user interface.

Since prototypic cells are the primary source for eventually existing replicas, and users are connected only to replicated cells, it is not required for prototypic cells to be equipped with a user interface. Prototypic cells stay in concealment, just waiting to be replicated. As a consequence it is possible to perform a simulation session without any representation on a computer monitor. In such a case, the whole simulation merely consists of prototypic cells. Figure 5.4 shows a simulation manager having a user interface attached. To get a prototypic simulation manager, i.e. cell, one simply has to drop the user interface part.

To supplement and to complete the list of responsibilities given above, the following itemizes the functionalities a simulation manager offers:

Manage Virtual Objects Managing virtual objects is one of the major responsibilities of a simulation manager. Since a simulation manager is the technical environment of a cell, i.e., a collection of virtual objects, it is in direct “contact” with this cell and, therefore, simulation managers have the possibility to manipulate virtual objects directly. Here, manipulating virtual objects stands for a variety of fundamental actions:

  - A simulation manager is in charge of timing its “entrusted” virtual objects. Logical time information is received from replica managers via messages and distributed among the convened virtual objects. As a consequence, each virtual object gets its own logical time information, upon which it can perform any imaginable action.

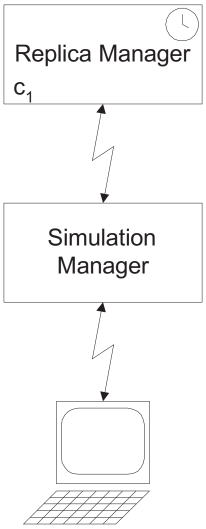

Figure 5.4: Block diagram of the simulation manager.

- Forwarding requests, e.g. messages and events, is a further functionality of a simulation manager. Virtual objects articulate "wishes" concerning other virtual objects by wrapping these wishes into messages and forwarding them to the simulation manager, asking it to pass the messages to the addressed recipient. From the perspective of a virtual object, a simulation manager is the address to turn to with requests it cannot handle itself.
- Forwarding responses, e.g. messages and events, is in charge of a simulation manager too. Responses are the answers a virtual object gets upon request. Since these responses are transmitted from one simulation manager to another, a simulation manager has to distribute arriving responses to the addressed virtual object.
- Upon commands from a replica manager, a simulation manager has to create or, respectively, destroy virtual objects it is responsible for. There are two different kinds of virtual objects: conventional virtual objects and observers. Whereas the treatment of conventional virtual objects results in no further difficulties, problems arise with the creation and the destruction of observers. This is because observers have user interfaces.

Manage User Interfaces Managing a graphical user interface is essential in performing simulations that can be watched and that allow users to interact. Seen from within our distributed simulation system, a user is represented by a virtual object called observer. Thus, all possibilities a user must have within a simulation session are mapped to observers. Observers perceive their surrounding virtual world. Therefore, this image has to be transported to the associated user. Observers move around in a virtual world according to commands. Therefore, these commands have to be transported to an observer. The way these mappings from an observer to a user and vice versa are done is part of the functionality of a simulation manager.

Check for Collisions Since a simulation manager knows everything about the virtual objects it is responsible for (including their "physical" extents), it is intended to detect collisions. So, a simulation manager checks a cell for any eventual collisions among virtual objects and performs certain actions in case a collision occurs. These actions depend on the type of participating virtual objects and usually include the destruction and the creation of virtual objects.

Simulation managers – in both described forms – are the "building blocks" of our distributed simulation system "burdened with all the hard work" of running a simulation session supported by the other building blocks. Especially the supervision of virtual objects and the handling of users with all their demands are up to simulation managers. Replica managers and cell managers are "just" supplemental.

5.4 Virtual Objects

Virtual objects – the "population" of a virtual world – represent those structures of the real world that are of particular interest to a virtual world's creator. In this section, we subdivide virtual objects into categories according to properties they have regarding technical aspects. Seen from a technical point of view, virtual objects can be subdivided into two different categories:

- The first category is made up of virtual objects having a behavior designated as a "standard behavior".
- The second category unites virtual objects having – besides the standard behavior – an additional behavior. Observers are typical members of this category.

Here, the term "behavior" denotes a technical behavior, i.e. functionality is implied. The standard behavior and functionality of virtual objects together with the standard properties comprises[6]:

Aging Virtual objects have the characteristic property of getting older. This enables them to change various items, e.g. appearance and motion, during the progress of their existence. It is even possible to limit the lifespan of a virtual object by killing it when its age has reached a predetermined value.

Motion and Extent It is possible for a virtual object to move around in a virtual world as it desires. The way this motion happens can be deterministic[7], it can be completely random, or it may depend upon other virtual objects. A property that roughly corresponds to the ability to move is the property to have a somewhat physical extension. This implies that virtual objects may collide.

Interaction A virtual object provides several ways to interact with other virtual objects in a virtual world. There are direct and indirect ways in performing interaction:

- The direct way of interaction is to urge a simulation manager to create a new virtual object or to destroy an existing virtual object.
- The indirect method in interacting with other virtual objects is inherent in a virtual object's nature: Since virtual objects have some kind of solid "bodies", it is likely for them to collide, especially if they are moving around in "space".

In the following, we describe the extended behavior and the functionality of a virtual object. Extended behavior does not replace the standard pendants, it extends them. Visual objects equipped with extended behavior are called observers:

Visual Feedback An observer has the extended functionality of supplying an associated user with an image of that part of the virtual world it is in and that can be "observed" by it. From a technical viewpoint this implies a connection between observers and some kind of user interface; an observer has to be capable of controlling screen output. Observers in our distributed simulation system are able to control screen output on a Unix-X11 terminal as well as on a Win-32 terminal within a single simulation session. Controlling screen output

[6] In this section, we deliberately mix the terms "behavior", "functionality", and "property". Anyway, throughout this section, we prefer to use the term "behavior" in denoting one of the three expressions.

[7] There are several different kinds of deterministic motions, e.g. linear motion, accelerated motion, etc.

means to have influence on the details of a virtual environment presented, the scale used, the kind of view transformation applied, etc.

Interaction Here, the term "interaction" denotes the possibility a user has in "steering" her/his "virtual pendant", i.e., observer, around some virtual world. Moving around in a virtual world means to direct the observer – via commands sent from a keyboard – to any location a user wants to move. But interaction is not restricted to the process of controlling the movement of an observer. Several other actions can be caused by users. The spectrum of these actions ranges from simple actions like the creation of a new virtual object to very complex actions like the picking up of some kind of workpiece in a factory simulation. Again, our distributed simulation system allows users to utilize Unix-X11 terminals as well as Win-32 terminals in order to interact with observers.

5.5 User Integration

Integrating users into a running simulation session is a process that comes about in several steps. The cause for such a sequence of actions is the launch of a simulation manager that has a user interface attached to it. A simulation manager is usually launched by a person that tends to participate in a running simulation session:

Connect to a Replica Manager A newly started simulation manager first has to find "its" replica manager – simulation managers are connected to replica managers as shown in Figure 5.5. This problem is solved by broadcasting a `Connect` message. The underlying communication structure provides mechanisms to emit reliable broadcasts to a dynamic set of processes that are members of a predefined communication group.

Create an Observer Once a simulation manager has found "its" replica manager by the receipt of a request to the previously sent `Connect` message, it is due to create an observer that represents the user who has launched the simulation manager in order to participate a simulation session. The newly created observer is sent to the replica manager, which, in turn, distributes it to any existing replicas. By the way, the replica manager that answers to the initial `Connect` message determines the cell in which an observer has to be created.

Receive a Cell Virtual objects within a cell are transmitted from the replica manager – more specific: from the prototypic simulation manager – to the new simulation manager. Now, the process of connecting to a running simulation session – seen from a network perspective – is finished.

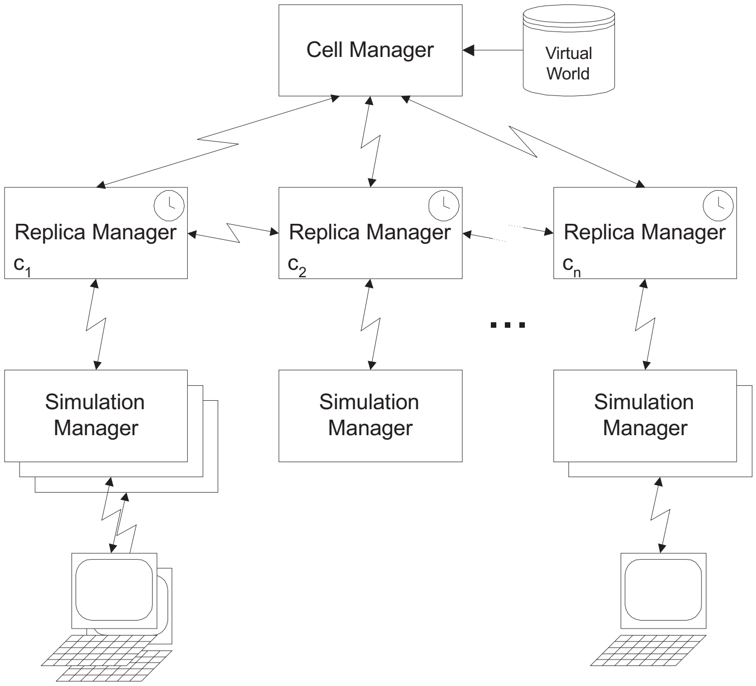

Figure 5.5: The overall technical design.

5.6 Dynamic Behavior

The dynamic-behavior section depicts – by means of a new kind of diagram language – the interaction of the three kinds of managers (cell, replica, and simulation manager) in the course of various different use cases. In this section we just focus on dynamic proceedings involving messages. Diagrams like flowcharts and similar forms of representing dynamic processes are neglected here.

5.6.1 Illustrating Concurrent Processes

In order to make it possible to adequately illustrate the dynamic behavior of various processes or managers respectively – either running "stand alone" or "playing in a team" – we had to use a new kind of diagram language[8] (cf. [Foster, 1994]) that allows us to:

[8]Commonly used *state diagrams* do not offer the possibility of introducing time nor is it possible to combine several state diagrams to illustrate tricky process interdependencies. Whereas

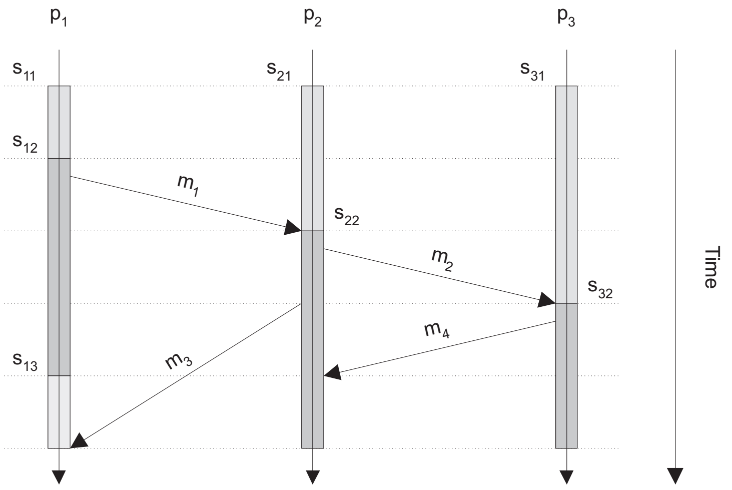

Figure 5.6: A diagram language for describing concurrent processes.

- depict the different states of processes,
- show state transitions of processes,
- illustrate the interplay of two or more processes, and
- arrange everything over a time coordinate, i.e., to put everything into correct relations concerning time.

All the issues itemized in the list above are contained within one single diagram language in a compact and lucid way (see Figure 5.6). Because it is not the subject of this thesis, we will not go into detail in explaining all the features of the diagram language. We will just focus on characteristics that will be used later on.

Figure 5.6 shows three processes p_1, p_2, and p_3. Each process is represented by a vertical bar that runs parallel to the time axis, from top to bottom. The horizontal dotted lines indicate positions having the same (global) time. A process consists of a set of states. Process p_1 consists of the states $S_1 = \{s_{11}, s_{12}, s_{13}\}$, process p_2 consists of the states $S_2 = \{s_{21}, s_{22}\}$ and respectively process p_3 consists of the states $S_3 = \{s_{31}, s_{32}\}$. Each state is initiated by its name and makes that state's associated vertical bar reflect the new state by changing its color.

object-interaction diagrams do offer the possibility of illustrating process interdependencies, they unfortunately lack the capability of describing states adequately.

State transitions may occur at any time, and they are normally caused by the receipt of an internal or an external event or, as this is true with the majority of our applications, with the receipt of a message. All state transitions occurring in process $p1$ are caused by internal events or by fulfilling some task, whereas, for example, the state transition from state s_{21} to state s_{22} in process p_2 is initiated by the receipt of the message m_1.

Figure 5.6 shows four messages: m_1, m_2, m_3, and m_4. Message m_1 and message m_2 cause state transitions, while message m_3 and message m_4 do not. The time a message needs to be transported from one process to another process is indicated by the slant of the associated arrow.

5.6.2 Booting Our Distributed Simulation System

Figure 5.7 shows the course of messages as they occur while our distributed system boots; the whole procedure starts by launching the cell manager on some computer in a network. The major assignments performed are:

- the launch of a number of replica managers,
- the launch of a number of simulation manager prototypes, and
- the distribution of the entire virtual world.

We describe the process of booting our distributed simulation system by means of a virtual world consisting of just one cell. With virtual worlds comprising more than one cell, the procedure – from a message-transporting perspective – is analogous. All messages are sent asynchronously, i.e., no answer upon the transmission of a message is expected, thus no process performing a `Send` operation gets blocked.

1. Immediately after a cell manager has been started and has performed some initialization procedures, it launches a number of replica managers – one for each cell. All considerations regarding the distribution of processes across some network apply here (see Section 5.3.1). Initially, a cell manager is in the state `Init`.
2. A replica manager – immediately after its start in state `Init` – switches to the state `Connecting` in order to await its connection to the evolving distributed simulation system.

3. After it has launched the replica managers, a cell manager transmits `Population` messages to them. These messages notify the replica managers of their connection to the simulation system. The contents of a `Population` message are those virtual objects that belong to a specific cell. The exclamation marks in Figure 5.7 indicate that all emitted messages are *notifications*.

4. After receiving the "population" of a cell, a replica manager launches a prototypic simulation manager. All considerations regarding the distribution of processes across some network apply here (see Section 5.3.1).

5. A simulation manager – immediately after its start in state `Init` – switches to the state `Connecting` in order to await its connection to the evolving distributed simulation system.

6. After it has launched the prototypic simulation manager, a replica manager transmits a `Population` message to it. This message notifies the prototypic simulation manager of its connection to the simulation system.

7. When all these processes have been finished successfully, all the participating managers, i.e. cell, replica, and prototypic simulation manager, switch to the state `Idle`; the process of starting up our distributed simulation system is finished.

5.6.3 Timing the System

After it has concluded the process of starting up, our distributed simulation system is going to operate in standard mode – a mode where no particularly exciting interruptions and exceptions occur. This standard-operating mode is characterized by the periodical transmission of `Time` messages from the replica managers to their associated simulation managers. Again, these messages are sent asynchronously, thus no process performing a `Send` operation gets blocked for an unforeseeable lasting period of time. Figure 5.8 shows in a diagram the timing of our distributed simulation system.

The `Time` messages periodically sent out by a replica manager are used by simulation managers to coordinate their course of events, i.e., to time those virtual objects, which are under supervision of a simulation manager. In other words: A simulation manager uses the `Time` messages it receives from a replica manager to *animate* its cell.

