Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes - France
and
Universidad De Chile - Chile
1999

Full distribution transparency, without losing control - AOP to the rescue.

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange
project funded by the European Community)

By: Johan Fabry

Promotor: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promotor: Dr. José Piquer (Universidad De Chile)

Acknowledgments

I would like to take this opportunity to thank the people who helped to make this work possible:

First of all, many thanks to all the organizers of the EMOOSE project. If it were not for their vision and their commitment, this masters thesis, and I suspect a large number of other master thesises, would never have existed. The EMOOSE project was a fantastic opportunity for me, and I strongly recommend it to all Computer Science students!

Thanks to my promotor Prof. Dr. Theo D'Hondt, for taking not only me, but all EMOOSE students under his wings. I never realized they were that large, but apparently they do extend halfway around the globe. Also thanks to my co-promotor Dr. José Piquer, for suggesting a subject, and allowing me to deviate a little from it. I especially appreciated the freedom I was given to work on my own, and the ability to have a thorough discussion with José when needed, even though his schedule always seemed to be overflowing.

A number of people have given some of their precious time to proofread this thesis: Xavier Alvarez, Marina De Vos, Kim Mens, and Patrick Steyaert. They all assure me that the fact that all but one went on holiday during the final phases of writing is a coincidence, and not a result of my writing style.

I would like to thank the people of the supporting institutions, the Ecole des Mines de Nantes and the Universidad de Chile for helping me whenever they could, and for generally being able to support me during the time I was at these locations.

Thanks to my fellow students, both in Nantes and in Santiago. We had some good times together, but now I'm afraid I have to move on. Rest assured, I will not forget you!

Last, but not least, thanks to my parents for allowing me to take this master, for fully supporting me all through it, and helping where they could. I will never forget their reaction when I told them I might get a chance to study abroad for a year: "Cool! Can we come and visit?" I said: "Of course." And so they did.

For the people wondering where I got the strange names I use for the servers in the examples: Akivi, Tongariki and Vinapu are ahus or platforms on Easter Island. It was on these platforms that a number of moai were erected in honor of the ancestors, which were buried underneath the platform. Akivi, also known as "the seven moai", is remarkable because it is the only ahu on which the moai do not stand with their backs to the sea. Tongariki is the largest restored ahu, an immensely long platform, containing fifteen moai, each moai easily over four meters tall. Vinapu is an ahu which is said to resemble much the walls of the Incas in the way the blocks are fitted together. Although it **is** possible to slide a razor blade between the blocks at some points, we'll not be picky about it.

Now enough archeology and anthropology, and on with Computer Science!

Contents

Chapter 1

Introduction

In the Beginning there was nothing, which exploded.
— Terry Pratchett, "Lords and Ladies"

The evolution of hardware technology, the economies of scale and the advent of cheap and convenient networks have significantly altered the view on computing in the last decade.

In the early days of computing the attention was focused more on a single, independent computer which performed all the processing needed to accomplish a certain task. However, it is now becoming increasingly more attractive to distribute this task amongst different machines in a network.

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.

An example of such a system which has recently been in the news, is the SETI@Home system [20], organized by the Search for Extra-Terrestrial Intelligence project (SETI). This project uses radio telescopes to scan the heavens for radio signals of extraterrestrial origin. The goal of this project is to locate radio sources of extraterrestrial origin which are clearly not natural phenomena and not generated by a human artefact. These signals would therefore indicate an extraterrestrial intelligence.

An important issue here is the processing of the signals received from the telescopes. The raw data produced by the telescopes needs to be filtered in various ways to eliminate background radiation, remove earth-based and

satellite-based signals, and to detect signals which are deemed to be 'interesting'. This filtering requires a high amount of computing power, for which the SETI project does not have sufficient resources.

The SETI@home system was built to overcome this problem. The system consists of a server program, dividing the data into small packets to be processed, and a client program, which uses the Internet to fetch the packets from the server, process them, and send the result back, again using the Internet. The idea is that people who are connected to the Internet donate their excess CPU time by running the freely distributable client as a background process on their machine.

This project has proven to be quite popular, given that at last count (8th of July 1999) there were 782999 participants, spread out over 216 countries. This leads the SETI@home organizers to proudly claim that "SETI@home is now our planet's largest supercomputer".

Another, less spectacular, class of distributed systems is the class of the, so called, "group-ware" applications. In these kinds of applications a number of users work together on a common task, but each user works on her individual computer. This class of distributed systems contains a wide range of applications. A basic application would be a simple messaging service for a group of users, allowing immediate communication between group members. More advanced are the systems where different users work concurrently on the same data, for example a word processor which allows different persons to work on the same document, at the same time. Another example of more advanced systems would be planning applications, where each user has certain rights and responsibilities in the planning process and the group as a whole creates the plan. And last, but not least, we can consider the software development process where a group of developers work together to create a program; these people collaborate to form the analysis, design and implementation of the program.

When building a distributed system, an important issue is how the aspect of distribution is treated. Ideally, the fact that the system is distributed should not pose an extra burden on the programmer. The programmer should primarily keep in mind the functionality of the system and not need to worry about secondary requirements generated by the distributed nature of the application. This unless she is specifically working on the distribution part, where it will be the primary concern. This idea is usually referred to as "distribution transparency" [2].

To achieve this transparency, a number of abstractions have been developed to separate the concern of communication between the different programs in the distributed system from the base functionality. The most

popular abstraction is the remote procedure call (or RPC [2]). When issuing a remote procedure call, execution of the procedure is not done on the machine issuing the call, but on a different, remote, machine. Parameters and return value are transported, in a more or less transparent fashion, between the calling and the called program, which allows the called program to work with the data, and provides the calling program with a return value, if any.

There are a number of RPC mechanisms available for Java, each providing a different degree of distribution transparency. However, none of these packages provide full distribution transparency while providing the programmer with a large degree of control. A recurring obstruction to full transparency is the error-handling required for remote procedure calls.

The reason for this symptom can be tracked down to the fundamental difference between a normal method call and a remote method call. The fact that the object executing the method is located on another machine has a number of consequences. Of these consequences, partial failures are considered by many [26, 7] to be the defining problem for distribution transparency. These people claim that it is impossible to achieve distribution transparancy because of partial failures.

In a non-distributed system partial failures do not occur; if the machine running the program fails to operate, the program will, obviously, cease to operate. However, in a distributed system failures are not always total, they may be partial: it is entirely possible that one computer fails while the others keep on running, or that a part of the network is down, making it impossible to communicate with a number of computers. Therefore it must be taken into account that all remote method calls may fail, even, and especially, where the local method calls would not fail.

To be able to achieve distribution transparency we should be able to reason about the program without continously having to worry about possible errors at every method invocation. However, this does not mean we can ignore that these errors may happen. At a certain point in the development we should be able to concentrate on these failures and specify what to do when an error occurs, i.e. we should be able to handle these errors separately from the main program.

The idea here is to separate the concern of distribution out of the main application. If this is achieved we can reason separately about, on one hand, the core functionality of the application, and on the other hand, the distributed properties of the application.

One technique proposed to achieve a clear separation of concerns [9] is Aspect-Oriented Programming or AOP [16]. In Aspect-Oriented Programming each concern is expressed separately in a (possibly special-purpose)

aspect language. This allows a programmer to reason more easily about each aspect. To create an executable program, the different aspect descriptions are compiled using a special-purpose tool, called an Aspect Weaver^{TM}.

In previous work [3, 4, 5] we have implemented a framework for replication using AOP. In that work we have shown that AOP can be used to achieve a high degree of replication transparency. Not to be confounded with distribution transparency, replication transparency ensures that a program using replicated data is unaware that any replicas of the data are being kept, and is completely unaware of what technique is being used to keep the replicas consistent [3]. It is clear that both transparency concepts are related, therefore we feel that AOP can succesfully be used to also achieve distribution transparency.

Applying the above technique to obtain a higher degree of distribution transparency entails defining the different languages and implementing an Aspect Weaver. The Aspect Weaver can then be used to combine the base program, defining the core functionality, with the aspect code, defining the distribution aspect which includes the error-handling. This allows the distribution aspect to be specified separately, making it possible to add this concern in a separate stage of the software development.

An alternate way of viewing the above is the following: If the source of a program is compiled with a standard Java compiler, which ignores the separate aspect specifications, the system will not be distributed and will consist of one process running on one machine. However, if the aspect weaver is used to compile the sources, the result will be a distributed system.

1.1 Goals

We will use the methods proposed by AOP to achieve a higher degree of distribution transparency for programs written in Java. The goal is to make it much easier for the programmer to write a distributed application than it currently is, using RMI or another distribution package. Specifically we will concentrate on what these packages do not handle adequately: error-handling.

This work differs from our previous work [3, 4, 5] in that the previous only addressed the specific concern of replication, with emphasis on easy implementation of different replication algorithms. Building on previous experience, this work is more general and addresses all kinds of distributed systems, placing emphasis on treating error-handling in a user-friendly fashion. In 8.2 we will discuss how the previous work can be combined with

the results of this thesis.

We will define three aspect languages: Dist, Serv and Fix, which allow the programmer to specify the aspect of distribution separately from the main program, which is written in Java.

We will implement an Aspect Weaver for these languages, which will generate Java source files and use the javac compiler to generate class files. Also, it will use Java's RMI as a transport layer for the remote method invocations.

To validate the functionality and ease of use of our solution, we will develop a number of small distributed systems using our aspect languages and our Aspect Weaver.

1.2 Overview

The next chapter will introduce a number of concepts relevant in the field of distributed programming. We will first present some characteristics deemed important for a distributed system, such as distribution transparency, and continue with an overview of how remote procedure calls are implemented.

Chapter three will discuss a number of packages which provide support for building distributed systems in Java. We will concentrate on user friendliness and the degree of transparency achieved when using these packages.

The fourth chapter will introduce aspect-oriented programming. We will show the need for separation of concerns, and show that a number of concerns can not be decomposed in the same fashion as others. Secondly, we will present aspect-oriented programming as a technique to achieve separation when this decomposition fails for some concerns. To conclude we will discuss why we feel that distribution transparency can be achieved using AOP.

In chapter five we will first present some general issues pertaining to the AOP system we developed, and will subsequently introduce the three aspect languages we defined: Dist, Serv and Fix.

Chapter six will discuss the modifications which the weaver makes to the base code, to integrate the different aspects. We will discuss the various steps of the process which transforms a local class into a remote class.

In the seventh chapter we will validate our claim that we achieve a higher degree of distribution transparency. We will create two distributed systems: a messaging application and a distributed library, using our AOP system.

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

Chapter 2

Concepts

It takes 20 years to make an overnight success.
— Eddie Cantor

The field of distributed systems has a number of concepts and some terminology which are not widely used in other fields of computer science. This chapter will introduce some of the most important concepts and terminology of the field. We will first discuss some characteristics which are considered as being important for distributed systems; resource sharing, fault tolerance and transparency. We will continue with describing the remote procedure calls, the widely-used paradigm for interaction within distributed systems, and detail the support needed to be able to invoke them.

2.1 Important Characteristics of Distributed Systems

The usability of a distributed system is primarily determined by a small number of base characteristics [2]. Of these the most important are resource sharing, fault tolerance and transparency, which we will now briefly introduce.

2.1.1 Resource Sharing

The term *resource* is widely used in computer science, and therefore has a large number of definitions. In the field of distributed systems, a resource

usually refers to a 'thing' that is shared amongst the different users of the distributed system. These 'things' can be hardware entities, such as disks or printers, or software entities such as databases or files, and even abstract concepts as CPU time.

It is clear that sharing physical resources, such as backup devices, printers, scanners, and other expensive hardware, implies a significant reduction in the cost of the system. Because of this cost-effectiveness a lot of computer systems, even those not normally considered as being a distributed system, provide a number of basic sharing services, such as a shared printer service.

Sharing a software construct, such a database or a file, is of utmost importance in group-ware applications (sometimes also referred to as Computer Supported Cooperative Working). Since all users are working towards a common goal, the data defining what that goal is, and the data pertaining to the status of the project has to be available to all users. Therefore this class of applications depend heavily on shared data.

Resource sharing is obviously a very important element of a distributed system. If there were no resources to share between the different machines of the distributed system, there would be not only no reason for them to cooperate, but also no way in which they could, and therefore there would be no distributed system at all.

Models for sharing resources

In a distributed system, we can make an abstract division between programs providing a service, i.e. a shared resource, and programs using this service. The users of a resource will communicate with a provider to access the shared resource which is managed by that provider.

This basic interaction can be placed in two different models of the working of a distributed system; the client-server model and the object-based model [2].

The *client-server model* is a simple, widely-adopted model for distributed systems: there is a set of server processes, each managing a certain number of resources, and a set of client processes, using the resources provided by the server processes. Server processes may themselves, at a certain point, need access to services which are provided by another server. In those cases, the first server process becomes a client process of the other server process. It is therefore possible that a given process is both a client and a server.

In the above model, all shared resources held are managed by server processes, while client processes use these resources by issuing requests to the servers whenever needed. The servers will perform whatever processing

required and return a result to the client process.

The *object-based model* is similar to object-oriented programming; each shared resource is viewed as an object. Objects are uniquely identified and all objects are accessed in a uniform manner. This is in contrast to the client-server model, where each service can have a different access scheme. The fact that all shared resources are viewed in the same manner adds a higher degree of simplicity and flexibility to the model.

In this model, shared resources are managed by objects. Whenever a program needs to access a shared resource, it sends a message to the corresponding object, which will perform the needed processing and return a result. Note that here also, a certain object may not only provide services, but also need to use services provided by another object.

2.1.2 Fault tolerance

Sadly, computers sometimes fail. Whereas on a non-distributed system these failures are usually total, i.e. the computer stops working, therefore the program stops working, distributed systems are also prone to partial failures. A partial failure occurs when only a part of the distributed system, say, a limited number of computers, has failed.

Note that, since in a distributed system a service on a given computer can depend on a number of other services provided by other computers, this implies that services on computers which have not failed can also fail. Therefore it is usually desirable that the system is able to handle partial failures in an adequate way.

The design of fault-tolerant systems is based on two concepts: *hardware redundancy* and *software recovery* [2]. By providing extra, redundant hardware, this redundant hardware can take over whenever a failure occurs. This solution is costly, and therefore not advised for all situations. In some cases a form of software recovery can be used; the software can be designed to recover from these failures in a more or less graceful way. This recovery can range from reverting to a previous, known, state when an error is detected, over providing some default values as the result of the failed request, to informing the user that an unrecoverable error has occurred and shutting down gracefully.

Of course, in a given system, these two concepts can be combined, for example: for the main server machines, such as database servers, redundant hardware can be allocated, while for less important machines, such as a print server, the distributed system software may mark the service as temporarily unavailable and queue requests until the problem is fixed.

Total fault tolerance, i.e. the guarantee that the system will be fully operational at all times, is an extremely difficult goal. Usually the tolerance of faults of a system is measured in availability. The *availability* of a system is a measure which defines the proportion of time of which the system can be used.

One possible use for a distributed system is providing a higher degree of availability when faced with hardware faults. The idea is, that if a certain piece of hardware fails, whatever task that was using that hardware is switched over to some other hardware. Ideally this process would happen with a high degree of transparency, i.e. invisible to the user.

