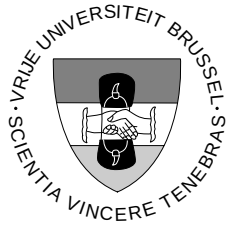

Vrije Universiteit Brussel
Faculteit Wetenschappen - Departement Informatica
Academiejaar 1997 - 1998

A Framework for replication of objects using Aspect-Oriented Programming

By: Johan Fabry

Promoter: Prof. Dr. Theo D'Hondt

Acknowledgements

I wish to take this opportunity to thank the people who made this work possible:

First of all, my promoter Prof. Dr. Theo D'Hondt for suggesting the subject and for steering me onto the right tracks when it was needed. Also Geert Lathouwers, who introduced me to AOP and continued to guide me throughout the work on this thesis, even though he discontinued his other work at the university.

I am grateful to Wolfgang De Meuter for giving me his Java parse tree generator, a big help in shortening the development time for the weaver, just when I needed it most. Roel Wuyts was helpful in providing a number of final hints and tips about writing this thesis.

The following people have graciously spent time in reading this thesis to get it 'just right': Kris De Volder, Geert Lathouwers, Kim Mens, Patrick Steyaert and Roel Wuyts. Thanks to my father, for searching for grammatical errors, even though he only understood "the part about the cars".

I must also thank all the people at DINF for giving me an education I can truly feel proud about.

Thanks to all my fellow thesis students at DINF for miscellaneous help, and thanks to everybody who helped, but who I've forgotten to mention here.

And last, but certainly not least, a big thanks to my parents, who gave me the freedom to be on my own when I wanted to, and the support to continue with my studies when I needed it.

Contents

Chapter 1

Introduction

Whereas in the early days of computing the view on computing was mostly concentrated on a single, independant computer perspective, the advent of the personal computer and convenient networks has made more distributed systems available as an option to handle many types of problems.

In these distributed systems, each component, be it a high-end server or a low-end PC, collaborates by means of distributed system software to reach a certain goal. Ideally, this distributed software system manages each components' resources and presents to the user an integrated computing facility, regardless of the number or kind of the different components.

A nice example of a distributed system is "Distributed.net". This system uses computers connected to the internet to perform certain tasks. The software consists of a small client application running on each machine and server software, which runs on a small number of selected computers. The client application performs some computing task, the server software ensures coordination between the different clients. Distributed.net has the computing power equivalent or larger than the world's fastest supercomputers and has been successfully used to crack the RC5-56 and DES-56 encryption algorithms.

This promise of unlimited computing power, sadly enough, is not as simple as it might seem. A large problem for such a distributed system is the sharing of data. With many computers working on a single goal, it is often the case that the data representing this goal must be used by multiple computers simultaneously, so it must be shared. In many cases it is possible to prevent the sharing problem by dividing the data in subparts, which can be processed completely independently by the different computers. Distributed.net imposes this restriction on the problems it attempts to solve, which makes it possible to reach such advanced computing power.

For many tasks, however, this restriction can not be imposed and some means of sharing data must be provided. There are two types of approaches for sharing the data: have the data available from one server computer or

have the data available from a larger number of computers, keeping this data identical on the different computers.

The first approach has two major problems: reliability and speed. In case of a server crash, the entire system will be inoperational, and also the speed of data requests from and updates to the server are relatively slow, due to the slow network speed.

The second approach, replication, tries to avoid these problems. Because the data is contained on several servers, a single failure will not render the system unoperational. The redundancy in the data sharing ensures higher reliability. Having multiple servers containing the data also allows us to place them at strategic locations in the network, optimizing client access speed to these servers.

Replication is not without its own drawbacks. The data shared on a server must be, at least approximately, identical to the data on every other server. This data consistency requirement creates a need for interaction between the different servers, introducing some overhead. This overhead depends on how strong the consistency needs to be and the frequency of changes occurring to the data. In some cases this overhead can become prohibitively high, making replication an unsuitable solution.

Until now, whenever a distributed application needed replication, often the replication algorithm was custom-built. It should be possible, however, to make a general replication algorithm, which will only need a limited number of customizations for a large number of applications. Having such a replication framework, implementing replication in new systems, or adding replication to existing systems, will reduce the amount of work to make this possible.

Previous work on adding replication to existing systems has shown that adding replication cannot be done using conventional modular or OO programming without requiring a large amount of work. This is because replication can not be encapsulated in a module or in an object, which can be transparently added to an existing system [20]. The code handling replication must be intertwined in a large section of the existing code because it must be added to each module requesting data from or updating data to the server.

The underlying reason for this can be explained using the concept of separation of concerns [11]. Replication can be seen as a separate concern from the basic algorithm. In the code for the basic algorithm, the programmer must 'keep in mind' the special-purpose concern of replication. Separating replication from the basic algorithm will allow her to deal with it separately, making it easier to design and implement the system or to add replication to an existing system.

One way of having this separation is aspect-oriented programming(AOP) [17]. In AOP every concern or aspect is expressed in a special-purpose language which allows the programmer to reason easily about an aspect. These

descriptions are then combined into an executable form by a tool called an Aspect Weaver™.

Taking an AOP approach to replication would entail creating a separate aspect language for replication and implementing an aspect weaver. This weaver would then be capable of weaving the code of the base algorithm with the special purpose aspect code to produce code which contains the concern of replication. This can be used to let a programmer add replication to the system in a separate stage.

1.1 Purpose

This dissertation will evaluate the possibility to create a framework for replication which, when instantiated, will add replication to a system being developed or to an existing system.

Whereas replication has usually been custom-built for certain applications we will try to create a framework which is tailored to suit the needs of different types of programs.

Also, we will use aspect-oriented programming to try to achieve the ultimate programmer-friendly replication. Providing good 'separation of concerns', the aspect "replication" must be completely hidden from the programmer when she is concerned with other concerns. Whenever this aspect must be modified, modifications should be easy to perform.

Finally, to validate that our framework is usable for different applications we will instantiate it for two different types of application: a distributed warehousing application and a chat application.

1.2 Overview

The next chapter will introduce frameworks: systems which are built specifically for reuse of their design. Some design patterns are well-suited to use in frameworks, we will introduce some of these patterns which will be used later on in the dissertation.

Chapter three will discuss replication. We will give a short definition, and show the need for replication in a number of distributed systems. A general overview of different strategies to achieve replication will be presented, and we will introduce an abstract model for replication.

The fourth chapter will introduce aspect-oriented programming. We will show the need for separation of concerns and present aspect-oriented programming as a technique to achieve this separation. To have a better 'feel' of AOP, the AOP system AspectJ will be presented and reviewed.

In chapter five we will present our aspect-oriented approach to replication, in order to achieve high replication transparency. The aspect languages for replication: Jav, Dupe and Fix will be introduced and the aspect weaver

which combines these languages will be discussed. These elements will be used to create a link between the algorithm of the clients and the core of the replication framework.

Chapter six will present our framework for replication. First we will analyze how this framework can be designed so it allows enough flexibility to be used in a wide variety of environments. Once the analysis complete we will discuss the implementation of the framework, which will include information on how the framework must be instantiated.

In the seventh chapter we will validate our claim that the combination of AOP and the replication framework provides easy replication for a variety of different applications. We will instantiate the framework for two, quite differing applications: a distributed warehousing application and an internet chat application.

The final chapter will present our conclusions and suggest further topics for research.

Chapter 2

Frameworks

Reuse as a means to boost productivity is considered important in software engineering. Two elements which have contributed to a shorter development cycle are use of frameworks and design patterns. This chapter will introduce frameworks; a means to reuse a design for applications within an application domain, and design patterns; reusable solutions for a wide variety of reoccurring design problems. Some of these design patterns can be very useful elements for a framework, we shall give a brief overview of three of these useful patterns which we will use in our replication framework.

2.1 Reusing Application Design

Achieving a high degree of reuse in software systems is considered an important issue in the software engineering community. Reusing parts of an existing application significantly reduces the time to create a new application.

In OO software, some of this reuse has been geared towards reuse of *components*. Such components can be considered as completely separate entities with a set interface and a set functionality, usually cleanly encapsulated in an object, method or API. A typical example is a set of user interface components, such as buttons, sliders, menus and radio buttons. In some cases, such component reuse has been facilitated by standards for these components, such as the JavaBeans architecture [28].

However, sometimes what has to be reused are not individual components of a system, but the overall design. System design is hard, so reusing an existing design will yield a faster turnover. The next logical step is to build these system designs in a fashion which ensures that they can be reused a number of times within an application domain.

Frameworks capture these reusable system designs and express them as sets of classes. These classes describe and enforce the way in which instances of their subclasses must interact. Frameworks express a design for a given

application domain, such as building graphical editors [1], broadcast planning systems [3], or order processing and warehouse inventory management [13].

A framework only dictates an overall architecture for applications: it only defines the general structure of the application, the responsibilities of the different objects in the application and the way in which they interact.

A framework must be *instantiated* to form the final application. Because it only provides a skeleton application for the domain, a number of abstract classes in the framework will need to be subclassed by classes which implement the behavior specific to the application. The aspects of the domain which differ between different applications and are represented by these abstract classes and methods are called *hot-spots*[6].

Consider the example of a car framework. We can define our problem domain as four-wheeled vehicles which transport a number of persons from A to B over land. Our framework can then include such abstract classes as Wheel, Engine, Gearbox, Transmission, Suspension, Body and Interior. These classes are required to interact in the following fashion: The Engine drives the Gearbox, which drives a number of Wheels through the Transmission. The Transmission is attached to the Body by the Suspension, and so on. This framework determines a number of design decisions for cars, but does not create a concrete vehicle.

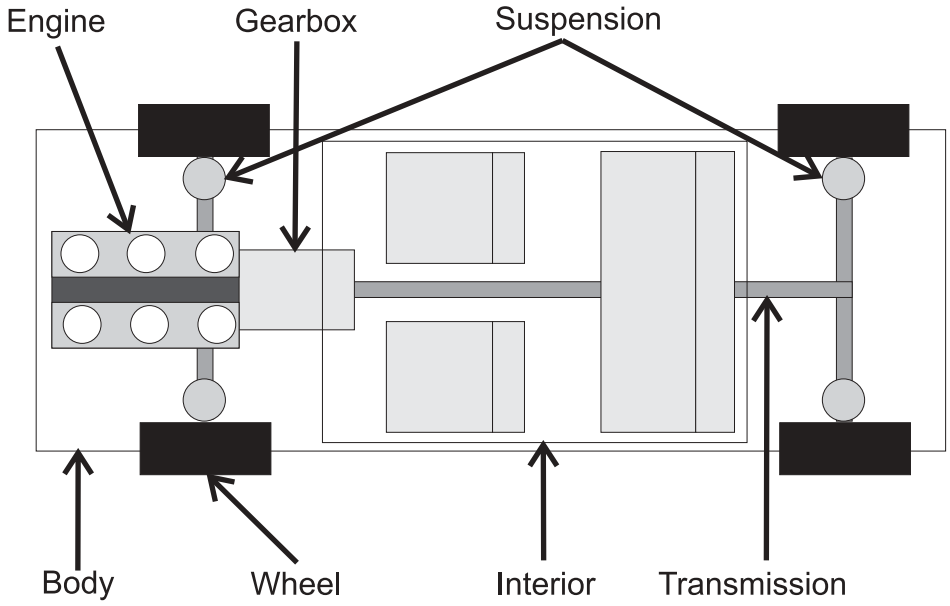

Figure 2.1: Schematic view of the car framework

In our car example, we can instantiate the car framework in a number of ways. To produce an off-road vehicle, we choose an Engine with a lot of

torque at low revs, a 4-speed Gearbox suited for off-road driving, a Transmission which drives the four Wheels, which are large and have deep grooves for better traction in sand and mud, and we suspend these to the Body with a high, soft Suspension. To produce a sportscar, we choose an Engine with a lot of torque at high revs, a 6-speed Gearbox suited for high-speed driving, a Transmission which drives the two back Wheels, which are broad with no grooves for better grip on the road, and we suspend the wheels to the Body with a low, hard suspension.

Using a framework leads to an opposite manner of code reuse. When reusing components for an application, the newly written code calls code in the components to achieve some functionality. When instantiating a framework, the code of the framework will call the newly written code to achieve some functionality. This is known as "Don't call us, we'll call you" [15], or *the Hollywood principle* [6]. This principle implies that the programmer must respect the calling conventions specified in the framework and the functionality which is required of the newly added code.

Because a framework severely limits the design possibilities of the programmer, it must necessarily be general enough to handle a wide variety of possible instantiations. Given that a framework represents the design, changes in a framework will severely impact applications written using the framework. However, a framework which is too general will be hard to reuse because of the higher amount of work necessary to instantiate it. So flexibility is also an important issue for a framework.

Our car framework demonstrates some of this flexibility: because Gearbox has been separated from Transmission, it is easy to create a four-wheel driven sportscar, by replacing the Transmission which drives only the back Wheels with a Transmission which drives all four Wheels. Had we specified that the Gearbox drives a number of Wheels, it would still have been possible to create a four-wheel driven sportscar, but it would have needed a new type of Gearbox, whereas we now can reuse the Transmission from the off-road. Assume our framework did not have this separation, were we to introduce it, we would need to adapt each instantiation of the framework (each different kind of car) to this new architecture, with a separated Gearbox and Transmission. This clearly can be an important change.

A number of techniques have proven to be successful tools for creating frameworks. One of these techniques is Design patterns.

2.2 Design Patterns

Design Patterns [10][2] originate from the observation that a number of solutions for certain design problems in software engineering seem to crop up repeatedly. This can mainly be attributed to a systematic manner in which experienced software designers reuse previous solutions to similar problems.

Whereas a person new to object-oriented software or to designing of OO software has no experience to lean on while designing a new piece of software, the experienced designer knows a number of general patterns of object hierarchies and their interaction which can be tailored to solve a number of specific problems. Instead of redesigning each application from scratch, the experienced designer remembers previous good solutions to a given problem and re-applies them to solve the problem at hand. These good solutions which can be re-applied often are called *Design Patterns*.

A number of books catalogue these design patterns [10] [2] [4] [31] [9]. In these books patterns are commonly described using their name, a description of the type of problem the pattern tries to address, a description of the objects which comprise the pattern and the relationships, collaborations and responsibilities of these objects.

Some design patterns have proven to be extremely useful not only for building frameworks, but also for understanding frameworks [1] [6]. Using design patterns to document a framework allows us to reason about the framework at a higher level of abstraction, which makes it easier to comprehend. Also, understanding the design patterns used in a framework implies understanding why those patterns were used and what these patterns achieve as a solution.

Three of these useful patterns appear quite frequently, and will be used later on. We shall summarize them, without covering them completely. For a complete discussion, see [10].

Factory method : Many frameworks consist mainly of abstract classes, and may instantiate any of these classes at will. To have a correct instantiation of the framework, the correct classes must be instantiated. Factory method defines an interface for creating an object, but lets subclasses decide which class to instantiate.

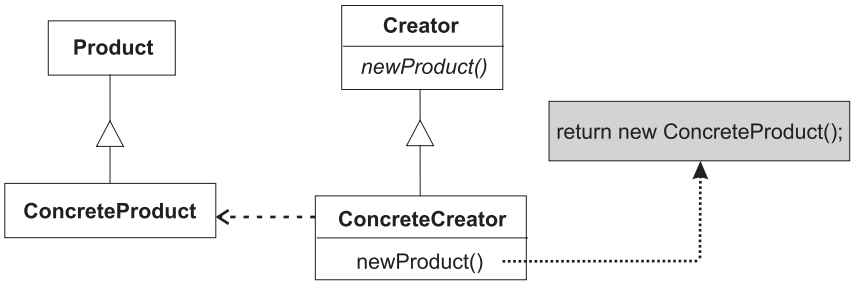

Figure 2.2: Overview of Factory method

This pattern consists of four classes: an abstract *Product*, an abstract *Creator* for this Product, a *ConcreteProduct* (subclass of Product) and

a *ConcreteCreator* (subclass of Creator). Creator and Product typically are classes of the framework. Creator will define an abstract method, say newProduct, which should return an instance of the correct subclass of Product. An instantiation of the framework will have ConcreteCreator return the correct ConcreteProduct.

Singleton : Often a requirement in a framework states that there may only be one instance of a certain class. Singleton ensures this, and provides a global access point to it, by making the class responsible for this sole instance.

A single class is required for the Singleton pattern. The *Singleton* class must contain its sole instance and provide access to it using a class method. Optionally, it may also be responsible for creating its unique instance. Each class which wishes to use the Singleton must however use the instance access method provided by the Singleton class to gain a reference.

Strategy : Because many algorithms may exist for a (part of a) given application domain, hardwiring a single algorithm in the framework would be a bad decision. Encapsulating each algorithm and making them interchangeable, allows the algorithm to vary across different instantiations of the framework.

The Strategy pattern requires an abstract *Strategy* class and any number of *ConcreteStrategy* subclasses. The Strategy class declares the operations for instantiation of the ConcreteStrategy and the operations for executing the algorithm. Each ConcreteStrategy encapsulates an algorithm, an instantiation of the framework will use the correct ConcreteStrategy. Creation of this correct ConcreteStrategy will often be achieved using a Factory Method.

These three patterns are only a small subset of all documented patterns, but are of some importance to us. This because the replication framework, which we will present later on, will use these three patterns extensively.

2.3 Conclusion

Reuse as a means to achieve faster software development can be achieved using multiple techniques. Two of these techniques are frameworks and design patterns. Frameworks speed up application development because of design reuse, and design patterns speed up software, and framework, development because of solutions reuse.

A possible application domain for which a framework could be developed would be replication. Replication is used by a large number of distributed

systems to share data amongst different components in the system. A framework for replication would significantly speed up the development process of a number of distributed systems.

Chapter 3

Replication

In distributed systems there is a need to share data amongst different computers. One possible method to achieve this data sharing is replication. We will provide a definition of replication, discuss its uses and the elements which need to be considered when opting for replication.

Although the concepts of replication are not linked to a programming paradigm, we shall primarily discuss replication for object-oriented software. This is because we are working towards a framework for replication, and frameworks are an OO concept. However, many of the topics discussed are also valid for non-OO software.

3.1 The Need for Replication

3.1.1 Replication

In a *distributed system* a, possibly large, number of computers, linked by a network, cooperate to achieve a common goal [5]. This goal and the various intermediate solutions required to achieve it, are encoded in some fashion into an amount of data. For each computer in the system to be able to perform its subtask correctly, it will need some form of access to this data.

There are a number of methods to achieve this access, one of which is *replication*. A system which uses replication copies the data to a number of servers. Each server makes the data accessible to the computers which need access to it, and ensures that changes made to the data are passed on to the other copies of the data [5].

Some kinds of problems do not require replication, or any form of data sharing. If each subtask can be performed completely independently, handling the data is easy: split the data up into parts, each part representing a subtask. These subparts can then be distributed to the different computers in the system, which then perform the subtasks [7].

A typical example of this class of problems is cracking encrypted data. Assume some data has been encrypted with a given algorithm which uses a

certain key. Given the encrypted data and the algorithm, the key is necessary to recover the original data. Finding the key can be done exhaustively: since the key is a value within a set range, try all possible values for the key. This search can be speeded up considerably by a distributed system. Divide the known key range by the number of computers in the distributed system, and let each computer try all keys in its subrange. This strategy has been used by the distributed system Distributed.net [8] to solve the RC5-56 and DES-II-1 56bit challenges issued by RSA labs [19].

However, this class of problems is but a small portion of all the tasks trying to be solved by distributed systems. For many tasks, multiple computers will need access to a single piece of data, so sharing the data amongst different computers is necessary.

Let us consider a well-known example: a distributed bulletin board system such as USENET. If there were no way in which the computers using USENET could access the same data (the posted messages), each computer would only have its separate bulletin board. This is clearly not how the system needs to work. All these separate bulletin boards should be one, united board, so each computer can access all messages posted on USENET.

A possible strategy is having the data available on one single server. Whenever a computer needs access to the data, it will connect itself to the server and transfer the data needed.

In our USENET example there would be a single server, we will call it "world.news.org", which contains all the USENET messages. A client computer, wanting to read messages or post new messages, needs to connect itself with world.news.org and fetch the needed messages from or put the posted messages on the server.