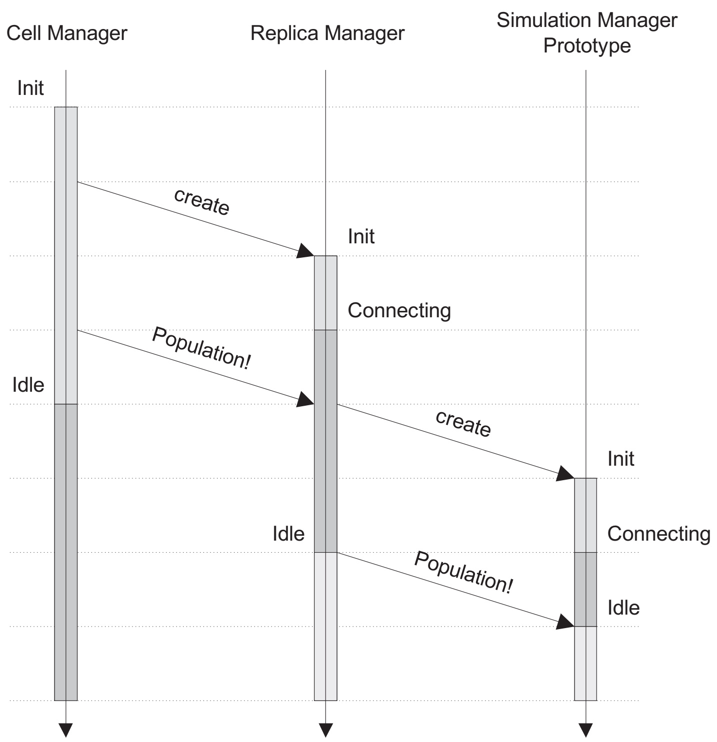

Figure 5.7: Starting up our distributed simulation system.

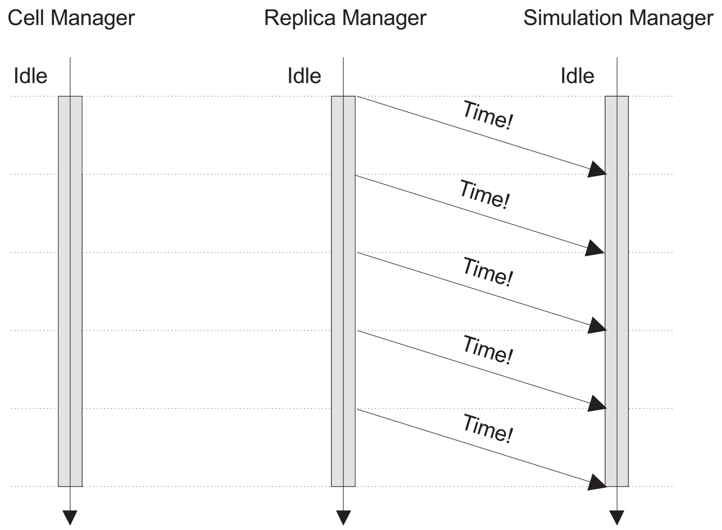

Figure 5.8: Supplying our distributed simulation system with time.

5.6.4 Terminating Our Distributed Simulation System

The termination of our distributed simulation system is a straightforward process initiated by a – concealed for its lack of importance – fourth kind of manager: the *Termination Manager*. The termination manager is executed by a person that runs a simulation session and that is responsible for its coordinated shut down. Here, two methods of sending messages are utilized:

Synchronous Communication In the synchronous form of communication, the sending and the receiving process synchronize at every message. In this case, both the `Send` and the `Receive` are *blocking* operations. This method of communication is utilized in the `Connect` sequence of messages in Figure 5.9.

Asynchronous Communication In the asynchronous form of communication, the use of the `Send` operation is *non-blocking* in that the process performing the send operation is allowed to proceed as soon as the message has left the computer. In the case of our distributed simulation system, the `Receive` operation is asynchronous too. The cascade of `Halt` messages is an example for asynchronous communications in Figure 5.9.

The following list describes the sequence of operations as they occur during the process of shutting down our distributed simulation system (see Figure 5.9).

1. After being started and having finished initializing, the termination manager broadcasts a `Connect` request message to all running processes of a simulation session in order to obtain a connection to the cell manager.
2. The cell manager answers – while all others ignore – the `Connect` request message via a `Connect` response message which holds the address of the cell manager. Now it is possible for the termination manager to address the cell manager directly.
3. The termination manager sends a `Halt` message to the cell manager and terminates afterwards.
4. The cell manager (and all other managers in succession) receives the `Halt` message, switches to the state `Exit`, sends, in turn, a `Halt` message to the "succeeding" manager and terminates.

This procedure removes the whole distributed simulation system from all computers it is distributed to.

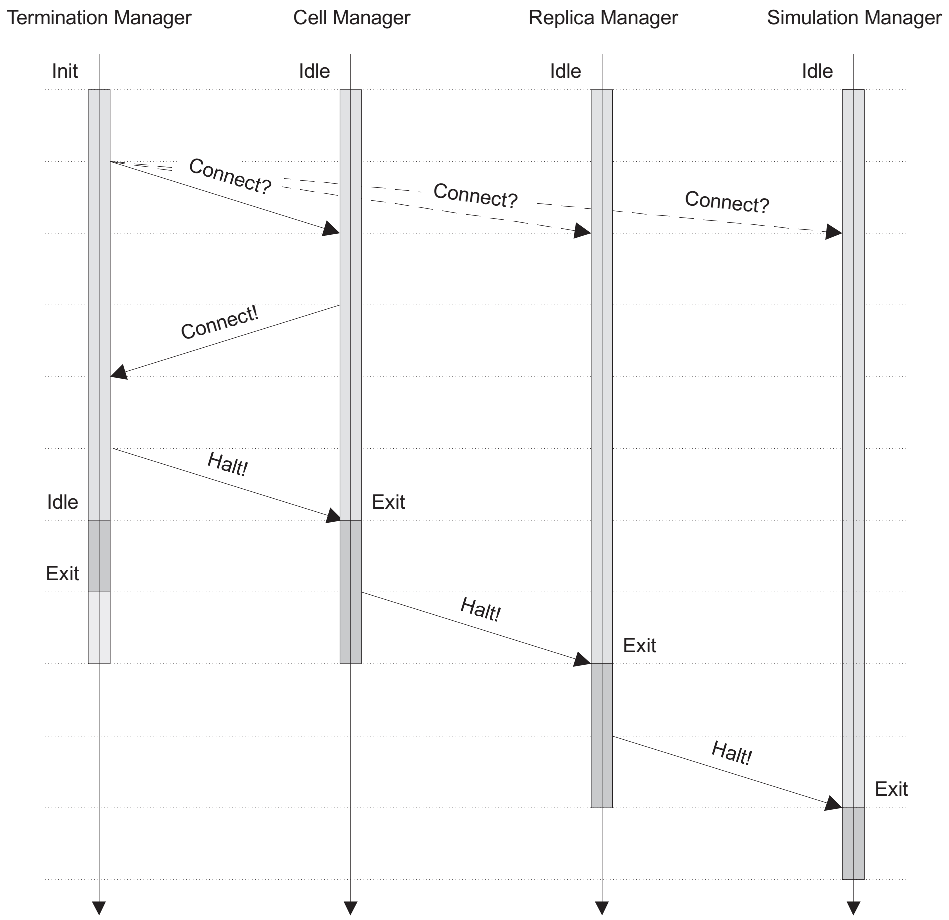

Figure 5.9: Shutting down our distributed simulation system.

5.6.5 Adding a User

A major design goal in planning our distributed simulation system was to allow users to join it at any time. To join our distributed simulation system means to start a new instance of a simulation manager – equipped with some sort of user interface, either a Unix-X11 or a Win-32 version – and entrust it with the job of getting connected. Like the process of terminating our distributed simulation system, the "new" simulation manager has to look for a suitable process, i.e., replica manager, in order to establish a network connection that will be used in the future for exchanging messages.

To summarize the operations which have to be carried out in order to add a new user to our distributed simulation system, we itemize them in the following list and depict it in Figure 5.10:

1. After being started and having finished initializing, the simulation manager – now in state `Init` – broadcasts a `Connect` request message to all running processes of a simulation session in order to obtain a connection to a replica manager. The `Connect` message contains information about which cell a user wants to join. This information is needed in order to give a hint for replica managers to be aware of which of them is addressed. After broadcasting, the simulation manager switches to the state `Connecting`.

2. That replica manager, which turns out to be responsible for the newly joining simulation manager, sends a synchronous `Transfer` request message to its prototypic simulation manager. This is done because replica managers do not have any knowledge about the contents of a cell – so they have to ask.

3. The addressed prototypic simulation manager answers the `Transfer` request message immediately, transmitting all the virtual objects in its cell to the inquiring replica manager. This is done by utilizing a `Population` message.

4. The replica manager, in turn, forwards the `Population` message to the simulation manager.

5. On receiving the `Population` message from the replica manager, the process of getting connected to a running simulation session has finished. The simulation manager switches to the state `Idle`.

As soon as a simulation manager has successfully been connected to a running simulation session, it initializes its user interface showing the region of the virtual world it is in, and participates in the animation by periodically receiving `Time` messages.

5.6.6 Migration of a Virtual Object

When a virtual object is going to cross cell boundaries, several actions regarding message transmissions have to take place. As explained in Section 4.4.1 and in Section 5.3, two major groups of operations have to be carried out in order to enable a virtual object to migrate successfully from one cell to another:

- the unification of cells, i.e., replica managers, and

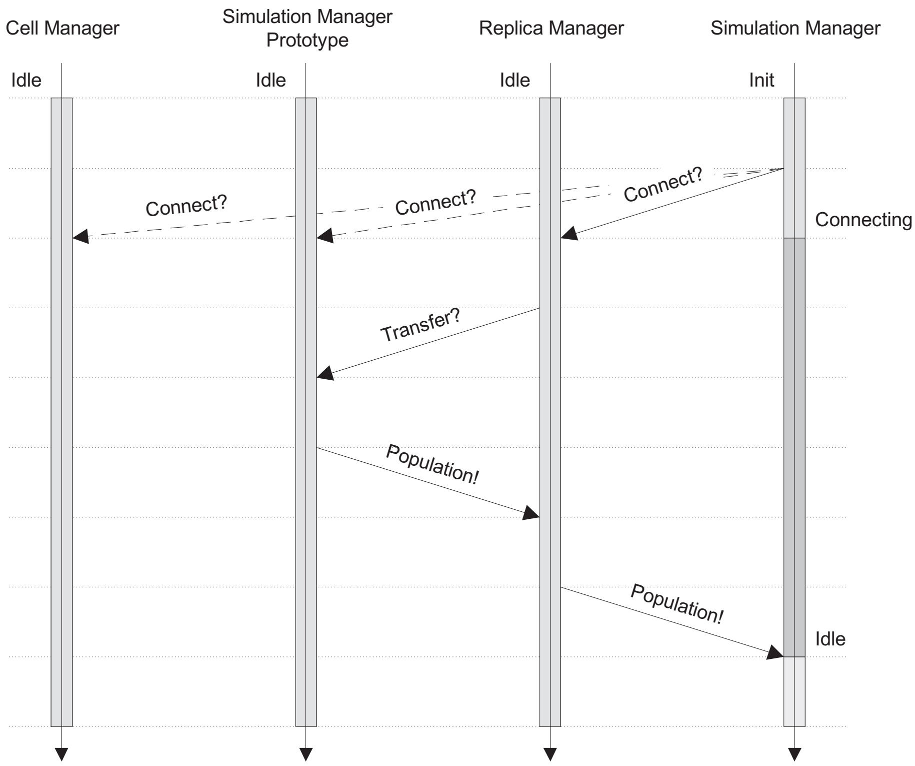

Figure 5.10: Adding a new user to our distributed simulation system.

- the synchronization of cells, i.e., replica managers.

The proceedings in both groups are depicted in Figure 5.12; the upper half of the diagram shows the unification of cells, whereas the lower half of the diagram shows how replica managers are synchronized. The unification of cells, i.e., the alternate transmission of virtual objects, involves several actions itemized in the list below. The example we give here is based on the use-case illustrated in Figure 4.5.

1. If a virtual object notices that its crossing of a cell boundary is imminent, it causes "its" prototypic simulation manager to send a `Cluster` message to the associated replica manager. The `Cluster` message contains information for the cell manager about which cells have to be unified.

2. The replica manager forwards the `Cluster` message it has received from the prototypic simulation manager to the cell manager. It does not add or drop information.

3. Arrived at the cell manager, the `Cluster` message causes the execution of two different algorithms:

   - The first algorithm determines which cells are to be replicated. The result of this algorithm is a list cells that have to be transmitted to the replica manager concerned.
   - The second algorithm determines the new clustering of all replica managers. The result of this algorithm is a list of *cluster supervisors* and *cluster subordinates*.

4. The cell manager initiates cell replication, i.e., it sends `Transmit` messages to the affected replica managers telling them where they have to send copies of their cells to.

5. Those replica managers, which have received the `Transmit` message, order their prototypic simulation managers to send their virtual objects to the given addressees. Again this is done by utilizing `Transmit` messages.

6. Now, the prototypic simulation managers transmit virtual objects to the specified destinations by using `Population` messages. This operation finishes the process of replicating cell contents, the first part of cell migration.

7. The second part of cell migration – the supervision of cell clusters, i.e., clusters of replica managers – is initiated by the cell manager by emitting `Supervise` messages. These messages contain instructions for replica managers to supply other replica managers with time, i.e., send `Time` messages not only to simulation managers but also to subordinate replica managers.

Figure 5.11 shows an example of how clusters are created and cluster supervisors are established. In Figure 5.11(a), there are two clusters:

$$\begin{aligned} C_A &= (c_1, c_2, c_5, c_6), \\ C_B &= (c_{10}, c_{11}). \end{aligned}$$

They exist because of the cell crossings of virtual object A and virtual object B. Both clusters are supervised by a replica manager: Cluster C_A is supervised by replica manager c_1 and cluster C_B is under supervision of replica manager c_{10}. To supervise a cell means to force to accept the time of another cell. In our example, c_1 stipulates the time of c_2, c_4, and c_5, whereas c_{10} stipulates the time of c_{11}. If it happens that a third virtual object crosses cell boundaries, the situation changes. Virtual object C associates cluster C_A and C_B. During the time of the migration of virtual object C, there is just one cluster $C_C = (c_1, c_2, c_5, c_6, c_{10}, c_{11})$.

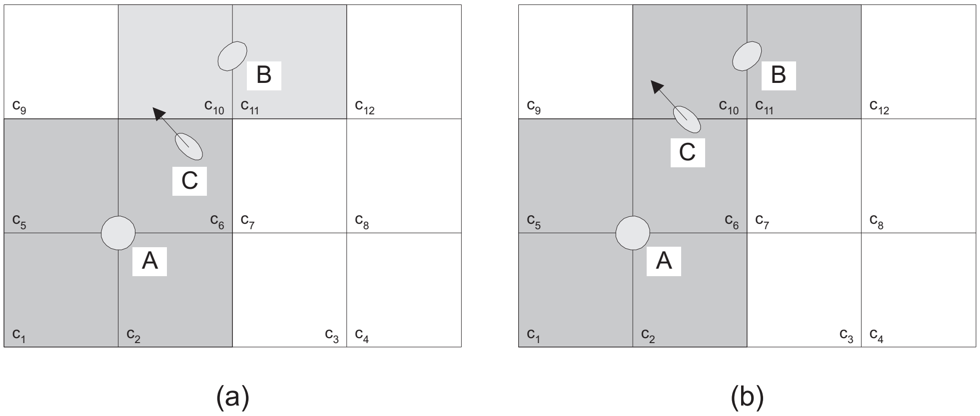

Figure 5.11: Supervising cell clusters.