2.1.3 Distribution Transparency

A distributed system is usually built out of a collection of independent machines, however, it is perceived by the user, and by the application programs, as one whole. If there is *distribution transparency*, the fact that the system is built out of different components is hidden, and the system is only perceived as one, whole, entity [2].

In the book "Distributed Systems, Concepts and Design" [2] eight forms of transparency are given, providing "a useful summary of the motivation and goals for distributed systems". We will now briefly describe these forms.

Access transparency. When local resources and remote resources are accessed using the same operations, we have access transparency. A frequently used method used to obtain this transparency is the remote procedure call, which we will discuss in the next section. Basically, the idea is that using a remote resource is performed by executing a procedure call which apperars identical to a 'normal' procedure call, ensuring access transparency.

Location transparency. If the actual location of a resource (i.e. on what machine it is located) need not be known to be able to access it, there is location transparency. The only thing the programmer, or the user, has to do, is to request a certain service, without regard as to what machine can deliver that service. The service will be provided without the programmer, or the user, knowing on what machine the processing was performed.

Concurrency transparency. There may be several processes using the same resource, concurrently. In many cases, if a resource which was not designed to be used concurrently, is used in a concurrent fashion,

unexpected behavior results. This is a consequence of a form of interference between the different processes using the resource. With concurrency transparency there will be no interference or unforeseen effects between the different processes.

Replication transparency. If we can have multiple copies of data entities in the system without the application programs, or the users, having knowledge of these replicas, we have replication transparency. Replicas are frequently used to increase the performance, or to increase the reliability of the system. As said in the introduction, this was an important element of our previous work [3, 4, 5]. In 8.2 we will discuss how the previous work can be combined with the results of this thesis.

Failure transparency. When faults in the system are concealed, allowing programs to keep on functioning despite failures in the system we have failure transparency. A number of alternatives exist to provide failure transparency; one example is having the program switch over to a different replica if one is available.

Migration transparency. In a number of cases, it is advantageous to migrate some data or executable code between different machines. One example is moving an item 'closer' to a process which uses it frequently, this to increase access times to the data. If an item can be moved between different elements in the system without affecting the processes using this item, we have migration transparency.

Performance transparency. The performance of a distributed system can often be improved by off-loading processes from machines which are heavily loaded onto machines which have a lower load. This process is also known as "load balancing". If we can reconfigure the system to improve performance as loads vary, there is performance transparency.

Scaling transparency. If the distributed system can expand in scale when this is deemed necessary, for example by adding new server computers or adding more user workstations, and this expansion does not require a change in system structure or client algorithms, we have scaling transparency.

Of the above eight, access and location transparency are considered to be the most important. These two have the strongest effect on how the resources in a distributed system are used, and therefore we will concentrate primarily on these two when trying to achieve distribution transparency.

The combination of access and location transparency is sometimes referred to as *network transparency* [2]. In a distributed system with network transparency resources are used without regard of their distributed nature, i.e. the system behaves as if it was a non-distributed system.

A good example of a network transparency is e-mail. When composing a mail, the user need only know the e-mail address of the receiver (besides, obviously, the message the user wants to transmit). Physical location of the machine to which the user connects to read her e-mail is not necessary, and the process of sending a mail to a user on a remote machine is exactly identical to sending it to a user on the same machine. Note, however, that this is has not always been so. In the early years of the Internet, if a mail needed to be sent to a user on a different machine, it was necessary to specify the physical path from the senders machine, through all machines forwarding the mail, to the recievers' machine. These paths were known as *"bang paths"* [19], and knowledge of a good bang path could mean the difference between a mail arriving in a few days, and a mail arriving in a week. It is obvious that the current e-mail system, using only the address, is much easier to use than the old one, where the entire path had to be known. This clearly shows what advantages can be gained by introducing location transparency.

As stated above, access transparency is an important element of a distributed system. One method which aids in achieving this transparency is the remote procedure call, which we will discuss next.

2.2 Remote Procedure Calls

As introduced above, in a distributed system clients request a service to be performed by a server by sending a request message, and the server returns a result after this operation has been performed. It is clear that this interaction is highly similar to a normal procedure call, where a certain service is performed for the caller by the callee, and a result is returned, if required.

Due to this similarity, the above request-reply interaction can be represented as a *remote procedure call* or RPC. Conceptually, a remote procedure call is an extension of the normal procedure call. The extension is that when issuing a remote procedure call, the code is not performed on the machine of the caller but on a different, remote, machine.

Generally, a server process will make a number of procedures available to the clients. Each procedure represents a part of a service the server provides, and the client need only call the corresponding procedure. A complete service

may then be viewed as a group of procedures, declared in one interface. This allows a service to be considered as 'just another module' which is used by the client program.

2.2.1 Differences With Normal Procedure Calls

Although the aim of an RPC is to preserve as far as possible the semantics of a normal procedure call, there are a number of important differences between a normal procedure call and a remote procedure call [2, 26, 7]. We will outline these below:

- A remote procedure call is executed in a different environment from the environment of the caller. Therefore a number of variables available to the caller, such as global variables, are usually not available to the callee.

- Parameter passing is handled differently. Parameters are usually divided in input, output and in/out parameters. *Input parameters* are parameters copied from the caller to the callee. Any changes made to them by the callee will not be seen in the caller. This is equivalent to parameters passed by value in a normal procedure call. *Output parameters* are analogous to input parameters, but these are passed from the callee to the caller, replacing any possible value they had before the call. A prime example of an output parameter is the return value. *In/out parameters* are the combination of input and output parameters, they are copied from the caller to the callee, at the beginning of the call, and copied back at the end of the call. This can be considered analogous to parameters passed by reference in a normal procedure call.

  Defining if a parameter is input, output, or in/out is either performed in a special interface definition language, or in the used programming language itself. The last is obviously only possible if the language provides sufficient support for declaring interfaces and the nature of the parameters of each procedure in the interface.

- Because the callee does not execute in the same environment as the caller, direct memory references (i.e. pointers) cannot be passed as parameters, as they would not be correct in the other environment.

- To pass data structures from caller to callee and back, the data must be converted into a format which can be successfully transmitted over the

network. This implies that hierarchical data structures must be 'flattened' to a stream of bytes, which can subsequently be used to recreate (a copy of) the original data structure 'on the other side'. The process of taking a, possibly hierarchical, data structure and transforming it into a format suitable for transport over the network is called *marshalling*. Similarly, the process of taking the network suitable format and recreating the original data structure is called *unmarshalling*.

- As mentioned above, a distributed system may suffer from a partial failure, i.e. a number of the machines of the system may fail. This means that, at any give time, an RPC cannot be completed because, either the remote machine cannot be contacted to invoke the procedure (if, for example, it is not functioning at this time), or there is no indication of termination of the procedure on the remote machine (if, for example, it has failed while executing the procedure). The possibility of failure of the RPC has to be taken into account here, and, preferably some recovery action has to be provided by the programmer.

Because of the differences mentioned above, some extra infrastructure is required to allow a RPC to be performed.

2.2.2 Required Infrastructure

Because of the differences between a normal, local, procedure call, and a remote procedure call, a certain amount of support is necessary to be able to perform remote procedure calls. This support code is provided by a special RPC package. This package is usually a part of the compiler of the used programming language, or a separately acquired package.

Stub and Skeleton

Based on the declared interface of the service, a certain amount of support code is generated. For example, the marshalling and unmarshalling algorithms used are generated using the declarations of the data types used as parameters in the procedure calls.

The support code comprises of two proxies for the object: the client stub and server skeleton [2]. Figure 2.1 presents a schematic representation of the interaction between stub and skeleton, which we will now discuss.

- The *client stub* acts as a 'stand-in' (i.e. a proxy) for the server in the clients' process. When the client performs the RPC, it actually

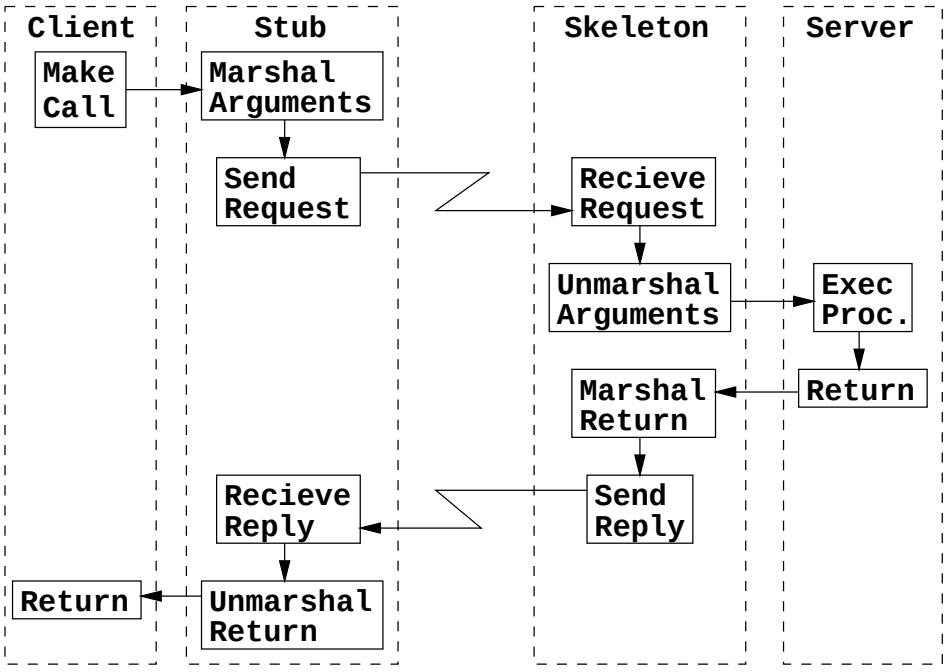

Figure 2.1: A schematic overview of the interaction between stub and skeleton. Zig-zag arrows represent network communications.

performs a normal, local, procedure call on the client stub. The client stub subsequently marshals the parameters of the call, contacts the server skeleton, and communicates that a certain method must be called with certain parameters, which are given in marshalled form. Also, the client stub is responsible for raising an error condition in case the RPC cannot be completed successfully.

- The *server skeleton* acts as a 'stand-in' (i.e. a proxy) for the client in the servers' process. When contacted by the client stub and informed to perform a procedure call, the skeleton unmarshals the parameters and proceeds to perform the, now local, procedure call on the server. When the procedure returns, the result, if any, is marshalled and communicated to the stub, which will unmarshal it and return it to the client process.

Generating the above support code is usually accomplished by running an interface compiler. This compiler has as input the declared interface of

the service, and produces as output the code for the stub and skeleton. The stub code must subsequently be included in the executable code of the client, and the skeleton code in the executable code of the server, to be able to invoke the RPC. Inclusion of this code is usually performed when the different object files of the programs are linked into an executable file.

Binding

We have now seen how a RPC is performed, by having stub and skeleton communicating the needed information amongst themselves, but we have not yet seen how the communication link between these two proxies is set up. It is obvious that for the stub and skeleton to be able to communicate, they must first get a reference to each other to be able to set up a link.

A two-step process is used to let the stub and skeleton get a reference to each other.

The first step in this process is what is known as the *binding* step. In this step, a server process makes a service known to the system. This is usually performed by providing an association between a name and a reference to the service. This reference mainly contains the hostname of the machine on which the server is running, and the port number on which the skeleton will listen for incoming connections.

The task of managing the above associations in a distributed system is performed by a service known as the *binder*. When in the above step, the server provides the association, it provides it to the binder.

The second step is performed by the client; whenever it wishes to access a certain service, it contacts the binder and requests a reference for a service by providing its name. The binder will return the requested reference, if the association exists, or an error if the association does not exist. When the client receives the reference, this is usually automatically converted to a stub, which is connected to the skeleton indicated in the reference.

Note that the binder is a crucial element for the operation of the system. If there were no binders, no references would be able to be obtained, and therefore no remote resources would be able to be accessed. Therefore it is advised to provide some fault-tolerance for the binder, for example by having multiple binders running on different machines. This would also solve a possible bottleneck, since all clients now need not pass through only one binder, but can use multiple, spreading the load over different binders.

However, we are now faced with a chicken-and-egg problem; to be able to contact the binder to get a reference to a service, we need to get the reference of the binder. There are a number of ways in which this can be

solved: we can let the binder run on a known port on a known machine, or let the operating system provide the location of the binder, for example in an environment variable, or we can let the client send a broadcast message asking for all binders to identify themselves.

To be able to use the binder, the code used to interact with it, and the code of the binder itself, must be provided as a part of the infrastructure delivered by the RPC package.

2.3 Conclusions

In this chapter a number of important concepts were introduced. We have defined a number of characteristics which are deemed important for a distributed system; the sharing of resources, the degree of fault tolerance and the transparency of the system. Subsequently we have introduced the remote procedure call as a mechanism for interaction of the different elements in a distributed system, and we have given an overview of the infrastructure required to be able to perform these.

Having presented a number of concepts of distributed systems, we will now introduce and evaluate a number of packages which aim to facilitate the building of distributed systems in Java.

Chapter 3

Distribution Mechanisms for Java

If at first you don't succeed, you're running about average.
— M.H. Alderson

A number of packages exist which provide support for building distributed systems in Java. In this chapter we will introduce and evaluate a number of these packages, ordered in function of increasing ease of use.

We will start with RMI [23]; the package which is included as standard in Java. Following this we will discuss Objectspace Voyager [10], a commercial package built as a replacement for RMI. We will conclude this chapter with a discussion of two academic research projects; JEDI [1] and JavaParty [18].

3.1 Java RMI

Java, as of version 1.1, includes a package for remote procedure calls, called Remote Method Invocations or RMI, which is contained in the `java.rmi` package and its sub-packages. Since Java is not a procedural language, but an object-oriented language, a conceptual difference exists between RPC and RMI. We do not call procedures, but invoke methods of objects, which implies a dynamic dispatch of the method invocation due to polymorphism. Also where in RPC procedures are declared to be a part of a module, in RMI methods are declared to belong to an interface which will be implemented by an object. A service delivered by a server class is therefore encapsulated

into an object. This conforms to the object-based model for the workings of a distributed system. Note that in this chapter, as in the rest of the text, we assume that the reader is familiar with Java.

We will now provide an overview on how the RMI package is used. We will create a small, distributed, "Hello, World" application, consisting of a `Hello` class, which can print out the well-known text on standard output, and a Runner class which will use this "greeting service". We will first define the hello service, and subsequently discuss on how we can make this service available to the clients. Once the service complete, we will define a client, and specify how the system can be compiled and run. As a last item, we will perform a short evaluation of the transparency of the RMI package.

3.1.1 Defining the Service

The first step we will take is defining the service which will be made available to client programs. This step consists of two parts: First we need to declare the remote interface which declares how to use the functionality of the service. Second, we need to define the class whose instances implement the functionality in the remote interface. We will discuss these parts in the following two paragraphs, and conclude this section with a discussion of parameter passing.

Declaring the Remote Interface

In RMI a remote service is declared by defining a Java remote interface for that service.

A *Remote interface* is an interface which implements the standard Java interface `java.rmi.Remote`. The `java.rmi.Remote` interface is an interface which does not declare any methods, its only purpose is being a flag to indicate that the interface implementing it is a remote interface. Also, all methods belonging to a remote interface must declare that they throw the exception `java.rmi.RemoteException`, in addition to any other exceptions they may throw.