There are some obvious problems with this single server strategy [5]. First of all, what would happen if the single server goes down? None of the clients can communicate with the server, so no data can be read from or written to the bulletin board. This means the entire distributed system has become unoperational. Secondly, consider the amount of traffic the single server has to cope with. With all clients using this single server for all data transfer, the server can easily be confronted with a large amount of work. Linked with this is the speed of data transfers from a client's perspective. If the single server is heavily loaded, data transfers will proceed at a slow rate. Moreover, it is likely that the network link between the client and the server has become slow due to the fact that many clients are using this network simultaneously. Also, as the number of clients grows, network congestion will rise, which often leads to slower connections.

Having one single server clearly generates a number of nontrivial problems. A logical step to avoid these problems is to have not one single server, but multiple servers. This multiple server solution is called replication. Now the different servers can share the load and act as backups for each other should one fail. The servers can also be placed at strategic locations in the

network to minimize network traffic. Because of this, replication is well-suited to handle problems where high reliability of the data or high speed access for the clients is important.

USENET uses replication to share the bulletin board messages all over the world. Each major internet access point (different universities, access providers, ...) has its own news server which contain copies of the messages. Users of the access point use these local servers to read the messages or post their messages.

One important element remains: there has to be a way in which the data on the different servers is kept in the same state. Clients can change the data on their server, and these changes on the different servers can accumulate until the data on the different servers is fundamentally different. When this happens, the different elements in the distributed system cannot cooperate with each other, and the common goal cannot be reached.

3.1.2 Data Consistency and its Overhead

To keep the data *consistent* i.e. in a similar state on all servers, a consistency algorithm must be chosen. The algorithm will determine the fashion in which the different servers are kept consistent with each other, according to how strong the consistency requirement is [5][20].

Strong data consistency usually means that the data on each server must be identical to the data on each other server at all times. *Weak data consistency* allows the data on the different servers to vary for a given time, the data must be synchronized at some moments in time, but not always. The times at which the data must be synchronized can be fixed points in time, or can vary: e.g. data can be synchronized after a given amount of inactivity time of the clients, or after a given number of accesses has occurred by the clients

So, a system using replication should not only have a complete setup of different servers handling the data, but also a consistency requirement and an algorithm ensuring this requirement [5].

The algorithm required to enforce this consistency requirement will need to perform a number of operations to ensure that the data is kept consistent on all servers. These operations will add a certain overhead to each data access to the servers. This overhead mainly depends on two factors: the data consistency requirement and the amounts of reads and writes within the system [5].

Let us consider the consistency requirement for the USENET example. In USENET, the consistency requirement is not very strong. The servers only update each other at given time intervals, usually ranging from once every hour to once every day. Now imagine USENET with the requirement for strong data consistency. Every time a client would post a new message to a local server, every other server would have to be updated immediately

with this new message. This would generate a much larger overhead for each posted message. With a looser consistency requirement, different updates can be bundled together to be treated at the other servers. By bundling these updates and sending these bundles instead of each message separately, the network overhead for sending the messages is smaller.

Now consider the rate at which data is read versus written. Data which is read a lot but almost never written will have a smaller overall overhead than data which is written frequently and read rarely. This is due to the fact that write operations imply that changes must be made to the data on other servers, whereas read operations do not.

The relationship between these two factors is simple: the consistency requirement determines the overhead per write operation, and the read/write ratio on the data will determine the overall overhead per access on the data.

In some cases, this overhead can become extremely large. All processing time on the servers will be spent on keeping the data consistent, which will prevent successful read or write operations on the data by the clients. When this happens, the distributed system will no longer be able to operate.

Therefore, whenever replication is considered as a solution for sharing data, the consistency and read/write ratio must be evaluated to determine if the overhead for ensuring data consistency will not be prohibitively high [20].

If the overhead proves to be acceptable, replication can be considered as a viable option. But to achieve replication there a still a number of choices which have to be made.

3.2 Choices in Replication

Replication is a quite complex strategy for sharing the data. For different sub-parts of the entire replication algorithm, there are a number of choices which have to be made. Some of the most relevant options are the degree of replication transparency, the choice of active or passive replication, determining the consistency requirement and the manner in which the servers communicate amongst themselves.

3.2.1 Replication Transparency

An important requirement for replicated data is *replication transparency*[5]. Computers using replicated data should not be aware that there are multiple copies of the data. As far as they are concerned there is but one instance of the data. Writing data should only require one write operation, although there may be many copies of the data on different servers, and reading data will only produce one result, although there may be many different values for the data on the different servers.

This does not imply that the client is unaware that the data is replicated, and therefore errors may occur while accessing the data, when e.g. the client's network link is severed. We feel that, as is suggested in [20], an extra transparency requirement could be added: this requirement would be that the client is completely unaware of the replication algorithm. This would allow the programmer to fully focus on the task at hand, and ignore the extra work needed to handle replication.

So, while minimal replication transparency is required, choices must be made on how far the client is 'shielded' from the replication algorithm.

3.2.2 Active vs. Passive Replication

Assume replication is chosen in order to achieve higher data availability or fault-tolerance. In such cases, whenever a server crashes, the system must still remain operational. Clients will have to switch over to another server. This requires a high level of data consistency, which will ensure a server crash will have a minimal impact.

The first choice that has to be made, is between active and passive replication [18][22]. In an OO system, the 'data' which is replicated is a collection of objects. Changes occurring to these objects are then caused by method calls on these objects. The difference between active and passive replication lies in where these methods are executed.

In *passive replication*, also called the "primary backup approach" the method effecting the change of data is executed on one server, called the primary server. To update the other — backup — servers, the changes in the data are passed on to them.

The drawbacks of passive replication are not only the high load on the single server, since all updates must happen at this server, but also the difficulties when the primary fails. Whenever the primary server fails, a new primary server must be elected from the backup servers and must be communicated to the clients. This process is time-consuming and may be unacceptable in some time-critical processes. In case of such a failure there can be a certain inconsistency between the state of the primary and the backups. This will happen when updates of the primary server have not yet been passed on to the backups. The new primary server will not be up to date and the clients will have to compensate for this.

In *active replication*, also known as the "state-machine approach" the method effecting the change of data is executed at each server, ensuring the update.

The main drawback of active replication is the replication of the computation needed to achieve the changes to the data. A certain amount of computing power is wasted on needlessly calculating the result values. There also has to be a restriction on the methods which are executed: they may not use properties specific to each server, only the replicated data. If this is

not the case, the data at the different servers will not be consistent.

3.2.3 Consistency Requirements

When replication is chosen for high data access speed, and not for fault-tolerance, a choice has to be made for the consistency requirement [5]. Ideally, one would want the consistency to be as high as possible. But, as has been shown in 3.1.2, a high consistency requirement introduces a high amount of overhead. Therefore, a tradeoff has to be established using projected read/write ratios on the data and the speed of the hardware to achieve a balance of an allowed consistency and the speed with which the clients will have access to the data [20]. Because of the high impact of the consistency requirement on the replication process, choice of consistency requirement can be seen as the most important choice that must be made when implementing replication.

One element of possible consistency requirements warrants further study: in many cases a consistency requirement may impose a sequence in which incoming updates on the server must be applied [5]. However, it is possible that updates will not arrive on the server in this given sequence. In such cases, incoming updates will have to be held back and not be processed before a number of other updates have been received and processed. The hold back algorithm must have the *safety* and *liveness* properties: data must not be applied to soon and it may not be held back forever.

Take the USENET example. In many cases it is possible to see a reply to a posted message before the message itself appears on the local news server. This situation can easily happen in a configuration of three news servers, as illustrated in figure 3.1: Suppose the original message is posted on news server O. Shortly after the message is posted, O sends its new messages, which include the original message to a second news server R. Somebody reads this new message from R and posts a reply to it. A while later, R sends its new messages, which include the reply but not the original message, to a third server B. This third server now has a reply to a message it has not yet received. Only later, when the originating server of the message sends its updates to B, will B see the original message.

A new requirement on USENET could be that replies to posted messages are not visible until the original message becomes visible.

This requirement could be met by having a hold-back queue at each news server [5]. Updates from other news servers are not directly applied to the data; replies are put in the hold-back queue, they will only be removed from the queue and added to the data after the original message has been added to the data. This algorithm has the safety property: replies will not appear before the original message. Assuming no messages are lost, this algorithm also has the liveness property: replies will be removed from the queue whenever the original message has been added to the data on the

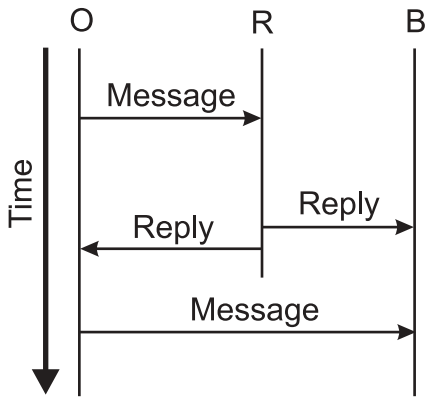

Figure 3.1: How a reply can appear before the original message in USENET.

server.

3.2.4 Inter-server Communications

A last important choice is inter-server communications [5] [20]. These communications are responsible for enforcing the chosen data consistency requirements. Choices regarding these communications are not limited to synchronous or asynchronous message passing between the servers. The timing of the communications, the content of these communications and the semantics of the communications must be established.

The timing of these communications must be defined. A possible choice is to let each update be passed on to all other servers whenever it is applied, either in a synchronous or in an asynchronous manner. Note that passing this update in a synchronous fashion will lead to a longer response time from a client's point of view. This is because the update to the server will not be completed before all other servers have also applied the update. Updates can also be collected into one block which is passed on only at certain points in time, or which is passed on whenever the server load is not too high.

What also must be defined is which data is passed on to the other servers. One possible solution is to send a complete copy of the local data to the other servers. This will lead to long communication times, increasing the overhead needed for the data consistency. It is better to keep the amount of data passed on between the servers as small as possible. In many cases only a subset of the data, containing the parts that have been changed, needs to be passed on to the other servers. Some cases allow a different approach, not unlike the active replication discussed above: instead of transmitting the changed data, transmit the operation which caused the data to change. Executing this operation on the other servers will result in these servers having identical copies of the data.

Until now, replication systems have been tightly linked to the choices of degree of replication transparency, type of replication (active vs. passive), the consistency requirement and the manner of inter-server communications. In many cases, this tight link need not exist, and the above options can be realized within the total design of a replication framework.

3.3 Replication Frameworks

Some general replication systems do exist, but they are mainly built to ensure high availability for the data [18][22]. Using a generalized approach for replication we will show that it should be possible to create a framework which can be applied both for high availability of the data and for fast data access by the clients, and which does not commit us to any of the choices in replication which we presented above.

The design of the framework is based on an abstract model for replication put forward in [5]. This model uses three abstract entities to describe replication: the Client, the FrontEnd and the ReplicaManager.

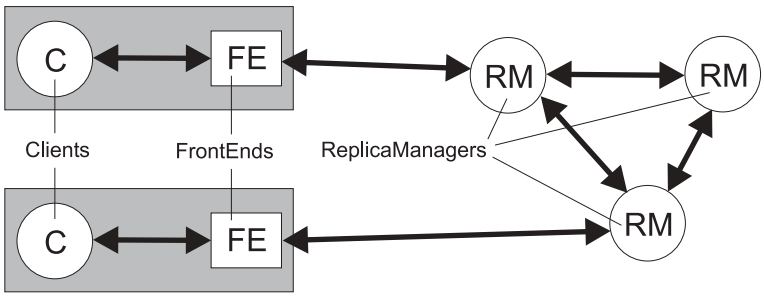

Figure 3.2: The elements in the abstract replication model

The *ReplicaManager* is the process which contains a copy of the data which is replicated and performs the read and write operations on this data. *Clients* utilize data from the ReplicaManager for their computations. The *FrontEnds* are responsible for the interactions between the Client and the ReplicaManager, ensuring replication transparency. ReplicaManagers communicate with each other to ensure the data consistency.

Although there has to be a strict one-to-one relation for Client and FrontEnd, the number of ReplicaManagers is not fixed. If there is only one ReplicaManager we have the single-server approach to data sharing, which we can consider as a special case of replication. We will not focus on this, but will assume a larger number of ReplicaManagers. Note that there is no strict upper limit, so we can easily have an amount of ReplicaManagers larger than the amount of Clients.

The location of Client and FrontEnd should be the same machine, so with respect to location and number, we will further refer to the Client-FrontEnd pair as the Client. Location and number of the ReplicaManagers is however not fixed. This allows us to place ReplicaManagers on separate, highly reliable, server computers, for high reliability, or to integrate the ReplicaManagers in the Client's process, for high speed access to the data.

We have now already hinted how this abstract model can be used for high availability of data or for high speed data access. Both cases can be handled easily.

High data access speed or data fault-tolerance

Consider the case of high speed data access. Here, the network distance between Client and ReplicaManager and the number of clients per ReplicaManager must be minimized. The best configuration will have a ReplicaManager communicate with just one Client and have both located on the same machine, within the same process. This immediately fixes both number and position of ReplicaManagers. Sub-optimal but still acceptable solutions are those where a number of Clients share a ReplicaManager, which is located within a small network distance of the Clients. Number and position of ReplicaManagers are not fixed, but should be chosen as to minimize network distance to their Clients, and, if possible, to reduce the number of Clients per ReplicaManager.

Now consider a setup for a high degree of reliability. Given a probability of failure $p \in [0, 1]$ for a ReplicaManager, the probability of simultaneous failure of all ReplicaManagers in the entire system is p^n with n the number of ReplicaManagers [5], assuming p is identical for all ReplicaManagers. So increasing the amount of ReplicaManagers will decrease the probability that the system will completely fail. Locating ReplicaManagers on computers with a high degree of reliability will decrease p, so it is advised to place them on highly reliable servers. Using active replication entails letting each Client communicate it's updates to all ReplicaManagers. Using passive replication requires selecting a primary ReplicaManager, with which all clients will communicate.

This shows that allowing the flexibility in location and number of ReplicaManagers in a framework will allow it to be instantiated both for high data availability and for fast client access.

Replication transparency

The abstract model defines a dedicated entity to ensure replication transparency: the FrontEnd.

As defined above, the FrontEnd is responsible for the interactions between Client and ReplicaManager. How the FrontEnd manages these inter-

actions will determine the degree of replication transparency. If the Client must make explicit calls to the FrontEnd, and perform some extra processing on the result of these calls to determine the required result, replication transparency will be very low. If the FrontEnd intercepts calls to replicated data within the Client and redirects them to the ReplicaManager without the client being aware of this, replication transparency will be high.

Active vs. passive replication

The choice of active or passive replication is determined by the communications between the FrontEnds and the ReplicaManagers and the interactions between the different ReplicaManagers.

For passive replication, every FrontEnd will communicate with one primary ReplicaManager. This ReplicaManager will update eventual changes to the backup ReplicaManagers.

For active replication, every FrontEnd will communicate simultaneously with all ReplicaManagers. The ReplicaManagers need not perform any interaction amongst themselves, as changes have been applied to all ReplicaManagers by the FrontEnd.

Consistency requirements and inter-server communications

The elements in the abstract model responsible for enforcing the chosen consistency requirement are the ReplicaManagers. Their responsibility can be divided in two parts: the manner in which the they interact with each other, and the manner in which they apply updates to their local replica.

If ReplicaManagers immediately pass on updates to other ReplicaManagers, the data consistency will be high. If ReplicaManagers do not immediately pass on data to other ReplicaManagers, the data consistency will be lower.

Recall the hold-back queue we have discussed in 3.2.3. This hold-back queue will have to be integrated in the ReplicaManager to determine when incoming updates are applied.

Conclusion

We have now defined an abstract model for replication, consisting of the abstract entities Client, FrontEnd and Replicamanager. We have seen how this model can be used to implement any of the choices in replication we have presented in 3.2 by defining how Client, FrontEnd and ReplicaManager interact, and how ReplicaManagers interact amongst each other.

Therefore the abstract model for replication can be used as a basis for a replication framework.

3.4 Conclusion

An important subject in distributed systems is how the data representing the problem at hand should be shared between the different computers. A number of strategies to share the data exist, one of which is replication. Replication is useful when fault-tolerance of or high access speed to the data is required.

Although there are a number of ways to implement replication, an abstract model provides a degree of flexibility which allows us to realize the choices of type of replication, consistency requirement and manner of inter-server communications.

As mentioned before, the purpose of the FrontEnds in the abstract model is to achieve replication in a transparent fashion for the Clients. Achieving replication transparency to the fullest: i.e. making a client totally unaware of the replication algorithm has been shown to be nontrivial [20]. A possible solution lies in the concept of separation of concerns, using aspect-oriented programming.

Chapter 4

Aspect-Oriented Programming

Current-day applications need to fulfill a large number of requirements, which makes developing these applications difficult for the programmer. We will present a principle which simplifies development of this software: separation of concerns. Using separation of concerns entails reasoning separately about subsets of the given requirements. As a possible technique to achieve separation of concerns we will also introduce aspect-oriented programming (AOP). To achieve a better understanding of the practical aspects of AOP, AspectJ will be presented as a case study.

4.1 Separation of Concerns

In many applications, the software has to meet a wide variety of requirements. Usually only a small subset of these can be considered as essential for the basic functionality, or the requirements for the basic algorithm, while other requirements deal with special concerns such as concurrency, distribution, persistence, ...

A typical manner in which programming languages handle these special purpose concerns is through the use of special programming constructs for these concerns. For example: in Java we have the keyword `synchronized` and the methods `wait` and `notify` to handle concurrency.

Writing code using these constructs often proves to be hard [11]. This also holds for understanding this code and for maintaining it. Having a program handle more concerns in this fashion, increases the difficulty of the produced code. This is due to the fact that the different concerns are mixed into the basic algorithm. If we were able to handle these concerns separately, not only at a conceptual level, but also at an implementation level, the ease of coding and maintenance would be greatly enhanced. This principle is known as *separation of concerns* [11].

The above problem can be restated in a different way: although the different concerns can be specified independently in an abstract fashion, integrating these concerns into final code is hard, as is extracting the original concerns from the produced code. This is largely due to the fact that although there is a loose coupling in the conceptual separation, this loose coupling does not hold true for the code integrating the different concerns.

Ignoring the difference between special purpose and basic concerns, a number of concerns have been explicitly named [11]: algorithm, data organization, process synchronization, location control, real-time constraints, persistence and failure recovery. This list is open-ended, newly 'discovered' concerns can be added to it any time.

A programmer trying to cope with these different concerns simultaneously will find this to become harder as the number of concerns increases.

Different special purpose concerns are often impossible to encapsulate in a single object. These concerns often have to be treated system-wide, it is impossible to add such a concern to the code without changing a large number of existing objects. For example: adding code handling the concurrency concern cannot be accomplished by simply adding a 'synchronization' object. For this to work, every method that needs to be synchronized will have to make calls to the 'synchronization' object, which clearly entails modifying the code. We can characterize this concern as "cross-cutting" the existing code [17] [16].

Were the different concerns not only separated at a conceptual level, but also have a loose coupling at an implementation level, e.g. in different objects for each concern, there would be a number of important benefits. Firstly, being able to handle different concerns completely independently leads to a higher level of abstraction. Secondly, since only one concern is handled at a time, the code is easier to understand. Thirdly, it can now become possible to reuse the code for certain concerns independently from other concerns.

One technique which has been proposed to achieve a clear separation of concerns is aspect-oriented programming or AOP.

4.2 Aspect-Oriented Programming

In aspect-oriented programming [17] [16] different concerns are termed as 'aspects'. These aspects are recognized to "cut across both each other and the final executable code" [16] and, although easy to work with conceptually, are hard to realize at implementation level. The code which integrates the different aspects is a "tangled mess of aspects" [16], which leads to a high level of complexity in the code.

These aspects must not only be reasoned about separately, but must also be implemented separately. To achieve this, AOP allows the programmer to express these aspects separately, in a natural form. Once these aspects

are programmed, a tool, called an Aspect Weaver, combines these different aspects into final executable code. Because the programmer will only be aware of the aspectual decomposition of the program, the program will be easier to write and to maintain.

Expressing the aspects in a natural form is made possible by using special-purpose aspect languages. These languages are specifically created to cover one aspect, thus allowing them to be expressed more easily. Some examples of aspect languages are languages for concurrency control [30], numerical accuracy (for mathematical problems) [17] and space optimization of data structures [14].