From a communication's point of view, a `Supervise` message is sent to c_1 , indicating that this replica manager is now supervisor, and five `Supervise` messages are sent to c_2, c_5, c_6, c_{10}, and c_{11} indicating, that, from now on, they are subordinates.

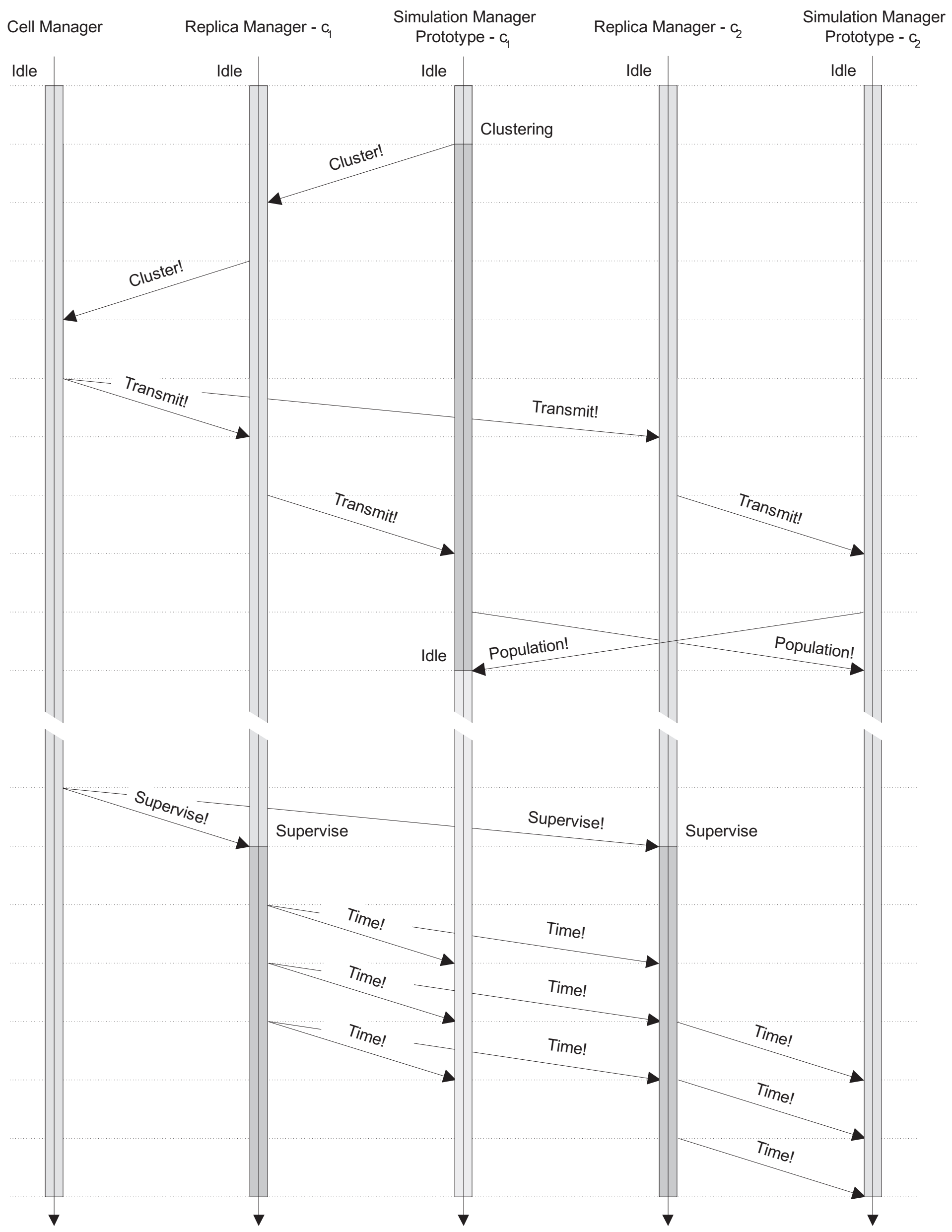

Figure 5.12: Migration of a virtual object.

Chapter 6

An Example: The "`Asteroids`" Game

In this chapter, the reader learns about our distributed simulation system's structure, functions, and features by creating an application for Unix-X11 and Win-32 called "`Asteroids`". Our distributed simulation system was realized within the framework of this thesis by developing an object-oriented application framework written in C++. And this application framework is the subject we are focusing on – by means of providing some sort of tutorial.

6.1 Basic Features

To get an idea of what the example application we give is about, a short description is provided.

The example application simulates an arbitrary section of space – subdivided into 25 cells arranged in a quadratic way – that is initially populated by a predefined number of virtual objects called *asteroids*. These virtual objects are activated upon system startup from a file called `World.dat`. Example 1 gives a model of how this file may look like. It is not necessary to create it by hand. It may be created by passivating a suitable model of the real world to be simulated.

In order to populate the scene with "intelligent virtual objects", it is possible to allow an arbitrary number of users – represented by *space ships* – to participate in the simulation. These users are in the position to interact and to make experiences

```
obj (int (1) str (12 "PopulationDT") siz (2)
   obj (int (1) str (10 "AsteroidDT")
      siz ( 0) int (-1) lng ( -1  ) int (  0) int (0)
      dbl ( 0) dbl ( 0) dbl (100  ) dbl (200) dbl (1)
      int (80) dbl ( 0) dbl (  0.1)
   )
   obj (int (1) str (10 "AsteroidDT")
      siz ( 0) int (-1) lng ( -1  ) int (  0) int (0)
      dbl ( 0) dbl ( 0) dbl (200  ) dbl (100) dbl (1)
      int (80) dbl ( 0) dbl ( -0.1)
   )
)
```

Example 1: The file structure of a virtual world to be activated.

within the virtual space simultaneously. The possible actions a space ship may perform are:

- turn left,
- turn right,
- accelerate,
- slow down, and
- fire *bullets*.

The ability to perform these actions is a special property of a space ship, thus this functionality has to be implemented in the class `ShipDT`. How this is done will be explained later on.

Figure 6.1 shows a screen shot of a simulation session as it took place at RISC-Linz participating three players (10 Mbit Ethernet LAN, two Unix-X11 workstations, one Windows NT 4 workstation). In this picture, a section of the virtual space as it is seen by player 2 is depicted. Player 2 launched several bullets onto a group of asteroids that, in turn, exploded and broke into parts[1].

[1] The ability of asteroids to explode and to break into parts when hit is specific behavior and has to be implemented in the class `AsteroidsDT`.

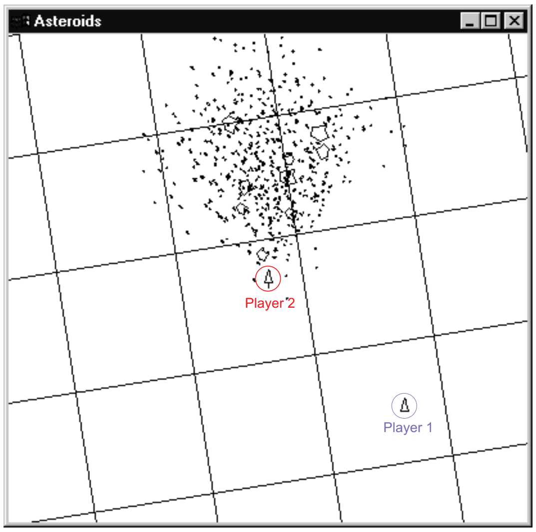

Figure 6.1: Screen shot of a simulation session (Win-32 terminal version).

6.1.1 Our Application Framework's Features

Our object-oriented application framework enables C++ programmers to write distributed applications conforming to the distributed simulation system described in the former chapters. It has the following features:

Manager Concept The concept of managers allows the individual adaptation of our application framework – on a modular level – to the individual needs of concrete distributed applications. Programmers are enabled to put emphasis on those "building blocks" their concrete distributed application needs.

Configuration Management Our application framework provides a mechanism that allows a network administrator to adjust the way how our distributed simulation system distributes processes among available computers – the system automatically keeps track of the entire distribution. Configuring the way how processes are allocated to computers is motivated by the need for balancing network and computational load.

Portability Our distributed application framework is capable of being executed simultaneously under two different operating systems within one simulation session: Unix-X11 and Win-32. C++ programmers are relieved of taking into account operating-system depending facts and peculiarities.

Graphical User Interface Two different graphical user interfaces – Unix-X11 and Win-32 – are supported. Both interfaces exploit double-buffering to ensure smooth animations. The user interface facilitates the interaction between a user and an application without bothering a C++ programmer with having to consider different window systems. Presently, graphical output is restricted to two dimensions. Extending the GUI to be capable of depicting three dimensional scenarios is imaginable and can be implemented easily by using graphic libraries, e.g. Open-GL.

TCP/IP Support Network communication – an essential functional part of our application framework – relies on the TCP/IP protocol (cf. [Tanenbaum, 1989] and [Santifaller, 1993]). TCP/IP was chosen for its guaranteeing a faultless information delivery.

MiniClass A class library named `MiniClass` was implemented in order to have a solid foundation upon which a large application framework can be built. It is designed to support a variety of application frameworks, i.e., its usage is *not* restricted to something similar to our concrete system (see Figure 6.2 and Figure A.1 for a class diagram). The class library `MiniClass` provides the following features offered by several methods in the root-class `ObjectMC`:

Meta-Information Meta-information is detailed information about all classes in an application framework. This information is available at runtime and comprises things like class-hierarchy information and class-attribute information. Meta-information is utilized by an application framework by exploiting special macros and methods.

Change Propagation The method of notifying a C++ object in the case of another object's state change is called change propagation. This mechanism is available to all heirs of the root-class `ObjectMC` and is implemented in an efficient and memory-saving way. Change propagation is utilized by an application framework by exploiting special methods.

Serialization It is possible to serialize, i.e., activate and passivate, all successors of the root-class `ObjectMC`. Serialization includes the dynamic instantiation of a class during runtime by just knowing its name. Examples of use are the passivation of C++ objects onto a file or the serialization of a C++ object across some network connection. A special quality of our method of activating objects is the instantiation by calling the object's default constructor. Other, exemplary and well-known, implementations like ET++ (cf. [Gamma, 1992]) lack of calling the default constructor when activating objects.

Cloning Cloning C++ objects is the process of creating identical copies of them. In theory there exist two kinds of cloning: deep and shallow.

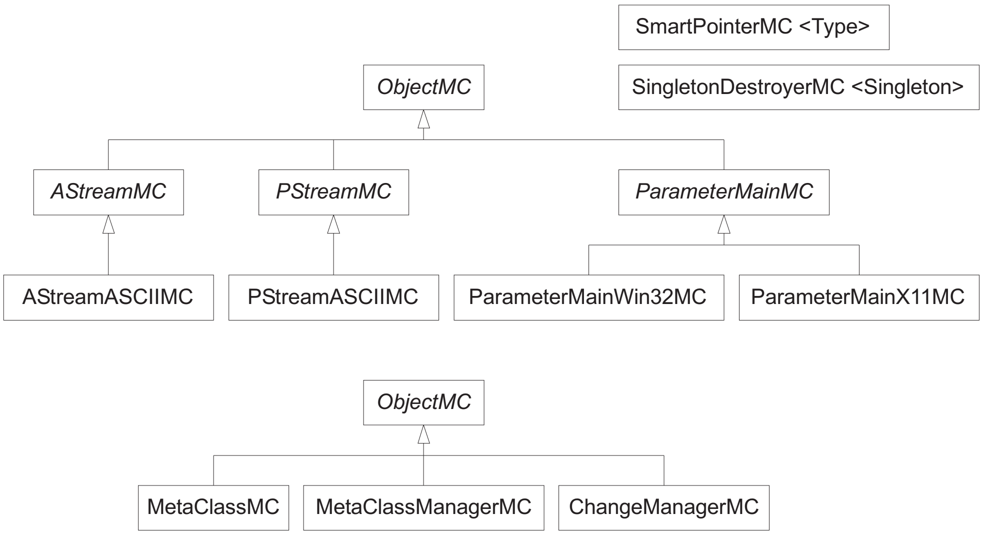

Figure 6.2: Schematic class diagram of `MiniClass`.

The method we have implemented is the creation of identical copies by performing a deep-clone operation.

Type-Safe Casts The class library `MiniClass` enables C++ programmers to cast pointers to objects in a type-safe manner. This means that runtime information is used to assure correct class-hierarchy relations prior to performing a type cast. Various `static` methods regarding type-safe casts can be used by C++ programmers.

Smart Pointers The concept of smart pointers prevents the programmer from running into serious problems when she/he misses to delete dynamically allocated C++ objects. Using smart pointers reduces program failures caused by memory violations, although it cannot replace garbage collection.

Standard Template Library Our class library fully supports the use of the Standard Template Library (STL) (cf. [Musser and Saini, 1996]) together with several other classes and libraries related to the STL, e.g. the class `string`.

6.1.2 Hook Methods

Our application framework provides anything a C++ programmer needs to write a full functioning distributed application equipped with the characteristics stated in the list above. As it is common with application frameworks, an experienced C++ programmer obtains the desired appearance, behavior, and functionality of a concrete application by deriving new classes and by overriding appropriate methods.

Methods intended to be overridden play a major role within our application framework, and thus have a special designation used throughout this chapter: they are called *hook methods* (cf. [Gamma *et al.*, 1995]). Example 2 sketches out the principle of how hook methods work.

```
class Base {
   public:
      void TemplateMethod () {
         HookMethod ();
      }

   protected:
      virtual void HookMethod () {
//       Default behavior.
      }
};

class Derived : public Base {
   private:
      void HookMethod () {
//       Modified behavior.
      }
};
```

Example 2: Changing behavior by means of hook methods.

The default behavior of method `HookMethod` in class `Base` is modified by a method with the same name in the class `Derived`. If this construct occurs in our application framework, `Base` will represent some basic framework functionality supplied by us, whereas `Derived` will be supplied by an application programmer. Consider that method `TemplateMethod` in class `Base` is the only method that can be invoked from “outside”: The interface and the sequence of method invocations in `TemplateMethod` are not changed, although the behavior definitely is.

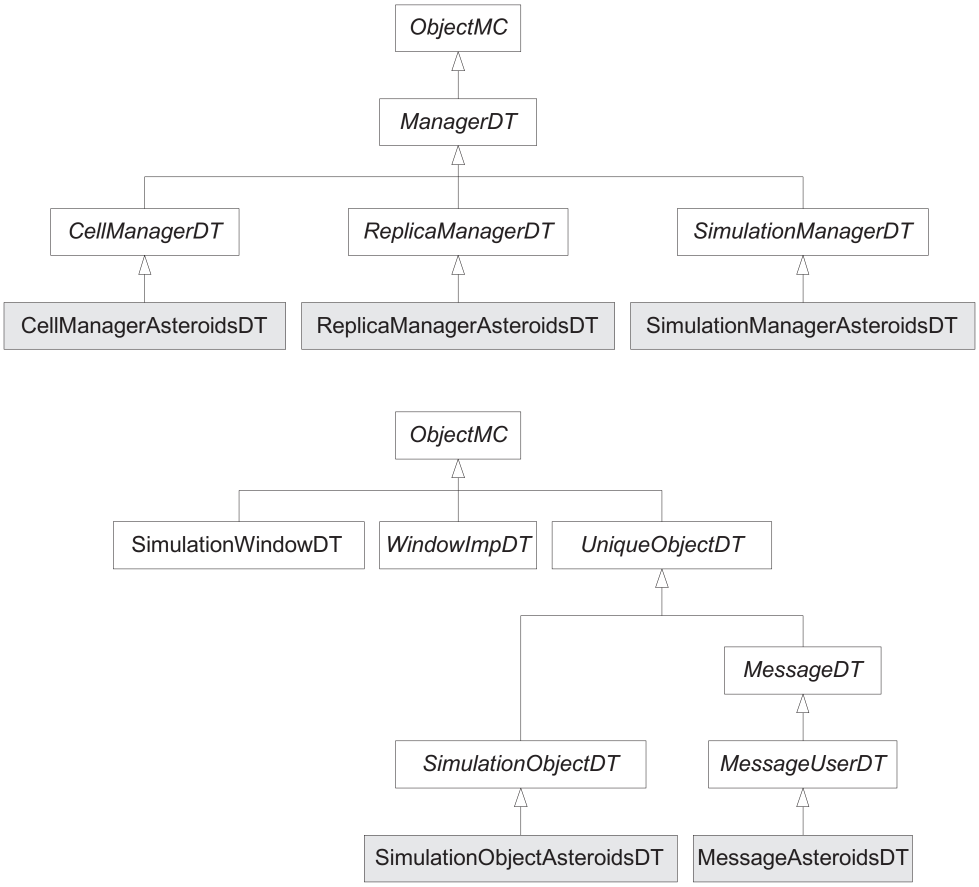

Figure 6.3: Schema of our framework pointing out where to place new code.