The methods declared in the interface represent the various parts of the service which can be required by the clients. In our example, the only functionality required is printing out hello, so the following remote interface can be used:

```
public interface Interf_Hello implements java.rmi.Remote {
   public void sayHello() throws java.rmi.RemoteException;
}
```

The above code is fairly straightforward; we define an interface named `Interf_Hello` which extends `java.rmi.Remote`, to indicate that it is a remote interface. The interface contains one method: `sayHello`, taking no arguments and having no return value. The method declares it can throw a `java.rmi.RemoteException`, because this is a requirement for all methods in a remote interface.

The importance of the remote interface will become clearer as we further develop the example. We will see that this interface is the only type that can be used to access the resource remotely.

Defining a Remote Class

Now the service we have declared above in the `Interf_Hello` interface must be implemented by a concrete class. This class, implementing the remote interface, is called a *remote class*.

The remote class will define all methods declared in the remote interface. Each method will perform whatever action is required to deliver the service specified by that method. It is, of course, allowed to define extra methods, but these will not be able to be called remotely; only the methods declared in the interface can be called remotely.

In our example, the class `Hello` implements the service defined in the `Interf_Hello` interface as follows:

```
public class Hello implements Interf_Hello {
   public void sayHello() {
      System.out.println("Hello, world!");
   }
}
```

Note that although the method `sayHello` has been declared in the interface as throwing `java.rmi.RemoteException`, the implementation need not declare it. This is because when the exception will be thrown, it will not be thrown by this code, but by code of the RMI package which handles the actual remote method invocation.

Parameter Passing

In our example we do not pass any parameters to the remote method and we do not have a return value. However this does not mean no parameters may be passed in a remote method invocation. A remote method invocation

may indeed include parameters, but these must conform to some rules which define the way they are passed.

Basically, parameters must either be primitive types, classes which implement the interface `java.io.Serializable`, or remote classes. If a remote method call includes a parameter which does not belong to one of the above categories (i.e. a class which is not serializable and not remote), the remote method invocation will fail.

Also, if a parameter is a remote class, the type of the parameter may not be the remote class, but it must be the remote interface declared by that class. We will discuss the reason for this below.

How a parameter is passed depends on which category it belongs to:

- Serializable classes and primitive types (which are also serializable) will be passed by value, i.e. it will behave as an input parameter if it is a parameter of the method call or it will behave as an output parameter if it is the return value of the method call. The marshalling and unmarshalling process used for the parameter passing is the standard Java serialization mechanism, which normally ignores static and transient fields (although this can be overridden, for more information, see [21]).

- Remote classes will be passed by reference; the parameter is not copied over, but a remote reference to its skeleton is handed out, which is why the declared type of the parameter must be the remote interface. The proxy of the process 'receiving' the parameter will automatically convert this reference to a stub referring to that skeleton. This implies that any method calls to that parameter will be remote method calls.

Having discussed the elements pertaining to the implementation of the service, we must now make sure it can be used by the clients, i.e. the service must be made available to the clients.

3.1.2 Making the Service Available

Once the code for the service completed, it now has to be made available to client programs so they can access it.

Depending on how the service will be accessed, we have either one or three steps. The first step is identical for both cases: we have to export the object, to make it available over the network.

If the object will need to be looked up by clients, two further steps are required, i.e. for objects which will only be passed as reference parameters of remote method calls we need not perform these step.

The second step is the binding step, where the object is bound to a name, so the object can be looked up by clients. The third step is the definition of the server process which performs the actual instantiation of the object.

We will now detail these three steps.

Exporting

To be able to use the service, encapsulated in an object, that object has to be exported. *Exporting* an object makes it available to be accessed remotely, concretely this entails creating the objects' skeleton, so remote references to this skeleton can be handed out.

An object can be exported in two ways: If the object is a subclass of `java.rmi.server.UnicastRemoteObject`, it is automatically exported at instantiation time. However, in this case, the constructors of the class must declare to throw `java.rmi.RemoteException`, because the export may fail. If the object is not a subclass of `UnicastRemoteObject`, the static method `exportObject` of `UnicastRemoteObject`, which takes one parameter of type Object, can be used to export this object.

Therefore, we need to alter the `Hello` class:

```
public class Hello
   extends java.rmi.server.UnicastRemoteObject
   implements Interf_Hello {

   public Hello() throws java.rmi.RemoteException {
      super();
   }

   public void sayHello() {
      System.out.println("Hello, world!");
   }
}
```

We have chosen to make the class a subclass of `UnicastRemoteObject`, which ensures instances are automatically exported whenever they are created. We have explicitly defined the constructor to draw attention to the fact that it throws `java.rmi.RemoteException`, also, we will extend this constructor in the following step.

The class `UnicastRemoteObject` is the unique subclass of the class `java.rmi.server.RemoteServer`. The `RemoteServer` class "is the common

superclass to all server implementations and provides the framework to support a wide range of remote reference semantics" [23]. The idea is that the various subclasses of this class define different semantics for remote references. At this moment only one subclass is defined; `UnicastRemoteObject`, which "provides support for point-to-point active object references using TCP-based streams" [23]. However, it is conceivable that later other subclasses are added, extending the functionality of the RMI package.

Note that an object may only be exported once, if the object has already been exported, a `RemoteException` will be thrown. The reasons for this behavior are not specified.

Binding

To make it possible for clients to obtain a reference to a remote service, the service has to be bound to a name, so it can be looked up.

Binding an object to a name is performed by calling the `bind` or `rebind` static methods on the `java.rmi.Naming` class. These methods take as arguments an URL and a object which implements the `java.rmi.Remote` interface.

To bind our service to the name "HelloService", we change the `Hello` class as follows:

```
public class Hello
   extends java.rmi.server.UnicastRemoteObject
   implements Interf_Hello {
   public Hello() throws java.rmi.RemoteException {
      super();
      java.rmi.Naming.rebind("///HelloService",this)
   }

   public void sayHello() {
      System.out.println("Hello, world!");
   }
}
```

The first argument of the `rebind` method is the URL of the object, it is a short form, omitting protocol and host name. RMI will use the default protocol to access this object, and the object will be available on the default port of the current host. If we assume that the host containing this object is named `tongariki`, the full URL would be `"rmi://tongariki/HelloService"`.

Note that "for security reasons" [23], the host which is used in the above URL must be equal to the local host. The idea is that a malicious program may not overwrite the registry of a remote machine, so that its services are used instead of the intended services.

The object which is bound to the name is not the object given as parameter, but the objects skeleton. Whenever a lookup is performed on the binder, a reference to a skeleton (or an error) will be returned.

It is also possible to specify a port number in the URL, for example we could use the URL `"//tongariki:4242/HelloService"`, to specify that the registry should be located on port 4242. If no port number is specified, the registry is located on the default RMI port, which is 1099.

Defining the Server

Once all objects the server will export are defined, we have to create the server program itself. This executable program will, when run, instantiate the correct objects ensuring they become available to the clients.

In our example, only one instance of one class, the `Hello` class, needs to be created. An easy way to achieve this, is to provide the class with a `main` method, as follows:

```
public class Hello
   extends java.rmi.server.UnicastRemoteObject
   implements Interf_Hello {

   public Hello() throws java.rmi.RemoteException {
      super();
      java.rmi.Naming.rebind("///HelloService",this)
   }

   public void sayHello() {
      System.out.println("Hello, world!");
   }

   public static void main(String[] args) {
      new Hello();
   }
}
```

The `Hello` class is now executable, running it results in the server being run, and one instance of the class being made available to the clients. Note

that, although the main method will finish after the creation of the instance, the program will not end. This is because a separate non-daemon thread has been created which handles incoming connections to the `Hello` object, and the Java VM does not quit until all non-daemon threads have ended.

Were we to have a number of objects of different classes on the server, it could be advisable to create a special `Server` class, responsible for instantiating the correct objects. All that would be necessary is a `main` method whose body simply creates the needed instances.

Also, it is not required to have the export and bind statements to be performed in the object being exported. The only requirement is that the object is given as a parameter. For example, we could remove the export and bind actions from the constructors of the classes, and perform them in the `main` method of the aforementioned `Server` class.

3.1.3 Making a Client

Now the server is fully defined, we can proceed with creating a client which will use this server. As this process is fairly straightforward, we will first present the full code, followed by a brief discussion.

```
public class Runner {

   public static void main(String[] args) {
      try {
         Interf_Hello hel = (Interf_Hello)
           java.rmi.Naming.lookup("//tongariki/HelloService");

         hel.sayHello();
      }
      catch(java.rmi.RemoteException ex) {
         ex.printStackTrace();
      }
   }
}
```

This executable class basically performs two operations: It first looks up the HelloService, and second performs the remote method invocation of the only method defined by the service. This two operations are enclosed within a `try - catch` construct because the lookup and the remote method invocation may fail, which results in them throwing a `RemoteException`. Should this happen, we print out a stack trace of the exception and quit.

A closer inspection of the `lookup` method is warranted here. Note that the argument of the lookup is again an URL, but here we specify a hostname; `tongariki`. This is because we assume that the HelloService will be running on that machine. Also note the explicit cast to the interface of the service; `Interf_Hello`. This is because the `lookup` method is declared to return an object of type `java.rmi.Remote`. We do not cast here to the class implementing the remote interface. This is because the object returned by the lookup method is not a instance of that class, but an instance of the stub class for that class.

The above allows us to perform a 'normal' method invocation on the result of the lookup. The result, which is the stub, will handle the process of transporting the method invocation to the server, and returning the eventual result (in this case there is none).

3.1.4 Compiling and Running the System

Once all classes in the system have been defined, we can proceed to create the executables and run the system.

First we need to create the `.class` files for the all the classes in the system, which can be done by running any standard Java compiler. However this does not conclude the compilation. We still have to generate the skeleton and stub files for all remote classes. This is performed by running the RMI compiler on those classes.

In the standard Java Development kit, this compiler is named *rmic*. rmic takes as arguments the names of remote classes, and produces as output the skeleton and stub classes of these classes. To be able to produce this output, rmic needs access to the `.class` files of the remote classes and to the `.class` file of the remote interface.

For our example, we need to run rmic on the class `Hello`, which will generate the `.class` files `Hello_Skel.class` and `Hello_Stub.class`

To run the system, we first have to place the `.class` files for all classes used by the server on the server machine, including the skeleton class file, and we have to place the `.class` files for all classes used by the client, including the stub class file, on the client machine.

Second we have to start the registry service on the server machine. When using the standard Java Development kit, this service is started by executing the *rmiregistry* program. It is possible to specify a specific port for the registry to run on, by specifying it as a command-line parameter. If no port number is given, the default RMI port (1099) is used.

Third we have to start the server program. This is done by running the

executable class which corresponds to the server program, in our example this is the `Hello` class. Once the server is started, the client programs can use the services provided by the server.

So as a last step, we can run our programs. In our example this means running the `Runner` class. The result of executing the program will be that the text `"Hello, world!"` is printed out on the console of the server.

Having completed the description of the usage of RMI, we can now proceed to a short evaluation of the transparency of this package.

3.1.5 Evaluation of RMI

When looking at ease-of-use, RMI is clearly a significant improvement over using sockets for accessing a remote service [1, 18]. RMI allows us to treat a remote object almost as if it were a local object, once a reference to the object is obtained we can invoke methods on it almost as if it were local.

Where RMI falls short, is in the two "almost"s above. We still have to make a significant distinction between remote and local objects. This is because of the following points:

- We have to declare an interface for the remote class, and we can only get a reference of an object which has as type this interface, and not the type of the remote class.
- There is no mechanism to create an instance of a remote object from within a client on a given server. This functionality can be needed if, for example, we want to create a new datum on a storage server. It is possible to create instances remotely, but to do this the programmer has to implement a proprietary instance creation mechanism. This can be done, for example by defining a `Factory` class which is placed on the server and which sole functionality is to create instances whenever the corresponding method is invoked.
- We cannot perform direct variable access on the instance variables of the object; we can only perform method invocations on the object. We only can perform variable access by defining accessor and mutator methods and use these.
- We cannot invoke static methods of the class of the object, because these cannot be declared in the interface, which is our only access to the functionality of the object.

- The rules for parameter passing put a number of restrictions on the types of the parameters; parameters have to be either serializable or remote, and if they are remote their declared type must be their remote interface and not the class.
- A method invocation on a remote object may throw an exception, where if it were local it would not. Therefore these method invocations must ultimately be inclosed in a `try-catch` statement which contains appropriate error-handling code.
- Bootstrapping the system is not straightforward. The remote objects must ensure that they are exported before they are used, and if necessary should be bound to a name using the binder service. The clients must obtain a reference to bound objects using the binder service.
- At compile time, the RMI compiler must be called on each remote class to create its skeleton and stub class .

Although each of these elements may individually be seen as a minor inconvenience, the combined result is a clear distinction between normal and remote objects, which makes RMI a somewhat complex package to use.

To address this complexity, a number of alternate remote method invocation packages have been developed. A very well known package is Voyager, by Objectspace, which we will discuss next.

3.2 Voyager

In this section we will introduce a commercial RPC package for java; Objectspaces' Voyager.

Voyager is a product family of Objectspace, built to ease distributed computing in Java. Voyager consists of a number of different packages, of which the Voyager ORB, also known as Voyager Core Technology, is the bottom layer upon which the other packages are built.

Voyager Core Technology consists of an Object Request broker that supports RMI and CORBA. A notable feature of this ORB is automatic runtime generation of the skeletons and the stubs of RMI and CORBA.

The Voyager ORB is freely available from Objectspaces' Voyager web site [10]. While it provides a wide number of features including multicasts and agents, we will only investigate the remote method invocation features relevant to our thesis. For more information, see [11].

For simplicity's sake, we will, from here on, refer to the Voyager ORB solely as Voyager.

We will now re-implement our previous example using Voyager. This re-implementation will follow the same sequence of steps as in the RMI example; first we define the service, then make it available to the clients, followed by implementing a client, and finally compile and run the system.

3.2.1 Defining the Service

As above, the first step is defining the service which will be used by the client programs. This step consists of two parts: First we need to declare the interface for the service, and second we need to define the class whose instances implement the service. We will also discuss the "default interface" feature of Voyager, and we will conclude with a discussion of parameter passing.

Declaring the Remote Interface

As in RMI, declaration of a service is achieved by specifying an interface for this service. However, whereas in RMI a remote interface must implement `java.rmi.Remote` and all methods belonging to that interface must declare that they throw the exception `java.rmi.RemoteException`, this is not required in Voyager.

Hence, we can define the interface for our example as follows:

```
public interface IHello {
   public void sayHello()
}
```

As we said above, it is not required for the methods to declare they throw a `RemoteException`, this is because handling of exceptions is performed according to the type of the exception.

Exceptions thrown during the remote method invocation which are a subclass of `java.rmi.RemoteException`, (and therefore caused by the process of performing the remote method invocation) can be handled in two ways. If the method which has been invoked declares that it can throw a RemoteException, no special action needs to be taken, and the exception is simply thrown. If the method does not declare that it can throw such an exception (as above), the RemoteException will be wrapped in a `java.lang.RuntimeException` and this `RuntimeException` is thrown.