Once all aspects and the functional code of a program have been expressed, the aspect weaver will combine these into executable code. The aspect weaver is able to do this because it not only knows how each aspect can be transformed into code, but also how the different aspects relate to each other and how they should be combined.

A number of AOP systems have been proposed [14] [24] [30] [32], suggesting AOP is a viable solution to achieve a good separation of concerns.

4.3 Case Study: AspectJ

4.3.1 Introduction

To have a practical understanding of the workings of an AOP system, we performed some experiments with AspectJ. "AspectJ is an object-oriented language framework designed for facilitating the development and maintenance of concurrent and distributed applications. It uses the aspect-oriented programming approach to allow the code for the basic functionality of a distributed application to be written without having to explicitly deal with remote interactions and synchronization." (AspectJ Specification [32]) AspectJ is also covered in [30] and [23], but under a different name: D.

AspectJ is an AOP extension of Java, using three different languages: 'JCore' for implementing the core functionality of the application, 'Cool' for programming the thread synchronization aspect and 'Ridl' for programming the remote method invocation aspect.

An AspectJ program consists of a number of JCore classes, a number of aspects written in Cool and a number of aspects written in Ridl. Processing the whole results in a number of Java .class files which can be executed on a Java Virtual Machine. Because AspectJ is an extension of Java, an AspectJ program containing only JCore classes is a valid program. It will not have any thread synchronization nor any remote method invocation.

We used a prerelease version, which the authors describe as "in a pioneer user phase of development". The specification of the different aspect languages was complete, but not yet fixed, the weaver was able to weave programmes written in the aspect languages. Several implementation issues

of the weaver, mostly constraint checking for the aspect languages, still had to be taken into account when programming for AspectJ.

We will now briefly introduce JCore, Cool and RIDL.

4.3.2 JCore

JCore is the aspect language which describes the core functionality of the program, omitting synchronization and remote method invocation. JCore is Java V1.0 from which some constructs have been removed: the keyword `synchronized` and the methods `wait`, `notify` and `notifyAll` have been removed because their functionality is implemented by Cool. Because the weaver bases its weaving on method names, overloading of methods in JCore has been disallowed. Overloading of constructors is permitted.

4.3.3 Cool

A Cool program describes a set of Coordinator modules or *Coordinators*. Coordinators are 'helpers' for a class: they take care of thread synchronization over the methods of a class. Coordinators are not classes and cannot be instantiated. Coordinators are associated with instances of JCore classes, on a name basis.

Classes are not aware of their coordinator, but coordinators have knowledge of the class they coordinate. More specifically, coordinators coordinating a class C are aware of all the methods of C, the non-private methods of the superclass of C, all variables of C and the non-private variables of the superclass of C. This awareness does not allow a coordinator to invoke a method nor to change the value of a variable. In other words, a coordinator may not modify C's state.

The interaction between a class and its coordinator is defined by the following protocol:

1. Within a thread, a method invocation is requested on C.
2. The request is passed on to C's coordinator.
3. The coordinator checks the exclusion constraints for the method. If any of these constraints is not met, the request is suspended until all constraints are met. Before the method is executed the coordinator performs the on_entry statements for the method. (The on_entry statements will be discussed later.)
4. The method is executed by the object.
5. The method invocation return is passed on to the coordinator.
6. The coordinator executes its on_exit statements for the method. (The on_exit statements will be discussed later.)

7. The method invocation returns.

All synchronization aspects are handled by this protocol, therefore the smallest units of synchronization are methods.

Describing a coordinator in Cool also implies describing its coordination strategy. A Coordinator declaration includes an number of *selfex* and *mutex* sets and a number of *MethodManagers*.

- Methods included in a selfex set are self-exclusive: each method in a selfex set can only be executed by at most one thread at a time. A self-exclusive, directly or indirectly, recursive method, however, will not deadlock.

- Methods included in a mutex set are mutually exclusive: if a method in a mutex set is executed by a thread, the other methods in this set cannot be executed by another thread.

  Mutual exclusion of a method M does not imply self-exclusion: while M is executed by a thread, other threads are also allowed to execute M.

- MethodManagers allow further coordination between methods, using guarded suspension of threads. Guarded suspension adds extra constraints on method invocations. These are expressed as a boolean expression of the internal state of the coordinator. Not only must the selfex and mutex constraints be met, but also the given boolean expression must return true. If not, the thread will be suspended until all constraints are met.

  As soon as a thread has the right to execute a method, but before it starts executing, the coordinator may update its' internal state using a number of statements specified in its' MethodManager. These statements are called the on_entry statements. As soon as a thread has finished executing a method, the coordinator may again update it's internal state using a number of statements specified in its' MethodManager, called the on_exit statements. Recall that modifying the state of the object which is coordinated is forbidden.

4.3.4 RIDL

The Remote Interface Aspect Language, a.k.a. RIDL, defines Portal modules or *Portals*. Portals are associated with classes on a name basis. There is at most one portal per class. Portals are 'helpers' for a class: they take care of method invocations between different address spaces. We define different address spaces as different programs, regardless of the fact if they are running on the same computer or on a remote computer.

As for coordinators, classes are not aware of their portal, but portals have knowledge of the class to which they apply. A portal for a class C is aware of all the methods of C, the non-private methods of the superclass of C, all variables of C and the non-private variables of the superclass of C. A portal cannot invoke a method of C or change the value of a variable. Put differently, changing C's state is not allowed.

Portal declarations identify classes whose instances may be referenced from another address space, and for these classes, which methods may be executed from a remote address space. These instances and methods are respectively called remote objects and remote methods.

Portals are not classes and cannot be instantiated. A portal is automatically associated with an instance of the class to which it applies if the instance is made available as a remote object to other address spaces. Making such an instance available is also known as "exporting an object".

Whenever an object is exported, the interaction between the object O and its portal is defined by the following protocol:

1. From a different (remote) address space there is an invocation request for a method on O.
2. The invocation request is passed on to O's portal.
3. The method is processed according to its declaration in the portal: parameters are extracted to O's address space according to the method's declaration in the portal.
4. The method is executed on O. This execution may be affected by O's coordinator, if it exists.
5. The method invocation return is passed on to the portal.
6. The return is processed according to the method's declaration in the portal.
7. The method invocation returns and the return value is passed on to the remote address space.

Extracting parameters from the remote address space and making them available to the local address space can be done in two ways: passing by reference or by copy.

When passing parameters by reference, the objects are exported by the remote address space and a reference to the objects is passed on to the local address space. Recall that to export an object, a portal must be declared for the class. This implies that for each parameter passed by reference, a portal must be declared, which will define the remote methods that may be executed on these parameters.

When passing parameters by copy, copies of the objects are passed on to the local address space. These copies consist of copies of the object's variables of primitive types and recursive copies of the object's variables of a reference type. These copies need not be complete, it is possible to skip some variables and to pass some variables by reference, as shall be shown later. Note that there is no consistency enforced between the two objects: any changes made to the copies are not applied to the original object, nor are any changes made to the original applied to the copy.

Portals pass all primitive types by copy, therefore directives for primitive types and variables of primitive types are not allowed in Ridl code.

A Portal declaration includes a list of all remote methods for a class, including passing modes for parameters and return type. For each remote method, it is possible to specify if the parameters which are not of a primitive type should be passed by copy or by reference. Non-primitive types can also be selectively passed on, allowing a finer degree of granularity for the passing parameters. For these objects, specified instance variables can be passed by copy, others by reference, and instance variables can be skipped i.e. not passed at all.

4.3.5 Results

We have performed some experiments to get the 'feel' of the system. The system performed as specified, although there were still a number of 'implementation notes' to be kept in mind.

Some of these notes were quite trivial, such as the restriction that names of classes should start with a capital letter, other were more severe. One of these merits special attention: it is unavoidable that every remote method invocation may throw an exception. This can be caused by factors such as the remote computer crashing, the network failing, ... This problem is not addressed by Ridl. A small note reads that these exceptions must be caught explicitly in the Cool code. This clearly breaks the separation between the different aspects. Although it might seem easy to solve by letting the weaver add a default exception handler for each remote method invocation, this solution is not optimal. Not each failed remote method invocation has the same impact on the code; some failures may be recovered from gracefully, while others may not be recovered from. The programmer should be able to specify the exception handler for a remote method invocation. This could be done by e.g. extending RIDL with an "errors" clause. In this clause a number of statements would specify the exception handler for the remote method.

Separating the code in these three concerns made implementing concurrent, distributed applications a lot easier. Development time was reduced not only by the ability to reason separately about the different aspects, but also by the ease in which these different aspects could be expressed. Although

the programmer may well have a grasp on the aspect she wants to add to the existing code, in many cases translating this abstract idea into actual code and inserting it into the already existing code proves to be nontrivial. Having expressive aspect languages and the aspect weaver allows the programmer to express her abstract idea much more easily, which shortens the development time.

The ideas of Coordinator modules and Portal modules, combined with their protocols bear great resemblance to meta-level objects and the meta-object protocols, with the base-level being the JCore program. This could lead us to believe that AOP is no more than meta-level programming. However, where composing different kinds of meta-level objects in to one meta-level object for each object is not resolved in many meta-level systems, AOP explicitly addresses these issues. Also, the resemblance between AOP and meta-level programming will decrease as more aspects can be expressed in different aspect languages. A single aspect will no longer dominate the system, making it hard to identify one aspect as base-level and other aspects as meta-level.

A final concern is the level of checking done by the weaver. We deliberately introduced semantic errors in Cool and RIDL code to see how the weaver coped with them. Such errors were e.g. mentioning methods or variables which did not exist in the cool code. In many cases the weaver did not report an error and went on to produce erroneous Java code. Although later versions of the weaver will introduce more error-checking this raises an interesting point. If an error is produced, by an incomplete or faulty weaver, or maybe because of the inability to fully verify the aspect code, how can the programmer debug the produced code? The produced code will contain elements from each aspect language, so the programmer must trace back the errors to the origin in the aspect language or in the weaver. Code generated by the AspectJ weaver proved to be fairly easy to debug, but if the weavers become more complex, this might not be the case.

4.4 Conclusion

Separating large applications into different concerns eases building and comprehension of these applications, due to the ability to reason separately about these concerns. Aspect-oriented programming allows the programmer to express each concern or aspect in a natural form, which makes it easier to reason separately about each concern. AOP is an emerging technique which shows a lot of promise, but some work still needs to be done to further develop the field.

Having some experience with separation of concerns and with AOP we can now envision how we can achieve replication in a transparent fashion. Using the AOP approach for replication transparency, we can consider repli-

cation as an aspect.

Chapter 5

Replication as an Aspect

To achieve a large degree of replication transparency in our framework for replication, we will now develop an AOP approach to replication.

First we will present a general introduction to how a framework for replication can be set up, and how the AOP methodology fits in this framework. Next we will perform a general analysis of how replication can be seen as an aspect, define the aspect languages for replication and discuss the implementation of the aspect weaver.

5.1 A Framework for Replication

Recall the abstract model for replication we introduced in 3.3. This model contains three abstract entities, the Client, the FrontEnd and the ReplicaManager. The ReplicaManager is the process which manages a number of copies of the data, called replicas, the Client is the code which uses this replicated data, and the FrontEnd is a special entity ensuring replication transparency.

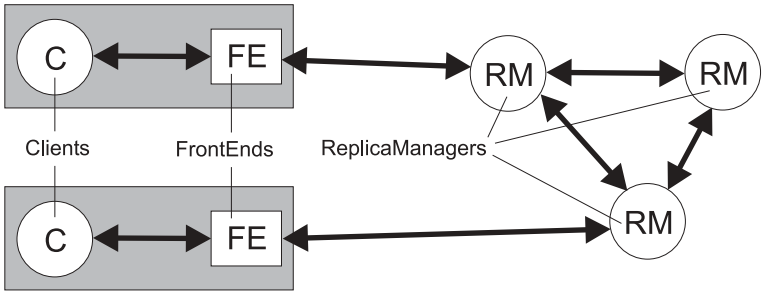

Figure 5.1: The abstract replication model

As in AspectJ we can make an AOP extension to Java. The base algorithm of the program using replication will be implemented in a variant

of Java and the replication aspect will be specified in a separate aspect language.

In 3.3, we have also argued why the abstract model can be used as a basis for a replication framework. We will now discuss how the three abstract entities and the AOP approach to replication fit together to form the basis of our framework.

Because the ReplicaManager is responsible for containing the data and managing accesses to it, the ReplicaManager can be considered as being a server application, with which will be interacted through a network.

Using the AOP approach, the base algorithm uses replicated data, without being explicitly aware of this. In the abstract model, the Client uses replicated data, without being aware of this. This means we can consider the base algorithm to be equal to the Client.

Using AOP, the replication aspect language determines how the base language interacts with the server to gain access to the replicated data. In the abstract model for replication, the FrontEnd takes care of communications between the Client and the ReplicaManager, ensuring replication transparency. From this analogy, we can deduce that the code produced by the aspect weaver can be considered to include the FrontEnd.

These three elements form the basis of our replication framework. The rest of this chapter will discuss how the AOP approach will be used to generate the FrontEnd. The next chapter will discuss how the FrontEnd will interact with the ReplicaManagers and how the ReplicaManagers can be implemented to build a framework for replication.

5.2 Analysis

To allow replication to be seen as an aspect, we will first analyze what will be replicated and how this can be achieved. Linked to this are the requirements for the aspect languages and the requirements for the weaver. These requirements will be put forward whenever the analysis reveals them.

5.2.1 Network Interactions

In most cases when using replication, a program will have a network link to its replicamanager, instead of a 'regular' association of an object with another object within the same program.

This means that interactions between the client and the replicamanager will usually have to occur over the network. A high-level approach to these interactions are remote method invocations. These remote method invocations have been introduced previously, in 4.3.4. Recall that when using remote method invocations a certain object exports its methods so they can be called by other objects over the network.

Java provides a standard solution to perform these method invocations over the network, known as RMI. To use RMI a large amount of work is required, and a number of alternative solutions exist which require less work by the programmer. We have chosen to use one of these solutions: Objectspaces' Voyager [12]. To use Voyager the programmer only needs to run a tool once for each class whose methods will be exported over the network.

This tool, vcc, will create a proxy class for the class whose methods are exported. If a client wishes to use a remote object, it needs to instantiate the objects' proxy with an extra argument which indicates the network location of the remote object. Now all method invocations on the remote object need to be invoked on the proxy. Voyager will ensure that method invocations on the proxy will be executed on the remote object.

This defines not so much as a task for the weaver as a property of the code generated by the weaver: whenever remote interactions with replicas or replicamanagers are needed, the weaved code will use Voyager to realize these remote method calls.

5.2.2 Replication of What?

To determine how we should approach replication, it is crucial to first analyze what will be replicated. Because we are working in an OO language, the maxim 'everything is an object' seems to give the answer. What we are replicating is an object. But an object is more than just data, objects also contain behavior. So must this behavior be replicated? This is the issue of active versus passive replication which has been discussed in 3.2.2.

Recall that the choice of active versus passive replication is important when replication is used for fault-tolerance. When using passive replication, all method calls on an object are executed on one, primary, replicamanager, and that changes are passed on to all other, backup, replicamanagers. When using active replication, method calls are executed simultaneously on all replicas, negating the need for updates to be passed on between replicamanagers.

Because the framework will not only be used for fault-tolerance, but also for high speed data access, we have chosen to use a variant of passive replication, which can be used for both purposes. Note that this choice does not rule out the possibility to later add active replication. The variation on passive replication is simple: instead of executing the method on the replicamanger, the method will be executed on the client, and changes to the data will be passed on to the replicamanager. There is no primary or backup replicamanager, a client may use any replicamanager as it sees fit.

Now what needs to be replicated is purely the data contained in the object, which is contained in its instance variables (also known as its *fields*). Note that these fields may either be Java primitive types or objects. The Java primitive types can be considered purely as data; they can either be

read or written. In case the variables contain objects (which we will call *sub-objects*), the sub-object's methods could be executed, and the sub-objects might themselves contain sub-objects which might contain sub-objects and so on ...

Method Executions

When a method is executed on a sub-object, the method will not be executed on the replicamanager, but on the client. Because sub-objects are fields within the replicated object, at some point they will be copied from the replicamanager to the client whenever a read access occurs. Amongst these read accesses are method calls on the sub-object. For the method to be executed, its code will have to be read by the virtual machine, so the sub-object containing this code will have to be copied from the replicamanager to the client. This leads to a problem whenever a method call changes the sub-objects state.

Method executions on the sub-objects may or may not alter the sub-objects' state. Should the sub-objects' state change, this change will have to be applied to the replicas of the object which is replicated. However, we have not defined how changes to these sub-objects are applied to the replica. The difficulty lies in determining when the sub-object is being altered by the method and when it is not.

We could compare the object after the method execution with a copy made before the method execution. For this, each replicated sub-objects must be cloneable, i.e. implement the standard Java `Clonable` interface and the sub objects' `equals` method must be defined in a relevant fashion. The sub-object must implement `Clonable` to ensure the `clone` method creates a deep copy, and not a pointer reference to the same object. Also, the standard implementation for `equals` is not adequate because it only performs pointer comparison of two objects. This means that an object which is cloned from an other object will not be considered equal by the default `equals` method.

These requirements will require the programmer to adapt the code for replicated sub-objects, which will greatly reduce the replication transparency for the object. Also, making the comparison after each method call on a replicated sub-object will add a certain overhead to each of those method calls. This overhead could range from small to extremely large, depending on how much data is be compared in the `equals` method.

To avoid these problems, the replicated sub-object could always be written back to the replicas after a method call is executed on a sub-object. However, this would be a cure worse than the disease. Always re-writing the sub-object, even when it would not be necessary, will significantly lower the read/write ratio on the data, and increase the total overhead for replication. This would in turn slow the replica system down.

So a distinction must be made between methods on replicated sub-

objects which alter the sub-objects' state and methods which do not. This distinction would then need to be expressed in an aspect language, and the aspect weaver would then need to be able to weave that language.

However, consider why these refinements to sub-objects should be made. If a method call on a sub-object is performed, and this method call alters the sub-objects' state, should the sub-object not be considered as a replicated object in its own right? We feel that this is indeed the case, so we have chosen not to implement these refinements.

Note that it can be argued that this choice decreases the replication transparency because the programmer would need to consider the operations which are performed on replicated sub-objects. However we feel that this does not necessarily reduce replication transparency, as the programmer can examine the method-calls on replicated sub-objects when she is implementing the replication aspect. In other words, reasoning about method calls on replicated sub-objects need not be done when implementing the base algorithm, but can be done when reasoning about the replication. As this does not impact the base algorithm, replication transparency is not affected.

So to avoid the method execution problems, we will treat objects similarly to primitive types: they can either be read or written. We will allow method calls on these objects, but will not actively support them. Method calls will occur on a local copy of the sub-object, not on the sub-object on the replica managers. Should these method calls change the internal state of the sub-object, these changes will be lost.

Graphs of Objects

Now consider what should be done when a replicated object is (an element of) a graph of objects i.e. one or more of its fields is a sub-object in the same graph. To correctly replicate the object, a traversal must be made through the graph to determine which part is reachable. This part of the graph must then be replicated, because it is a part of the objects' state. In the traversal care must be taken with respect to loops in the graph. These loops may not cause the traversal to be caught up in an infinite loop.

Once the traversal is made, the different objects on the subgraph must be encoded into data which can be transferred over the network. This will require access to the structure and to each field of all objects in the subgraph, so the complete state can be read out. This requires a nontrivial algorithm to serialize these objects into data.

Luckily, Java contains a standard object serialization mechanism [25] which handles these problems. To be able to use this serialization mechanism, all replicated sub-objects and their respective sub-objects must be serializable. To be serializable, an object must implement the `Serializable` interface, which contains no methods, and the fields of the object must themselves be serializable. This means the fields of a sub-objects must be either

primitive types or objects which are serializable. In the rare case that some of the sub-objects' fields can not be made serializable, the sub-object can define `writeObject` and `readObject` methods, which are then responsible for, respectively, serializing the state of the object into data and recreating the state from some data.