6.1.3 Our Application Framework's Class Hierarchy

Because our application framework provides so much standard functionality, it is easy to write distributed applications. At the same time, its flexibility and extendibility does not lock a programmer into our system's conformance, although deviating from the "standard" may take a little more work. Figure 6.3 shows very roughly the class hierarchy of our application framework. It gives a clue of where to put new code in order to change behavior. Shaded rectangles in Figure 6.3 denote classes to be supplied by a programmer. These classes can be subdivided into three categories:

- The first category is made up of three abstract manager classes:

  - `CellManagerDT`,
  - `ReplicaManagerDT`, and
  - `SimulationManagerDT`.

- The second category consists of an abstract class representing the base class of any virtual object: `SimulationObjectDT`.
- The third category of classes intended to be overridden is the abstract base class for all messages: `MessageDT`.

The most important category for programmers to write a distributed application is the category of virtual objects. Manager classes are to be altered seldom and message classes are to be altered almost never. This is obvious, since virtual objects are the major reason for performing distributed simulations. Everything that happens in a virtual world is based on these entities; managers and messages support them in a very basic way – it makes not much sense to manipulate them, although the need for flexibility requires it.

Our example application "`Asteroids`" overrides the manager classes and the virtual-object classes. It does not modify the message classes.

6.2 Customizing Managers

Figure 6.4 shows schematically the hierarchy of manager classes – it does not give a detailed description of the manager classes' interface. We refer to Figure A.2 instead. Further more, programmers are recommended to go through the source code available at [Kulczycki, 1997]. Shaded rectangles in Figure 6.4 denote classes not part of our application framework. They are supplied by us in order to implement the application example "`Asteroids`".

For implementing the application example "`Asteroids`", it is not necessary to alter the default behavior of the classes `CellManagerDT` and `ReplicaManagerDT`. The only manager class to be overridden and modified in order to achieve the desired behavior is the class `SimulationManagerDT`.

Since it is impossible for our application framework to know anything about its later use, it has to provide a mechanism that allows application programmers to specify which observer object to create. So, our application framework offers the `protected` method `DoCreateObserver`. This method is invoked whenever the framework intends to create an observer object. Example 3 gives a sample of how this may look like.

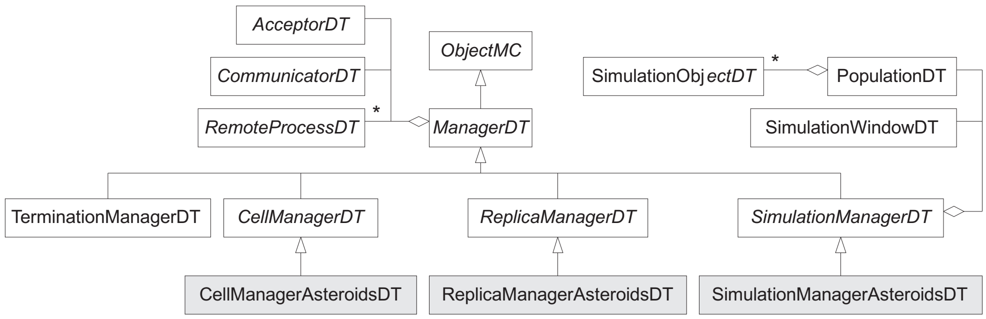

Figure 6.4: Schematic class diagram of the manager classes.

```
SimulationObjectDT *
SimulationManagerAsteroidsDT::DoCreateObserver () const {

// Overridden Hook Method
// ----------------------
   return new ShipDT;
}
```

Example 3: Creating an observer object via an overridden hook method.

As stated above, the representation of a user inside the virtual space is an instance of the class `ShipDT`[2]. The method `DoCreateObserver` – it is responsible for creating observers – is a typical example of a hook method. It is not necessary for the programmer of this overridden method to implement any defensive program construct, e.g. `assert`. Everything is checked by the calling template method inside our application framework. If, for instance, the execution of the command `new ShipDT` fails, it is up to the framework to "assert".

6.3 Customizing Virtual Objects

On the contrary to manager classes virtual-object classes are much more flexible – and thus much more powerful – in allowing programmers to alter the behavior of a distributed simulation system.

[2]How to create ships and other virtual objects flying around in our virtual space is explained in the following.

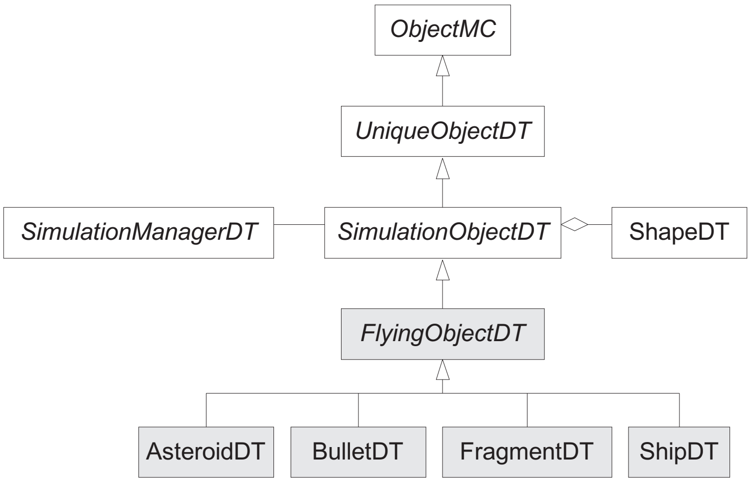

Figure 6.5: Schematic class diagram of the virtual-object classes.

Figure 6.5 shows schematically the hierarchy of virtual-object classes – it does not give a detailed description of the virtual-objects' interface. We refer to Figure A.3 instead. Furthermore, programmers are recommended to go through the source code available at [Kulczycki, 1997]. Shaded rectangles in Figure 6.5 denote classes not part of our application framework. They are supplied by us in order to implement the application example "`Asteroids`".

The following sections explain how to add code to the class `SimulationObjectDT` – by deriving new classes and by overriding `virtual` methods – in order to adapt our framework to achieve the demands made by the application example. Code fragments are given when it is necessary to support the applied techniques. Programmers interested in our complete C++ source code regarding the application example are again recommended to go through the source code available at [Kulczycki, 1997].

6.3.1 "Flying" Virtual Objects

Since the virtual space in our example application is populated by *flying virtual objects*, we introduce the abstract class `FlyingObjectDT` and derive it from the abstract class `SimulationObjectDT`. The class `FlyingObjectDT` will serve as a base class for any concrete virtual object that "flies" around in our virtual space.

There exists just one property characteristic of all concrete virtual objects: the property to react upon collisions by generating a number of *fragments*. So we have to override the method `OnCollision`, since this method is called for a virtual object when it collides with a "colleague". Example 4 shows how this is implemented.

```
01   void
02   FlyingObjectDT::OnCollision (SimulationObjectDT const & SimObj) {
03
04   // Overridden Hook Method
05   // ----------------------
06      Inherited::OnCollision (SimObj);
07
08      if (IsPrototype ()) {
09         AskForCreation (new FragmentDT (i_Position, ...));
10         AskForCreation (new FragmentDT (i_Position, ...));
11         AskForCreation (new FragmentDT (i_Position, ...));
12      }
13   }
```

Example 4: Creating fragments upon object collisions.

The argument `SimObj` of the method `OnCollision` represents the opponent a virtual object collided with. In lines 09–11, three fragments, i.e., three initialized, temporary instances of the class `FragmentDT`, are created. These new C++ objects are passed as arguments to the method `AskForCreation`, which – as its name suggests – asks (via message passing) the corresponding replica manager for a definitive, synchronous creation of a C++ object of the desired class in all replicas of the affected cell (see Figure 6.6). It is possible for a virtual object to react upon the answer from a replica manager by overriding the method `OnCreate`. Since fragments are very simple virtual objects, it is not necessary to do so.

6.3.2 Fragments, Asteroids, and Bullets

In addition to flying virtual objects, a fragment has two more properties stipulated by our application example:

- it is mortal and therefore grows older, and
- it has a shape.

Example 5 shows how to vest fragments with a limited lifespan. The instance variable `i_Age` in line 07 inherited from `SimulationObjectDT` denotes the actual age

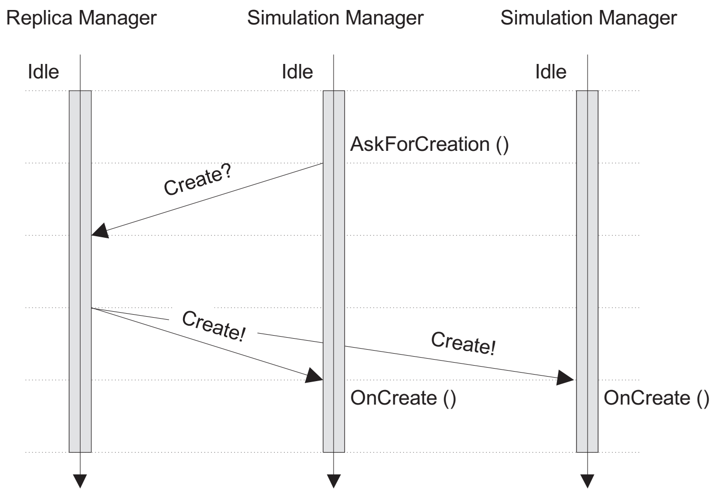

Figure 6.6: Teamwork of `AskForCreation/OnCreate`: Creating a virtual object synchronously.

of a virtual object. If the value of this instance variable exceeds `AgeFragment`, the fragment commits suicide by calling the inherited method `Die` (line 08). A perfect place to perform time dependencies like aging is the hook method `DoTime`, since it is called whenever a simulation manager receives a `Time` message from its replica manager.

```
01   void FragmentDT::DoTime () {
02
03   // Overridden Hook Method
04   // ----------------------
05      Inherited::DoTime ();
06
07      if (i_Age > AgeFragment) {
08         Die ();
09      }
10   }
```

Example 5: The aging of fragments.

To give a fragment a body – and therefore make it visible to observers – we have to initialize the instance variable `i_Body` inherited from `SimulationObjectDT`. How this is done is shown in Example 6. Lines 05–08 append two-dimensional points to the body of a fragment resulting in a closed 2D-polygon. A polygon representing a fragment's shape has to be specified in local coordinates – these coordinates are adjusted later on, when certain view transformations involving matrix operations are applied.

Line 10 initializes a fragment's radius in order to enable effective collision checks to be performed collectively for all virtual objects by a simulation manager.

```
01   void FragmentDT::PartInitialize (...) {
02
03   // Part Method
04   // -----------
05      i_Body.Add ( 1.0,  1.0);
06      i_Body.Add (-1.0,  1.0);
07      i_Body.Add (-1.0, -1.0);
08      i_Body.Add ( 1.0, -1.0);
09
10      i_Body.SetRadius (RadiusFragment * i_Scale);
11   }
```

Example 6: Vesting a fragment with a body.

Similar to fragments, the behavior of asteroids is quite straightforward, and therefore relatively simple to be implemented. Asteroids differ from fragments in that they do not grow older, and in that they break into parts upon collisions. Example 7 shows how the "crumbling" of an asteroid is implemented.

In line 05, an asteroid commits suicide by calling the inherited method `Die` in order to "rise from the dead" in threefold copy immediately (line 09–11). Again, the creation of a C++ object is announced by calling the method `AskForCreation` in line 10 in order to synchronize the creation in all replicas (schematically illustrated in Figure 6.6). One annotation has to be made to the conditional construct in line 08: The inherited instance variable `i_Strength` denotes the "stability" of a virtual object and ranges from 0 to 100. Now, if an asteroid's stability drops below the threshold value of 25, it is no longer capable of creating new asteroids – it dies definitely.

6.3.3 Space Ships

Space ships constitute "intelligent objects" as they are described in Section 3.3.1. As a consequence, they demand increased effort – compared to the other virtual objects

```
void AsteroidDT::OnCollision (SimulationObjectDT const & SimObj) {

// Overridden Hook Method
// ----------------------
   Inherited::OnCollision (SimObj); Die ();

   if (IsPrototype ()) {
      if (i_Strength > 25) {
         for (int i = 0; i < 2; i++) {
            AskForCreation (new AsteroidDT (...));
         }
      }
   }
}
```

Example 7: The "crumbling" of an asteroid.

– when implementing them by employing our application framework. A space ship, it is represented by the class `ShipDT`, comprises the following characteristics:

- it is sensitive to collisions and therefore mortal,
- it has a shape,
- it can be navigated by a user through virtual space, and
- it may fire, i.e., create bullets.

To implement a space ship to die upon collisions is straight forward. The procedure to achieve this has already been shown above. Example 8 is just given for completeness.

The same is true with a space ship's body (Example 9).

The situation changes when interaction comes into play: Interaction is a quality completely new to virtual objects in our application example. The central method in achieving interaction is the `virtual` method `DoKey` inherited from the class `SimulationObjectDT`. This method is called whenever the user presses a key on her/his keyboard. To start with, we first give a skeleton of how we implemented the overridden method `DoKey` in Example 10.

```
void ShipDT::OnCollision (SimulationObjectDT const & SimObj) {

// Overridden Hook Method
// ----------------------
   Inherited::OnCollision (SimObj); Die ();
}
```

Example 8: The behavior of a space ship upon collision.

```
void ShipDT::PartInitialize () {

// Part Method
// -----------
   i_Strength = StrengthShip;

   i_Body.Add ( 20.0,   0.0);
   i_Body.Add (-10.0,  10.0);
   i_Body.Add (-10.0, -10.0);

   i_Body.SetRadius (RadiusShip * i_Scale);
}
```

Example 9: Vesting a space ship with a body.

When a user presses the button `KeyFire` on her/his keyboard, an initialized, temporary instance of the class `BulletDT` is created (see line 13). This newly created C++ object is passed as an argument to the method `AskForCreation`. On the contrary to the class `FlyingObjectDT`, which did not react upon the answer of the creation request, the class `ShipDT` has to. It has to react appropriately by overriding the method `OnCreate`, since the new bullet has to be initialized with the space ship's *current* position.

We want to place emphasis upon this point, since in our opinion it is very fundamental in understanding the way how our distributed simulation system works.

So, why is it *not* allowed to pass a fully initialized instance of the class `BulletDT` to the method `AskForCreation` and rely on our application framework to distribute it correctly among any affected replicas? Of course, it would do so, but the position of the bullet makes troubles.

```
01  void ShipDT::DoKey (int const Key) {
02
03  // Overridden Hook Method
04  // ----------------------
05     Inherited::DoKey (Key);
06
07     if (i_IsWatched && !i_PendingUpdate) {
08        switch (Key) {
09           case KeyLeft:  AskForUpdate   (...);           break;
10           case KeyRight: AskForUpdate   (...);           break;
11           case KeyUp:    AskForUpdate   (...);           break;
12           case KeyDown:  AskForUpdate   (...);           break;
13           case KeyFire:  AskForCreation (new BulletDT); break;
14          }
15       }
16  }
```

Example 10: Interacting with a space ship.