RuntimeExceptions are defined as "exceptions that can be thrown during the normal operation of the Java Virtual Machine." [22] Also, "A method is not required to declare in its throws clause any subclasses of RuntimeException that might be thrown during the execution of the method but not caught." [22]

This means that the program using remote method invocations need not declare to handle any exceptions caused by the remote method invocations themselves. If such an exception occurs, a RuntimeException will be thrown, which, if not caught somewhere (which is not unusual, considering their nature), will cause the VM to print out a stack trace of the exception and exit.

The following section, where we define the class, will clarify why we have followed this convention.

Defining a Remote Class

Now the service we have declared above in the `IHello` interface must be implemented by a concrete class, providing the implementation for the service.

The code for the example is straightforward:

```
public class Hello {

   public void sayHello() {
      System.out.println("Hello, world!");
   }
}
```

There is one noteworthy difference between this class and the class in the RMI example: this class does not declare that it implements the `IHello` interface.

The reason why we can omit this declaration is discussed in the following section.

The Default Interface

Usually, when an object implements a remote interface, this interface is written by the programmer of the object and it represents the methods which will only be called remotely. However, Voyager provides a tool, called *igen* which automatically generates the "default interface" for a given class. The default interface contains all the `public` methods of the class, and its name is derived from the Voyager interface naming convention, which is the

letter `I` followed by the name of the class (for example, the interface for a class `Foo` will be named `IFoo`).

In our example the default interface of the class `Hello` as generated by igen is remarkably similar to the interface we have defined manually above:

```
/**
 * IHello.java
 * <p>
 * @version 1.0
 * @author generated by igen 2.0.2 [rest deleted]
 */

public interface IHello extends java.rmi.Remote
  {
  void sayHello();
  }
```

The default interface can be used to remote-enable classes of which the source code is not available or cannot be changed, and which do not implement a suitable remote interface. Whenever a Voyager skeleton object is created (i.e. when a remote object is exported), Voyager verifies if the default interface for that class is available. If the default interface is available, the skeleton will also implement the default interface.

For example, if in our application we would want to create a Java `Vector` in a remote program, we would first need to run igen on the `Vector` class, which results in `IVector` interface source code being created. Once this is compiled, and made accessible to Voyager, whenever a `Vector` is created using `Factory.create()`, the returned proxy will implement the `IVector` interface, i.e. all `public` methods of the `Vector` class.

The automatic usage of the default interface has one interesting advantage: Using the default interface, it is not necessary to declare that a class which will be used remotely implements a remote interface, as we have done above, since the default interface can be used for remote operations. This implies however, that public methods which were not meant to be used remotely can now be used remotely, if a default interface for the class has been created and made available to Voyager.

Parameter Passing

As in RMI, a Voyager remote method invocation may include parameter passing, but these must conform to some rules which define the way they

are passed.

As in RMI, an object is either passed by reference or passed by copy.

If the object implements the `java.rmi.Remote` or Voyagers' `IRemote` interface it will be passed by reference. Concretely, a Voyager `Proxy` object (a skeleton) will be created at the site executing the method which will refer to the passed parameter. Because a `Proxy` object is created, the type of the object in the executing method cannot be the original class but must be a remote interface. This has as a consequence not only that the object must implement a remote interface, but that the executing method must declare as the type of the parameter the remote interface and not the class of the object.

If the object does not implement one of the two above interfaces, is is passed by copy. The copy is performed using the Java serialization mechanism, which requires the object and its contained objects to implement the `java.io.Serializable` interface (recall that primitive types are defined as being serializable). If this is not the case, serialization will fail, the object cannot be copied, and the remote method invocation will fail.

The implementation of the service now being complete, we will proceed with discussing how this service may be made available to the clients.

3.2.2 Making the Service Available

There are two ways in which the defined service can be made available to the clients: we can proceed analogous to RMI; by using a binding service to make it known to the system, or we can use the remote instantiation feature of Voyager. We will now discuss both of these options.

Exporting and Binding

When using Voyager, there is no need to export an object, Voyager will automatically export the object when a remote reference to the object is handed out. However, there is the possibility to manually export an object, by using the static method `export` defined in the `voyager.Proxy` class.

Voyager provides a simple naming service. As is RMI, the naming service allows objects to be bound to a name and exported, so that they may be looked up by other applications by providing the given name. All operations of the naming service are defined as static methods on the `Namespace` class.

The `bind()` method will, given an name expressed as an URL and an object, bind that object to the name. Expressing the name as an URL requires a port and a name for the object to be given. To lookup a bound object, the

`lookup()` method is invoked with argument the URL of the bound object. If the object can be resolved, a `Proxy` to the object is returned, which can be cast to the remote interface. A `NamespaceException` will be thrown if the object cannot be found.

For our example, we could create a server class which instantiates the required object and binds it to a name so it can be looked up. We will however use the second option available to us, which is remote instantiation.

Remote Instantiation

Voyager provides a `Factory` class, which contains a `create` method. This method can be used to create instances of a given class. There is, however, a significant difference between `Factory.create` and the `new` keyword in Java: the Factory returns a Voyager `Proxy` object to the object and not the object itself. This implies two things: First; the type of the result is not of the generated class, and second; the created object need not be local.

Since the type of the result is a proxy, the object which has been created must implement a remote interface. Since the proxy will also implement the remote interface, the result has to be cast to the remote interface, and messages can be sent to the object.

Also, since the result is a proxy, the created object may well reside on a different machine. The `create` method provides facilities for this: an extra parameter can be given which specifies the URL where the object should be created. This URL should point to a port belonging to a Java program which is running a Voyager server, and which has access to the `.class` file defining the class.

At this point we should note that Voyager also provides a minimal server which can be started up on the command line. This server performs one action: it opens a Voyager port and waits. The port can then be used, for example, by other programs to create objects in the server. The minimal Voyager server can access `.class` files in the standard Java classpath, in an extra directory given as argument upon startup, and files served by a http server of which the URL is given as argument upon startup.

This combination allows us to let our client create the needed object on a Voyager server, by invoking the `Factory.create` method. The concrete implementation will be given in the following part.

3.2.3 Making a Client

Having finished the server, we can now proceed with creating a client for the services provided by the server.

The code for our example is as follows:

```
public class Runner{

   public static void main(String[] args){
      try {
         Voyager.startup();

         IHello hel = (IHello)
            Factory.create("Hello","//tongariki:4242");

         hel.sayHello();
         Voyager.shutdown();
      }
      catch (Exception ex) {
          ex.printStackTrace();
      }
   }
}
```

This executable class performs four operations: it first starts up Voyager, then creates an object remotely, subsequently performs a method invocation on the remote object, and finally shuts down Voyager. These operations are contained in a `try-catch` statement, because the startup method may throw a `voyager.StartupException` (if Voyager has already been started), the create method may throw a `java.lang.Exception`, and the remote method invocation may throw a `java.lang.RemoteException`.

To be able to use the Voyager package, Voyager first has to be started up, which is performed by calling the static method `startup` on the `Voyager` class. Voyager can also be shut down, by calling the `shutdown` method.

Once Voyager is started, we can create the object for the remote greeting service by using the `create` method. We specify that the server for our example was named `tongariki`, and that the port should be number `4242`. If we do not provide a port number, Voyager will use its default port number. Note that, the result of this operation is an object of the `Proxy` class, which has to be cast to the appropriate interface, in our case `IHello`.

When the reference is obtained we can simply perform the method invocation.

With all the code for the system done, we can now proceed with compiling and running the system.

3.2.4 Compiling and Running the System

To compile the system, we can use any standard Java compiler, which will produce the necessary `.class` files.

Unlike RMI, Voyager does not require us to compile any skeleton or stub files, as it performs dynamic proxy generation. This means, that whenever a proxy object is required in the operation of the program, Voyager will on-the-fly generate the class for the proxy (if it is not already in memory) and instantiate a proxy. The advantage of this is that the programmer does not need to concern herself with the proxy classes when creating the applications. There is however also a disadvantage, which is the runtime overhead. Because the proxy classes are generated on-the-fly each time the program is run, there is a certain overhead involved with the creation of these proxies. Sadly, there is no facility for saving the proxy classes for a subsequent execution of the program. To eliminate the overhead, the "static proxy generation" feature of the Voyager ORB Professional has to be used.

To run the system, we first have to place the `.class` files for all classes used by the server, including the Voyager class library, on that server machine and we have to place the `.class` files for all classes used by the client, including the Voyager class library, on the client machine.

To start up the server, the executable class for that server has to be run, or if using the minimal Voyager server, as in our example, this server has to be run by running the `voyager` executable. The minimal server can be given a command-line argument which indicates the port number it should be running on, in our example this is `4242`.

Having started the server, we can now run the clients. In our example, this is achieved by running the `Runner` executable class.

Having finished and run the system, we can now perform an evaluation of Voyager.

3.2.5 Evaluation of Voyager

Looking at ease of use, Voyager scores higher than RMI. This is because we need to perform less work to make a class remote than in RMI and because of some extra features. We do not need to manually declare the interfaces,

the igen tool can deduce them for us. With Voyager we can also create classes remotely, which we could not do with RMI. Voyager automatically exports objects as needed, RMI does not. Also we do not need to run a separate proxy compiler, as Voyager generates the proxies on the fly.

However, there is still a clear distinction between remote and local objects; as can be seen from the following:

- If we get a reference to a remote class, this is to the classes' interface and not to the class itself. We still have to handle remote classes by their interfaces. There is also an inconsistency between the requirements for interfaces of classes which are created remotely, and classes which are obtained as by-reference return values. This will be discussed in some detail below.

- We can create instances remotely, but this mechanism is different from Java's instance creation mechanism (the `new` statement).

- Although we can mask exceptions thrown by a remote method invocation (which will probably cause the VM to exit if an exception occurs), we cannot mask exceptions thrown by an object creation.

- As in RMI, we cannot perform direct variable access on the instance variables of the object.

- As in RMI, we cannot invoke static methods of the class of the object.

- We have analogous rules for parameter passing as in RMI, which are different from normal parameter passing rules.

Note that the last point; choice of passing by reference or by copy is quite simplistic. This simple choice mechanism causes inconsistencies between object creation and parameter passing. Whereas any kind of object (which implements any interface) can be created remotely by using the `Factory.create()` method, and be used "by reference" as it were, only objects which implement the `Remote` or the `IRemote` interface can be passed by reference.

Furthermore, if the object passed as a parameter does not implement one of the above interfaces it must implement the `Serializable` interface or the remote method invocation will fail. This is a stark contrast to object creation, where there is no such constraint at all.

Also, when passing objects by reference, the object must explicitly declare it implements a remote interface, whereas when creating an object

remotely, this does not need to be done. The implicitly implemented default interface can be used for remote object creation, but not for parameter passing.

These inconsistencies imply that the programmer must keep foremost in his mind what interfaces the arguments of all remote method invocations implement and not implement. This obviously needs not be the case, because when creating remote objects all of this is done transparently.

A root problem here is the difference in which a remote object creation and a remote method invocation are specified in the program text. A remote object creation is specified by calling a method on a Factory object, which declares it will return a `Proxy`. A remote method invocation is, in the program text, identical to a standard method invocation. The interaction between the need to satisfy the type checker in the remote method invocation, and the building of a `Proxy` for arguments passed by reference can be described as follows:

- Voyager needs to create a `Proxy` object on the site executing the method.
- This proxy is not of the same class as the original object.
- The type checker cannot allow the type (in this case the class) of the formal parameter to be different from the type of the actual parameter. (Excluding subtype relationships.)
- Although the proxy is generated on the fly, and can therefore implement the default interface (which the class does not need to specify), again the type checker does not allow this. This is because the declared type of the parameter is the class and the actual type is the default interface, which are different types.
- Therefore the type of formal and actual parameter must be changed in the program text to a common type, which is the remote interface.
- If this is changed, the class of the parameter must explicitly declare that it implements the remote interface to satisfy the type checker at compile time.

This problem does not occur in object creation, since `Factory.create()` declares it returns a `Proxy` object, which it does, satisfying the type checker. The proxy can then be cast to an interface it supports, such as a default interface.

To solve this problem, either remote method invocations must be implemented through a special syntax (which would be a case of the cure being worse than the disease), or the proxy system must be changed to satisfy the type checker, or we should have a pre-processor change the source code of the class before it is compiled.

Having discussed the most important non-academic packages; RMI and Voyager, we will now introduce some noteworthy academic solutions for transparent distribution, starting with JEDI.

3.3 JEDI

The Java Environment for Distributed Invocation (JEDI) [1] is an alternative system for remote method calling, using dynamic method invocation. With dynamic method invocation no statically compiled interfaces are necessary, it is possible to call any method of a remote object at run-time.

Dynamic method invocation was chosen to avoid a number of preprocessing steps, such as defining the interface, exporting the objects and running the RMI compiler. This was done because these steps are considered to introduce unnecessary complexity.

The primary goal here was to provide a remote method call system which is simple to use. Whereas other remote procedure call packages may require a nontrivial amount of steps to be performed before a remote procedure call can be performed, JEDI simplifies or eliminates as many of the steps as possible.

JEDI is a library-based system; there is no extra compiler involved, only an extra library is necessary. All interactions required to call a method remotely are performed through methods of this library. This simplifies the compilation step, making it equal to the compilation step of a non-distributed program.

Using this system, any public method of an exported object can be called by a client, without the object having to be modified in any way. This means that objects for which source is not available can also be made remote, for example, it is possible to remotely call methods on objects belonging to the Java class library.

We will now implement our "Hello, World!" example using JEDI. We will first define the service, subsequently define the client and run the system, and we will conclude with a small evaluation.

3.3.1 Defining the Service

As said above, we do not need to declare an interface for the service, we can directly proceed to implement the class.

For our example, the code, including the server, would be as follows:

```
public class Hello {

   public void sayHello() {
      System.out.println("Hello, world!");
   }

   public static void main(String[] args) {
      Hello hel = new Hello();
      info.jedi.Repository.local().bind(hel,"HelloService");
   }
}
```

The code for the service is straightforward and does not need any clarification. The only notable point is in the `main`, where after we create a `Hello` object, we bind the new object using the local binder service, which automatically exports the object. This does not throw any exceptions (whereas using RMI or Voyager exporting and/or binding may throw exceptions).

As can be expected, methods may contain parameters and a return value. However there is a strict rule for the parameters (and return value); they must be serializable. Non-serializable objects may not be passed as parameters, as all parameters are passed by copy. This might seem to exclude remote objects to be passed by reference, but it does not. As we shall see later, the stub for a remote object is a JEDI `Proxy` object. These objects are serializable, which allows the remote object to which they refer to be passed by reference.

Having defined the server, we can now proceed with making a client for the service offered by the server.

3.3.2 Defining the Client

The client is fairly straightforward; we only need to obtain a reference to the remote object and perform the method invocation:

```
public class Runner {

   public static void main(String[] args) {
      try {
         info.jedi.Proxy hel =
            new info.jedi.Proxy("tongariki","HelloService");
         hel.function("sayHello");
      }
      catch (Exception ex) {
         ex.printStackTrace();
      }
   }
}
```

Obtaining a reference is performed by instantiating a new `Proxy` object, with arguments the name of the server running the service (in our example `"tongariki"`), and the name of the service (`"HelloService"`). This operation may throw an `UnknownHostException` if the host cannot be found.

Once the reference is obtained, a method call can be performed by calling the `function` method on the proxy. This method takes as argument the name of the method to be executed remotely, and optionally a Java Vector containing the arguments for the remote method. Exceptions thrown in the remote method will be re-thrown by `function`, which is why `function` declares it throws `Exception`. Exceptions due to the process of executing the method call remotely will also be thrown by `function`.