Rendering sub-objects serializable might require adapting the code of some replicated sub-objects, which will reduce the replication transparency. However we can consider this reduction to be minimal, because in most cases for an object to be serializable, no structural change needs to be performed. No methods need to be added or changed, nor do any variables need to be added or changed.

Once each replicated sub-object is serializable, the Java serialization mechanism will resolve conflicts in the replicated object's graphs, and transporting sub-objects over the network will succeed. This will allow these sub-objects to be replicated as part of a replicated object.

Conclusion

Should the programmer want to replicate fields which contain objects, she should insure the replicated sub-objects are serializable, and regard them purely as data. This data can be copied locally, and if needed, changed locally and written to the replica. If the sub-objects will be used and changed frequently, the programmer should replicate these sub-objects and not keep them as replicated sub-objects of a replicated object.

We can now define a *replicated object* as an object containing a number of fields which will be replicated. A *replica* for an object is the collection of the *replicated fields* of an object. The collection of all replicas on the replica-managers for a given replicated object is the replicated objects' *replicagroup*.

5.2.3 Replication: How?

Replicating an object is replicating a number of its fields, which we treat as primitive types. These variables can either be read from, or assigned to. Since these fields will no longer be contained within the object, but in its replicagroup, access to the fields will have to be changed into accesses of the fields on the appropriate replicas within the replicagroup.

In a number of cases, it might not be necessary to replicate all fields of an object. In these cases, replicating only the fields which need to be replicated will speed up the system.

To provide this greater control over replicated objects, a means must be provided to specify which fields of the object must be replicated and which fields must not. This will be the role of the aspect language; using the aspect language, a programmer will be able to specify which fields of a class must be replicated.

This defines the first task for the weaver: the weaver must use the aspect language to determine which fields of a class must be replicated, and for these fields it must modify all reads and assigns into reads and writes on the appropriate replicagroup. The weaver must also create the classes of the replicas which will make up the replicagroup.

Field Visibility

The visibility of a field within an object is determined by its modifiers, as shown in 5.2. Should we allow `public` fields to be replicated, all objects in the application will be allowed to access these fields. This means all classes of the application will have to be weaved, so the accesses to the replicated fields must be modified into accesses to the replicagroup.

Whenever an class is added to the application, or a class is changed, this class will need to be weaved to ensure the replication aspect is weaved correctly into the entire application. This will need to be done regardless of the fact that the class accesses replicated fields or not. This implies a lot of work for the weaver, resulting in slow and awkward development.

Field	Visibility
`public`	all classes
`protected`	Class' subclasses, all classes in Class' package
`default`	all classes in Class' package
`private`	Class

Figure 5.2: Visibility of a field in an object of class Class.

The same argument can be made for `default` and `protected` fields, so to minimize weave time and the amount of weaves needed while developing, we have chosen to only allow replication for `private` fields.

This does reduce replication transparency somewhat, because it requires the programmer to ensure that all replicated fields are private. Although making all the fields `private` can be considered as a good programming style, decoupling interface from implementation, the programmer may not have done so, which will require a number of rewrites in the code. Should the programmer want to replicate non-private fields, these fields can be made private and be accessed through accessor and mutator methods.

5.2.4 Naming and Location

In a system where multiple replicagroups are active, a replicagroup must be identified uniquely to be able to access it. Some sort of 'naming service' must allow the accesses of replicated fields to be executed on the correct replicagroup based on a name given for this replicagroup.

Example

Consider the following example: a Counter class, containing an integer field `count` and a method `add()` which adds 1 to `count` needs to be replicated, to e.g. be able to count the number of bottles of beer produced by different production lines in a beer factory. One Counter object would be replicated and each production line would increase the replicated Counter object whenever a bottle of beer is produced. Now suppose the Counter should not count the overall number of bottles produced, but a different Counter should exist for each type of beer (say Pils, Kriek and Witbier). How will the production line be able to distinguish between the different counters? This must be possible to make sure each line only updates the product counter of its own kind of beer, and not of some other kind. In other words, lines producing Pils should not increase the counter for Kriek or Witbier and vice-versa.

One possible solution would be to create a different Counter class for each type of product, say Pils_Counter, Kriek_Counter, and Witbier_Counter. This is not a workable solution in many cases, because of its inflexibility. Each time a new kind of beer is added to the range of products manufactured by the production lines, the counter program will need to be partially reprogrammed, by implementing a new Beer_Counter.

A better solution would be to allow different instantiations of the Counter class on the replicamanager, where each instantiation represents a different kind of beer. However, how can we now distinguish the different Counter replicated objects? How will the different production lines know which counter to increase?

To be able to distinguish the counters, each counter should have an unique identifier, say the type of beer it is counting. Using this unique identifier, the client can ensure it will interact with the correct replicated Counter object.

The need for an unique identifier is not unique to replication, all forms of data sharing in distributed systems need to be able to uniquely identify a part of the data, so it can be correctly read or written. The algorithm which establishes the correct identity based on a given 'name' is usually called the *naming service*. Providing the name for a replica is also known as 'naming' the replica.

Naming

As we have seen in our example, the identity of a replicagroup cannot be determined per class, because different objects of the same class may be replicated at the same time, each representing a different replica.

So providing the name of the replicagroup for an object and finding it must be done at run-time. The algorithm which provides the name must know what the replicated object will be used for, so it can provide the

name of the correct replicagroup. The only algorithm which 'knows' for which purpose the replicated object is created, is the base algorithm. In our example, only the production lines will know what kind of beer they are producing, by e.g. scanning the bar code of the bottles that are produced.

This leads us to require that naming the replicagroup has to be performed by the base algorithm, because only this algorithm knows what the replicated object will be needed for. All the objects used by different clients for the same purpose will then have the same replicagroup, ensuring correct data sharing. In our example, production lines producing Pils should share the counter for Pils amongst themselves, production lines for Kriek should share the counter for Kriek, and lines for Witbier should share the Witbier counter. To achieve this, each production line producing Pils, will increase the Counter named "Pils", the lines producing Kriek will increase the counter "Kriek", and the lines producing Witbier will increase "Witbier".

Note that the naming of the replicagroup from the base algorithm has serious consequences for the replication transparency: the replication aspect is no longer completely separated from the base algorithm. The base algorithm still has to contain code for a part of replication, namely providing the name of the replicagroup, which excludes complete replication transparency.

Solutions which contain no explicit naming can be imagined, but cannot prevent the need for a replicated object to be associated with a replicagroup. The association could be based on the class of the replicated object, but this excludes the possibility to have different objects of the class to be replicated.

Inferring the name from the state of the object when it is instantiated would be a possibility, but would require a complex aspect language able to capture a wide variety of inference rules. But even with these inference rules, there can be cases where the correct name for a replicated object cannot be deduced. The counter example would be such a case: all counters would be initialized to zero, even though they represent different kinds of beer.

Therefore we have chosen not to develop this approach, and we keep our original requirement: The base algorithm will need to specify a name for each replicated object when it is instantiated. This name will be used to associate the replicated object with a replicagroup.

Non-existent Replicagroups

Now consider what should happen if there were no replicagroup for the replicated object. This will happen if the replica group has not been created yet, or if the programmer provided a wrong name for the replicagroup.

We have chosen to consider a missing replicagroup not as a mistake by the programmer, but as an indication that the replicagroup has not been created yet. A possible option in handling these missing replicagroups would be to produce some kind of error. However, now we will need to be able to create the correct replicagroups to allow replicated objects to be used.

This would require a special construct which would create new replicagroups when these are needed. This construct would then have to be called from the base algorithm, because only the base algorithm is aware of the name for a replicated object and therefore is the only element able to know if a replicagroup exists for a replicated object.

But this would mean that the base algorithm would first need to keep track of all existing replicagroups. This is needed to verify if a replicated object can be associated with a replicagroup. If this is not the case, the base algorithm will need to create the replicagroup, to ensure the replicated object can be instantiated.

Using this entire construction clearly decreases the replication transparency. To avoid this, we have chosen an other option. If in cases of a missing replicagroup, the replicagroup is created on the fly, the client will not need to explicitly create new replicagroups when these are needed.

This solution removes the need for explicit managing of replicagroups from the base algorithm, thus providing a greater degree of replication transparency, which is one of our goals. Therefore we have chosen to create replicagroups on the fly whenever they are named.

Location of Replicagroups

To perform accesses to the replicagroup, not only must the group be named, but also a network location for the replicamanager to be used for this group must be given. The location of the correct replicamanager depends on the location of the program in the network. It is not advisable to hard-code this location into the program, as it would lead to a separate build for each copy of the program. Each copy needs to be able to determine at run-time which replicamanager it should use. The easiest way for this would be to provide a (list of) replicamanagers in a configuration file which can easily be changed for each copy, without requiring a rebuild.

Conclusion

We have now defined a third task for the weaver: The weaver must ensure that whenever a replicated object is instantiated, it is associated with the named replicagroup for the object, which may have to be created, and with the correct replicamanager, all according to a configuration file.

5.2.5 Initalization

One important factor which can easily be overlooked in replication is initialization of the replicas. Whenever a replica is created, it will have to be initialized to certain values.

The case for initialization of the replica is similar to naming the replicagroup, which we have discussed in 5.2.4. Initialization of the replica should

be done from the base algorithm, because only the base algorithm 'knows' what the replica will be used for, and thus knows the relevant initial values. This is what happens in the objects' constructor: the initial values for fields of the object are set, according to what the object will be used for.

Now consider when the initial values should be written to the replica. Clearly this should only happen immediately after it has been created, certainly not after the replica has been used by a client. However, because replicas are accessed in a transparent fashion by the base algorithm, it is not aware of their existence. Therefore, the algorithm can not write the initial values to the replica only when it just has been created. Instead, every time a client will instantiate a replicated object, the initial values for its fields will be written to the replica, essentially re-initializing it.

To avoid this unwanted re-initialization, initializing replicated objects must be considered with some care. There are two possible options to perform this initialization: The first option would be to create a special initialization program, which explicitly initializes the replica. The second option would be to let the program check if the replica contains appropriate values and, if not, assign correct values to the replicated fields.

Unfortunately, this need to re-think initializers for the replicated fields further reduces the replication transparency. A possible solution would seem to be to consider the initializers in the replicated fields' declarations and the assignments in the objects' constructors as initializers for the replica which need only be executed when the replicagroup is created. However this somewhat alters the semantics of a constructor, as some initializations in the constructor will only be performed if the replicagroup does not exist. This implies that the constructors of the replicated object should be reviewed to evaluate the impact of these changes.

So in either case, how the replicated object is initialized should be considered with great care. Therefore we have chosen not to alter the semantics of a constructor, and to always perform assignments in the constructors of replicated objects.

5.2.6 Error-Handling

Accesses to a replicamanager to read values from the replicagroup or to update values in the replicagroup, will usually occur over the network. As we have seen in 4.3.5, these network accesses can fail and throw an exception.

To achieve a high degree of replication transparency, these errors should be caught in the FrontEnd and not in the Client program. We could provide only a default error-handler, but this is clearly not a good solution. The severity of being unable to access the replicas will be different for different types of replicated data. The programmer will want to specify different exception handlers for different kinds of replicated data, so she can take appropriate action. These actions will also depend on what kind of application

the programmer is writing. So providing only one, or a limited number of default error handlers, should be avoided.

The straightforward solution would be to add to the replication aspect language a construct which allows the programmer to specify exception handlers for the network accesses. However, error-handling could itself be considered as an aspect, since it is a special-purpose concern in the code, and error-handling code tends to be interspersed throughout all the code of the base algorithm. Therefore it makes sense to specify the error-handling in its own aspect language. At the moment only the errors caused by the replication aspect will be handled by the error-handling aspect, but it should be straightforward to later add other sources of errors to the error-handling aspect.

Let us now concentrate on when these errors can occur. Since the only accesses to the replicated data are either reads or writes, the only moments at which errors can occur seem to be at reads or at writes of the data. This overlooks one type of error: the error which might occur when the client first tries to access the replicamanager to locate the correct replicagroup. So these three types of errors need to be handled, errors when first accessing the replicamanager, errors while reading the data and errors while writing the data.

Whenever these errors occur, the actual source of the error can be one of many; the network might be faulty, the replicamanager may be inoperative, ... In Java different kinds of exceptions are thrown for these errors, so it should be possible to specify an error-handler for each kind of exception which might be thrown.

To cover the combination of kinds of operations which might throw an exception, and the kind of exceptions thrown, the error handling aspect language should enable the programmer to specify an exception handler, based on a type of exception, for each kind of operation (read, write and contact).

The final task of the weaver would then be to add these exception handlers to the code responsible for accessing the replicas.

5.2.7 Conclusion

We have now determined the different requirements for the aspect languages and for the weaver.

1. The replication aspect language should provide a means to specify which fields of an object must be replicated.

2. The error-handling aspect language should provide a means to specify exception handlers for the errors which might occur.

3. The weaver must create a link between the replicated object and its replicagroup, modify all accesses to replicated fields to accesses to the replicagroup, and add the error-handling code to these accesses.

Now the requirements for the languages and for the weaver have been described, we can specify each language and discuss how the weaver will combine them.

5.3 The Aspect Languages

We will now discuss the three aspect languages Jav, Dupe and Fix.

As we define each aspect language, we will implement the corresponding aspect of the counter example we introduced in 5.2.4.

This distributed counter is a replicated object of different clients, and each client will increase the counter when it sees fit. This counter could be used e.g. to count the number of items produced by different assembly lines of a factory.

5.3.1 Jav: The Base Algorithm Language

The base algorithm language is the language which will be used to describe the algorithm for the clients, excluding the replication aspect. Because we are building an extension to Java, our base algorithm language must be a variant of Java.

The base algorithm language, Jav, is equivalent to Java1.1, but excluding interface specification. This because interfaces do not specify instance variables, so interfaces can not be replicated. Should the base algorithm require interfaces to be specified, these interfaces can be specified in Java, but will not be weaved.

A requirement for each replicated object is that all the objects' constructors have an extra formal parameter `String repl_id`. This extra parameter will be used to associate the replicated object with its replicagroup.

Due to a number of implementation issues for the weaver, a number of extra constraints have been introduced: per file only one class can be replicated, local variables in a method or block may not have the same name as a replicated variable and replicated variables may not be accessed using a `this` prefix. Also a number of method names and variable names are restricted, let X be a number between 0 and 1000 and VAR a name of a replicated variable. Method names init_X_vars, buildX_Proxy, set_X_VAR and get_X_VAR, may not be used, variable names rep_X_proxy are not allowed. Note that the fact that X lies between 0 and 1000 does not imply a limit on the number of replicated variables, this number is used for internal purposes and does not impose any limit whatsoever.

Example

The Counter's base algorithm, as specified in the Jav file is fairly straightforward:

```
public class Counter
{
   private int count;

   public Counter(String repl_id)
   {
      System.out.println("Counter started on "+repl_id);
   }
   public void add()
   {
      count = count + 1;
   }
   public int read()
   {
      return count;
   }
   public void reset()
   {
   count = 0;
   }
}
```

Whenever the counter is instantiated, it will print this out on standard output. The counter can be incremented, read out and reset. Note that `count` is not explicitly initialized. We rely on the default initial value of int, which is 0.

5.3.2 Dupe: The Replication Aspect Language

The replication aspect language, Dupe, specifies which fields of the replicated object need to be replicated and which fields need not be replicated.

Syntax

The abstract syntax of Dupe is quite simple:

```
DupeProgram :
   "Replicate" Classname
   "{"
       Field* [Default]
```

```
   "}"

Field :
   "field" Fieldname ("replicate" | "skip") ";"

Default :
   "default" ("replicate" | "skip") ";"
```

The class `Classname` must be replicated, fields specified as `"replicate"` will be replicated, fields specified as `"skip"` will not be replicated. The `Default` production allows to set a default replication mode for fields which are not specified. If no default replication mode is given, the default `"skip"` mode will be used.

Example

In our distributed counter, the field `count` will need to be replicated to achieve the distributed counter. The following Dupe file ensures this:

```
Replicate Counter
{
   field count replicate;
}
```

5.3.3 Fix: The Error-Handling Aspect Language

The error-handling aspect language, Fix, specifies exception handlers. At the moment only exception handlers for errors occuring during replication operations can be specified. It is possible to extend the language so exception handlers for other kinds of exceptions can also be specified.

Syntax

We now present the abstract syntax of Fix, annotated with some semantics:

```
FixProgram :
   "Replicate" Classname
   "{"
       (Contact | Field)*
   "}"
```

Exception handlers for the class `Classname`'s replication operations will be specified. These exception handlers can either be for errors occurring when contacting the replicamanager, or while accessing a field.

```
Contact :
   "contact" "(" ExceptionType ExceptionName ")"
   "BEGIN"
      Statements
   "END."
```

If an exception of type `ExceptionType` is thrown while contacting the replicamanager to link the replicated object to the correct replicagroup, it will be given the name `ExceptionName` and the `Statements` will be executed. These statements can refer to the exception using the given `ExceptionName`, these statements will be executed as if they were the method body of a method of the object, meaning they have full access to all the object's variables and methods. These statements may not access any replicated variables, because this will often lead to an infinite loop of errors.

```
Field :
   "field" Fieldname
   "{"
      (ReadFix | WriteFix )*
   "}"

ReadFix :
   "read" "(" ExceptionType ExceptionName ")"
   "BEGIN"
      Statements
   "END."

WriteFix :
   "write" "(" ExceptionType ExceptionName ")"
   "BEGIN"
      Statements
   "END."
```

The `ReadFix` and `WriteFix` productions are similar to the `Contact` productions, here exception handlers for exceptions thrown while reading the data, respectively writing the data are specified.

Recall that in Java assignments have return values, namely the value which has been assigned. Should the exception handler want to return a value for the instance variable which has been read, or has been written, the handler should use the `return` statement to do so.

Regardless of returning a default value, due to the strong checking of the Java compiler, the `Statements` which specify the exception handler, must end with a `return` statement. If this statement is omitted, the weaved file

will not compile. We have chosen not to automatically add a `return` statement at the end of the exception handler, as this might add erratic behavior to programs when the statement has been forgotten by the programmer. In these cases, we feel it is better to have a compiler error instead of runtime erratic behavior.

Example

In our counter example, exception handlers need to be specified for when the counter field cannot be accessed. The following Fix file will print an error on the standard output and end the program.

```
Replicate Counter
{
   contact (VoyagerException ex)
   BEGIN
   System.out.println("Could not contact replicamanager:");
   ex.printStackTrace();
   System.out.println("Exiting");
   System.exit(1);
   END.
   field count
   {
      read (VoyagerException ex)
      BEGIN
      System.out.println("Could not read counter value:");
      ex.printStackTrace();
      System.out.println("Exiting");
      System.exit(1);
      return 0;
      END.
      write (VoyagerException ex)
      BEGIN
      System.out.println("Could not write counter value:");
      ex.printStackTrace();
      System.out.println("Exiting");
      System.exit(1);
      return 0;
      END.
   }
}
```

Note that the code for the exception handlers has been kept simple, basically repeating the same action for all errors. Other exception handlers could be provided, which e.g. pop up dialog boxes, stop the assembly line, or

perform some other useful action, depending on the type of exception and the type of access to the replicagroup which generated this exception.

5.4 The Aspect Weaver

We will now present the aspect weaver, which will combine the aspects defined in Dupe and Fix programs with the base algorithm defined in Jav into Java files. These Java files will contain the code for the FrontEnd of the replication framework, which ensures the replication transparency, as discussed in 3.3.

As revealed in the analysis, the weaver must perform three distinct tasks: The weaver must create a link between the replicated object and its replicagroup, modify all accesses to replicated fields to accesses to the replicagroup, and add the error-handling code to these accesses.

The weaver takes as argument a base filename `name`, and will read three files which define the base algorithm and the aspects. The file named `name.jav` will contain the Jav file, the file named `name.dupe` will contain the Dupe file, and the file named `name.fix` will contain the Fix file.

The aspect weaver will add methods and instance variables to the Jav file, add code which links the replica to the correct replicagroup, modify variable access to replicated fields into accesses to the replicagroup and include exception handlers for errors occurring during accesses to the replicagroup.

When this is complete, the weaver will write this modified code to the file `name.java`, generate the code for the replica and write it to the file `nameReplica.java`.

We have chosen not to generate Java class files, but to generate Java source files. This will allow other Java pre-processors to process the Java code, and the programmer to use her favorite Java compiler or Integrated Development Environment to produce the executable class files.