When performing the command `new BulletDT`, a space ship initializes the new bullet's position with its current position in space. But during the transmission of the `Create` request and the re-transmission of the `Create` reply, the space ship moves (see Figure 6.6)! If we allow the application framework to instantiate the bullet without intervention, it would be created on the wrong, i.e., old position. To prevent this, we have to adjust the new bullet's position to the space ship's current location. Example 11, lines 01–10, show how this has to be implemented.

Line 07 gives an example of how to use meta-information provided by the class library `MiniClass`. The method `IsA` verifies the relationship between two given classes to agree with the *is-a* relation.

Creating a bullet as a response to a key-press on a keyboard is one aspect of user interaction. The modification of specific attributes of a virtual object, e.g. speed or the direction of movement is the other aspect. How certain space ship's attributes are modified is depicted in Example 11, lines 12–19. Let us focus on how a space ship accelerates.

When a user presses the button `KeyUp` on her/his keyboard (Example 10, line 11), the method `AskForUpdate` is invoked, passing a set of parameters describing the new desired attributes. This causes the simulation manager to transfer an `Update` request to its associated replica manager that, in turn, answers with an `Update` reply to all affected replicas (schematically illustrated in Figure 6.7).

```
void ShipDT::OnCreate (SimulationObjectDT & SimObj) {

// Overridden Hook Method
// ----------------------
   Inherited::OnCreate (SimObj);

   if (SimObj.IsA ("BulletDT")) {
//    Initialize some of the bullet's attributes.
   }
}

void ShipDT::OnUpdate (...) {

// Overridden Hook Method
// ----------------------
   Inherited::OnUpdate (...);

// Update some of the ship's attributes.
}
```

Example 11: The methods `OnCreate` and `OnUpdate`.

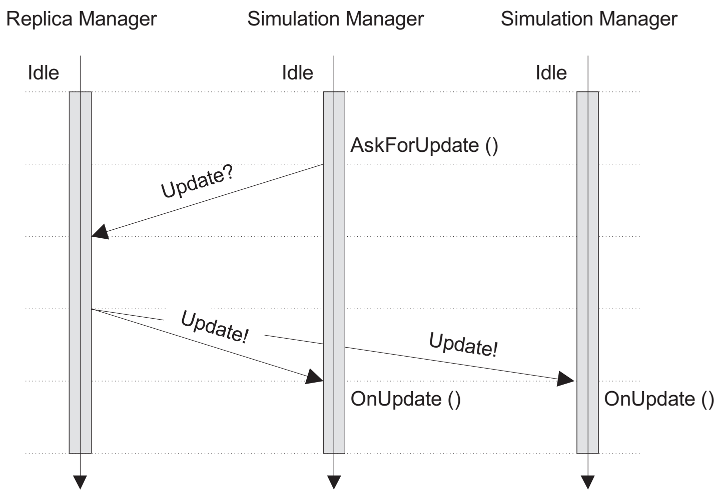

Figure 6.7: Teamwork of `AskForUpdate`/`OnUpdate`: Updating a virtual object synchronously.

The receipt of the answer message causes the inquiring virtual object's method `OnUpdate` to be invoked. Since this method is called in any replica, global consistence of the virtual space is assured (see Example 11, line 18).

6.4 Implementation Status

To complete the description (the tutorial) of how to write an application based on our framework, we want to remark on some software characteristics in order to illustrate the current implementation status. This information will be of use to programmers in allowing them to assess the expenditure and "risk" when relying on our framework.

The entire software which has been developed within the framework of this thesis – interested programmers are referred to [Kulczycki, 1997] – consists of the three components:

- class library `MiniClass`,
- object-oriented distributed application framework, and
- application example "`Asteroids`".

It comprises the following four characteristics:

- 68 classes,
- 145 files,
- approximately 25,200 lines of code exclusively written in C++, and
- approximately 622,000 Bytes of code altogether.

Table 6.1 shows how these characteristics are shared between the individual software components.

The functional basis – as far as communication via some network is concerned – is based on the public-domain C library *Parallel Virtual Machine*, commonly abbreviated as PVM (cf. [Geist *et al.*, 1994a] and [Geist *et al.*, 1994b]). PVM[3] is a message

[3] Available at `http://www.epm.ornl.gov/pvm/pvm_home.html`.

Software Component	**Classes**	**Files**	**LOC**	**Size (in Bytes)**
Class Library `MiniClass`	13	33	5,500	127,000
Application Framework	47	94	17,000	430,000
Example "`Asteroids`"	8	18	2,700	65,000
Total	68	145	25,200	622,000

Table 6.1: Software components by numbers.

passing system that enables a network of Unix and Windows NT computers to be used as a single distributed-memory parallel computer. This network is referred to as the *virtual machine*. PVM can be run in three different modes:

Transparent Mode At this highest level, tasks are automatically executed on the most appropriate computer.

Architecture-Dependent Mode In the architecture-dependent mode, the user specifies which type of computer is to run a particular task.

Low-Level Mode In low-level mode, the user may specify a particular computer to execute a task.

Functionality	**PVM Functions**
Buffer Management	`pvm_bufinfo`, `pvm_freebuf`, `pvm_getrbuf`, `pvm_getsbuf`, `pvm_mkbuf`, `pvm_setrbuf`, `pvm_setsbuf`, `pvm_upkstr`
Group Communication	`pvm_gsize`, `pvm_joingroup`, `pvm_lvgroup`
Initialization	`pvm_mytid`
Send & Receive	`pvm_bcast`, `pvm_probe`, `pvm_recv`, `pvm_send`
Task Management	`pvm_kill`, `pvm_spawn`

Table 6.2: Utilized PVM functions.

In all of these modes, PVM takes care of necessary data conversions from computer to computer (XDR) as well as low-level communication issues (TCP/IP). Our application framework is driving PVM in low-level mode. XDR and TCP/IP are those functionalities offered by PVM we were taking advantage of when encapsulating them in our communication and process classes (class `PVMCommunicatorDT` and class `PVMProcessDT`). All PVM functions exploited are summarized in Table 6.2.

Software Component	**Maturity Level**
Class Library `MiniClass`	Fully implemented and tested.
Application Framework	Fully implemented and tested. Some problems regarding the unification of cells are not removed at the moment.
Example "`Asteroids`"	Fully implemented and tested.

Table 6.3: The software components' levels of maturities.

Table 6.3 lists the levels of maturities of all individual software components of the entire software. Due to the limited amount of time, it was not possible for the authors to eliminate some minor flaws from the application framework.

Chapter 7

Conclusions

7.1 Results

The entire software consists of fourteen different functional class groups. Functional class groups and their affiliations to the three existing software components are shown in Table 7.1.

Meta-information, in general, is the information about classes. This information is available during runtime and can be used in various different ways. Several things, for instance obtaining a class's name or a class's size are quite straight forward. Retrieving information concerning a class's position in a class hierarchy, e.g. inquiring base classes, is more complex. Real power lies, for instance, in the ability to deep-clone objects or to allocate class instances dynamically – just by specifying a class name via a string.

Activation and *passivation*, so-called serialization, extend meta-information beyond dynamic object instantiation in that they expand this principle to be capable of working with streams of any kind. Having this possibility available in larger application frameworks is extremely valuable, especially in conjunction with disk I/O and networking.

Communication classes facilitate the transmission of information between processes. They enable programmers to send and receive any instance of a C++ class derived from the root class `ObjectMC`. In conjunction with the process classes, they ease interprocess communication tremendously. A *process object* acts as a proxy for any "real" and possibly distant running process in a distributed system. These objects allow programmers to "remote control" tasks on different computers in a network,

Software Component	**Class Groups**
Class Library `MiniClass`	Activation Classes
	Change-Propagation Classes
	Meta-Information Classes
	Passivation Classes
Application Framework	Acceptor Classes
	Communication Classes
	Event Classes
	Manager Classes
	Message Classes
	Process Classes
	Virtual-Object Classes
	Window Classes
Example "`Asteroids`"	Manager Classes
	Virtual-Object (Ship) Classes

Table 7.1: Software components and class groups.

i.e., to launch and terminate them. Communication classes and process classes cooperate very tightly.

Manager classes represent the supporting structural skeleton of our application framework. There is a one-to-one relationship between manager classes and processes. These classes, i.e., processes, are programmed in a portable way so that it is possible to execute them on Unix as well as on Windows NT workstations. *Acceptor classes* provide a way to vest managers with distinct states.

Virtual-object classes allow entities existing in a real world to be mapped into the virtual space of a simulation system. Several forms of virtual-object classes are available, covering various different behaviors.

During the entire process of developing our software, i.e., while designing, implementing, and testing, we attached great importance on several major issues:

- The use of new standards in software engineering covering all phases of software development as they are taught in the Department of Software Engineering of the Upper Austrian Polytechnic University (cf. [Mayr, 1997a] and [Pomberger and Blaschek, 1996]).

- The employment of object-oriented analysis paradigms, object-oriented design paradigms, and object-oriented programming paradigms as they appear in [Coad and Yourdon, 1991a], [Coad and Yourdon, 1991b], [Coad and Nicola, 1991], [Blaschek, 1994], and [Pree, 1995]. Emphasis is put on object-oriented design patterns (cf. [Gamma *et al.*, 1995]).
- The use of standards in general, e.g. the use of the *Unified Modeling Language* UML (cf. [Rational, 1997a], [Rational, 1997b], [Rational, 1997c], [Rational, 1997d], and [Rational, 1997e]) and the use of the *Standard Template Library* STL (cf. [Musser and Saini, 1996]) – just to mention two of them.
- The observance of several "trivial" rules when programming in C++. For example the use of programming guidelines as described in [Henricson and Nyquist, 1992] or the demand for a defensive style of programming.

We suppose that in consequently applying these few major guidelines, the everlasting demand for quality in software development can be met accordingly.

7.2 Perspectives

The distributed simulation system described and documented in this thesis is just a tiny contribution to the tremendously large scientific field of distributed computing. Chiefly, it served to introduce the *authors* to distributed technologies and to distributed programming. It is our intention to gain deeper insight, especially into the following topics:

- communication structures,
- distributed algorithms in general,
- distributed programming in general,
- distribution with respect to object-orientation,
- distributed design patterns,
- fault tolerance in distributed systems,
- distributed software engineering, and
- distributed debugging and testing.

Some of these topics can be applied when improving our distributed simulation system, whereas for others, our distributed software system serves as an excellent starting point.

Appendix A

Class Diagrams

This chapter summarizes the class diagrams of all functional class groups of our distributed application framework. It includes all class diagrams of the example application "`Asteroids`" as well as diagrams of all "stand-alone" classes, i.e., classes not derived from the root class `ObjectMC`. Here, functional class groups are[1]:

- the class library `MiniClass`,
- manager classes,
- virtual-object classes,
- window classes,
- event classes,
- communication and process classes,
- message classes,
- acceptor classes,
- "`Asteroids`" classes, and
- miscellaneous classes.

The lists of methods and instance variables appearing within the context of the UML class diagrams on the following pages do *not* claim for completeness. There

[1] Chapter 7 subdivides them in a slightly different way.

are just items necessary to illustrate the class functionalities and relations stated. Programmers interested in more details are referred to the source code available at [Kulczycki, 1997].

A.1 Class Library `MiniClass`

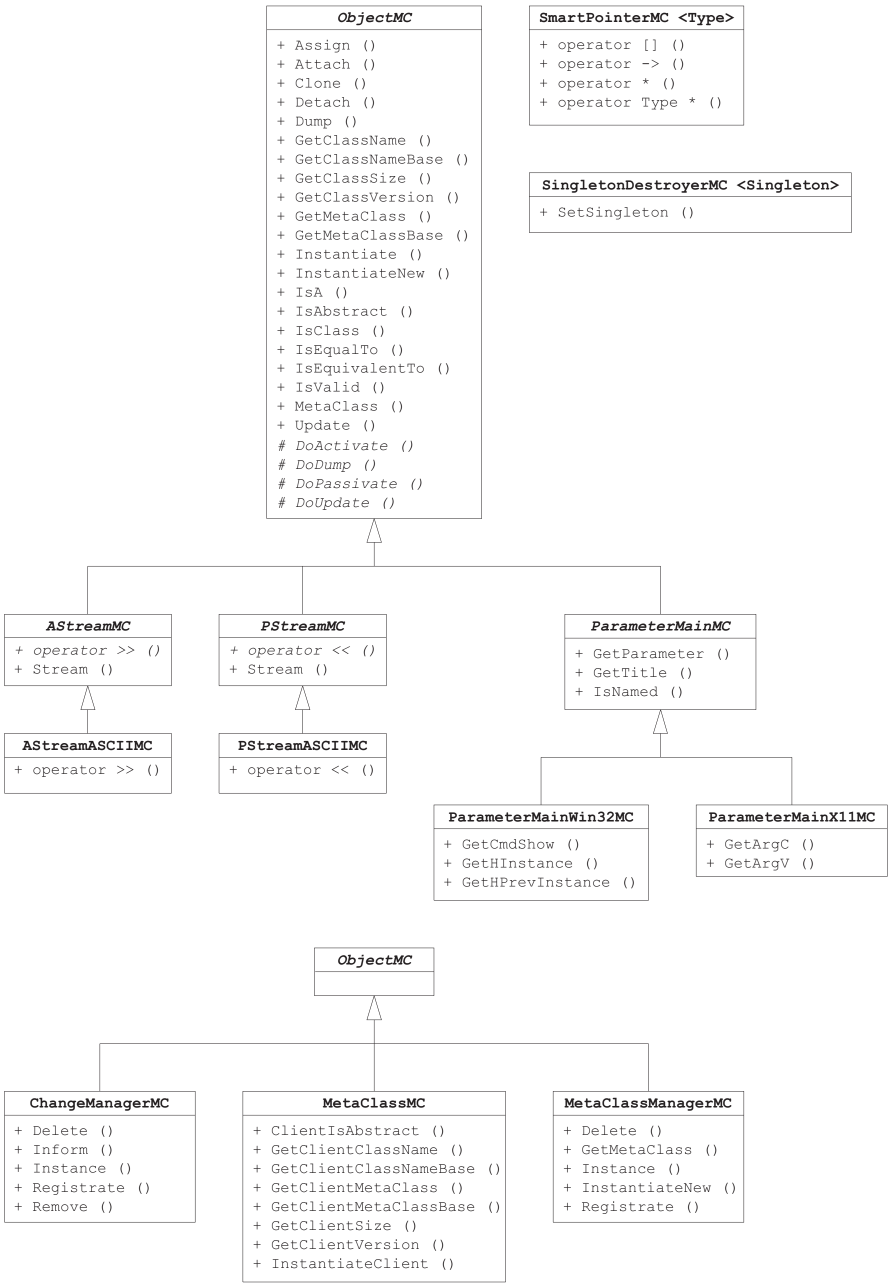

Figure A.1: Class diagram of `MiniClass`.