Once all code is written, it can be compiled by any Java compiler. As expected we need to place all classes needed by the server, including the JEDI libraries on the server machine, and perform the similar action for the client on the client machine. Running server and clients is performed by simply running the corresponding executable classes.

3.3.3 Evaluation

JEDI provides yet another improvement in the user-friendliness of remote method invocations, but we have to pay a price; the syntax for remote method invocations is radically different from normal method invocations.

JEDI still falls short in the following areas:

- As in RMI, we have no standard protocol to create objects remotely
- As in RMI and Voyager, we cannot perform direct variable access on the instance variables of the object.

- We have analogous rules for parameter passing as in RMI and Voyager, which are different from normal parameter passing rules.
- The syntax for remote method invocations is radically different. Also, there is no static type checking on the remote method calls, since these are performed by using the `function` method, which takes as argument the name of the method and a Vector of parameters, and returns an Object.
- Remote method invocations may throw an `Exception`.

We will now introduce the final package; JavaParty, which promises truly transparent remote objects in Java.

3.4 JavaParty

An extremely interesting package for easing distribution is JavaParty. JavaParty adds remote objects to Java in a transparent fashion but avoids the "disadvantages of explicit socket communication, the programming overhead of RMI, and many disadvantages of the message-passing approach in general" [18].

This package extends Java with one class modifier; `remote`, and adds a preprocessor and a run-time system. JavaParty is primarily aimed at parallel programming in workstation clusters of a heterogeneous nature, i.e. containing different kinds of machines.

JavaParty implements distributed shared memory; all processes part of the system share their memory, i.e. there is no distinction between local and remote object references and all memory is accessible from all processes.

The `remote` class modifier was introduced because classes for which source is not available, such as the Java class library, cannot be transformed into remote classes, partly because a preprocessor approach is used, and partly because if the classes contain native code these are designed specifically to be run in a non-remote fashion [18].

By using the `remote` class modifier, a programmer indicates the objects of this class can be placed on a remote machine if needed. Placing of these objects is achieved by a separate "distribution strategy" algorithm. These strategies are implemented using the well-known "Strategy" design pattern [6] and can be selected at runtime. Also, the runtime system monitors the interaction of the remote objects, and may migrate these to other servers if considered appropriate.

3.4.1 The "Hello, World!" application

Using this system, our example application becomes extremely simple, as can be seen below:

```
public remote class Hello {

   public void sayHello() {
      System.out.println("Hello, world!");
   }
}

public class Runner {

   public static void main(String[] args) {
      Hello hel = new Hello();
      hel.sayHello();
   }
}
```

This code is completely equal to a non-distributed version, save the use of the `remote` class modifier for the `Hello` class. Because of the distributed shared memory, we do not have to concern ourselves with the remote-ness of the instances of Hello.

There is only one hitch in the system, which is in the referencing of objects on remote machines. Objects on remote machines must be either `remote` objects, or be serializable, in which case access is performed on a local copy of the object. If an object is not `remote` and not serializable, it cannot be accessed remotely and an error will occur. This is highly similar to the restrictions we have seen for parameter passing in all three previous packages.

3.4.2 Evaluation

JavaParty is the most user-friendly package for distributed applications, it provides a distributed shared memory system which makes remote objects almost identical to local objects. Because of this, almost no distinction has to be made between local and remote objects, making the distribution transparent.

Note the "almost" above, there is one distinction left, which is the restriction on remote references; they must either be remote or serializable

objects. It can be argued that this restriction is not that severe: objects which will be placed remotely will in most cases be objects developed by the programmer, and therefore can be made remote or serializable. Also the most used 'primitive classes' such as String, and all numeric classes of the `java.lang` package are serializable.

However, in making remote objects identical to local objects, we have lost something. Namely, we lost the ability to specify error-handlers for failing remote method invocations. In the environment JavaParty is aimed for, this is not a significant problem, as errors are not so likely to occur when using clustered workstations. This is due to the high reliability of the local area networks (or special-purpose interconnections) they are connected to and to the high reliability of the workstations themselves.

But if we leave this environment, errors are much more likely to occur, and to have a stable distributed system we must be able to specify what to do in case a method invocation fails. Yet, as we have seen above, this diminishes the ease-of use of the distribution package. This is because at each method invocation we must consider if it is local or not, and if necessary add error-handlers. Having to keep this in mind significantly decreases the ease of use of the distribution package. Therefore we need to find an alternate way in which the error-handlers can be specified.

We will propose such an alternative in the following chapter, where we will discuss Aspect-oriented Programming.

3.5 Conclusions

This chapter gave an overview of four packages which can be used to simplify the creation of distributed Java programs.

We have discussed the RMI package, available as standard in Java. We have seen that, although it significantly eases the development of distributed systems, compared to sockets, it is still somewhat lacking.

The next package we discussed, Voyager, was built to make the development of distributed systems easier than when using RMI. We found that, indeed Voyager is easier to use, but there is still room for improvements.

The third package, JEDI, delivered most of these improvements. However, an important shortcoming is that the syntax for remote method invocations is radically different form normal method invocations. This places a great emphasis on the kind of object the method is invoked, and lowers the ease of use.

The last package, JavaParty, provides the easiest solution for building

distributed systems. However, this ease of use made us lose some control. More specifically, we have lost the ability to specify error-handlers for the remote method invocations.

What we need to find now, is a method which can give us the ease of use delivered by JavaParty, combined with the control we need to be able to handle errors. We believe Aspect-oriented Programming can be used to achieve this goal.

Chapter 4

Aspect-Oriented Programming

Do not pray for easy lives. Pray to be stronger men!
Do not pray for tasks equal to your powers.
Pray for power equal to your tasks.
— Phillips Brooks

Applications which need to fulfill a large number of requirements are generally difficult to develop. One principle can be used to ease the development of these applications: separation of concerns. However it is not always easy to separate these concerns, as some of them have a system-wide impact. A technique to separate out these system-wide concerns is Aspect-Oriented programming (AOP).

In this chapter we will introduce the principle of separation of concerns, and discuss AOP as a technique to achieve this separation where, until now, it was hard, if not impossible, to achieve. We will conclude the chapter with a discussion on how AOP can be used to achieve distribution transparency.

4.1 Separation of Concerns

When developing an application, the programmer has to ensure that a wide variety of requirements be met. To be able to meet this diversity of requirements, various techniques and methodologies have been developed to

reduce the complexity of the software, to increase its comprehensibility, and therefore also to ease its maintenance.

Common to many of the above techniques is the "divide and conquer" concept. The software is decomposed into smaller pieces, which individually are easier to comprehend and manage, and these pieces are later combined into the full system.

A good decomposition allows each separate piece, to address a specific concern of the application, i.e. a specific subset of related requirements, such as the core algorithm of the application, the organization of the data, concurrency, persistence, etc. ... In OO programming, we divide into classes, the idea being that each (group of) classes will address one concern. Handling these concerns separately, not only conceptually but also at an implementation level, greatly enhances coding and maintenance. This is because the programmer only needs to keep in mind the exact concern she is working on, and can ignore the other concerns, which significantly reduces the complexity of the task at hand. The above principle is aptly named *separation of concerns* [9]. The ideal of this separation is that each concern of the application is addressed in one and only one module, or class, of the program.

However, this ideal is hard to achieve. This is because a number of 'special-purpose' concerns, such as concurrency, distribution, persistence, etc. ... can not be decomposed into these modules. This is because these concerns have a system-wide impact, they can not be added by simply adding a new module to the system. An example would be concurrency; adding a 'synchronization' object to the system is not enough, every method that needs synchronization will have to make a call to this object. This means that the code in a wide number of existing classes will have to be adapted, breaking the separation of the concurrency concern. The special-purpose concerns mentioned above are said to *cross-cut* the class structure [15, 16].

For a number of these special concerns, the programming language provides constructs which handle this concern. For example, in Java synchronization is handled with the `synchronized` keyword and the `wait`, `notify` and `notifyAll` methods. However, it still proves to be hard to write and understand code using these constructs [9, 8]. Also, if the number of special-purpose concerns in the program increases, it will progressively become more difficult to comprehend the code, increasing the difficulty of developing and maintaining the program.

The above problem occurs because although the different concerns can be specified independently in an abstract fashion, integrating them 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 between

the concerns at a conceptual level, at code level this coupling becomes strong when the concerns have been integrated.

Were the different concerns not only separated at a conceptual level, but also at a code level, by e.g. having one class for each concern, we would have a full separation of concerns. This full separation would lead to a higher level of abstraction, making the code easier to understand and maintain.

A number of techniques have been developed, addressing the problem of these cross-cutting concerns, achieving a higher separation of concerns. One of these technique is Aspect-Oriented Programming, which we will discuss in the next section.

4.2 Aspect-Oriented Programming

In this section we will introduce Aspect-Oriented Programming (AOP) [15, 16]. We will detail how concerns are split into aspects and components, discuss how aspects are treated separately and how they are combined into executable code. We will conclude with an example showing how this technique achieves separation of concerns.

In *Aspect-Oriented Programming*, the concerns which cross-cut the class structure are called aspects. They are said to "cut across both each other and the final executable code" [15]. As said above, although these aspects can be easily separated at a conceptual level, the code which integrates these aspects is "a tangled mess of aspects" [15]. This tangling of aspects indicates that in the code the concerns are not separated, which makes the program hard to comprehend.

In AOP, a concern is either implemented in a component, or in an aspect. A concern is called a *component* if it can be encapsulated cleanly into a module (such as a class or a group of classes). A concern that can not be encapsulated cleanly into a module, is called an *aspect*. In a program, all concerns which are components are sometimes also called the *base aspect*.

An example of an aspect would be the concurrency concern we mentioned above. It is clear that this concern can not be addressed solely by adding a 'synchronization' object; calls to this object must be added in various other objects, at the point where synchronization is required. This prohibits clean encapsulation of the concurrency concern, making it an aspect.

AOP allows the programmer not only to reason separately about the aspects, but also to implement them in separate modules. This is achieved by specifying the code pertaining to an aspect in a separate aspect file, in a special aspect language.

The aspect languages are designed to allow the aspect to be easily expressed by the programmer. The first aspect languages which were designed were aspect-specific, for example, there are languages for concurrency control [25], numerical accuracy [13], and space optimization [17]. Because of this specific nature of the languages, the programmer can express the code in a natural form, leading to a greater ease of use. Current AOP research, however, is heading towards a more general aspect language, in which various aspects can be expressed [14].

Once all components and aspects are defined, a special tool, called an Aspect Weaver, combines these into executable code. The weaver is able to do this because it knows not only how each aspect can be transformed into code, but it also knows the relationships between the different aspects, and the correct way to combine them.

Example

Now let us consider how our previous example, the concurrency concern, is handled using AOP. One AOP tool which handles this concern is AspectJ [14], a previous version of which was known as D [25].

Using this tool, the programmer writes the base aspect code, i.e. ignoring the concurrency aspect, in Java. This implies that the special Java constructs for concurrency (`synchronized`, `wait`, `notify`, `notifyAll`) are not used in this code. Addressing concurrency is done by defining the concurrency aspect in a separate aspect file, not in Java, but in a special aspect language.

In D, a special language named *Cool* is used to specify concurrency control. Using Cool, a so-called "coordinator object" is described for objects of a given class. This coordinator object has a coordination strategy which is applied to the methods of the object. Methods can be declared to be self-exclusive, mutually exclusive, or a guard can be placed on the method's execution.

A method which is declared to be self-exclusive, will not be run concurrently on an object, i.e. if multiple threads try to simultaneously run this method on the same object, this execution will not occur in parallel, but serially.

A method belonging to a group of methods which are declared to be mutually exclusive, prohibits the other methods of that group to be run concurrently. In other words, if multiple threads try to simultaneously run a number of methods of the same mutually exclusive set, on the same object, this execution will not occur in parallel, but serially.

Last but not least, finer control can be obtained by using guarded sus-

pension of threads. A boolean expression is provided which specifies if the method can be executed or not; if the expression is true, the method will be executed, if not, it will be suspended until the expression becomes true.

Using these declarations, the coordination strategy can be declared outside of the class definitions. This allows the concern of coordination to be addressed separately from the other concerns, giving us true separation of concerns.

To produce the executable code, the aspect weaver included in D has to be run, with, as arguments, the names of the Java files, and the name of the aspect file. The weaver will subsequently produce a number of Java `.class` files, which integrate the base aspect and the concurrency aspect.

Having seen how AOP can be used to achieve separation of concerns, we will now introduce how we can use this technique to achieve distribution transparency.

4.3 AOP and Distribution Transparency

Using AOP, a concern which can not be encapsulated in one module can be separated out of the application, so it can be reasoned about separately.

Should we try to encapsulate the concern of distribution in a separate 'distribution' module without using AOP, we will encounter a number of problems. Making an application a distributed application, entails placing objects on separate machines, and changing the method invocations on these objects into remote method invocations.

Even when using the most powerful tool we have mentioned, JavaParty, we still need to adapt the code of a large number of classes; we need to add the `remote` modifier to the classes. Should we elect not to use JavaParty because of the impossibility to perform recovery from partial failures, the changes to the class structure are even more widespread. Therefore we can conclude that distribution has to be considered as being an aspect.

Now consider the definition of distribution transparency we used previously: "If there is distribution transparency, the fact that the system is built out of different components is hidden, and the system is only perceived as one, whole, entity."

Seen from the "separation of concerns" principle, this definition can be redefined as: "If there is distribution transparency, the concern that the system is built out of different components is separated out. The system is perceived as one entity when the programmer is not working on the distribution concern."

This definition leads us to state that we can achieve full distribution transparency, by using AOP.

We propose, and will develop in the rest of this thesis, the following approach to achieving full distribution transparency:

- Remote classes have to be specified separately, in a special aspect file. As in JavaParty, method calls to these objects will be remote method calls.
- The rules for parameter passing in methods of these remote classes must be as straightforward as possible.
- Location of these remote classes will also be defined separately from the main program, by defining it in the aspect file described above.
- To provide a possibility to recover from partial failures, error-handlers can optionally be specified in a second aspect file. If no error-handlers are specified, a default error-handler will be used.

To achieve this, a number of aspect languages have to be defined and an aspect weaver for these languages has to be implemented. In the following chapter we will introduce the aspect languages we have defined, and the subsequent chapter will detail the transformations performed by the aspect weaver.

4.4 Conclusions

This chapter introduced the principle of separation of concerns, which is used to ease the development of software. Separation of concerns is akin to "divide and conquer": complexity is reduced to manageable pieces which can be handled individually.

However, some concerns can not be separated out using OO programming, because they have a system-wide impact and cross-cut the chosen decomposition [24]. Many of the pieces mentioned above need to contain elements pertaining to these concerns. Aspect-Oriented Programming solves this problem by letting these special concerns, called aspects, be defined separately. A tool, called the Aspect Weaver, combines these different aspects together with the programs' 'normal' code into the final executable.

We have deduced why distribution has to be considered as being an aspect, and have stated that full transparency can be achieved using AOP. To achieve this goal, we have determined and outlined a strategy, which will be developed further in the following chapters.