To be able to discuss the actions of the replicamanager in further detail, some assumptions will have to be made.

5.4.1 Preliminaries

Because replication will be performed by the replication framework, we need to make some assumptions regarding the interaction of the code generated by the aspect weaver, and the code of the replication framework..

We will assume that a replicated object will communicate with its replicagroup through one replica, located on a replicamanager. The replicated object will perform read and write operations on this replica, which will be trapped by its replicamanager. The replicamanager will ensure that these operations proceed correctly, and that the consistency requirement for the replicated data is ensured.

For clarity, we will assume a replicated object of class `repclass` is being weaved, and `RNUM` is a given number between 0 and 1000.

5.4.2 Linking to a Replicagroup

Whenever a replicated object is instantiated, it must connect itself to a given replicamanager to establish a link between itself and the correct replicagroup.

The first task of the weaver is to generate code which will perform these actions. Recall that in most cases, the replicamanager will not be located on the same machine as the client. Therefore, as has been put in the analysis, some sort of network link will need to be established between client and replicamanager.

To establish a network link between the replicated object and its replicagroup we will use Voyager. This entails instantiating the Voyager proxy object for a replica within the replicagroup. As we will show in 6.3.2, the replicamanager will ensure that read and write methods on the data of the replica will result in correct reads and writes on the entire replicagroup.

Instantiating the proxy requires knowledge of the network location of the replica. The class `RepManLocator`, which is included in the replication framework, provides a means to determine this location by reading it from a configuration file.

The weaver will add a `buildRNUM_Proxy` method to the class, which will instantiate the correct proxy and assign it to a `repclass_RNUM_proxy` instance variable, ensuring the correct link between the replicated object and its replicagroup. Should the replicagroup not exist, it will automatically be created.

5.4.3 Modifying Variable Accesses

Once the link between the replicated object and its replicagroup can be created, the weaver must ensure that all reads from replicated fields and assigns to replicated fields are changed into reads and writes on the replicagroup.

The weaver will first verify that all fields which must be replicated are declared as `private`. The weaver will print a warning for each non-private field which has to be replicated, and ignore the replication directive. In other words, these non-private fields will not be replicated.

For each replicated field called `fieldname` a `get_RNUM_fieldname` and `set_RNUM_fieldname` method is added to the code. These methods contain the code which respectively reads the values and writes the values to the replicagroup, by calling the correct methods on the proxy.

Once these methods have been added, each reference to a replicated field `fieldname` is changed by a method call `get_RNUM_fieldname()`,

and each assignment `fieldname = value` is changed by a method call `set_RNUM_fieldname(value)`.

Also, to warn the programmer that replicated sub-objects are to be treated only as data, for each method call on replicated data the weaver will issue a warning.

5.4.4 Error-handling

Recall that each remote method invocation can throw an exception, and that these must be caught and handled in a transparent fashion.

To realize this, the weaver includes the exception handlers specified in the Fix code in the `get_RNUM_fieldname` and `set_RNUM_fieldname` methods which have been introduced above.

Note that the weaver currently performs no checking on the validity of the exception handler, nor does it check whether the exception handler uses replicated variables, which has been forbidden in the fix specification. This because we did not consider it essential for a first version.

5.4.5 Extras

Once the class specifying the replicated object has been processed, the class for the replica needs to be created.

Because this class is tightly linked to the replication framework, it will not be discussed here, a full discussion will be given in 6.3.2. Suffice it to say here that the class will be named `repclassReplica` and will provide correct implementations for the method calls requesting a read or a write of a replicated field.

To allow any version of Voyager to be used the weaver will not automatically generate the Voyager proxy class for the replica. The programmer will need to run the appropriate Voyager tool to generate the proxy class manually.

Using AOP it is now possible to create a FrontEnd for a replicated object, as specified in the abstract model for replication. This FrontEnd consists of the proxy object, the variable accessing methods containing exception handlers, and the modifications from variable references to the respective accessing methods.

5.4.6 Example Output

As an example output, we will use the code generated for the distributed counter which we presented while introducing the aspect languages.

We will not repeat the Jav, Dupe and Fix aspect code here, as it can easily be looked up in the sections about each aspect language.

The weaver will combine the three aspect languages into the following Java program, in which we have manually interspersed a number of comments to enhance clarity:

```
import com.objectspace.voyager.*;

public class Counter
   {
   private int count;

   public Counter(String repl_id)
      {
      // Call proxy building code.
      build782_Proxy(repl_id,
            repserver.replicas.RepManLocator.RMLocation());
      System.out.println("Counter started on "+repl_id);
      }

   public void add()
      {
      set_782_count(get_782_count()+1);
      }

   public int read()
      {
      return (get_782_count());
      }

   public void reset()
      {
      set_782_count(0);
      }

   // Method which retrieves counter value from the replica
   protected int get_782_count()
      {
      try
         {
         return rep_782_proxy.read_count();
         }
      // Exception handler specified in Fix code
      catch(VoyagerException ex)
         {
         System.out.println("Could not read counter value:");
```

```
      ex.printStackTrace();
      System.out.println("Exiting");
      System.exit(1);
      return 0;
      }
   }

// Method which sets counter value on the replica
protected int set_782_count(int value)
   {
   try
      {
      rep_782_proxy.write_count(value);
      return value;
      }
   // Exception handler specified in Fix code
   catch(VoyagerException ex)
      {
      System.out.println("Could not write counter value:");
      ex.printStackTrace();
      System.out.println("Exiting");
      System.exit(1);
      return 0;
      }
   }

// Field which contains the replica's proxy
protected repserver.replicas.VCounterReplica rep_782_proxy;

// Create the proxy
protected void build782_Proxy(String repl_id, String host)
   {
   // If proxy has not been created, create it. Is needed
   // if a constructor calls another constructor.
   if(rep_782_proxy == null)
      {
      try
         {
         // Try to connect to existing replicagroup
         rep_782_proxy=(repserver.replicas.VCounterReplica)
            com.objectspace.voyager.VObject.
                        forObjectAt(host + "/" + repl_id);
         rep_782_proxy.createReplicas();
         }
```

```
        // Fails to connect to existing replicagroup
        catch(com.objectspace.voyager.VoyagerException ex)
           {
           try
              {
              // Create new replicagroup
              rep_782_proxy =
                 new repserver.replicas.VCounterReplica
                              (repl_id,host + "/" + repl_id);
              rep_782_proxy.createReplicas();
              }
           // Could not create new replicagroup
           // Exception handler specified by fix file
           catch(VoyagerException ex)
              {
              System.out.println
                      ("Could not contact replicamanager:");
              ex.printStackTrace();
              System.out.println("Exiting");
              System.exit(1);
              }
           }
           // Perform first initializations on replica
           init_782_vars();
        }
     }

  // Performs initializations specifed in field initializers.
  protected void init_782_vars()
     {
     }
  }
```

Note that we made a good choice to not initialize `count` in the Jav code. Would we have done so, whenever a Counter object would be instantiated, the replica would be re-initialized, as we have described in 5.2.5.

5.5 Conclusion

In this chapter we have discussed an aspect-oriented programming approach to replication. We have introduced three aspect languages: Jav for the base algorithm, Dupe to specify what should be replicated, and Fix to catch errors generated by the replication algorithm. Further we have introduced an aspect weaver which combines the three aspect languages into Java code.

We have encountered two problems which exclude full replication transparency. First it is necessary to name the replicagroup for a replica, which needs to be done from the base algorithm, thus excluding full replication transparency. Second, it is necessary to reconsider all initialization code for replicated objects, to ensure that the replicas are not re-initialized whenever a replicated object creates a link with its replicagroup.

Now the FrontEnd of the abstract model for replication has been realized, we can move on to the ReplicaManagers in the model, which leads us to the core of the replication framework.

Chapter 6

A Framework for Replication

In the previous chapter we have discussed how AOP can be used to create a FrontEnd for the framework for replication. This chapter will discuss the core of the framework, which is located in the ReplicaManagers.

First we will perform an analysis to determine the overall properties of the framework, and once completed we will discuss the implementation of the framework.

6.1 Replication as an Aspect

In the previous chapter we have seen how the framework for replication will be based on the abstract model for replication presented in 3.3. We have defined the ReplicaManager as a server application containing the data, the Client as the base-level algorithm and the FrontEnd as a part of the code generated by the aspect weaver, which is specified in the aspect languages. Also a part of the code generated by the aspect weaver are the replicas for each replicated object, which contain the replicated data of a replicated object.

Recall that instance variables of objects, called fields, are replicated, and that these replicated fields can either be read from or assigned to. Whenever the Client accesses a replicated field, the FrontEnd will intercept this access, and redirect it to the replica by making a method invocation on the replica. These method invocations are specific to each replica, and either read a value from, or assign a value to the replica.

6.2 Analysis

Creating a framework for replication first requires some analysis to determine how we can allow a variety of different instantiations of the framework.

We shall analyze the different choices in replication: Replication transparency, Active vs. passive replication, Inter-server communications and con-

sistency requirements, which we presented in 3.2.

Additional elements need also be analyzed: there is a need for network interactions between the replicamanagers, and for statistics and locks.

6.2.1 Replication Transparency

The choice for replication transparency has been fixed in our thesis. Our goal is to achieve the highest possible replication transparency.

To achieve the highest possible replication transparency, we have used aspect-oriented programming. By considering replication to be an aspect, the FrontEnd of the abstract model can be constructed by the aspect weaver.

A full discussion on this approach, including the obstacles preventing full replication transparency, has been provided in the previous chapter (chapter 5).

6.2.2 Active vs. Passive Replication

For the first implementation of our framework, we have chosen to use a variant of passive replication. This because this variant can both be used in replication to achieve fault-tolerance and to achieve high access speed to the data. Our variation on passive replication consists of two parts: First, methods which modify replicated fields are not executed on the replicamanager, but on the client. Second, there is no distinction between primary and backup replicamanagers.

For a full discussion of this topic, see 5.2.2.

6.2.3 Inter-Server Communications

In the previous chapter we have introduced what will be replicated and how the frontend for the replication framework is constructed.

Recall that reading and writing replicated fields will be transformed into method calls on the replica by the frontend. These method calls have to be caught by the replicamanager, so they can also be applied to the other replicas in the replicagroup. We will refer to these 'other' replicas in the replicagroup as *remote replicas* and the replicamanagers on which they reside as *remote replicamanagers*.

What needs to be implemented is the way in which the different replicamanagers communicate so the accesses to the replicas can be passed on between them. A straightforward solution would seem to be to repeat the method calls invoked by the client on the replica to access the data between the replicamanagers. However, recall that these method calls accessing the data are generated by the aspect weaver and are specific to each replica. For each different replica contained on the replicamanager a different set of data accessing methods would need to be implemented by various classes within the replicamanager.

This in turn implies that the aspect weaver would need to generate not only the replicas, but also a number of classes of the replicamanager to ensure that the replicamanager implements the method calls on all replicas. So the aspect weaver would need to generate a large number of classes of the framework.

We have chosen an alternative solution, using a fixed number of method calls, which avoids the need to generate specific methods for classes in the framework. To achieve this we have used the Java Reflection API [27]. Reflection allows us to handle a classes' field as an object (appropriately called a Field). The Field object is aware of the class which declared its corresponding field, and contains a number of interesting methods. Two of these methods allow reading from and writing to the corresponding field in a given object. These methods on the Field object take as argument an object of the class containing the corresponding field, and read or write the field's values in that object.

We now need only use these Field objects and the name of the replicagroup if we want to read a value from a remote replica. Defining one method, which can be called between replicamanagers and takes as parameters the Field and the replicagroup, is sufficient.

Similarly, if we wish to write a value to a remote replica, we need only use the Field objects, the name of the replicagroup and the value which needs to be set. This requires defining a second method which can be called between replicamanagers, now with parameters the Field, the name of the replicagroup and the new value.

Defining these read and write methods between replicamangers proves to be flexible enough to allow reads and writes between any replicas in the framework.

6.2.4 Consistency Requirement

The manner in which updates and reads can be performed between replica managers has now been presented, but when these operations will occur has not been discussed.

Recall that in replication a consistency requirement determines how the values in the different replicas are kept consistent i.e. containing the same values, as we have discussed in 3.1.2. Keeping replicas consistent is achieved by applying changes made in one replica to each other replica in the replicagroup. The consistency requirement determines how and how quickly these changes must be applied to the replicas in the replicagroup.

A strong consistency requirement would have the changes applied quasi-immediately to all the replicas, a weak consistency requirement would e.g. have changes applied whenever the replicamanager would see fit. The algorithm ensuring consistency would then determine how, where and when

updates and reads are actually performed between the different replicamanagers.

It is clear that the consistency requirement can vary between different applications, so the consistency algorithm may not be fixed in the framework.

Note that in the previous discussion we have both mentioned reads and writes as being influenced by the consistency requirement. Indeed, also reads my be influenced: consider the case where a replicated field may have different values on different replicas. A requirement may be that in these cases the value returned by a read is the result of merging the values on the different replicas. For example, were the values integer values, the result could be the mean of these different values.

Once these updates are passed on between replicamanagers, they still must be applied to the data. Recall that here also the consistency requirement may impose some restrictions, as we have introduced in 3.2.3. As an example we have used the extra restriction in USENET stating that replies may not appear on a bulletin board before the original message has appeared. A hold-back queue was presented in 3.2.3 to allow these requirements to be met. It is obvious that different instantiations of the framework will need different queues, so the queue may not be fixed in the framework.

Locks on replicas and on fields might also be needed to implement some consistency requirements. A consistency requirement might state that while replicas are being updated, no other changes may be made to any replica. This means that no client may access the replicas and no other updates may be applied to the replica. To achieve this, locks on the replica must be made possible.

The semantics of the locks might be of different kinds. Classical read and write locks [5] might be necessary, or more advanced strategies might be needed, such as intent-to-read and intent-to-write [5]. Again, these locking strategies must be able to vary between different implementations.

This leads to a total of four elements which can vary for the consistency: a read and write element for the timing and location of reads and writes, a queuing element for application of these reads and writes, and a locking element for different kinds of locking. To implement a certain consistency requirement, the programmer can mix and match these different elements to obtain the correct consistency algorithm with relative ease.

6.2.5 Network Interactions

Systems using replication inevitably rely on a network between different replicamanagers which is used by the consistency algorithm to enforce the consistency requirement. The different replicamanagers will use this network to pass data amongst themselves to ensure the data in the replicagroup is kept in a consistent state.

We will use remote method invocations to allow replicamanagers to enforce the chosen data consistency. We will rely on Voyager [12], which we have discussed in 5.2.1 to provide the possibility to use remote method invocations.

6.2.6 Metadata

When replicating data, some data which describes elements of the replicated data can be useful. This metadata could consist of a variety of elements. We shall restrict ourselves to two types of elements: statistics and locks.

Statistics of usage of the replica could take on a variety of forms. They can range from basic statistics such as number of data accesses to sophisticated statistics which include number of reads and writes per field, elapsed time between accesses, and so on.

These statistics could be used for a variety of purposes, an interesting application would be management of replicas. Using these statistics, a management algorithm could determine which replicas are intensively used by some clients, and which are not. This could be used to perform load-balancing between replicas, by e.g. adding new replicas on replicamanagers which do not contain them, or moving unused replicas to other replicamanagers where they will be used by a number of clients.

The statistics must of course be kept up to date, so each call made by a frontend should be registered in the statistics. As has been stated above, the statistics can take on various forms, and the statistics used will depend on the instantiation of the framework.

Although locking strategies are an element of the consistency algorithm for the replication, they must also be considered as metadata. Locks on a replica are data which describe if and how the data of a replica can be accessed. So locking can also be classified under metadata.

6.2.7 Conclusion

We have now seen how we can allow any types of replicas to be kept consistent by making using Java Reflection in the inter-server communications. Also a wide variety of consistency requirements can be permitted, by splitting up the consistency algorithm into read, write, queuing and locking elements. Finally, some statistics might be needed about replica usage.

6.3 Implementation

With the analysis complete, we can now discuss the implementation of the framework.

We will first present a small overview of the framework's design, followed by in-depth discussions of each of the framework's elements.

6.3.1 Overview

The analysis has revealed that each replica must be accompanied by a variety of classes which must be subclassed to provide the correct behavior. We have seen that each replica will need a read and write element, a queuing element, a locking element and a statistics element. Each of these elements can be realized using a Strategy design pattern, so they all are encapsulated into an object and are easily interchangeable.

To ensure correct coordination between the tasks performed by these five objects, we have chosen to add a 'director' object to each replica. This director will instantiate the correct five strategy objects, obtain a list of the remote replicamangers for the replica, notify the replicamanager of the replicas' existence, and further coordinate actions of the strategies.

Figure 6.1 gives an overview of the replicamanager, assuming the replicamanager contains only one replica.

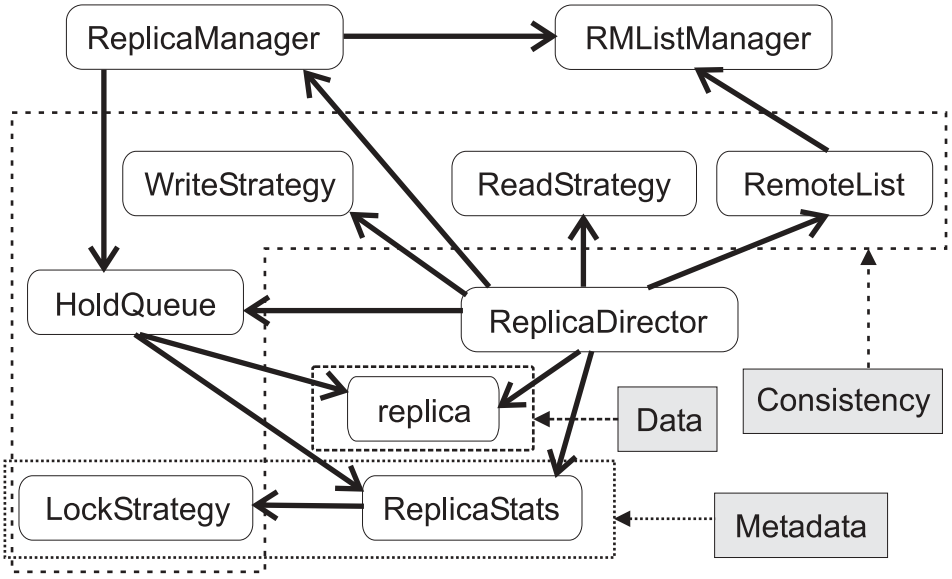

Figure 6.1: An overview of the replicamanager, assuming one replica. For multiple replicas, all classes except ReplicaManager and RMListManager must be duplicated per replica. Arrows indicate directed associations.

We will document each class in the framework using the call traces of a read and write call. As the call trace 'enters' a new class, this class will be discussed. An example call trace of a read call is given in the figures 6.2 and 6.3. Figure 6.2 treats the method called within the local replicamanager, and figure 6.3 treats methods called within remote replicamanagers.

For easy reference while documenting the diverse classes, we will repeat the trace in each class in a textual version, as given in 6.4, which also includes

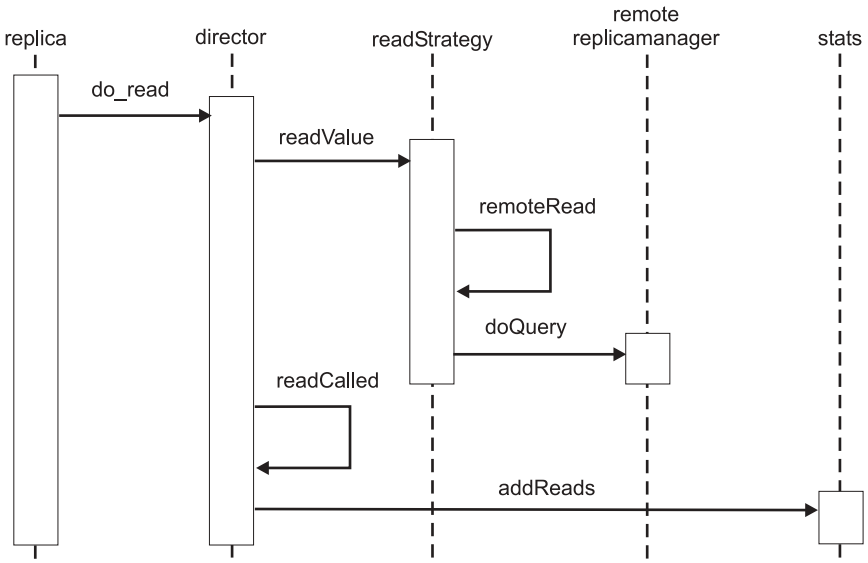

Figure 6.2: OMT interaction diagram for a read call `read_varname`, within the local replicamanager.