A.2 Manager Classes

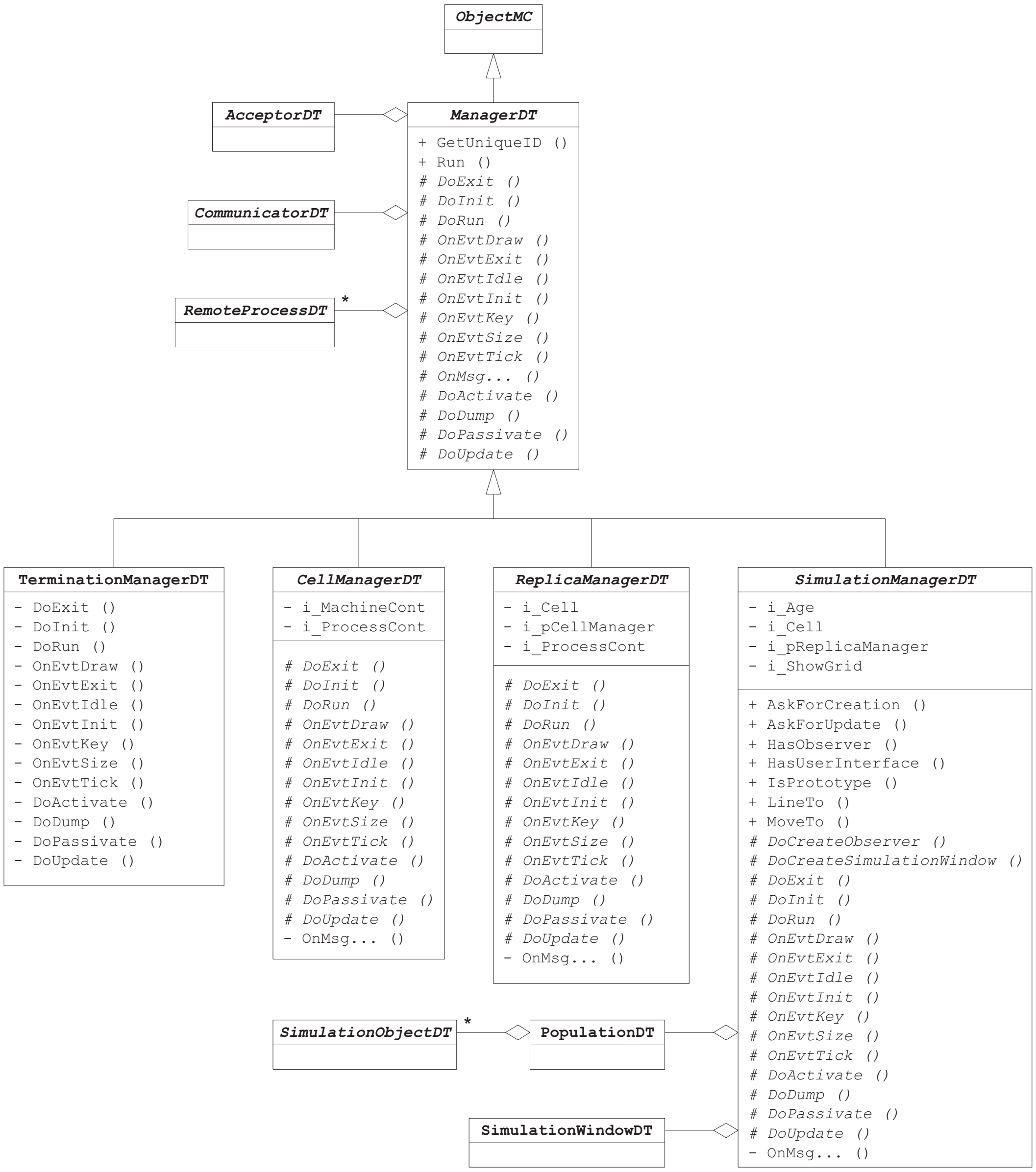

Figure A.2: Class diagram of the manager classes.

A.3 Virtual-Object Classes

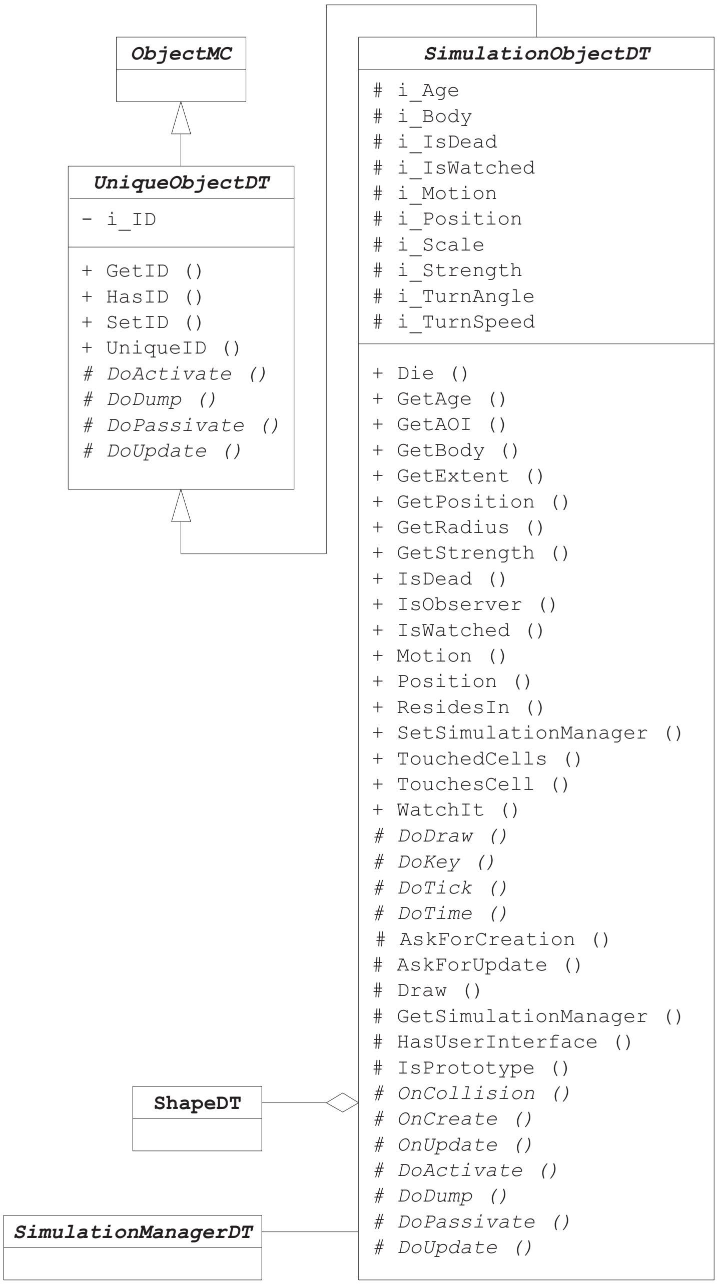

Figure A.3: Class diagram of the virtual-object classes.

A.4 Window Classes

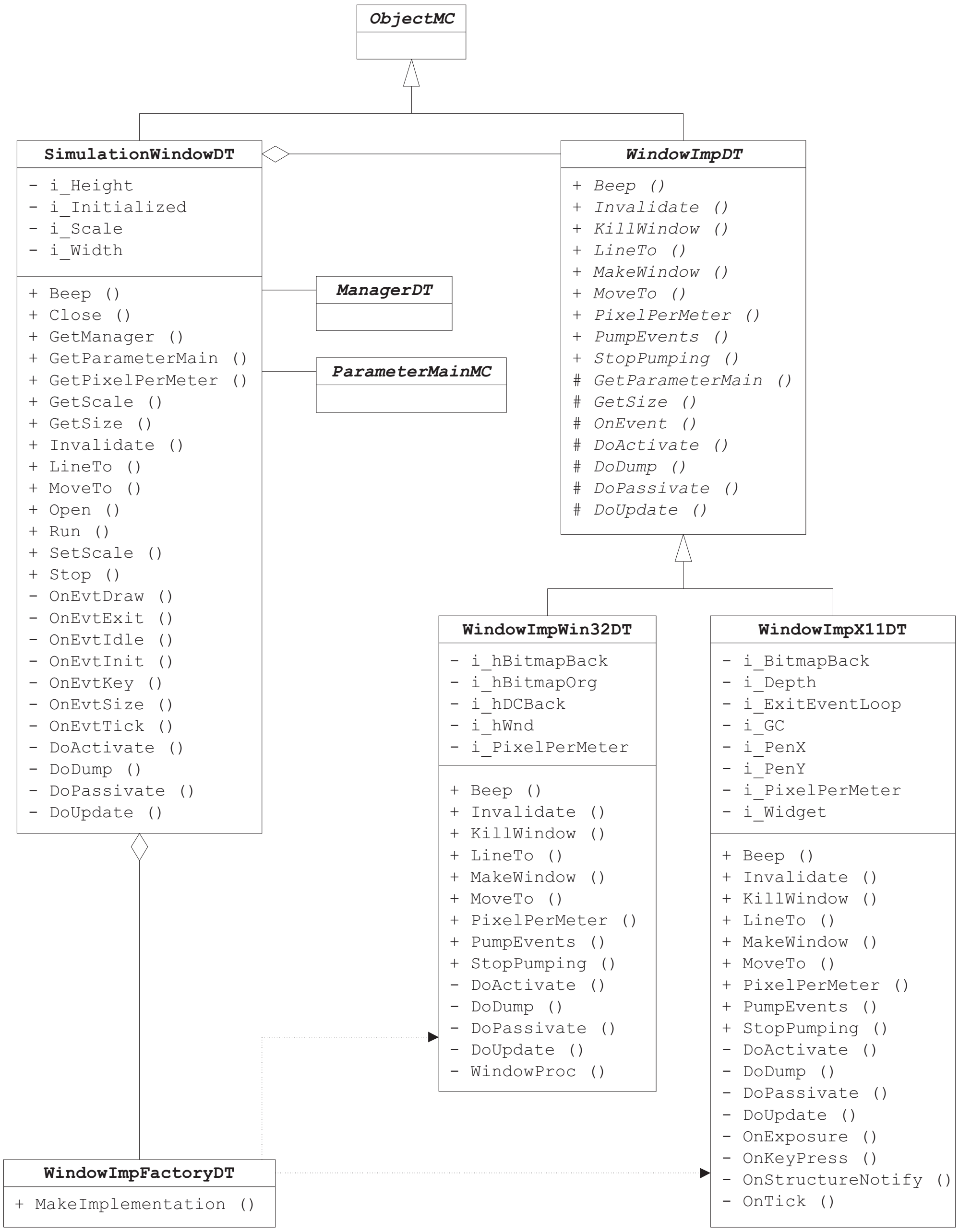

Figure A.4: Class diagram of the window classes.

A.5 Event Classes

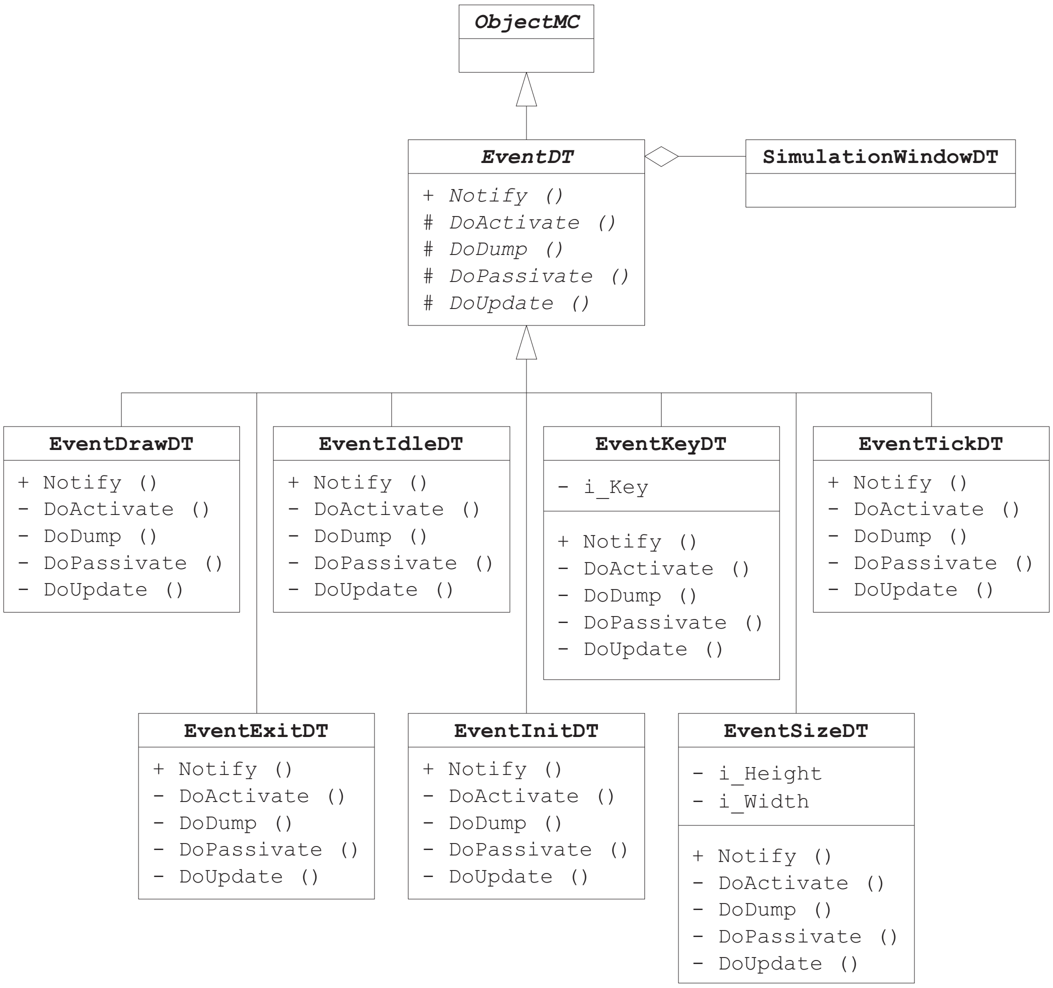

Figure A.5: Class diagram of the event classes.

A.6 Communication and Process Classes

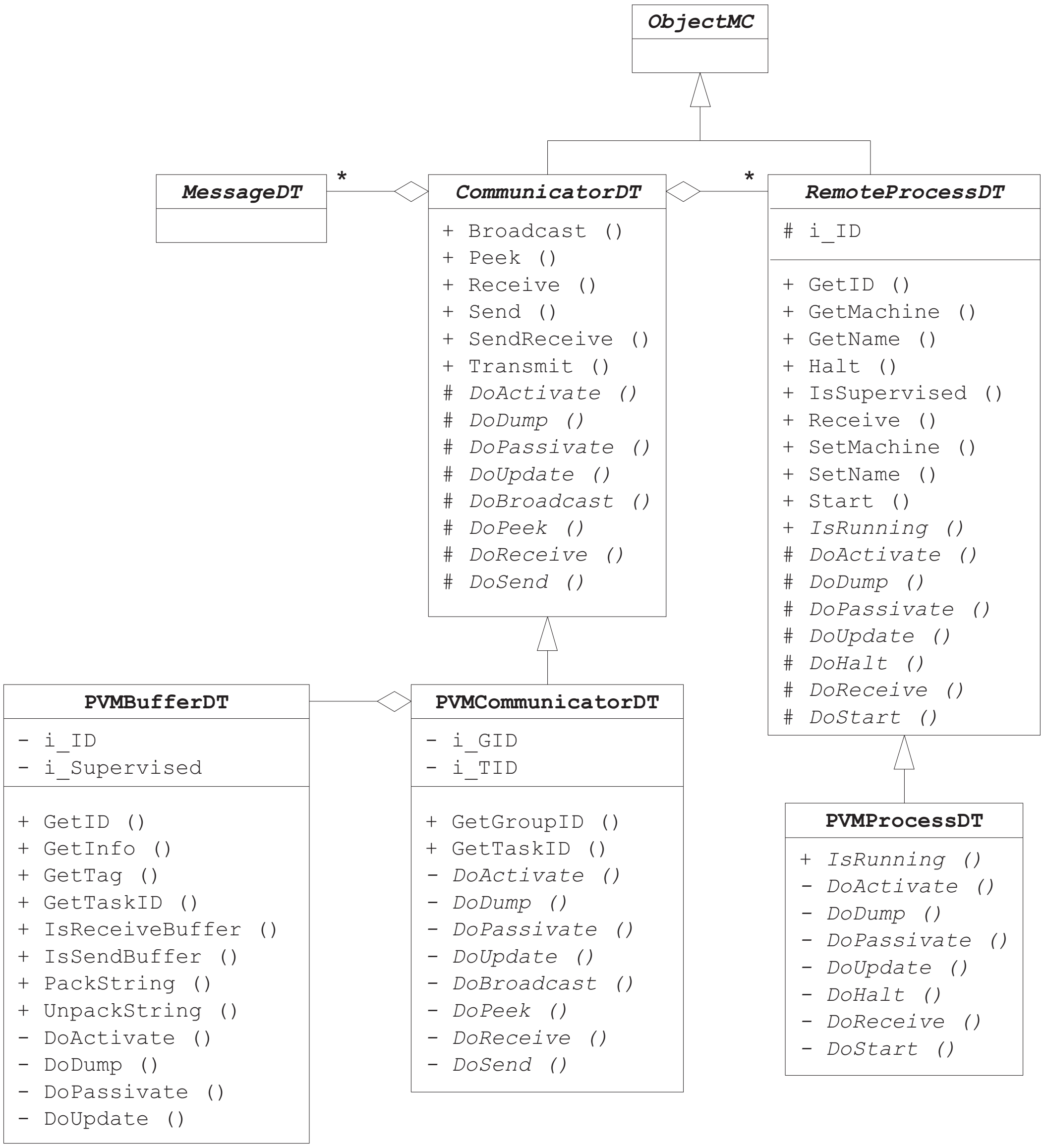

Figure A.6: Class diagram of the communication and process classes.