Chapter 5

The Aspect Languages

Language ... it's a virus.
— Laurie Anderson, "Home of the brave"

To make it possible to achieve distribution transparency, we have created an AOP system named Distra. It consists of three aspect languages and an Aspect Weaver. In this chapter, we will first discuss some general issues and subsequently introduce these three languages. The next chapter will give an overview of the transformations performed by the weaver, which we named DistraC.

5.1 General

Before we introduce the aspect languages, we will first discuss some general issues pertaining to the functionality of the system. We will first talk about the transport layer which is used in the generated systems. Secondly, we will discuss the remote instantiation ability and thirdly we will introduce the parameter passing rules. To conclude this part we will detail the default error-handlers.

RMI and Remote Instantiations

To ease development of our aspect weaver, named DistraC, we have decided to use RMI as a transport layer for the distributed systems generated by DistraC. RMI has the advantages that it is much easier to use than sockets and TCP connections, and that it is included as standard in Java. We have

chosen not to use one of the more advanced solutions we discussed in chapter three, as RMI provided us with sufficient functionality, while having the convenience to be included as standard in Java.

As mentioned in chapter three, RMI does not include a mechanism to create instances of objects remotely. However, to make remote classes behave identically to local classes, we must be able to instantiate instances remotely. To be able to do this, we have included a remote instantiation behavior which is totally transparent to the programmer of the base aspect, i.e. instantiating an object remotely is identical to instantiation of an object locally.

However, when handling the distribution aspect, the programmer should be aware that a remote instantiation is a two-step process: first a connection to the remote class is made, and second an instance of that class is created. This process will have an impact on the specification of the location of the class, and on error-handling.

More specifically, it is important to know that on every instance creation a connection to a remote class will be made, which allows a fine control on the placement of the instances, and since instantiation implies two remote operations, two classes of errors can occur here.

Parameter Passing

Also important for the behavior of remote classes is how parameters are passed. Because we are using RMI, we are confined to the parameter passing rules of RMI. However, we have managed to simplify these rules somewhat. Recall that using RMI, a remote method invocation may fail if a parameter is not serializable nor a remote class.

To avoid these failures from occurring, DistraC will attempt to transform non serializable, non remote classes into remote classes. Sadly, this transformation cannot be applied to classes for which the source is not available (which we will call system classes, for sake of brevity). Examples of these classes are the classes of the Java class library.

This gives us the following rules to determine which passing mode is used (the first rule that matches is the rule which is used):

1. All classes declared as remote are passed by reference
2. Serializable classes (which are not declared as remote) are passed by copy
3. All other non-system classes, (which are non-serializable and not declared as remote) are passed by reference.

The third rule implies that some classes will automatically be transformed into remote classes. Except for the system classes described above, transformation into remote classes will happen transparently, the user need not worry about this. However, if DistraC encounters a system class which should be made remote for the generated program to work, DistraC will print out an error message and stop.

We estimate that this need not be a serious problem. A large number of the classes in the Java class libraries are serializable, so passing them as parameters will not be a problem. Many of the classes which are not serializable are classes which have a strong connection with the VM the program is running on, such as `java.lang.System`, and therefore may not make much sense in an environment where multiple VM's are running the program.

However, should the programmer wish to use a non-serializable system class in the program, this can be achieved by writing a 'wrapper' class for this class, whose objects would simply contain a reference to an object of the system class and forward the method calls to that object. This wrapper class would then be used in the program wherever the original class is used, and as for this class the source is available, it can be made remote.

Creating this wrapper class and replacing the references to it could be performed automatically by DistraC, but as we estimate that it would only be used very infrequently, and due to time constraints, we have not implemented this feature.

The Default Error-handler

To ease the development of the distributed system, especially in the early phases of implementing the distribution aspect, Distra provides a default error-handler.

The idea here is that at some points of the development, the programmer does not want to concern herself with failures. The primary concern initially is the layout of the objects in the system, partial failures need not be handled gracefully at this point.

To facilitate this, a default error-handler is provided. If for a given remote method invocation, no error-handler is specified by the programmer, the default error-handler will be used.

The default error-handler produced by DistraC is identical to the default error-handler of Voyager we mentioned in chapter three; it wraps the exception thrown by the remote method invocation in a Java RuntimeException and rethrows it.

We can now proceed with introducing the aspect languages used by Distra. We will start with the language for the base functionality. Not surprisingly, this language is Java.

5.2 Java

The language in which the base functionality is written is Java. However, we have to impose one important restriction on the programs: the exception `java.rmi.RemoteException` may not be thrown nor caught in the code.

This restriction guarantees that the concern of distribution is completely separated out. The semantics of the `RemoteException` are that it is thrown as a result of a remote method invocation. If this exception is thrown in the base aspect code, this would mean that the concern of distribution is included in the code. This is exactly what we want to avoid by specifying the distribution concern as an aspect, therefore we cannot allow `RemoteException`s to be thrown or caught in the base aspect code.

Writing our "Hello, world!" example from chapter three is now extremely simple, as we need not concern ourselves with the distribution aspect. (Note that more extensive examples are contained in chapter seven.) The code is totally identical to a non-distributed version, as can be seen below:

```
public class Hello {

   String thetext;

   public Hello(String text) {
      thetext = text;
   }
   public void sayHello() {
      System.out.println(thetext);
   }
}
public class Runner {

   public static void main(String[] args) {
      Hello hel = new Hello("Hello, world!");
      hel.sayHello();
   }
}
```

We have altered this example somewhat, so we can later illustrate some of the features of Distra. The example now includes a constructor, which takes as argument the text to be said.

To make this program a distributed program, we need to specify this in a separate aspect file, written in the Dist language, which we will now introduce.

5.3 Dist

Specification of the remote classes is done in a separate aspect file. This file contains a number of declarations of remote classes and their location, written in the *Dist* language. This section will give an overview of this language, guided by its' abstract grammar.

The Dist language, of which the grammar is given in appendix A, is a language we designed to easily allow specification of remote classes, including specific attributes of these classes. We will now discuss this language, guided by the grammar.

5.3.1 Basics

The first three productions form the basic elements of the specification, they are used to specify that a number of classes are remote:

```
Distfile:
   (Classdist)*

Classdist:
   "class" FullyQualdName [Location] [Extends]

FullyQualdName:
   <IDENTIFIER>
```

The Dist file contains the specifications of a number of classes, the name of each class being a fully qualified name, i.e. the name includes the package of the class. This is done to avoid errors in case of duplicate class names. Only non-system classes can be specified here, i.e. can be made remote. The reason for this restriction will be discussed in depth in the following chapter. Basically, we need to modify the source of a class to make it remote, so if we cannot access the source, we cannot make the class remote.

The most basic remote classes does not include a Location description. Instances of these classes cannot be created remotely. They can only be used as reference parameters for remote method invocations. If we want to instantiate objects remotely, we need to specify a location, as will be explained in the next subsection.

5.3.2 Locations

To enable instances of classes to be created remotely, a location must be specified.

```
Location:
    <HOSTNAME>":"<PORTNUMBER> | Block
```

Adding a Location part to a remote class makes it possible for instances of this class to be created remotely. To specify which machine the instances will be located on, an identifier indicating host name and port number is required. This can either be specified literally or in a block of Java code.

The literal specification has to be of the form hostname:portnumber e.g: `tongariki:4242`. If the specification is a block of Java code, this block will be executed every time an instance of the class is created. The return value of the block will indicate the server name, therefore it must be a Java String of the above form, e.g. {`return "akivi:"+(4240+2);`} If the String equals `null` (e.g. the result of {`return null;`}), no connection to a remote class will be made, and the instance will be created locally.

A more extensive example of using a block to specify a location, will be given in 5.3.4, as in the experiments detailed in chapter seven.

5.3.3 Additions

Last but not least, we can add some extra functionality to the class, which is relevant to the concern of distribution.

```
Extends:
   "extends" ClassBody
```

In some cases, extra functionality only pertaining to the distribution aspect will have to be added to a non-remote part of the class. Extends allows extra variables and methods to be declared in the body of the part, ClassBody corresponds to a block declaring instance variables and defining methods.

Note that this functionality is placed on a local class, i.e. on the client, and not on the remote class, i.e. on the server. This allows this functionality to be used before a connection to the remote class is made, for example it can be used to determine the location of the remote class to connect to.

The following subsection describes an example of the usage of the dist file, including the use of an extends clause.

5.3.4 Example

Let us now transform our "Hello, world!" example into a distributed system.

The simplest way in which to do this, is to create a Dist file with the following contents:

```
class HelloWorld
   tongariki:4242
```

This specifies that the `HelloWorld` class is a remote class, and that instances of this class will be created on the server named `tongariki`, on port number `4242`.

As we have not specified any error-handlers, DistraC will include the default error-handlers. For our example, this means that the program will stop when an error occurs.

We can extend this example with some extra features, such as a primitive form of load-balancing. Suppose we have not one but two servers named `tongariki` and `akivi`, and our client will create a large number of `HelloWorld` objects. We can split the creation of the objects over the two servers by specifying a Dist file with this contents:

```
class HelloWorld {return split();}
   extends {
      private static int index = 0;
      private static String[] hostnames =
                    {"tongariki:4242","akivi:4242"};

      private String split() {
        if (index == 0)
           return hostnames[index++];
        else
           return hostnames[index--];
      }
   }
```

This file specifies that the method named `split` should be called to determine the hostname of the remote class and defines two static variables and the `split` method. The return value of this method will alternate between the two host names, providing a form of load balancing.

Should we now want to add error-handling to this example, we would need to specify error-handlers. These specifications are performed in the Fix language, which we will introduce next.

5.4 Fix

Specification of error-handling of remote classes is done in a separate file. This file contains a number of declarations of error-handlers for remote classes, written in the *Fix* language. This section will give an overview of Fix, guided by its' abstract grammar.

We designed the Fix language, of which the grammar is available in appendix A, specifically to allow easy specification of exception handlers for exceptions thrown by remote invocations.

Note that in this file exception handlers can be specified for classes which are not explicitly specified as being remote in the Dist file. This is useful when classes are automatically transformed into remote classes due to the rules of parameter passing.

It is not an error to have Fix specifications for a class which will not be made remote. Fix specifications for classes which are not remote are simply ignored. Also, if no Fix specification is given for a remote class, the default error-handlers will be generated for that class.

5.4.1 Basics

```
Fixfile:
   (Classfix)*

Classfix:
   "class" FullyQualdName [Extends][New][Bootstrap]
   [Creates][Invokes]
```

The Fix file contains the specifications of a number of classes, the name of each class being a fully qualified name, i.e. the name includes the package of the class. This is done to avoid errors in case of duplicated class names.

There are three major categories of error-handlers; Bootstrap, Creates and Invokes. Each catches errors occurring at specific categories of remote

method invocations.

1. Bootstrap handlers catch errors occurring when the connection to the remote class is made, i.e. when the program connects to a remote class with the intent to subsequently create an instance of this remote class. (These handlers only apply to classes for which instances can be created remotely.)

2. Creates handlers catch errors occurring when creating an instance remotely. (These handlers only apply to classes for which instances can be created remotely.)

3. Invokes handlers catch errors occurring during an invocation of a method of the remote class.

5.4.2 Exception Handlers

We will now discuss the different categories of exception handlers;

```
Bootstrap:
   BootCatchBlocks

BootCatchBlocks:
   "{"(BootCatchBlock)*"}"
```

As stated above, exceptions can occur whenever the program tries to connect to a remote class, so an instance can be created. Once a reference to the remote class is obtained, a constructor of this class will be invoked.

The type of exceptions which can be thrown here are:

java.rmi.RemoteException if the remote host could not be contacted.

java.net.MalformedURLException if the name of the remote host, as specified in the Dist file contains illegal characters, such as slashes ("/").

java.rmi.UnknownHostException if the host, as specified in the Dist file, is unknown.

Note that subclasses of the above exceptions may also be thrown. To catch these exceptions a number of BootCatchBlocks can be specified, which are grouped by surrounding curly braces. This allows multiple exception handlers, each for a different kind of exception, to be specified.

Note that if an error-handler is specified, error-handlers have to be specified for all of the above exceptions, DistraC will not generate the default error-handlers for the remaining exceptions. For example, if only an error-handler for `java.rmi.RemoteException` is specified, an error will occur at compile time, because no handlers are specified for the two other exceptions. DistraC could perform a scan of the defined error-handlers, to define which are not included, but we did not implement this due to lack of time.

```
BootCatchBlock:
   "catch" "(" <TYPENAME> <PARAMNAME> ")"
   (Block | BootBreak | BootSwitch)

BootBreak:
   "break" [Block]

BootSwitch:
   "switch" [Block]
```

The specification of the error-handler is highly similar to how the "catch" part of a Java "try-catch" is specified. A class of exception is specified in `<TYPENAME>` and a name for the formal parameter in `<PARAMNAME>`. This name can then be used in the block of Java code which contains the actual error-handling code.

We have extended the Java "catch" part here by allowing a `"break"` or a `"switch"` keyword (which may be followed by a optional Block) instead of a Java Block.

Using the `"break"` keyword indicates that the instance should not be made remotely, but should be made locally. When using a BootBreak no connection will be made to the remote class at this time, and the instance will be made locally.

The `"switch"` keyword indicates that the connection should be switched to another server. If no Java Block is given, the server used is the one indicated in the Dist file. If a Java Block is given here, this block should return a String of the form hostname:portnumber which indicates the server to connect to. As in the Dist file, if the String indicating the remote server equals `null`, no link will be made to the server and the instance will be created locally.

Switching to another server entails trying to establish a connection to the alternative server. Note that this process can also fail. The error-handlers used to handle this failure will, of course, be the bootstrap error-handlers.

This can of course lead to the program looping if all the servers it tries to connect to are down, and there is no other alternative provided (such as a BootBreak). We assume it is the responsibility of the programmer to determine if this is the required behavior, and, if not, to change the error-handling code.

```
Creates:
   (Create)+

Create:
   FormalParameters CatchBlocks
```

As there can be multiple constructors of a class, an exception handler can be specified for each. To identify for which constructor the handlers are intended, the formal parameters are used. Note that the parameters must be placed within braces, e.g. `(String first, int second)`.

Whenever an instance is created remotely, java.rmi.RemoteException and any of its subclasses can be thrown.

```
Invokes:
   (Invoke)+

Invoke:
   MethodDeclarator CatchBlocks

MethodDeclarator:
   <METHODNAME> FormalParameters
```

To specify for which method an exception handler is intended, the method name and the formal parameters (as above) are required. (This to take into account overloaded method names.)

Whenever an invocation occurs, java.rmi.RemoteException and any of its subclasses can be thrown.

Note that if the method has a declared return value different from `void`, the CatchBlock has to return an object (or primitive) of the declared type. If this does not happen, a compile error will occur.

```
CatchBlocks:
   "{" (CatchBlock)* "}"

CatchBlock:
```

```
   "catch" "(" <TYPENAME> <PARAMNAME> ")"
   (Block | Break | Switch)

Break:
   "break" Block

Switch:
   "switch" Block
```

CatchBlocks are similar to BootCatchBlocks, therefore we will only describe the differences between the Break and Switch parts and the BootBreak and BootSwitch parts.

The Break part allows the link to a remote instance of the class to be broken, and a local instance to be created. The difference between a BootBreak and a Break is that here the Block is required. This is because actual creation of the local instance is achieved by calling the method with the name `"distrac_break_instance"` from within the block. For each constructor a private `"distrac_break_instance"` method, with formal parameters identical to the formal parameters of the constructor, will be included in the object. Whereas the constructors may create the object remotely (depending on the specifications in the Dist file), these methods will always create an instance locally.