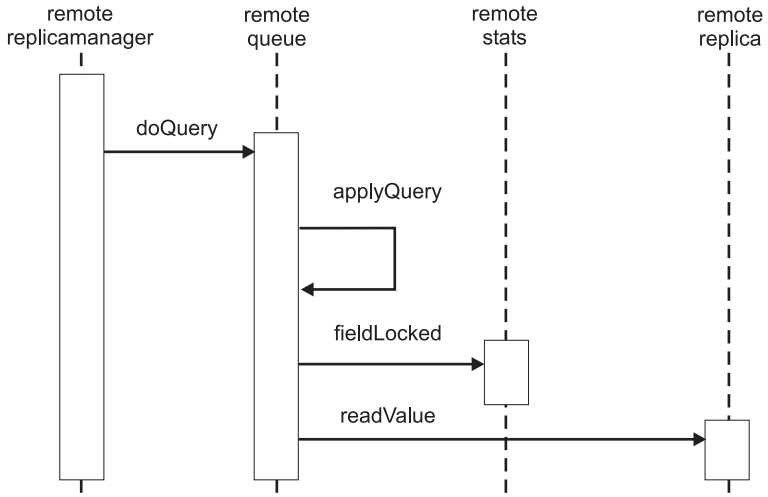

Figure 6.3: OMT interaction diagram for a read call `doQuery`, recieved by a remote replicamanager.

a trace for a write call.

```
read:                                write:

replica: read_varname()              replica: write_varname()
 director: do_read()                  director: do_write()
  readStrategy: readValue()            writeStrategy: writeValue()
   readStrategy: remoteRead()           writeStrategy: remoteWrite()
    remote RM: doQuery()                 remote RM: doUpdate()
     queue: doQuery()                     queue: doUpdate()
      queue: applyQuery()                  queue: applyUpdate()
       stats: fieldLocked()                 stats: fieldLocked()
       replica: readValue()                 replica: writeValue()
  director:readCalled()                director:writeCalled()
   stats: addReads()                    stats: addWrites()
```

Figure 6.4: Full traces of a read and write call through the framework. An indent level up the call trace means the method has been called by the method one indent level down in the call trace, e.g. `do_read()` is called by `read_varname()`, `fieldLocked()` and `readValue()` are both called by `applyQuery()`.

6.3.2 The Replica

```
read:                                write:

replica: read_varname()              replica: write_varname()
```

Figure 6.5: Call trace up to replica

The replica, subclass of AbstractReplica, is generated by the aspect weaver when it processes an object which needs to be replicated. The replica contains the replicated fields of the replicated object, and a number of methods used to read data from an write data to the replicagroup.

For each replicated field `fieldname`, two methods are available in the replica: a method `read_fieldname` to read the field's value and a method `write_fieldname` to write a new value to the field. These methods will not interact with the data of the replica itself, but call methods in the director which will act on the data of the replicagroup, according to the read and write strategies.

Because the replica is the only element in the framework which is generated separately for each application, the replica is responsible for transform-

ing the read and write calls made by clients to a general form. This form, as has been discussed in 6.2.3 uses Java Reflection. To realize this, each replica will, when instantiated, create Field objects for each replicated field. When a client wishes to read a value from a replicated field its ReplicaDirectors' `doRead` method will be called, with as argument the corresponding Field object. Similarly, when a client wishes to write a value the directors' `doWrite` will be called with arguments the corresponding Field object and the new value.

Two methods defined in AbstractReplica allow data in the replica to be read or written by the read or write strategies, or by the hold-back queue. To read a value `readValue` is used, which takes as argument a Field object to determine which replicated field needs to be accessed. To write a value `writeValue` is used, which takes as arguments a Field object as above, and the value which needs to be written.

Instantiation of replicas is always done by a frontend. Recall that when the frontend sees there is no replicagroup for a certain replica, it will create this replicagroup. Creating a replicagroup naturally entails instantiating the replicas in this group.

When instantiated, the replica will immediately instantiate its ReplicaDirector, which will perform a number of setup operations. The replica and the AbstractReplica are also responsible for creating links to the ReplicaManagers of the remote replicas in the replicagroup. The location of these remote replicas is given by a RemoteList. If a remote ReplicaManager in this list does not contain a replica, it will be created on that ReplicaManager. The links to the ReplicaManagers will be stored in a RemoteList and passed on to the replica's ReplicaDirector.

6.3.3 The ReplicaDirector

```
read:                              write:

replica: read_varname()            replica: write_varname()
 director: do_read()                director: do_write()
```

Figure 6.6: Call trace up to ReplicaDirector

The ReplicaDirector is an object which directs the interactions between the different strategies associated with an object. We have chosen this approach to achieve a cleaner separation between the data which is replicated (the replica), and the actions which have to be taken to keep the replica consistent.

Whenever a replica is instantiated, it will immediately instantiate its director, which in turn will instantiate all the different strategies for the

replica and register the replica with the ReplicaManager. This will also allow an expansion of the framework with minimal impact: it can later be made possible to assign different strategies i.e. different consistency algorithms per replica, by making a number of changes in the director.

Two methods in the director are responsible for read and write operations on the data and are called by the clients: `doRead` and `doWrite`. These methods will respectively call the `readValue` method on the ReadStrategy and the `writeValue` method on the WriteStrategy. Each will pass on its given parameters and will add two new parameters : the replica and the RemoteList of ReplicaManagers containing the remote replicas.

When the `readValue` or `writeValue` call has terminated, the director will also update the statistics for the replica.

6.3.4 The Read - and WriteStrategies

```
read:                                write:

replica: read_varname()              replica: write_varname()
 director: do_read()                  director: do_write()
  readStrategy: readValue()            writeStrategy: writeValue()
   readStrategy: remoteRead()           writeStrategy: remoteWrite()
```

Figure 6.7: Call trace up to Read- and WriteStrategies

The ReadStrategy and WriteStrategy are objects responsible for determining when reads and writes to the replicas will occur, according to the consistency requirements.

We will extensively cover the WriteStrategy. Read- and WriteStrategy are similar to a large extent, so the structure of the ReadStrategy can be easily deduced.

WriteStrategy is the implementation of a Strategy design pattern. The abstract class WriteStrategy declares an abstract method `writeValue` which must be implemented by its subclasses. These subclasses will be concrete write strategies. WriteStrategy also defines a method `remoteWrite` which immediately writes a value to a given ReplicaManager. Subclasses can use this method to perform the writes to the remote ReplicaManagers.

Subclasses, implementing concrete write strategies should implement the method `writeValue`, taking as parameters the AbstractReplica from which the write originated, the Field in which is written, the new value for this field, and a RemoteList of ReplicaManagers which must be written to. When this method is called, this indicates that a new value must be written to the remote replicas. The strategy's algorithm should then decide when and to which ReplicaManagers to write the value. To achieve the writes, the

`remoteWrite` method can be used, which will call the `doUpdate` method in the ReplicaManager.

Writes can cause lock exceptions to be thrown, this happens when the replica contains a lock which conflicts with the write operation. The WriteStrategy should handle this exception how it sees fit. Note that the `remoteWrite` method in AbstractReplica will handle lock conflicts on the replica (not on the fields) by waiting a random time and retrying the write. Should this throw another lock exception, the wait-retry cycle will be repeated until the write is successful.

The AbstractReplica parameter in the `writeValue` method on the concrete write strategy can be used to write to the local replica. This can be done directly to the replica, bypassing the queue, or a reference to the queue can be obtained, which can be used to write the value to the replica obeying queueing regulations.

6.3.5 The ReplicaManager

```
read:                              write:

replica: read_varname()            replica: write_varname()
 director: do_read()                director: do_write()
  readStrategy: readValue()          writeStrategy: writeValue()
   readStrategy: remoteRead()         writeStrategy: remoteWrite()
    remote RM: doQuery()               remote RM: doUpdate()
```

Figure 6.8: Call trace up to ReplicaManager

The ReplicaManager is the main process of the replicamanager entity in our abstract model for replication. When starting a replicamanager, a ReplicaManager object will be instantiated.

Because there need only be one ReplicaManager per process, ReplicaDirectors need access to the ReplicaManager to register the replica, and read or write strategies also need access to the local ReplicaManager, ReplicaManager is an implementation of the Singleton pattern.

Whenever a replica is registered by a ReplicaDirector, the HoldQueue for the replica is added to a dictionary. This dictionary uses the name for the replicagroup as a key and will be used to direct incoming read and write operations to the queue for the appropriate replica.

The methods `doQuery` and `doUpdate` are called by remote replicas to indicate read or write operations must be performed on a replica. These methods include the name of the replicagroup for which a replica must be read from or written to. This name will be looked up in the dictionary which

will provide the replicas' HoldQueue. The method `doQuery` or `doUpdate` will be executed on this queue to place the read or write request in the queue.

Methods which lock and unlock fields of replicas or entire replicas at once are also provided by the ReplicaManager. `lockField` and `unlockField` can be used to lock and unlock fields for the local replica of a given replicagroup, and `lockReplica` and `unlockReplica` will lock and unlock the entre replica. These four methods will not be executed on the replica itself, but on its HoldQueue, as has been discussed for the `doQuery` and `doUpdate` methods.

Whenever new replicas are added to a replicagroup, i.e. when new replicas are added to ReplicaManagers which did not contain them, the 'old' replicas in the group must be made aware of these new additions. To allow this, the ReplicaManager contains a `registerReplicaManager` method. This method allows other ReplicaManagers to signal this ReplicaManager that they have constructed a 'new' replica for a given replicagroup. The ReplicaManager which contains the 'new' replica will be added to the RemoteList of the local replica, ensuring that the 'new' replica is kept consistent with the local replica.

6.3.6 The HoldQueue

```
read:                                write:

replica: read_varname()              replica: write_varname()
 director: do_read()                  director: do_write()
  readStrategy: readValue()            writeStrategy: writeValue()
   readStrategy: remoteRead()           writeStrategy: remoteWrite()
    remote RM: doQuery()                 remote RM: doUpdate()
     queue: doQuery()                     queue: doUpdate()
      queue: applyQuery()                  queue: applyUpdate()
```

Figure 6.9: Call trace up to HoldQueue

As has been mentioned in the analysis, a hold-back queue for the replica is also required to implement some consistency requirements. The HoldQueue allows such queueing to be performed.

To allow different instantiations of the framework to use different queueing mechanisms, HoldQueue is an implementation of the Strategy design pattern.

The correct queueing strategy is instantiated by the ReplicaDirector, associated with the correct replica and registered with the ReplicaManager when the ReplicaDirector is instantiated.

HoldQueue declares a number of abstract methods which are called by the ReplicaManager, indicating data must be read or written, or locks must

be set or released. Furthermore, HoldQueue defines methods `applyQuery` and `applyUpdate` which will perform the actual reading and writing of the data, after verifying if the current locks on the field allow the operation. If current locks do not permit this operation, a locking exception will be thrown, which will need to be caught by at least the Read- or WriteStrategy. A number of locking methods are also provided in HoldQueue: `applyLockField`, `applyUnlockField`, `applyLockReplica` and `applyUnlockReplica` which lock and unlock replicas or fields, after verifying if it is permitted. If the locking operation is not permitted, a locking exception will be thrown, which, again, will need to be caught by at least the Read- or WriteStrategy.

Concrete queueing strategies, subclasses of HoldQueue, will need to implement the abstract `doUpdate`, `doQuery`, `lockField`, `unlockField`, `lockReplica` and `unlockReplica` methods. These methods can rely on their corresponding `apply`-method to perform the actual operation when permitted by the consistency requirement.

6.3.7 Statistics and Locking

```
read:                                write:

replica: read_varname()              replica: write_varname()
 director: do_read()                  director: do_write()
  readStrategy: readValue()            writeStrategy: writeValue()
   readStrategy: remoteRead()           writeStrategy: remoteWrite()
    remote RM: doQuery()                 remote RM: doUpdate()
     queue: doQuery()                     queue: doUpdate()
      queue: applyQuery()                  queue: applyUpdate()
       stats: fieldLocked()                 stats: fieldLocked()
```

Figure 6.10: Call trace up to ReplicaStats

The ReplicaStats and LockStrategy contain the metadata about a replica. ReplicaStats contains statistics about the replicas' usage, and LockStrategy manages the locks on the replica.

To allow different instantiations of the framework to use different kinds of statistics for the replicas, a Strategy design pattern is used. Subclasses of ReplicaStats, implementing concrete statistics, must implement the methods `addReads` and `addWrites` which are called by the ReplicaDirector when a read or a write has been executed. The methods which read out the statistics : `giveReads` and `giveWrites` must also be implemented.

ReplicaStats also contains the current LockStrategy. As for statistics, different instantiations of the framework may use different strategies for

locking fields. This might be no locking at all, or standard read-write locks, or any other locking method. To allow these different locking strategies for the fields of replicas, LockStrategy is also a Strategy design pattern.

Each subclass of LockStrategy must implement the locking and unlocking methods `lockField`, `unlockField`, `lockReplica` and `unlockReplica` and also `fieldLocked`, which verifies if a lock conflict is occurring. These methods are called by the ReplicaStats, when the corresponding methods in the ReplicaStats are called by the HoldQueue.

6.3.8 Lists of ReplicaManagers

The RemoteList holds a number of Voyager proxy objects for remote ReplicaManagers, insuring inter-server communications can be performed whenever neccessary.

RemoteLists are created by RMListManagers. Given a name of a replicagroup, RMListManagers will create the RemoteList containing replicamanagers which manage the replicas of this replicagroup.

Different instantiations of the framework can require different allocation strategies for replicas within a replicagroup. Some might require replicas to be contained on all replicamanagers, others might require replicas to be contained on a select number of replicamanagers, and so on. To allow these different allocations for each instantiation of the framework, RMListManager also implements the Strategy design pattern.

Each concrete listmanager, subclass of RMListManager, must, amongst others, implement the method `listRMsFor` which will create the correct RemoteList given the name of a replicagroup.

To access the elements of the RemoteList, an iterator must be created on the list. This iterator is a construct which can traverse the list, and return the element of the list to which it is currently pointing. Using these iterators allows us to safely modify a RemoteList while it is being traversed by a number of iterators.

Currently two types of iterators are provided: a ForwardRLIterator and a BackwardRLIterator. The forward iterator starts at the start of the list, and can advance to the end of the list, the backward iterator starts at the end of the list, and can advance to the start of the list. When iterators are no longer needed, they must be explicitly removed from the list.

6.3.9 RMFactory

The Factory methods, which instantiate the right objects for the current instantiation of the framework, are all defined in one class: the RMFactory.

We use a variation of the Factory method design pattern, all Factory methods are defined as class methods of the RMFactory class. To instantiate the correct objects, the code of the RMFactory must be adapted.

RMFactory can instantiate objects of the following classes: HoldQueue, ReplicaStats, ReadStrategy, WriteStrategy, LockStrategy and RMListManager. In short, all instantiations of the different Strategy design patterns are done here.

6.4 Conclusion

We have seen how we can develop a framework for replication which can be instantiated for a variety of applications.

Using reflection to handle inter-server communications we can keep a wide variety of replicas consistent, using only two method calls. Also, encoding the consistency requirement into four strategies enables us to mix and match strategies to achieve a large number of consistency requirements with relative ease.

To verify these claims, we will now instantiate the framework for two, widely different applications.

Chapter 7

Main Experiments

To validate the claim that our framework can be used for a wide variety of applications, we will now instantiate it to create two widely differing applications.

First we shall create a distributed warehousing application. The main requirement here is strong data consistency. The stock contained in the warehouses must always be identical on all replicas, so e.g. stock can not be removed twice from a warehouse.

Second we shall create a internet chat application. The main requirement here is not the data consistency, but high interactivity. To allow people to talk easily, the latency between different messages must be as low as possible.

7.1 The Distributed Warehouse

Our first experiment is a distributed warehousing application.

We will first provide a short description of the situation, the requirements for the replication and the proposed general solution.

Next, the implementation of the distributed warehousing will be discussed application. We will present the Warehouse class, which will be replicated, followed by the aspect languages specifying the replication, the instantiation of the framework, and finally three small applications.

7.1.1 Situation

Imagine the following situation: a company has a number of warehouses, distributed over a large area. Each warehouse contains a variety of products, which will change over time. In each warehouse different applications must be able to check the stock not only of the local warehouse, but also of the other, remote warehouses. These applications must also be able to modify the stock of all the warehouses. This might be necessary when a product has

to be delivered to a customer, but it is not available in the local warehouse, or when products are moved from one warehouse to another.

Because removing the same products twice from a warehouse may not happen, and the total amount and variety of products has to be correct at all times, the consistency requirement has to be very high. Were this not so, stock might be removed twice in the following scenario: two clients working on a different replica of the stock of a warehouse both remove all items of a product from the local replica at approximately the same moment in time. Later on, these changes are propagated to the other replicas. At each replica, all the items of the product will be removed twice, which can obviously not be allowed. Would the changes to a replica be immediately propagated to the other replicas, once the first client had removed the items, the second would not be allowed to remove the non-existent items.

Also, the company would like to know how intensively the warehouses are used, how many times a stock listing is requested, and how many changes are made to the stock.

Replication is used in this system to achieve good client access speed to the data. We assume the stock data is read quite frequently, but written less frequently. A large overhead for the consistency algorithm is allowed, but must not be elevated unnecessarily.

Let us consider the number and location of the replicamanagers. Having few replicamanagers eases the work of the consistency algorithm, decreasing the overhead for replication. All clients accessing the replicamanagers will be applications running in a warehouse, so locating a replicamanager inside the warehouse will improve client access speed. To balance all the requirements, we will use one replicamanager per warehouse, each managing a replica of each warehouses' stock.

7.1.2 The Warehouse Class

The Warehouse class represents the stock of a warehouse and allows this stock to be listed and to be modified.

Stock is an amount of products of different varieties. The products are represented by a Product class. Product represents the code of a product and the amount of that product in the warehouse. Products may only be instantiated and printed. Other operations on products are of course possible e.g. changing the amount of products, but have been omitted for sake of clarity.

The Product class is defined in Java as follows:

```
public class Product implements java.io.Serializable
{
   public int id, amount;
```

```
   public Product (int id, int amount)
   {
      this.id = id; this.amount = amount;
   }
   public void print()
   {
      System.out.println(id + " : "+amount);
   }
}
```

Because a Warehouse must contain different Products, and this amount will change over time, we have chosen to use a Vector, called `thestock` to store the different Products. Since `thestock` contains the values needed by the clients, it will be the replicated field of the Warehouse objects. Recall the requirement for replicated fields, as specified in 5.2.2, is that they are serializable, so they can be transferred over the network. A Vector is serializable if each of the objects it contains is serializable. Because Product implements `Serializable` this is indeed the case, and `thestock` can be replicated.

We will now present the Jav code for Warehouse, followed by some notes.

```
import java.util.Vector;

public class Warehouse
{
   private Vector thestock;

   public Warehouse(String repl_id)
   {}
   public void init()
   {
      thestock = new Vector();
   }
   public int size()
   {
      return thestock.size();
   }
   public void removeFirst()
   {
      Vector stock = thestock;
      stock.removeElementAt(0);
      thestock = stock;
   }
   public void addProduct(Product prod)
   {
```

```
      Vector stock = thestock;
      stock.addElement(prod);
      thestock = stock;
   }
   public void print()
   {
      System.out.println("========================");
      Vector stock = thestock;
      int i;
      for(i=0; i<stock.size(); i++)
      {
         Product prod = (Product)stock.elementAt(i);
         prod.print();
      }
   }
}
```

Recall that the formal parameter `String repl_id` of the constructor is used to associate the Warehouse to its replicagroup. In this case, each replicagroup can have the name of a 'real life' warehouse.