A.7 Message Classes

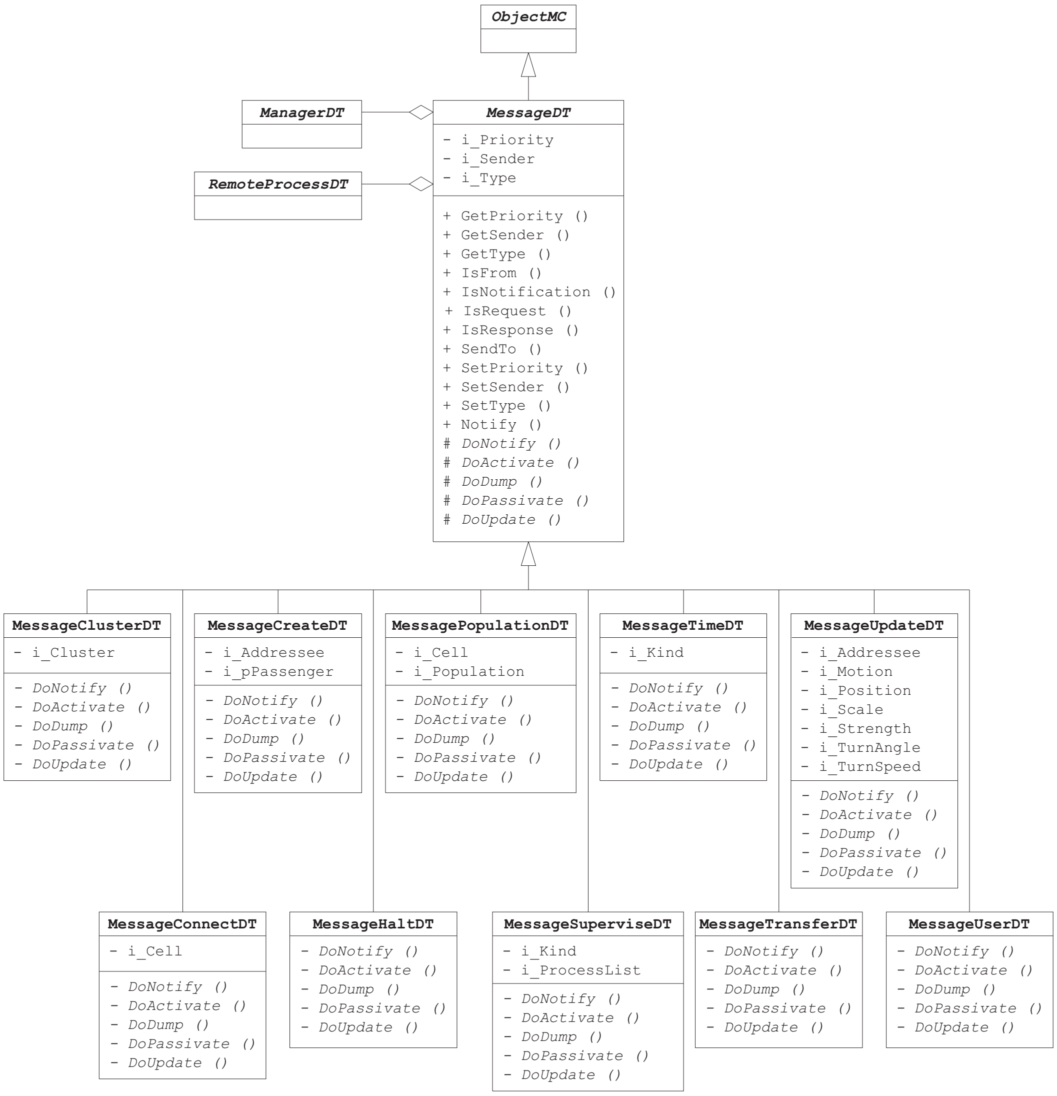

Figure A.7: Class diagram of the message classes.

A.8 Acceptor Classes

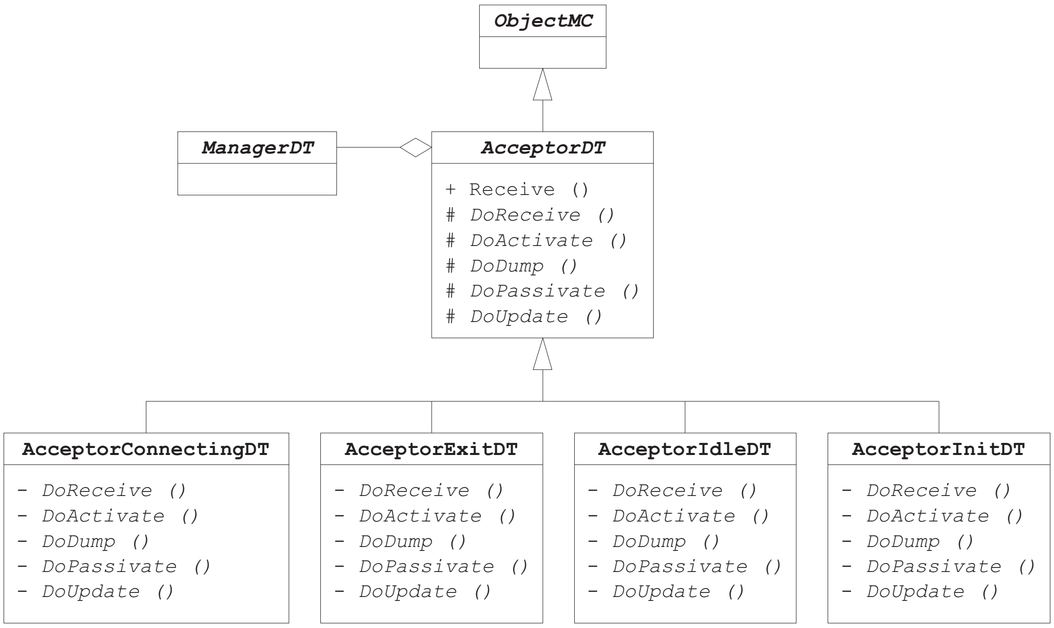

Figure A.8: Class diagram of the acceptor classes.

A.9 “`Asteroids`” Classes

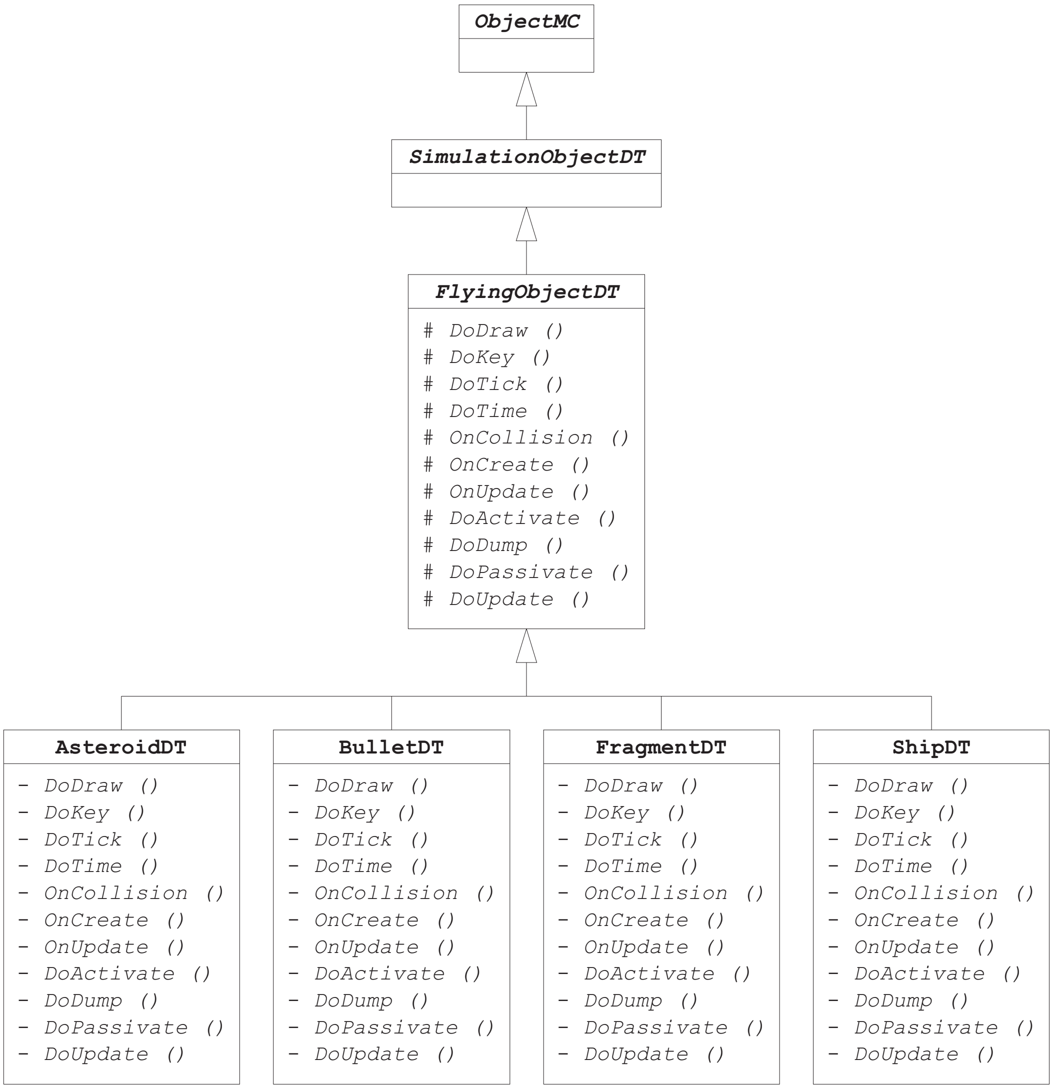

Figure A.9: Class diagram of the “`Asteroids`” classes.

A.10 Miscellaneous Classes

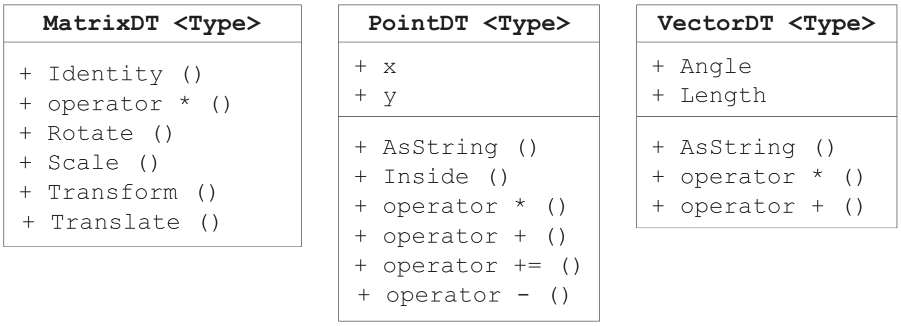

Figure A.10: Class diagram of miscellaneous classes.

Bibliography

[Agrawala and El Abbadi, 1991] Agrawala, D., El Abbadi, A. (1991). An Efficient and Fault-Tolerant Solution of Distributed Mutual Exclusion. *ACM Transactions on Computer Systems (TOCS)*, 9:1–20.

[Benford and Greenhalgh, 1997] Benford, Steve, Greenhalgh, Chris (1997). Introducing Third Party Objects into the Spatial Model of Interaction. CRG Internal Report, Department of Computer Science, The University of Nottingham, University Park, Nottingham, NG7 2RD, U.K. Accessible through the WWW (visited on August 29, 1997): `ftp://ftp.crg.cs.nott.ac.uk/pub/papers/CRG97-model2.ps.gz`.

[Benford *et al.*, 1994] Benford, Steve, Bowers, John, Fahlén, Lennart E., Greenhalgh, Chris (1994). Managing Mutual Awareness in Collaborative Virtual Environments. In *Proceedings VRST'94*, Singapore, August. ACM Press. Accessible through the WWW (visited on August 29, 1997): `ftp://ftp.crg.cs.nott.ac.uk/pub/papers/VRST94.ps.gz`.

[Benford *et al.*, 1997] Benford, Steve, Greenhalgh, Chris, Lloyd, David (1997). Crowded Collaborative Virtual Environments. Technical report, Department of Computer Science, The University of Nottingham, University Park, Nottingham, NG7 2RD, U.K. Accessible through the WWW (visited on August 29, 1997): `ftp://ftp.crg.cs.nott.ac.uk/pub/papers/CHI97a.ps.gz`.

[Birman and Renesse, 1997] Birman, Kenneth. P., Renesse, Robert van (1997). Software für zuverlässige Netzwerke (Software for Reliable Networks; in German). *Spektrum der Wissenschaft*, 1997(9):76–81.

[Birman, 1993] Birman, Kenneth. P. (1993). The Process Group Approach to Reliable Distributed Computing. In *Communications of the ACM*, volume 36, pages 36–53.

[Blair and Lea, 1992] Blair, Gordon S., Lea, Rodger (1992). The Impact of Distribution on the Object-Oriented Approach to Software Development. *IEE/BCS Software Engineering Journal (Special Issue on Object-Oriented Systems)*, 7(2).

[Blaschek, 1994] Blaschek, Günther (1994). *Object-Oriented Programming with Prototypes*. Springer-Verlag Berlin Heidelberg New York London Paris Tokyo.

[Bubendorfer, 1996] Bubendorfer, Kristian Paul (1996). Resource Based Policies for Load Distribution. Master's Thesis in Computer Science, Victoria University of Wellington, New Zealand, August. Accessible through the WWW (visited on July 19, 1997): `http://www.mcs.vuw.ac.nz/~kris/thesis/thesis.html`.

[Chow and Johnson, 1997] Chow, Randy, Johnson, Theodore (1997). *Distributed Operating Systems & Algorithms*. Addison-Wesley Longman, Inc.

[Coad and Nicola, 1991] Coad, Peter, Nicola, Jill (1991). *Object-Oriented Programming*. Object International, Inc.

[Coad and Yourdon, 1991a] Coad, Peter, Yourdon, Edward (1991). *Object-Oriented Analysis*. Object International, Inc., 2^{nd} edition.