The Switch part allows this instance to refer to another remote instance, this instance would, as it were be, "switched" to another remote instance. The important conceptual difference between a BootSwitch and a Switch is that here we do not switch servers, but switch instances. The second difference here is, as in the Break, that the block is required. To switch to another remote instance, the private method `"distrac_switch_instance"` must be called from within the block. The method has one parameter, the type of which is the class of this instance. (Note that this method will only be generated by DistraC if there is at least one Switch specified for this class in the Fix file.)

If the Break or Switch parts are part of an Invoke error-handler, the call which caused this error will be re-executed on the new instance. The call, in case of a Switch, may of course also fail. The error-handlers used to handle these failures will be the ones specified for this call, i.e. the ones the Switch is a part of.

5.4.3 Additions

Last but not least, as in the Dist file, some additions to the class can be specified.

```
Extends:
   "extends" ClassBody

New:
   "new" Block
```

In some cases, extra functionality only pertaining to the error-handlers aspect will have to be added to a non-remote part of the class. Extends allows extra variables and methods to be declared in the body of the part, ClassBody corresponds to a block declaring instance variables and defining methods. New allows a block of statements to be executed locally whenever a new instance is created. This can be used to e.g. initialize instance variables declared in the Extends.

Note that, as in Dist, this functionality is placed locally. This allows error-handling code to be performed locally, avoiding that new errors are caused by the error-handling code.

5.4.4 Example

Let us now extend our example with some error-handling. Suppose we have two servers available: `tongariki` and `akivi`. We are not using them for load balancing, the first server is the 'main' server, the second is a backup. To assure automatic roll-over to the backup server at instance creation time we specify the following Fix file (assuming we are using the Dist file which does not use load-balancing):

```
class HelloWorld
   extends { private boolean first = true; }
   {
      catch(Exception ex} switch {
         if(first) {
            first = false;
            return "akivi:4242";
         }
         else return null;
      }
   }
```

The file is fairly straightforward; we first declare an instance variable `first` which determines if we are trying the first server or not. Upon failure to connect to the remote host we specify that a switch to another host will be made. If we tried to connect to the first server, we switch to the second server. If we tried to connect to the second server, we will not connect to a remote server, but create the instance locally.

Recall that remote instance creation is a two-step process, here we have only handled the first step: connecting to the remote class. We can handle the second step: creating the instance, and handle errors occurring during the method invocation, by specifying the following in the Fix file:

```
class HelloWorld
   extends {
      private boolean first = true;

      private void doSwitch() {
         HelloWorld hello = new HelloWorld("Hello, world!");
         distrac_switch_instance(hello);
      }
   }
   {
      catch(Exception ex}
      switch {
         if(first) {
            first = false;
            return akivi:4242;
         }
         else
            return null;
      }
   }
   (String text) {
      catch (RemoteException ex)
      switch {doSwitch();}
   }
   sayHello() {
      catch (RemoteException ex)
      switch {doSwitch();}
   }
```

We specify here that if an error occurs when invoking the constructor, or when invoking the method, we should switch to another instance of the object, which will be performed in the `doSwitch` method. This method, which we defined in the `extends`, is simple: it creates a new instance and switches over to this instance. Because the error-handler for the first part of instance creation will switch over to the backup server, or to a local instance, we need not specify anything extra here.

Now to ensure that the servers contain the remote classes of which we want to create an instance, we need to specify this. Our last aspect language; Serv, was created to facilitate these specifications.

5.5 Serv

Because instances can be created on a remote machine, there is a need for a "server" program in which these classes will reside. DistraC will create these server programs, one for each host and port combination.

However, as the host and port names can be given by a block of code, there is no practical way to let DistraC deduce on which computer instances of a remote class can be created. Therefore this information has to be specified separately.

We have designed the `Serv` language, of which the grammar is given in appendix A, to easily specify which classes are placed on which host.

A Serv file defines a number of servers, by providing host name and port number, in the form hostname:portnumber e.g: `tongariki:4242`.

For each server a number of classes can be specified as being contained on that server, so that instances of these classes can be created there. The name of each class should be a fully qualified name, i.e. the name includes the package of the class. This is done to avoid errors in case of duplicate class names.

Example

For our example, in both the load-balancing and the fault-redundant version, we need to specify the following in the Serv file:

```
tongariki:4242 {
   HelloWorld; }
akivi:4242 {
   HelloWorld; }
```

We have now seen all the aspect languages, and can now proceed with detailing how the aspect weaver will transform the given code, integrating all the aspects. These transformations are the subject of the next chapter.

5.6 Conclusions

In this chapter we detailed some general design decisions and introduced the aspect languages, Dist, Fix and Serv.

We first discussed the use of RMI as a transport layer for the generated programs and how remote instantiations become a two-step process. Furthermore we introduced our rules for parameter passing, and the default error-handler provided by DistraC.

We talked about the languages used in the system; Java, to which we have added a restriction of not allowing any `RemoteExceptions`, and our three aspect languages: Dist, Fix and Serv, which we specified fully.

With full knowledge of the languages we defined, we can now detail how our aspect weaver transforms the input files, integrating these different aspects into an executable file.

Chapter 6

Transformations to the Code

If everything's under control,
you're going too slow.
— Mario Andretti

Having introduced the aspect languages in the previous chapter, we can now discuss how DistraC, our aspect weaver, will transform the input code, to provide code which integrates these aspects.

We will first introduce the interfaces generated by the weaver, which we have called classfaces. Secondly we will discuss how a class is made remote by splitting it into a local proxy and a remote part. Thirdly we will talk about how correct parameter passing is achieved, according to the rules we have introduced in the previous chapter. Following this, we will detail how instances of remote classes are created, with special emphasis on remote instantiation. Finally, we will give an overview of the servers created by DistraC.

6.1 Classfaces

As the programs generated by DistraC will use RMI as a transport layer, all remote classes will have to implement a remote interface. Recall that in RMI the type of a reference to a remote class is not the type of the remote class, but the type of a remote interface, implemented by the class.

Therefore, DistraC automatically generates a remote interface for each remote class. We call this interface a *classface*, and its name is the name

of the remote class, to which `Classface_` is prepended. Note that DistraC verifies that a class with this name does not exist. If such a class exists, the name of the classface will be changed such that there are no conflicts.

A classface extends the interface `java.rmi.Remote`, contains all non-static methods defined in the remote class and extends the throws clause of each method with a `java.rmi.RemoteException`. As said in chapter three, a remote interface needs to extend `java.rmi.Remote`, and have all declared methods throw `java.rmi.RemoteException`.

RemoteExceptions

Extending the throws clauses of all methods may lead to incorrect code. This is because, in Java, a sub-interface (or subclass) of a given interface (or class) is not allowed to extend the throws clauses of the declared methods. This might not seem to be a problem, because no methods are declared in the `java.rmi.Remote` interface, but this is not the case.

The problem lies with `Object`; all interfaces are subclasses of `Object`, therefore, for a number of methods, such as `equals` and `toString`, we may not extend the throws clause.

We have a straightforward solution for this problem: instead of using the original name of the method in the classface, we prepend the string `distrac_` to the name of the method. This ensures that these methods have not been declared in a super-interface, and therefore we can extend the throws clause with `java.rmi.RemoteException`. Note that, for consistency reasons, we do not only prepend the names of the methods declared in `Object`, we prepend `distrac_` to all the method names in the interface.

Now, because we have changed the method names, this implies that the remote class cannot implement the classface. Therefore, extra methods are added to the remote class. These methods, with name and parameters declared in the classface, forward the method call to the original method.

We could have chose to rename the methods in the class, but this would imply that we would need to change all the method invocations using the old method names to the new method names. We have not done this because the 'forwarding' methods which are added will be used to enable correct parameter passing, which we will discuss in section 6.3.

There is one disadvantage to this solution, which is that method names in the code may not start with `distrac_`. If this is the case, errors may occur. For example, consider a remote class containing the methods `distrac_boo` and `boo`. The classface will declare the methods `distrac_distrac_boo` and `distrac_boo`. This implies that a method `distrac_boo` will be created in the

remote class, which redirects the method call to `boo`. However, the remote class already contains a method called `distrac_boo`, therefore the method is declared twice, and the code is incorrect.

We could have the weaver search for these cases and handle them correctly. But, considering the extra processing this would imply, and the fact that this is not relevant to the core of our work, we have elected to impose the rule that no methods may start with `distrac_`

Static Methods

Note that we have specified that only non-static methods of the remote class are contained in the classface, which means that the static methods will not be executed remotely. The reason for this is that there is an unresolved issue if several remote objects of the same class exist on different servers.

With all objects of a remote class on one server, the class methods can be executed by the class on that server, which conserves the semantics of class methods and class variables. However, when these remote objects are distributed over different servers, it is not clear where the class methods must be executed: do we consider the class methods to be limited in effect to the server on which that class is located, or should the effect be on all occurrences of that class, spread out over the different servers?

It seems more logical to consider the second, i.e. we should have a 'global' class, of which all occurrences on the different servers are a reflection. However, this poses a twofold problem: firstly, where should the processing of a remote class be performed; on one occurrence, or on all occurrences, and secondly, where are the values of the class variable kept; on one occurrence, or on all occurrences?

The first option, centralizing all class information in one occurrence seems the most straightforward. However, this implies that if the server containing this occurrence fails, invoking static methods will be problematic because of the possibility that the other occurrences will be in an invalid state since their static variables have not been updated. This implies that, at least, we must perform replication on the static variables to achieve a system which can recover if a server fails.

Because of the extra work this replication would entail, we have decided not to perform static methods remotely. However, this does not prohibit this feature to be added to the system later.

Having introduced the classface for the remote class, we can now proceed with detailing the transformations needed to make a class truly a remote class.

6.2 Remote Classes

As our goal is to achieve distribution transparency, we want accesses to the remote class to be as transparent as possible, i.e. a remote class should be indistinguishable from a local class.

To achieve this, we have two options: we could change all references and accesses to the remote class so they handle the 'remote-ness', or we could replace the remote class with a proxy, which encapsulates the 'remote-ness'.

We have chosen the second option, because of this encapsulation. It limits the impact of the class changes, making the code easier and faster to process, and making resulting code smaller and easier to read.

This entails that making a class a remote class first consists of splitting it up into two parts, a local proxy and the real remote class. The local proxy will contain a reference to the real remote class, forward method calls to this remote part, and handle the extra processing required due to the remote method invocations. The remote part will perform the actual processing, and return the result, if any, to the proxy, which will return it to the caller. A schematic overview of this transformation is given in figure 6.1.

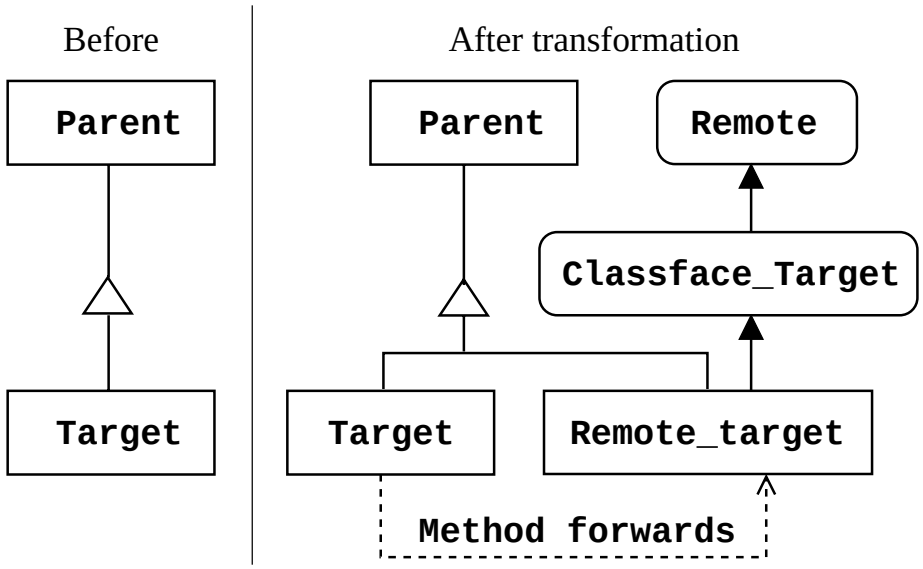

Figure 6.1: The transformation process making a class named "Target" remote. On the left hand side the situation before, on the right hand side the situation after the transformation. Rounded boxes represent interfaces.

Figure 6.2 gives an overview of how these two classes interact when a method is invoked on the proxy.

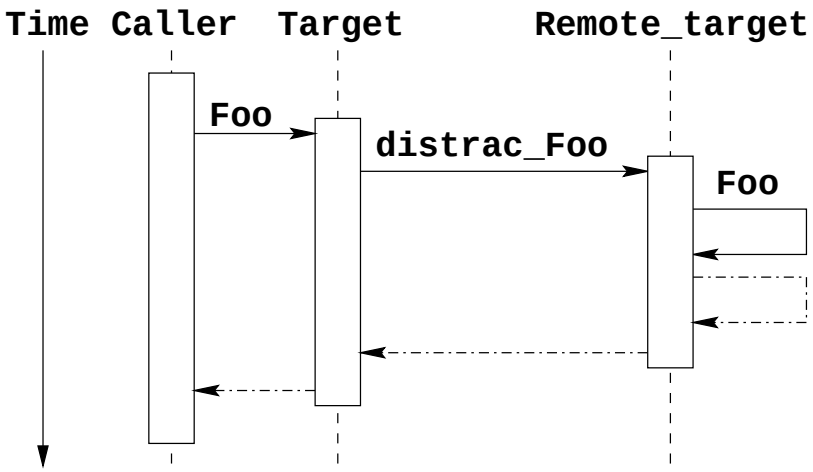

Figure 6.2: The interaction between the Proxy (named "Target") and the remote part (named "Remote_Target") when a object ("Caller") invokes a method "Foo" on the Proxy. We have included the passing of return value as dotted lines. Note that the method call "distrac_Foo" is a remote method invocation.

We will now discuss these two classes in more detail. We will firstly talk about the proxy and secondly detail the remote part.

The Proxy

The proxy is the part of the remote class which takes care of the overhead generated by the remote method invocations, encapsulating the 'remoteness' of the class.

Basically, the proxy forwards incoming method calls to the remote part of the class, and handles all overhead induced by RMI. This makes the proxy a 'drop-in' replacement for the original remote class, i.e. it replaces the original class and provides the same functionality. Therefore, the name of the proxy class is identical to the name of the original class, and the proxy class has the same superclass and implements the same interfaces as the original class.

To forward the method calls, the proxy contains, internally, a reference to the remote part, which has as type the classface of the remote class. All

methods of the proxy will call the corresponding method on the reference, making it a remote method invocation. Also, these methods will perform some transformations needed for correct parameter passing, which we will discuss in 6.3, and will handle exceptions thrown by RMI. Recall that these exception handlers are, or the handlers specified in the Fix file, or the default error handlers generated by DistraC.

Note that, as the error-handlers of the Fix file are added here, the contents of the Extends in the Fix file, are also added to the proxy, and not to the remote part.