We have chosen to allow only two changes to the stock: removing the first Product in `thestock` and adding a new product at the end of `thestock`. Making other changes to the stock, such as removing a given product from the stock or changing the amount of a given product, can be easily implemented, but have not been included to keep the example uncluttered.

Note that `thestock` is a replicated Vector, so a replicated sub-object. As mentioned in the analysis (5.2.2), method calls on this Vector are allowed, but if these method calls change the state of the Vector, these changes will be lost. To allow changes to be made to `thestock`, the procedure applied in the method `addProduct` must be followed: `thestock` must be copied to a non-replicated field, say `stock`, changes must be made to `stock`, and saved to the replica by assigning `thestock` the value of `stock`.

The method `print` also makes a local copy of the stock, this to print out an accurate snapshot of the stock. Would we iterate over the replicated stock, additions and deletions from this stock might still occur while we are iterating over it. This would lead to a printout which would not accurately represent a snapshot of the stock.

Because we had only to keep in mind including the `String repl_id` formal parameter in the constructor, and not making any changes directly to `thestock`, we feel we can safely state that the degree of replication transparency is high for the Warehouse object.

7.1.3 The Replication Aspect Code

Once the class of the replicated object has been defined, we must decide which fields must be replicated and provide the exception handlers which will handle possible errors occurring while accessing the replicated fields.

As we have state above, the replicated field is `thestock`. This is specified in the following Dupe file:

```
Replicate Warehouse
{
   field thestock replicate;
}
```

Now consider what must be done when an error occurs. Because of the high data consistency, we can not provide default recovery values for the stock and must inform the user a serious error occurred. To maintain simplicity, we have chosen to print out an error message and end the program. While this is a quite radical and user-unfriendly action, other exception handlers can be specified which could easily be more graceful, for example by using dialog boxes to signal the problem.

```
Replicate Warehouse
{
   contact (VoyagerException ex)
   BEGIN
   System.out.println("Error while connecting, aborting.");
   ex.printStackTrace();
   System.exit(0);
   END.
   field thestock
   {
      read (VoyagerException ex)
      BEGIN
      System.out.println("Error in reading stock, aborting.");
      ex.printStackTrace();
      System.exit(0);
      return null;
      END.
      write (VoyagerException ex)
      BEGIN
      System.out.println("Error in writing stock, aborting.");
      ex.printStackTrace();
      System.exit(0);
      return null;
      END.
```

```
    }
}
```

Note that because we are using Voyager for the remote method invocations, all exceptions thrown will be subclasses of VoyagerException. This means the specified exception handlers will catch all possible exceptions thrown while accessing the replicated data.

After defining the replicated object in the base aspect language Jav and defining the replication and error-handling aspect in the Dupe and Fix, the aspect weaver must be used to generate the Java code for the replicated object and for the replica. We will not include this code here, as it can be easily deduced from the discussion of the aspect weaver (5.4).

7.1.4 Instantiating the Framework

To instantiate the framework, six hot spots must be filled in by creating concrete classes for the strategies. Four concrete classes are needed for the consistency algorithm: read and write strategy, queuing strategy and locking strategy. Fifth and sixth strategies are the RMListManager and the statistics algorithm.

The requirement for these statistics are that they record usage of reads and writes on the stock, so usage of the application can be determined. This can be achieved by implementing a statistics class where the `addReads` and `addWrites` methods store these reads and writes on some permanent storage medium, which can be accessed from the local replicamanager. Other specific statistical applications can then analyze this data to provide the required information.

Recall that the RMListManager is responsible for creating a RemoteList of the replicamanagers containing replicas of a given replicagroup. In this case, the list will be identical for all replicagroups and will contain all ReplicaManagers. The list of all ReplicaManagers can be easily specified in a config file, since the number and location of ReplicaManagers is fixed (one per warehouse). The `listRMsFor` method of the RMListManager can read this config file and create the appropriate RemoteList.

Now consider what must be done to satisfy the high consistency requirement. A high consistency requirement implies that changes made to a replica must be immediately applied to the other replicas, before the client can proceed. To ensure this, the writestrategy must immediately update the other replicamanagers, which will put this update in the hold-back queue for their replica. This queue may not hold back the changes to the replica, they must be applied immediately. Applying these changes must be possible at all times, so locks on the fields of the replica are not allowed. Also, since the values on all replica will be kept identical, reading the value from the replicagroup must only be done at the local replica, and not on any remote

replicas.

We will now present the implementation of the objects of the write, queue, lock and read strategies.

As stated above, the writestrategy must immediately apply the new value to all the replicamanagers. To realize this, it will first apply the change to the local replica, and afterwards iterate over the list of remote replicamanagers to pass on the change to the other replicamanagers.

This algorithm is implemented in the following class:

```
public class SynchWrite extends WriteStrategy
{
   public void writeValue
      (AbstractReplica repl, String thefield,
       java.io.Serializable value, RemoteList rmlist)
      {
         // Write locally
         repl.writeValue(thefield, value);

         // Obtain an iterator for the RemoteList
         RLIterator iter = rmlist.newForwardIterator();
         if (!iter.isEmpty())
         do
         {
            try
            {
               // Pass on this change to the remote
               // replicamanager
               this.remoteWrite(iter, repl,
                  "", thefield, value, new Integer(0));
            }
            catch (LockedException lex)
            {
            // We assume no locking is used,
            // so this exception will not be thrown
            }
         }
         while(iter.next());
         rmlist.delIterator(iter);
   }
}
```

The queuing strategy must immediately apply the incoming update to the replica. This is realized by using the following `doUpdate` method.

```
public Serializable doUpdate
```

```
   (String owner, String field,
    Serializable value, Integer level)
   throws LockedException
{
   this.applyUpdate(owner, field, value, level);
   return null;
}
```

Note that the implementation of the other methods is irrelevant, as they will not be called, since we use no locking, and read all data locally.

For the locking strategy, only the implementation of the `lockConflict` method is relevant. Since we use no locking, the `lock` and `unlock` methods are not called, so their implementation is irrelevant.

The `lockConflict` method should always return false, to indicate there is no locking conflict, and changes can be applied to the replica.

```
public boolean lockConflict
   (String owner, Integer level,
    Vector lockowners, Vector locklevels)
{
  return false;
}
```

Finally, the readstrategy must be implemented. Since the data must only be read from the local replica, the code for the readstrategy is simple:

```
public class LocalRead extends ReadStrategy
{
   public Serializable readValue
      (AbstractReplica repl, String thefield,
       RemoteList RMList)
   {
       return repl.readValue(thefield);
   }
}
```

Now all six strategies have been implemented, the framework for replication has been instantiated. What remains is to write the client applications which will use the replicated Warehouse objects.

7.1.5 Some Client Applications

We will now present three client applications for the distributed warehouse. The first client will initialize a given Warehouse, a second client will monitor

the stock of a Warehouse, and the third will apply random changes to a Warehouses' stock.

The first client is an application which initializes the stock. Considering the fact that warehouses will we initialized only very rarely, we have chosen to make a separate initialization client application. The code for this client is extremely simple:

```
public class StockInit
{
   public static void main(String [] args)
   {
      if (args.length != 1)
      {
         System.out.println("Need warehouse name");
         System.exit(1);
      }
      Warehouse wh = new Warehouse(args[0]);
      wh.init();
      System.exit(0);
   }
}
```

This application takes as argument a name of a Warehouse, and will initialize it by calling its `init()` method. Note that when instantiating the Warehouse and a replicagroup with the given name does not exist, the replica framework will transparently create it.

The second application simulates repeated consulting of the stock by printing a snapshot of the stock every three seconds. Again the code is straightforward:

```
public class StockMonitor
{
   protected Warehouse wh;

   public StockMonitor(String zone)
   {
      wh = new Warehouse(zone);
   }
   public void monitor()
   {
      while(true)
      {
         synchronized(this)
         {
```

```
            try
            {
               wait(3000);
            }
            catch(InterruptedException ex)
            {}
         }
         wh.print();
      }
   }
   public static void main(String [] args)
   {
      if (args.length != 1)
      {
         System.out.println("Need Warehouse name");
         System.exit(1);
      }
      StockMonitor sm = new StockMonitor(args[0]);
      sm.monitor();
   }
}
```

The application takes as argument the name of a Warehouse and instantiates a Warehouse which will be linked to the replicagroup with this name. Once the Warehouse is instantiated, the application will call its `print()` method every three seconds.

The third client simulates random changes to the stock. To keep this application simple, we only remove the first Product from the stock and add new Products to the stock. More elaborate applications, which add, remove or change a random Product in the stock can be easily implemented.

The following code will, with a probability of 50% remove the first Product in the stock and, with a probability of 50%, add a new Product to the stock. This will be repeated every five seconds, and a printout of the current stock will be created.

```
public class StockChanger
{
   public Warehouse wh;

   public StockChanger(String zone)
   {
      wh = new Warehouse(zone);
   }
   public void change()
```

```
   {
      while(true)
      {
         synchronized(this)
         {
            try
            {
               wait(5000);
            }
            catch(InterruptedException ex)
            {}
         }

         int siz = wh.size();
         if ((siz != 0) && ((Math.random() * 100)>50))
         {
            wh.removeFirst();
         }
         if((Math.random() * 100)>50)
            wh.addProduct(
               new Product((int)(Math.random()*10000),
                           (int)(Math.random()*30)));
         wh.print();
      }
   }

   public static void main(String args[])
   {
      if (args.length != 1)
      {
         System.out.println("Need Warehouse name");
         System.exit(1);
      }
      StockChanger sc = new StockChanger(args[0]);
      sc.change();
   }
}
```

Note that all three applications treat Warehouse as any other object, and need not concern themselves with replication. The only element of replication which they come into contact with is the naming of the Warehouse object.

However the naming of the replicagroup does not have a significant impact on the application. It can be seen as 'just another formal parameter'.

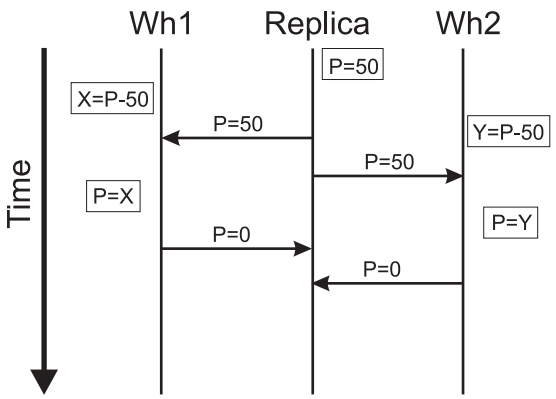

Figure 7.1: Dual removal of product P on the same replica by Warehouses Wh1 and Wh2.

Therefore, we feel we can state that for these three applications replication is fully transparent.

7.1.6 Omissions and Improvements

An important element in the distributed warehouse is the requirement for high data consistency. To fulfill this requirement, we have implemented a consistency algorithm which should ensure this. However, there are some cases where the consistency requirement will not ensure correct operation.

Consider the operations which are performed to remove a number of products from the stock. First, the stock will be consulted to see how many of these products are available. Next a number of these products will be removed. In the time elapsed between consulting the stock and removing a number of products from the stock, some other client may already have removed all products in the stock. This client will not be aware of this, and remove the same products again from the stock, which can not be done. Clearly, this situation must be avoided, even though it does not depend on the consistency requirement, as illustrated in figure 7.1.

A potential solution would be to let the clients perform locking on the replicated fields. To ensure that the replication transparency is not further reduced, setting and releasing of locks must be done from the aspect languages This would entail extending Dupe with some locking constructs. The most straightforward manner would be to allow locking of a number of replicated fields when certain methods are executed. This would then be combined with the requirement that locked fields can only be accessed by the lock's owner. Accesses performed by replicated objects which do not own the lock must wait until the lock is released.

A possible extension of Dupe would be:

```
DupeProgram :
   "Replicate" Classname
   "{"
       (Field | MethodLock)* [Default]
   "}"

MethodLock :
    "method" MethodName "lock" Fieldname ";"
```

Where `MethodName` is a method of `Classname`, and `Fieldname` is a replicated field.

Currently this extended version of Dupe is not supported, but adapting the weaver and the framework to support it will not pose any significant problems. A `lock` and `unlock` method must be defined on the replica. The `lock` method will be called whenever `MethodName` starts executing, and the `unlock` method will be called when `MethodName` finishes. Both these `lock` and `unlock` methods can use the locking facilities which are already provided by the locking strategy.

Locking could be used in our distributed warehouse as follows. In the Warehouse class, we can define a method `removeProducts` as described below.

```
boolean removeProduct(int prod_id, int amount)
{
    [read the products]
    [locate product with id prod_id]
    if([products' amount] > amount)
       [decrease products' amount and save it]
       return true;
    else
       return false;
}
```

Now, were we to specify that removeProduct requires a lock on the field `thestock`, the problem would be solved. This implies adding the following line to the Dupe code:

```
method removeProduct lock thestock;
```

With this locking implemented, the errors described above will not occur.

7.1.7 Conclusion

We have seen how we can create a distributed warehouse application using the framework for replication.

The main requirement for the distributed warehouse was the high degree of data consistency, which we have achieved by using the correct consistency algorithm.

To ensure no errors occur during updates of stock in spite of the high degree of data consistency, locking of fields will have to be introduced. Locking fields is not possible at the moment, but can be added to the framework without difficulties.

Also, we have seen that replication transparency for the different client applications can be considered to be very high, which was one of our main goals.

7.2 The Chat Application

Our second experiment is a chat application, an application with which different users can talk to each other over the network in real time.

We will first provide a short description of the situation, the requirements for replication and a general solution.

Next we will discuss the implementation of the chat application. We will present the two classes of the clients: Zapper and TalkWindow, followed by the aspect languages specifying the replication and the instantiation of the framework.

7.2.1 Situation

In a chat application, users on different computers want to have a real-time written conversation with each other. These conversations are usually not one-to one conversations, but many to many. A often recurring analogy is that the users are sitting in the same room, and every user can hear what the other users are saying. Users also have the possibility to sit in different rooms at the same time and direct what they are saying to one particular room. We will call each of these rooms a `channel`.

For the conversation to run smoothly, people must be able to type a message whenever they want to. This means that even immediately after they have written a previous message, they must be able to send a new one. So the client may not wait for the message to be propagated to the other clients before a new message can be typed in and sent. The order in which new messages arrive at each client is not important, the only requirement is that messages coming from one person do not arrive out of order. In exceptional situations, it is even allowed for messages to be lost, as the users can easily ask someone to repeat something.

We will use replication in this system to achieve high client access speed to the data. Given that there are few restrictions on how the data must be kept consistent, the data consistency requirement can be kept low. This will

keep the overhead for replication low, and speed up client accesses to the data.

Consider the number and location of the replicamanagers. As we have said in 3.3, having one replicamanager per client, located as close to the client as possible, will achieve the fastest data access speed. Therefore, we will integrate the replicamanager in the clients' process, fixing both location and number of replicamanagers.

Although each chat application will contain both a client and a replica manager, these two elements are still considered as separate entities. Therefore we will still use the client and replicamanager terms to refer to the two parts of the application

7.2.2 The Zapper and TalkWindow

The chat client consists of two classes: Zapper and TalkWindow. Zapper allows channels to be listed and creates, and can also open a TalkWindow on a channel. A TalkWindow monitors the conversation in the channel, displaying each message as it appears, and allows the user to put a new message on the channel. First we will discuss the Zapper class, followed by the TalkWindow class.

Zapper

The Zapper class contains a Vector of Strings, named `thechannels`, which is the list of the names of the available channels. Because each Zapper must have the same list of available channels, `thechannels` will have to be shared amongst all Zapper objects. We will replicate `thechannels` to achieve this sharing of channels.

The constructor of the Zapper class mainly performs a lot of layout work for the user interface, the final statement however does not, and is of some importance. This statement will verify if `thechannels` equals `null`. If this is so, this means that `thechannels` has not been initialized yet, so it must be initialized.

After Zapper has been instantiated, it will mainly perform event handling. Whenever the "Quit" button has been pressed, the application will quit, when the "New channel" button has been pressed, a new channel will be added to the channel list using the name typed in the appropriate TextField. Also, when the "Connect" button has been pressed, a new TalkWindow will be opened on the channel selected in the appropriate Choice.

The complete Jav code of Zapper is included below:

```
public class Zapper extends Frame implements ActionListener
{
   private Vector thechannels;
```

```
protected static Zapper thezapper;

protected String username;
protected Button newbutton, connectbutton, quitbutton;
protected TextField newfield;
protected Choice channels;

public Zapper(String user, String repl_id)
{
   //Layout main window
   this.setLayout(new GridLayout(3,2));
   channels = new Choice();
   newfield = new TextField();
   newbutton = new Button("New channel");
   newbutton.addActionListener(this);
   connectbutton = new Button("Connect");
   connectbutton.addActionListener(this);
   quitbutton = new Button("Quit");
   quitbutton.addActionListener(this);
   add(channels); add(newfield);
   add(connectbutton); add(newbutton);
   add(quitbutton);
   setTitle("IRC App"); pack(); show();
   username = user; thezapper = this;

   //initialize channels only if needed
   if (thechannels == null)
      thechannels = new Vector();
}
public void actionPerformed(ActionEvent evt)
{
   Object source = evt.getSource();
   if(source == quitbutton)
      System.exit(0);
   else if(source == newbutton)
   {
      Vector chans = thechannels;
      String newchan = newfield.getText();
      if (chans.indexOf(newchan) == -1)
      {
         chans.addElement(newchan);
         thechannels = chans;
         refresh();
      }
```

```
      }
      else if(source == connectbutton)
      {
         new TalkWindow(username,channels.getSelectedItem());
      }
   }
   public static void refr()
   {
      if(thezapper != null)
         thezapper.refresh();
   }
   public void refresh()
   {
      channels.removeAll();
      Vector newchan = thechannels;
      for(int i=0; i<newchan.size(); i++)
         channels.add((String)newchan.elementAt(i));
   }
}
```

Note that when the Zapper is instantiated it will be given the name of the user using the application, this to identify the messages each user sends. Also, we have chosen that the name of Zappers' replicagroup will always be `Chan_list_keeper`. This because all Zapper instances must all share the same list of channels. Whenever a Zapper is instantiated, the name for its replicagroup will be `Chan_list_keeper`.

Now consider the following problem: the Zapper must be made aware when the list of channels has been changed, so it can update its display of available lists. The `refr` class method will ensure the updated version is displayed, whenever Zappers' replica has been changed. How and when this method is called will be discussed in 7.2.4.

As for replication transparency, the transparency for Zapper can be considered to be high. This because the only things to be kept in mind while writing the Zapper class is to include the `String repl_id` formal parameter in the constructor, and not to make any changes directly to `thechannels`.

TalkWindow

TalkWindow displays the ongoing discussion of channel in a TextArea, by adding new messages to the text in the TextArea whenever these new messages are received. The user can also participate in the discussion by typing in text in a TextField, which will be sent to all other participants in the discussion.

New messages being received and sent are stored in one String variable

called `lastmessage`. This String will be replicated, so whenever a user writes a message to `lastmessage`, the other clients will automagically be made aware of this new message, and add it to the TextArea. Once this message has been added to the TextArea, it is no longer needed, so it can safely be overwritten by another message, which will then be added to the TextArea, and so on.

Whenever TalkWindow is instantiated, it will perform some user interface setup, and will add itself to a dictionary of all TalkWindows, the purpose of which will be discussed later.

After TalkWindow is instantiated, it will mainly perform event handling. When the enter button is pressed, the message in the TextField will be read. If this message is equal to "/quit", the application will quit, if the message is equal to "/leave" the TalkWindow will be closed and the user will leave the channel. If the message does not equal either of these strings, the message will be prepended with the users' name and put into `newmessage`, so all other clients will be updated with this new message.

Whenever a new message is received by a TalkWindow replica, the `newMessage` class method will be called. This method will determine which TalkWindow replica is changed, and ensure this TalkWindow adds this new message to its TextArea .How and when the `newMessage` method is called will be discussed in 7.2.4.