[Coad and Yourdon, 1991b] Coad, Peter, Yourdon, Edward (1991). *Object-Oriented Design*. Object International, Inc.

[Coplien and Schmidt, 1995] Coplien, Jim, Schmidt, Douglas C. (1995). *Pattern Languages of Program Design*. Addison-Wesley Longman, Inc.

[Coulouris *et al.*, 1994] Coulouris, George, Dollimore, Jean, Kindberg, Tim (1994). *Distributed Systems – Concepts and Design*. International Computer Science Series. Addison-Wesley Longman, Inc., 2^{nd} edition.

[Flanagan, 1992] Flanagan, David (1992). *X Toolkit Intrinsics Reference Manual*, volume 5. O'Reilly Verlag Deutschland, 3^{rd} edition.

[Foster, 1994] Foster, Ian (1994). *Designing and Building Parallel Programs – Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman, Inc., Reading, MA. Accessible through the WWW (visited on August 29, 1997): `http://www.mcs.anl.gov/dbpp/text/book.html`.

[Gamma *et al.*, 1995] Gamma, Erich, Helm, Richard, Johnson, Ralph, Vlissides, John (1995). *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Longman, Inc.

[Gamma, 1992] Gamma, Erich (1992). *Objektorientierte Software-Entwicklung am Beispiel von ET++ – Design-Muster, Klassenbibliothek, Werkzeuge*. Springer-Verlag Berlin Heidelberg New York London Paris Tokyo.

[Garcia-Molina, 1982] Garcia-Molina, H. (1982). Elections in a Distributed Computing System. In *IEEE Transactions on Computers*, volume 31, pages 48–59, January.

[Geist *et al.*, 1994a] Geist, Al, Beguelin, Adam, Dongarra, Jack, Jiang, Weicheng, Manchek, Robert, Sunderam, Vaidy (1994). *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing.* Scientific and Engineering Computation Series. The MIT Press Cambridge, Massachusetts London, England. Accessible through the WWW (visited on August 29, 1997): `http://netlib.cs.utk.edu/pvm3/book/pvm-book.html`.

[Geist *et al.*, 1994b] Geist, Al, Beguelin, Adam, Dongarra, Jack, Jiang, Weicheng, Manchek, Robert, Sunderam, Vaidy (1994). PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, September. Accessible through the WWW (visited on August 29, 1997): `http://www.netlib.org/pvm3/ug.ps`.

[Greenhalgh and Benford, 1995a] Greenhalgh, Chris, Benford, Steve (1995). MASSIVE – A Collaborative Virtual Environment for Teleconferencing. *ACM Transactions on Computer Human Interfaces (TOCHI), ACM Press*, 2(3):239–261. Accessible through the WWW (visited on July 19, 1997): `ftp://turing.cs.nott.ac.uk/pub/papers/TOCHI95.ps.gz`.

[Greenhalgh and Benford, 1995b] Greenhalgh, Chris, Benford, Steve (1995). MASSIVE – A Distributed Virtual Reality System Incorporating Spatial Trading. In 15^{th} *International Conference on Distributed Computing Systems Incorporating Spatial Trading*, pages 27–34, Vancouver, Canada, May. IEEE, Inc., IEEE Computer Society Press, Los Alamitos, California. Accessible through the WWW (visited on July 19, 1997): `ftp://turing.cs.nott.ac.uk/pub/papers/DCS95.ps.gz`.

[Greenhalgh, 1994] Greenhalgh, Chris (1994). An Experimental Implementation of the Spatial Model. In 6^{th} *ERCIM Workshop*, Stockholm, Sweden, June. Swedish Institute of Computer Science (SICS). Accessible through the WWW (visited on July 19, 1997): `ftp://turing.cs.nott.ac.uk/pub/papers/ERCIM94-massive1.ps.gz`.

[Greenhalgh, 1996a] Greenhalgh, Chris (1996). Dynamic Embodied Multicast Groups in MASSIVE-2. Technical Report NOTTCS-TR-96-8, Department of Computer Science, The University of Nottingham, University Park, Nottingham, NG7 2RD, U.K., December. Accessible through the WWW (visited on July 19, 1997): `ftp://turing.cs.nott.ac.uk/pub/papers/NOTTCS-TR-96-8.ps.gz`.

[Greenhalgh, 1996b] Greenhalgh, Chris (1996). Spatial Scope and Multicast in Large Virtual Environments. Technical Report NOTTCS-TR-96-7, Department of

Computer Science, The University of Nottingham, University Park, Nottingham, NG7 2RD, U.K., September. Accessible through the WWW (visited on July 19, 1997): `ftp://turing.cs.nott.ac.uk/pub/papers/NOTTCS-TR-96-7.ps.gz`.

[Henricson and Nyquist, 1992] Henricson, Mats, Nyquist, Erik (1992). Programming in C++ – Rules and Recommendations. Technical Report, Ellemtel Telecommunications Systems Laboratories, Box 1505, 125 25 Älvsjö, Sweden.

[Hubbold *et al.*, 1997] Hubbold, Roger, Murta, Alan, West, Adrian, Howard, Toby (1997). Design Issues for Virtual Reality Systems. Technical Report, Advanced Interfaces Group, Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, UK. Accessible through the WWW (visited on July 19, 1997): `http://www.cs.man.ac.uk/aig/publications/barcelona.ps.Z`.

[Koenig, 1995] Koenig, Andrew (1995). Programming Language C++. Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++ X3J16/95-0087, WG21/N0687, AT&T Bell Laboratories, PO Box 636, 600 Mountain Avenue, Murray Hill, NJ 07974 USA. `ark@research.att.com`.

[Kramer, 1997] Kramer, Jeff (1997). Distributed Software Engineering – Invited state-of-the-art Report. Technical Report, Department of Computing, Imperial College, London SW7 2BZ.

[Kulczycki, 1997] Kulczycki, Peter W. (1997). An Application Framework for the Distributed Simulation of Virtual Worlds by Spatial Decomposition – Source Code and Documentation. Master's thesis, Research Institute for Symbolic Computation (RISC), Johannes Kepler Universität, A-4040 Linz, Austria, Europe, August. Accessible through the WWW (visited on August 29, 1997): `http://www.risc.uni-linz.ac.at/people/reports.cgi?query=pkulczyc`.

[Ladin *et al.*, 1992] Ladin, R., Liskov, B., Shrira, L., Ghemawat, S. (1992). Providing Availability Using Lazy Replication. In *ACM Transactions on Computer Systems (TOCS)*, volume 10, pages 360–391.

[Lamma *et al.*, 1994] Lamma, Evelina, Mello, Paola, Wegner, Peter (1994). The Role of State in Open Systems (Extended Abstract), September.

[Lamport, 1978] Lamport, Leslie (1978). Time, Clocks, and the Ordering of Events in a Distributed System. In *Communications of the ACM*, volume 21, pages 558–565. ACM Press, July.

[Lamport, 1997] Lamport, Leslie (1997). Concurrent Reading and Writing of Clocks. In *ACM Transactions on Computer Systems (TOCS)*, volume 8, pages 305–310. ACM Press, November.

[Lomdahl and Beazley, 1993] Lomdahl, P. S., Beazley, D. M. (1993). Message-Passing Multi-Cell Molecular Dynamics on the Connection Machine 5. *Parallel Computing*.

[Lynch, 1996] Lynch, Nancy A. (1996). *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc. San Francisco, California.

[Macedonia *et al.*, 1995] Macedonia, Michael R., Zyda, Michael J., Pratt, David R., Brutzman, Donald P., Barham, Paul T. (1995). Exploiting Reality with Multicast Groups: A Network Architecture for Large-Scale Virtual Environments. In *Proceedings 1995 IEEE VRAIS'95*, RTP, North Carolina, March.

[Masoud and Sloman, 1992] Masoud, Mansouri-Samani, Sloman, Morris (1992). Monitoring Distributed Systems (A Survey). Research Report DOC92/23, Department of Computing, Imperial College of Science Technology and Medicine, 180 Queen's Gate, London SW7 2BZ, U.K., September.

[Mayr, 1997a] Mayr, Herwig (1997). Ingenieursmäßige Projektentwicklung im Software Engineering. Lecture Notes, FH-Studiengang Software Engineering, A-4232 Hagenberg, Austria, Europe.

[Mayr, 1997b] Mayr, Herwig (1997). Using Software Sensors for Migrating From Classical Simulation Systems Towards Virtual Worlds. In *Proceedings ECBS'97*, Monterey, CA. IEEE.

[Mizuno *et al.*, 1994] Mizuno, M., Raynal, M., Zhou, J. Z. (1994). Sequential Consistency in Distributed Systems – Theory and Implementation. Internal Publication 871, Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA), Campus de Beaulieu – 35042, Rennes Cedex, France, October.

[Mullender, 1993] Mullender, Sape (1993). *Distributed Systems*. ACM Press Books. Addison-Wesley Longman, Inc., ACM Press, 2^{nd} edition.

[Musser and Saini, 1996] Musser, David R., Saini, Atul (1996). *STL Tutorial and Reference Guide – C++ Programming with the Standard Template Library*. Addison-Wesley Professional Computing Series. Addison-Wesley Longman, Inc.

[Nye and O'Reilly, 1992] Nye, Adrian, O'Reilly, Tim (1992). *X Toolkit Intrinsics Programming Manual – Motif Edition*, volume 4M. O'Reilly Verlag Deutschland, 2^{nd} edition.

[Nye, 1992a] Nye, Adrian (1992). *Xlib Programming Manual*, volume 1. O'Reilly Verlag Deutschland, 3^{rd} edition.

[Nye, 1992b] Nye, Adrian (1992). *Xlib Reference Manual*, volume 2. O'Reilly Verlag Deutschland, 3^{rd} edition.

[Petzold, 1992] Petzold, Charles (1992). *Programming Windows 3.1.* Microsoft Press, 3^{rd} edition.

[Pomberger and Blaschek, 1996] Pomberger, Gustav, Blaschek, Günther (1996). *Software Engineering – Prototyping und Objektorientierte Software-Entwicklung.* Carl Hanser Verlag München Wien, 2^{nd} edition.

[Pree, 1995] Pree, Wolfgang (1995). *Design Patterns for Object-Oriented Software Development.* ACM Press Books. Addison-Wesley Longman, Inc., ACM Press.

[Quercia and O'Reilly, 1993] Quercia, Valerie, O'Reilly, Tim (1993). *X Window System User's Guide – Standard Edition*, volume 3. O'Reilly Verlag Deutschland, 4^{th} edition.

[Rational, 1997a] Rational (1997). Unified Modeling Language (UML) – Notation Guide. Technical Report, Rational Software Corporation, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951, January. Accessible through the WWW (visited on July 19, 1997): `http://www.rational.com`.

[Rational, 1997b] Rational (1997). Unified Modeling Language (UML) – Process-Specific Extensions. Technical Report, Rational Software Corporation, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951, January. Accessible through the WWW (visited on July 19, 1997): `http://www.rational.com`.

[Rational, 1997c] Rational (1997). Unified Modeling Language (UML) – Semantics. Technical Report, Rational Software Corporation, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951, January. Accessible through the WWW (visited on July 19, 1997): `http://www.rational.com`.

[Rational, 1997d] Rational (1997). Unified Modeling Language (UML) – Semantics Appendix and Glossary. Technical Report, Rational Software Corporation, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951, January. Accessible through the WWW (visited on July 19, 1997): `http://www.rational.com`.

[Rational, 1997e] Rational (1997). Unified Modeling Language (UML) – Summary. Technical Report, Rational Software Corporation, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951, January. Accessible through the WWW (visited on July 19, 1997): `http://www.rational.com`.

[Raynal and Singhal, 1995] Raynal, M., Singhal, M. (1995). Logical Time – A Way to Capture Causality in Distributed Systems. Internal Publication 900, Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA), Campus de Beaulieu – 35042, Rennes Cedex, France, January.

[Raynal, 1988] Raynal, M. (1988). *Distributed Algorithms and Protocols.* Wiley.

[Ricart and Agrawala, 1981] Ricart, G., Agrawala, A. K. (1981). An Optimal Algorithm for Mutual Exclusion in Computer Networks. In *Communications of the ACM*, volume 24, pages 9–17. ACM Press.

[Roehl, 1995a] Roehl, Bernie (1995). Distributed Virtual Reality – An Overview, June.

[Roehl, 1995b] Roehl, Bernie (1995). Some Thoughts on Behavior in VR Systems, August.

[Santifaller, 1993] Santifaller, Michael (1993). *TCP/IP und ONC/NFS in Theorie und Praxis – Unix in lokalen Netzen.* Addison-Wesley Longman, Inc., 2^{nd} edition.

[Schroeder, 1993] Schroeder, Michael D. (1993). A State-of-the-Art Distributed System: Computing with BOB. In Mullender, Sape, editor, *Distributed Systems*, pages 1–16. Addison-Wesley Longman, Inc., ACM Press.

[Severance and Enbody, 1996] Severance, Charles, Enbody, Richard (1996). Dynamic Load Balancing with Integrated Data Migration for Spatially Decomposable Problems. Technical Report, Computer Science Department, Michigan State University, East Lansing, MI 48824-1027. Accessible through the WWW (visited on July 19, 1997): `http://www.egr.msu.edu/~crs/papers/ipps_96/`.

[Shuey *et al.*, 1997] Shuey, Richard L., Spooner, David L., Frieder, Ophir (1997). *The Architecture of Distributed Computer Systems – A Data Engineering Perspective on Information Systems.* Addison-Wesley Longman, Inc.

[Silberschatz and Galvin, 1994] Silberschatz, A., Galvin, P. (1994). *Operating System Concepts.* Addison-Wesley Longman, Inc. Reading, MA.

[Sinha, 1997] Sinha, Pradeep K. (1997). *Distributed Operating Systems – Concepts and Design.* IEEE Computer Society Press.

[Snowdon and West, 1994] Snowdon, David N., West, Adrian J. (1994). The AVIARY VR-System – A Prototype Implementation. In 6^{th} *ERCIM Workshop*, Stockholm, Sweden, June. Swedish Institute of Computer Science (SICS). Accessible through the WWW (visited on July 19, 1997): `ftp://ftp.crg.cs.nott.ac.uk/pub/dns/aviary/aviary-ercim94.ps.gz`.

[Stroustrup and Ellis, 1991] Stroustrup, Bjarne, Ellis, Margaret A. (1991). *The Annotated C++ Reference Manual.* Addison-Wesley Longman, Inc., AT&T Bell Telephone Laboratories, Inc.

[Stroustrup, 1994] Stroustrup, Bjarne (1994). *The Design and Evolution of C++.* Addison-Wesley Longman, Inc., AT&T Bell Telephone Laboratories, Inc.

[Stroustrup, 1997] Stroustrup, Bjarne (1997). *The C++ Programming Language.* Addison-Wesley Longman, Inc., 3^{rd} edition.

[Tanenbaum, 1989] Tanenbaum, Andrew S. (1989). *Computer Networks.* Prentice Hall, Inc., Englewood Cliffs, New Jersey 07632, 2^{nd} edition.

[Tanenbaum, 1995] Tanenbaum, Andrew S. (1995). *Distributed Operating Systems.* Prentice Hall, Inc., Englewood Cliffs, New Jersey 07632.

[Wallmüller, 1997] Wallmüller, Ernest (1997). *Software-Qualitätssicherung in der Praxis.* Carl Hanser Verlag München Wien.

[Wegner, 1997] Wegner, Peter (1997). Interactive Models of Computing. Lecture Notes, Brown University, July.

[Wolf, 1996] Wolf, Werner A. (1996). OASIS – An Object-Oriented Approach to Sensor-Integrated Simulation. Diploma Thesis, RISC-Linz, Institut für Mathematik, Technisch-Naturwissenschaftliche Fakultät, Johannes Kepler Universität, A-4040 Linz, Austria, Europe, September.