The complete Jav code of TalkWindow is included below.

```
public class TalkWindow extends Frame implements KeyListener
{
   private String lastmessage;

   protected static Hashtable thewindows = new Hashtable();

   protected String my_id;
   protected TextField newfield;
   protected TextArea msgarea;
   protected String username;

   public TalkWindow(String name, String repl_id)
   {
      username = name;
      BorderLayout layout = new BorderLayout();
      newfield = new TextField();
      newfield.addKeyListener(this);
      msgarea = new TextArea(20,80);
      msgarea.setEditable(false);

      layout.addLayoutComponent(newfield,BorderLayout.SOUTH);
```

```
      layout.addLayoutComponent(msgarea, BorderLayout.NORTH);
      setLayout(layout);
      add(msgarea); add(newfield);
      setTitle("Channel "+repl_id);
      pack(); show();
      my_id = repl_id;
      thewindows.put(repl_id, this);
   }
   public void keyTyped(KeyEvent evt)
   {}
   public void keyPressed(KeyEvent evt)
   {
      if(evt.getKeyCode() == KeyEvent.VK_ENTER)
      {
         String newtext = newfield.getText();

         if(newtext.equals("/quit"))
            System.exit(0);
         else if(newtext.equals("/leave"))
            dispose();
         newtext = username+" : "+newtext+"\n";

         lastmessage = newtext;
         newfield.setText("");
      }
   }
   public void keyReleased(KeyEvent evt)
   {}
   public void addMessage()
   {
      msgarea.append(lastmessage);
   }
   public static void newMessage(String id)
   {
      TalkWindow win  = (TalkWindow)thewindows.get(id);
      if(win != null)
         win.addMessage();
   }
}
```

Note that the formal parameter `String repl_id` is equal to the name of the channel on which this TalkWindow is opened, so each replicagroup is a channel.

Also, TalkWindow need not know that `newmessage` is replicated, only

that this field may be changed by some external object, which will be indicated by a call to the `newMessage` class method. The only replication element which must be kept in mind is providing the `String repl_id` formal parameter in the constructor, which is the name of the channel being used. Therefore we feel we can state that the replication for TalkWindow is very highly transparant.

7.2.3 The Replication Aspect Code

We will now present the replication aspect code for both Zapper and TalkWindow.

Zapper

Recall that Zapper contains a Vector of Strings named `thechannels`, which contains the list of channels currently available. Because these channels must be accessible from all clients, this list must be replicated.

The following Dupe code will insure this:

```
Replicate Zapper
{
   field thechannels replicate;
}
```

Now consider the errors that can occur while accessing this replicated field. Because the replicamanager and the client are integrated in the same process, no errors will occur, so the exception handlers may safely be left empty. However, as we have described in 5.3.3, because the Java compiler requires a return statement at the end of the code, we have to provide the `return null` statements for the exceptions thrown while reading or writing the replicated data.

```
Replicate Zapper
{
   field thechannels
   {
      read(VoyagerException ex)
      BEGIN
      return null;
      END.
      write(VoyagerException ex)
      BEGIN
      return null;
      END.
   }
}
```

TalkWindow

As has been said above, TalkWindow replicates the new messages, so all clients can add these to their transcript of the conversation.

To replicate the new messages, the following Dupe file is needed:

```
Replicate TalkWindow
{
   field lastmessage replicate;
}
```

As stated for Zapper, there is no need for exception handlers in the code, but they are still required to be able to compile the code correctly. The following Fix file will ensure this:

```
Replicate TalkWindow
{
   field lastmessage
   {
      read(VoyagerException ex)
      BEGIN
      return null;
      END.
      write(VoyagerException ex)
      BEGIN
      return null;
      END.
   }
}
```

With all aspects defined for both Zapper and TalkWindow, the aspect weaver must be used to generate the final Zapper and TalkWindow Java file, along with their replicas. We will not include this code here, as it can easily be deduced from the description of the aspect weaver (5.4).

7.2.4 Instantiating the Framework

To instantiate the framework, six concrete classes must be created. Four concrete classes are needed for the consistency algorithm: read and write strategy, queuing strategy and locking strategy. Fifth and sixth strategy are the RMListManager and the statistics algorithm.

As the requirements for the application include no statistics, the class which is used for keeping these statistics can simply do nothing. The method bodies of all methods in this class can be left empty.

The number and location of ReplicaManagers is not fixed in the chat application, since there is one ReplicaManager per client, integrated in the

clients' process. Therefore, the RMListManagers' `listRMsFor` method will have to dynamically create a RemoteList of the ReplicaManagers which are currently active. This could be done by first having a list which specifies the possible network locations of clients, and verifying if on these network locations a client is active. We have used this technique in our experiments, other options are e.g. to have a known server which keeps track of active clients, and let the RMListManager request a list of active clients.

Recall that the data consistency requirement is low for the chat application, and that the focus is primarily on data access speed. To achieve this, the writestrategy can asynchronously pass on updates to the remote replicamanagers, ensuring the client can immediately write a new update to the local replicamanager.

Given the low consistency requirement, there is no need for the hold-back queue to hold back incoming updates, so these will be applied immediately. Also, given the low consistency requirement, there is no need for any locking. To ensure high data access speed by the clients, reading the data must be done only from the local replica, and not from any remote replicas.

This allows us to reuse parts of the code for the previous instantiation of the framework, speeding up development time. The previous locking and read strategy can be reused without modifications. Because of this, we will not repeat this code here.

The code of the queue will be discussed in 7.2.4, because the queue will also be used to notify the client of changes made to the replicas. We will now provide the code for a possible writestrategy. This strategy will not do asynchronous updates as suggested above. A strategy which writes updates in an asynchronous fashion can easily be implemented, by letting the updates to other ReplicaManagers run in a separate thread. This somewhat clutters the code, so we have included the synchronous version for clarity.

```
public class SynchWrite extends WriteStrategy
{
   public void writeValue
      (AbstractReplica repl, String thefield,
       java.io.Serializable value, RemoteList rmlist)
   {
      // write locally to the queue
      try
      {
         repserver.ReplicaManager.giveRM().
            doUpdate(repl.repl_id, "", thefield, value,
             new Integer(0));
      }
      catch (NoReplicaException ex)
      {
```

```
         //will not be thrown
      }
      catch (LockedException ex)
      {
         //will not be thrown
      }
        // Obtain an iterator for the RemoteList
         RLIterator iter = rmlist.newForwardIterator();
         if (!iter.isEmpty())
         do
         {
            try
            {
               // Pass on this change to the remote
               // replicamanager
               this.remoteWrite(iter, repl,
                  "", thefield, value, new Integer(0));
            }
            catch (LockedException lex)
            {
            // We assume no locking is used,
            // so this exception will not be thrown
            }
         }
         while(iter.next());
         rmlist.delIterator(iter);
   }
}
```

The important element of this code is that changes to the local replica are not made directly to the replica, but put in its queue by making the appropriate call on the replicamanager. This will ensure the client is notified of its own changes, which allows it to add its own messages to the transcript of the conversation.

Change Notification

Recall that both Zapper and TalkWindow need to be notified of changes to the replicated data, by respectively calling the `refr` or `newMessage` methods.

So whenever Zappers' replica is modified, `refr` must be called, and whenever a TalkWindow replica is modified, `newMessage` must be called. These methods will then ensure the change is properly processed.

The best place for this to happen is in the hold-queue. Because the queue determines when updates are applied, it can easily perform a method call

after these applications.

For the chat application, only the `doUpdate` method on the queue will be called, so we only include this code:

```
public Serializable doUpdate
   (String owner, String field,
    Serializable value, Integer level)
   throws LockedException
{
   synchronized(therepl)
   {
      this.applyUpdate(owner, field, value, level);
      if(therepl.repl_id.equals("Chan_list_keeper"))
         Zapper.refr();
      else
         TalkWindow.newMessage(therepl.repl_id);
   }
   return null;
}
```

This code will either call the `refr` or `newMessage` method, based on the id of the replica. Recall that `Chan_list_keeper` is the name of the replicagroup containing the list of available channels, and that the name of a channel is equal to the name of its replicagroup.

Having the code block synchronized on `therepl` ensures that the value of the replica will not be changed by any other concurrent process while the `refr` or `newMessage` method are executed. This ensures correct updating by the clients at all times.

With this implementation of the queue, the instantiation of the framework is complete, and the chat application can be used.

7.2.5 The Complete Application.

Since the chat application integrates both client and replicamanager, both elements will have to be started up from one application.

The main code for the application has already been given above, when Zapper, TalkWindow and the instantiation of the framework were discussed. What remains to be done for the application is to start up the replicamanager and to instantiate a Zapper object, providing the users' name and `Chan_list_keeper` as the name of the objects' replicagroup.

7.2.6 Omissions and Improvements

The interesting element of the chat application proved to be change notification. Change notification has been possible because the replicamanager and the client were running within the same process, so the replicamanager could call a number of class methods on the clients' code.

Calling the clients' class methods is however not included in the protocol we defined between the frontend of the client and the replicamanager. If client and replicamanager were not running inside the same process, this would not have been possible. So to allow change notification in these cases, the protocol between frontend and replicamanager must be extended.

This would require a number of substantial changes to the code of both the frontend and the replicamanager. This because the current protocol between frontend is basically one way: the frontend invokes calls on the replicamanager, never the reverse. Were we to extend the protocol, the replicamanager would need to be able to make calls to the frontend to indicate changes in the replica. This would, amongst others, require the replicated object to export some of its methods so they can be called by the replicamanager and the replicamanager to connect itself to this replicated object on the client.

Although these changes are feasible, they are quite extensive, and we estimate they will be used rarely. Therefore they have not been implemented and will not be covered here.

A possible intermediate solution would let the clients use polling to determine changes to the replicated fields, but this would be both CPU and network-intensive for both client and replicamanager, which would slow down the system.

7.2.7 Conclusion

We have now implemented a distributed chat application, where multiple users can converse not just on a one to one basis, but also in a many to many fashion.

The main requirement for the distributed chat application was the need for high interactivity of the clients. We have achieved this by integrating client and replicamanager into one process and by using a consistency algorithm which did not result in a high degree of data consistency.

To notify the chat application changes had been made to the replicas, a form of change notification has been implemented. This implementation can however not be used in all instantiations of the framework. Extending the framework to enable event notification is possible, but will require some work.

Also, we have seen that replication transparency for the different client applications can again be considered to be very high.

7.3 Results

In this chapter we have validated the claim that the replication framework can be instantiated for a wider range of applications.

We have implemented two widely differing applications requiring replication: a distributed warehouse and a distributed chat application. Both implementations were easily realized and had a high degree of replication transparency.

From these results, we feel we can state that our replication framework can easily be used to implement the replication part of a variety of distributed systems. However, due to the lack of explicit locking from the client, a number of applications can not be implemented correctly.

To further increase the usefulness of the framework we have proposed two possible improvements: explicit locking and change notification. Explicit locking can easily be implemented, change notification will require some more work.

Chapter 8

Conclusions and Further Research

8.1 Summary

In a distributed system a number of computers, linked by a network, cooperate to achieve a common goal. An important problem in these distributed systems is how data can be shared between the different computers which make up the distributed system.

One possible methodology to share this data is replication. Until now, replication has been custom-built for each distributed system which required it. In this thesis we developed a framework for replication to ease development of such distributed systems.

We first defined frameworks as a means to reuse the same design in many applications within a given domain. A useful element in making these frameworks and in instantiating them into applications are design patterns. Design patterns capture expert knowledge into customizable solutions for a variety of reoccurring problems. Some of these patterns can be used quite successfully in the creation and instantiation of frameworks. We discussed three of these useful patterns.

Creating a framework for an application domain requires knowledge of the commonalities and variabilities in the domain. To gain this knowledge we studied the domain of replication. This study showed that replication is useful when fault-redundancy of the data or high speed access to the data is required. We also studied a number of possible choices which have to be made for realizing different parts of the replication strategy, of which the consistency requirement is the most important. The last element of the study of replication was an abstract model for replication in which the above choices are not fixed, which makes it a suitable candidate to base a framework on.

Our study also revealed that an important element in replication is replication transparency. Replication transparency defines the 'awareness' of an

application that its data is replicated. For ease of development, the application should not be aware that its data is replicated, so it need not be concerned with the possible complications resulting from replication.

A possible option to realize a high level of replication transparency lies in the concept of separation of concerns. Separation of concerns entails separating the different requirements which an application must meet into different concerns. Because these concerns can be treated separately, reasoning about them is easier, as the programmer need not cope simultaneously with the different concerns and reason about the fashion in which they interact.

A technique which has been developed to ensure a clean separation of concerns is aspect-oriented programming. In aspect-oriented programming each concern, called an aspect, is specified separately. These specifications are typically done in special-purpose aspect languages, which allow the programmer to reason about the concern in a natural form. A tool, called an Aspect Weaver, then combines these different aspects into an executable form.

We used aspect-oriented programming to achieve a high degree of replication transparency in our framework for replication. We first analyzed how replication could be seen as an aspect and how we could define an aspect language in which this could be specified. This analysis revealed two major obstacles to totally transparent replication: naming of the replicagroups and initialization of the replicas. Once the analysis was completed, we developed two special-purpose aspect languages: Dupe and Fix. These languages can respectively be used to specify what must be replicated and what must be done in case of errors. We have also implemented an aspect weaver which combines code written in Jav: a variant of Java, with Dupe and Fix files to produce Java code which includes the replication aspect. Although we could not achieve full replication transparency due to the naming and initialization obstacles, replication transparency is high, which allows this approach to be used.

Basing ourselves on the abstract model for replication we have discussed in our replication study, we developed a framework for replication. In this framework, a number of the choices presented in the study have been fixed, but the most important choice of data consistency has not. The consistency algorithm has been split up into four parts, which can easily be modified to meet the required data consistency requirement. This allows the framework to be instantiated for a wide variety of possible applications.

To validate this claim, we have created two differing applications: a distributed warehouse and a chat application. Whereas the distributed warehouse had a requirement for high data consistency, the consistency requirement for the chat application was low. Conversely, the client interaction speed for the chat application had to be as high as possible, whereas client interaction speed for the distributed warehouse did not need to be high. We have shown how the framework can be instantiated to satisfy the re-

quirements for each of these two applications. An evaluation revealed that an explicit locking facility for the clients is needed for the distributed warehouse, which is not provided by the framework. Therefore, although a wide variety of applications can be realized, a number of applications can not, due to the lack of explicit locking. We have however argued that adding this explicit locking to the framework will not be difficult.

8.2 Further Research

An important area which needs to be developed further is aspect-oriented programming. At the moment only a limited number of concerns can be described in aspect languages, leaving concerns which are not addressed, to be coded in the 'base aspect'. This implies that not all concerns are fully separated, so the programmer cannot reason separately about all concerns.

Also, since the aspect weaver combines several aspect languages to produce executable code, tracing bugs in this code back to their origin in the aspect languages is nontrivial.

A possible solution would be to develop a debugger which would trace back errors to the aspect languages, and to the code generated by the weaver. We can envision how this could be done for the code generated by our weaver, however other aspects may be hard to trace back from weaved code. It could also be possible that multiple pre-processors have sequentially performed a number of code transformations, which will significantly complicate the work of the debugger.

Some extra work can be performed on our aspect weaver, most importantly the weaver could perform some checking of the validity of the exception handlers specified in the fix code, and it could be made possible to replicate all fields, not just `private` fields, by letting the weaver weave all files of the client application.

Further research on increasing replication transparency can be performed, an area which merits some attention here is naming the replicagroup for a replicated object and performing initialization of the replicas.

Last but not least, some work can be performed to increase the usefulness of the replication framework. Adding explicit locking to the framework is an element which certainly must be considered. Also, the framework could be extended with active replication, event notification, and it could also be made possible to assign different consistency algorithms to different replicas.

8.3 Conclusions

In this dissertation we created a framework for replication which can add replication to a system with a minimal impact on existing code.

To be able to make replication as transparent as possible to the code using replicated data, we used an Aspect-oriented approach. We defined three aspect languages and implemented an aspect weaver capable of weaving code in these languages into Java code.

Although the transparency of replication was very high, there were two hurdles which prevented full replication transparency: naming of replica-groups and initialization of replicas. These hurdles require further study to determine if and how they can be eliminated.

In the implementation of our framework for replication, we allowed for a wide range of data consistency requirements by splitting up the data consistency algorithm in four parts. In an instantiation of the framework, implementations for these parts can be mixed and matched to form the required consistency algorithm.

We have shown that this framework can be instantiated for a variety of distributed applications, although adding explicit locking would increase the frameworks' usefulness.

Bibliography

[1] Kent Beck and Ralph Johnson. Patterns generate architectures, 1994.

[2] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns.* John Wiley & Sons, 1996.

[3] Wim Codenie, Koen De Hondt, Patrick Steyaert, and Arlette Vercammen. From custom applications to domain-specific frameworks. *Communications of the ACM*, october 1997.

[4] James O. Coplien and Douglas C. Schmidt. *Pattern languages of program design.* Addison-Wesley, 1995.

[5] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems, Concepts and design.* Addison-Wesley, second edition, 1994.

[6] Serge Demeyer. *Zypher: Tailorability as a link from object-oriented software engineering to open hypermedia.* PhD thesis, Vrije Universteit Brussel, Faculteit Wetenschappen - Departement Informatica, 1996.

[7] Erik Dirkx. Concurrent systems. Course notes, Vrije Universiteit Brussel.

[8] Distributed.net. Node zero. http://www.distributed.net.

[9] Martin Fowler. *Analysis patterns : reusable object models.* Addison-Wesley, 1997.

[10] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[11] Walter L. Hürsh and Cristina Videira Lopes. Separation of concerns, February 1995. College of Computer Science, Northeastern University.

[12] Objectspace Inc. Objectspace Voyager core technology. http://www.objectspace.com/voyager.

[13] International Business Machines Corporation. The San Fransisco Project. http://www.ibm.com/sanfransisco/.

[14] John Irwin et al. Aspect-oriented programming of sparse matrix code, 1997. Xerox Palo Alto Research Center.

[15] Ralph E. Johnson. How to develop frameworks. OOPSLA 1993 Tutorial notes, ACM press.

[16] Gregor Kiczales et al. Aspect-oriented programming, a position paper, 1996. Xerox Palo Alto Research Center.

[17] Gregor Kiczales et al. Aspect-oriented programming, 1997. Xerox Palo Alto Research Center.

[18] Jürgen Kleinöder and Michael Golm. Transparant and adaptive object replication using a reflective Java. Technical report, Friedrich-Alexander-University, Erlangen-Nürnberg, 1996.

[19] RSA laboratories. RSA laboratories challenges. http://www.rsa.com/rsalabs/html/challenges.html.

[20] Geert Lathouwers. Ontwerp-technieken voor een transparant gedistribueerd objectmodel gebruik makend van replicatie. Licentiaatsthesis, Vrije Universiteit Brussel, Faculteit Wetenschappen - Departement Informatica, 1997.

[21] M. C. Little and D. L. McCue. The replica management system: a scheme for flexible and dynamic replication, 1994. Department of Computing Science, University of Newcastle upon Tyne.

[22] Mark C. Little and Santosh K. Shrivastava. Object replication in Arunja, August 1993. Department of Computing Science, University of Newcastle upon Tyne.

[23] Cristina Videira Lopes. *D: A language framework for distributed programming.* PhD thesis, Xerox Palo Alto Research Center, 1997. Draft version.

[24] Anurag Mendhekar, Gregor Kiczales, and John Lamping. RG: a case-study for aspect-oriented programming, 1997. Xerox Palo Alto Research Center.

[25] Sun Microsystems, Inc. The Java object serialization documentation. http://java.sun.com/products/jdk/1.1/docs/guides/serialization/.

[26] Sun Microsystems, Inc. Java platform application programming interface. http://java.sun.com/products/jdk/1.1/docs/api/packages.html.

[27] Sun Microsystems, Inc. The Java platform reflection documentation. http://java.sun.com/products/jdk/1.1/docs/guide/reflection/.

[28] Sun Microsystems, Inc. JavaBeans: the only component architecture for Java. http://java.sun.com/beans/index.html.

[29] Werner Van Belle. Short term transactions. Report for MediaGeniX, January 1998.

[30] Cristina Videira Lopes and Gregor Kiczales. D: a language framework for distributed programming, 1997. Xerox Palo Alto Research Center.

[31] John M. Vlissides, James O. Coplien, and Norman L. Kerth. *Pattern languages of program design 2.* Addison-Wesley, 1996.

[32] Xerox Palo Alto Research Center. *AspectJ user manuals*, december 1997. Draft version.