We have not yet discussed how the reference to the remote part is obtained. This reference can be obtained in two ways, either through a remote instantiation, which we will discuss in 6.4 or it can be given as a parameter to a remote method invocation, which we will discuss in 6.3.

However, before we discuss this, we must first detail the functionality of the remote part.

The Remote Part

The easiest way to envision the remote part, is to consider it as the original class, with a different name, and some extra methods.

The name of this remote part is the name of the remote class, to which `Remote_` is prepended. Note that DistraC verifies that a class with this name does not exist. If such a class exists, the name of the remote part will be changed such that there are no conflicts.

The extra methods added to the class, are those we described when introducing the classface; they enable the class to implement the classface, handle extra processing required for parameter passing, and forward method invocations to the original method.

To be able to be used in a remote method invocation, firstly the remote part is also set to implement the classface which was generated for the remote class, and secondly, the remote part is immediately exported when it is created.

Having divided the class in proxy and remote part, we must now ensure that the parameters and return values of the remote method invocations are passed correctly between the different machines.

6.3 Parameter Passing

Recall that in the previous chapter, we introduced the following parameter passing rules:

1. All classes declared as remote are passed by reference
2. Serializable classes (which are not declared as remote) are passed by copy
3. All other non-system classes, (which are non-serializable and not declared as remote) are passed by reference.

We will now discuss the transformations required for these rules, in reverse order.

To satisfy rule three, the classes which satisfy this rule will be transformed into remote classes, which will result in rule one being applied. Note that this implies that the parameters of the method of these classes will also have to satisfy these rules, which implies that some of these classes may also be made remote. DistraC performs this transitive closure to determine which classes have to be made remote in a special, pre-processing stage and will return an error here if a class is encountered for which the source is not available.

Satisfying rule two is straightforward. This is because the rule is identical to the RMI rule which states that serializable classes are passed by copy. As we are using RMI as the underlying transport layer, no extra processing has to be made.

Rule one requires a larger amount of work; if we determined that a parameter is a remote object, that parameter is not a reference to the remote part, but to the proxy of the part. Therefore, the proxy must be unwrapped from the remote object, so the remote part may be passed as reference, and the part must be re-wrapped in a proxy at the receiving end.

These wrapping and unwrapping operations will be performed by the forwarding methods in the proxy and by the forwarding methods in the remote part.

Note that passing the remote part as parameter entails that the method in the classface cannot have as type the proxy of the remote part. The type will be the type of the remote part. Also, this requires that the proxies provide an alternate instantiation, which just wraps an existing remote object.

A final remark of parameter passing is how interfaces are handled. The above parameter passing rules apply, of course, not only to objects whose type is a given class, but also to objects whose type is an interface. This implies a certain overhead when passing an interface type as a parameter. As DistraC generates proxies for remote classes and not for remote interfaces, the interface is not wrapped in a proxy for the interface but an interface for the class.

To allow this, the interface will be transparently extended with a method which allows the class of the object to be determined. When the remote part needs to be wrapped in a proxy, the class of that proxy will be determined using the above method, and the object will be wrapped in a proxy of that class.

This procedure allows casts from the interface of the parameter to the actual class of the parameter to be performed in the methods using that object, which is a situation which can occur quite frequently in Java.

Having a proxy wrap an existing remote part is a first way in which a proxy can refer to a remote part. A second way to have this reference in a proxy, is when the proxy is instantiated normally, which we will discuss in the next section.

6.4 Remote Instantiation

As a proxy is a drop-in replacement for the original class, this proxy will be instantiated by the base code, instead of the original class, whenever the base code intends to instantiate the original class. It is obvious that because, essentially, the proxy forwards method calls to a remote part, this remote part must also be instantiated when the proxy object is created.

Instantiation of the remote part can be performed in two different ways:

- If the Dist file does not contain a specification for remote instantiation, the constructors of the proxy will create a remote part locally by invoking the corresponding constructors in the remote part.
- If the Dist file does contain a specification for remote instantiation, the constructors of the proxy will create a remote part on a remote machine.

However, RMI does not provide a mechanism to instantiate an object remotely, therefore we must implement a mechanism for remote instantiation ourselves. The mechanism we used is inspired by the well-known Factory design pattern [6].

For each remote class which can be instantiated remotely, we create a *Bootstrap class*. This class contains methods which create an instance of the remote part of the remote class, and returns this remote part. For each constructor of the remote class, a corresponding method exists in the bootstrap class.

But to be able to create a remote instance, an instance of the bootstrap class will have to be placed on the server which will contain the remote part

of that instance. This is because instantiation can not be done across virtual machines. We need to place the bootstrap object in the Java virtual machine where the remote part should be.

The above implies that the bootstrap object is itself a remote object. To make this remote object available to the proxies, each server will instantiate an instance of this object, as indicated in the Serv file, and will register this object in the RMI binder under a predefined name (the name of the class).

An overview of the structure for a remote class which can be instantiated remotely, is given in figure 6.3.

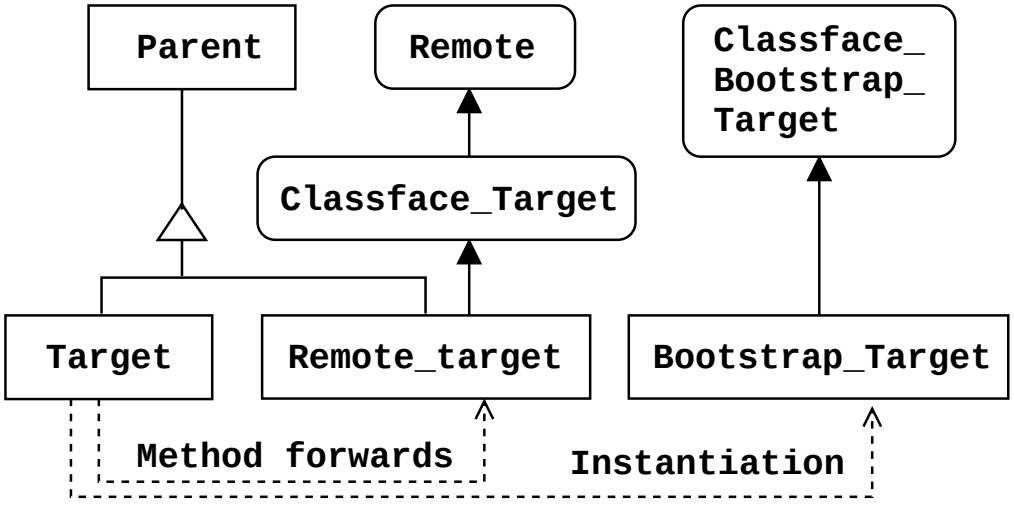

Figure 6.3: An overview of the classes resulting from the transformation process if the class can be instantiated remotely. Rounded boxes represent interfaces.

To instantiate an object remotely, the constructors of the proxy now have to:

1. Obtain a reference to a bootstrap object by using the lookup method provided by the RMI binder. The host name used in the lookup is the host name given in the Dist file, either literally or by a block of code.

2. Invoke the factory method corresponding to the constructor. This will return a reference to the remote part of the object, which will be used by the proxy as the target object of the method redirections.

These two steps, which use RMI, may throw exceptions. The error-handlers used to catch these errors are, respectively, the Bootstrap handlers and the Creates handlers declared in the Fix file. Recall that if the Fix file does not declare any error handlers, the default error handlers are used.

Also note that, as determining the host name when obtaining a reference to a bootstrap object is performed in the proxy, the contents of the Extends clause of the Dist file is placed in the proxy.

In this section we have briefly mentioned the servers which will be created by DistraC. We will now give an overview of these servers.

6.5 The Servers

As each object must be contained in a Java VM to be able to access and use it, we must provide a virtual machine for the remote objects. These virtual machines will be our servers.

Using the Serv file, DistraC will generate a number of class files, one for each server declared in the Serv file. These executable class files will, when run, enable remote objects to be created on that server.

The code of the server class is straightforward, it consists of two main steps:

1. The RMI binder service, called the registry, is started on that server, i.e. on the local host, with the port specified in the Serv file. Although only a single RMI registry is needed for a distributed system, and all bindings can be placed in that binder, this is not advisable. This is because, if the machine holding the registry fails, no bindings can be resolved, and the entire distributed system can become un-operational. Therefore we use one binder per server, and only the objects located on that server are registered in that binder. If the machine containing that server fails, only lookup of objects on that server will fail (which is acceptable because the server itself will be un-operational).

2. For each remote class of which instances can be created on this server, as declared in the Serv file, an instance of the bootstrap class will be created, exported and bound to a predetermined name. When the clients obtain a reference to the bootstrap object, they will use host-name and port defined in the Dist file to address the binder service on that host machine with corresponding port, and request the object of the predetermined name.

Exporting the bootstrap objects will launch a number of non-daemon threads which handle the RMI connections to the bootstrap objects. This implies that no extra waiting or looping code must be defined in the server class, and therefore or discussion of the code of the server is complete.

Having now seen the transformations performed by the weaver, we can, in the next chapter, introduce the experiments we performed to determine the achieved degree of distribution transparency.

6.6 Conclusions

In this chapter we described the transformations performed by the weaver, which integrate the different aspects into executable code.

We first introduced classfaces, the interfaces through which the remote classes will be accessed. This was followed by an overview of how a class is made remote by splitting it up into a local proxy and a remote part. Next, the issue of parameter passing was discussed, detailing how the rules described in the previous chapter are applied. Following this, we described how instances of remote classes are created, emphasizing remote instantiation. And last, but not least, we have given an overview of the servers generated by DistraC.

Having fully described the aspect languages and given an overview of the workings of the weaver, we can now give an overview of the experiments we performed to determine if we have a high degree of distribution transparency, while still retaining sufficient control of error-handling.

Chapter 7

Main Experiments

"That's why it's always worth having a few philosophers around the place. One minute it's all Is Truth Beauty and Is Beauty Truth, and Does A Falling Tree in the Forest Make A Sound if There's No one There to Hear It, and then just when you think they're going to start dribbling one of 'em says, Incidentally, putting a thirty-foot parabolic reflector on a high place to shoot the rays of the sun at an enemy's ships would be a very interesting demonstration of optical principles."

— Terry Pratchett, "Small Gods"

To validate our claim that a higher degree of distribution transparency can be achieved though AOP, we have performed a number of experiments, of which two will be discussed in this chapter.

The first experiment is a distributed messaging application, in which users can send short, textual, messages to other users, the second is a simulation of a distributed library, where books are searched for in a distributed fashion.

7.1 The Messaging Application

Our first experiment is a messaging application, where users can send messages to each other in real time. We will first give an overview of the application, second we will describe the implementation of the base aspect, and third is a description of the distribution aspect. We will conclude with a discussion on some points of interest.

7.1.1 Situation

A currently quite popular and reasonably small class of distributed systems are messaging applications. These allow a user, when registered, to send short, textual, messages to other registered users, which arrive immediately.

Our first experiment is such a messaging system. We have decided to only provide basic functionality, to keep the example short and easier to understand.

The system is structured as follows:

- There is one central server for the system, which will keep a list of registered clients. The server will inform all clients when this list changes, i.e. when a client has subscribed or unsubscribed itself.
- As implied above, a client can register and unregister itself on the server. Registering implies providing a unique name for the client (usually the username or her alias) and a reference to the mailbox used by the client.
- A client can request a list of the currently registered clients to the server. The server will return a list containing the names of the clients. Also, a client can obtain a reference to another clients' mailbox.
- To send a message to another client, a given client obtains the reference to the other clients' mailbox, and places this message in this mailbox.

Also, the following items are relevant for error-handling:

- The server is assumed to be a machine with a high availability that can recover from a failure. This means that down time will be minimal, and the state of the server after a failure is resolved, will be identical to the state immediately before the failure occurred.
- Clients are more error-prone, and may fail at any time. It is acceptable for a message to a client to be lost, but only if the sending user is informed of this loss.

Having seen the overall structure and functionality of the system, we will now discuss the code for the base aspect, i.e. ignoring the issues created by the distributed nature of the system.

7.1.2 The Base Aspect

We will now introduce the code for the classes of the base aspect. We will first discuss the `ContactManager` and the `ContactSingleton` classes, which are the classes responsible for managing the list of users. Second, we will introduce the `Message` class, which contains the actual messages passed around, and the `MailBox` class, which is the representation of a user in the system. Third, and last, we will briefly discuss the user interface.

ContactManager

Management of the list of users and their mailboxes is performed in the `ContactManager`.

The ContactManager keeps the associations of users and their mailboxes in a standard Java Hashtable, with as key the user name and value the mailbox. A user can only add her mailbox to the Hashtable, if the username is not already a key in the table.

Upon modifications of the hashtable, the manager will notify all mailboxes in the table by calling the `usersChanged` method on each mailbox. A Java Vector containing the names of the connected users can be obtained by calling the method `getUsers`.

To get the mailbox for a given user, the method `getMailBox` must be called, with as parameter the name of that user. If the user does not exist, a `UserNotFoundException` will be thrown. This exception is a simple subclass of the standard Java `Exception`, which does not add any extra functionality. (Therefore we will not include the code for this exception here.)

The code for the `ContactManager` is straightforward, as can be seen below:

```
package dmsg; import java.util.*;

public class ContactManager {
   private Hashtable users;

   public ContactManager() {
      users = new Hashtable();
   }

   public boolean addUser(String name, MailBox box) {
      if (users.containsKey(name))
         return false;
```

```
      else {
         users.put(name, box);   updateUsers();
         return true;
      }
   }

   public void removeUser(String name) {
      users.remove(name);   updateUsers();
   }

   private void updateUsers() {
      Enumeration boxes = users.elements();
      while (boxes.hasMoreElements()){
         MailBox mb = (MailBox) boxes.nextElement();
         mb.usersChanged();
      }
   }

   public Vector getUsers() {
      Vector retval = new Vector();
      Enumeration usernames = users.keys();
      while(usernames.hasMoreElements())
         retval.addElement(usernames.nextElement());

      return retval;
   }

   public MailBox getMailBox(String name)
   throws UserNotFoundException {
      Object box = users.get(name);
      if (box == null)
         throw new UserNotFoundException
                        ("User "+name+" not registered here");

      return (MailBox) box;
   }
}
```

ContactSingleton

To register herself, and to be able to contact other users, a user must first obtain a reference to the `ContactManager`. It is obvious that only one manager may be allowed in the system, since it centralizes all user contact information.

A well-known design pattern that guarantees uniqueness of an object and an easy way in which this unique object can be accessed is the Singleton design pattern [6]. The class `ContactSingleton` is used to implement this pattern for the `ContactManager` class.

The idea here is that `ContactManager` is never instantiated directly, but always through the `ContactSingleton` class. This class has a static variable which contains a reference to the unique `ContactManager` instance. The method `getInstance` returns this unique instance.

Getting a reference to the unique class is easily performed through the following code: `(new ContactSingleton()).getInstance();`. The advantage of this implementation of the Singleton pattern is that no static methods are used, which is important in our case, since DistraC does not handle static method invocations correctly.

The code for `ContactSingleton` is quite simple:

```
package dmsg;

public class ContactSingleton {
   private static ContactManager themanager;

   public ContactSingleton() {
      if (themanager == null)
         themanager = new ContactManager();
   }
   public ContactManager getInstance() {
      return themanager;
   }
}
```

Message and MailBox

The actual data which is passed around between the different users, are simple messages, represented in the `Message` class. This class is a simple wrapper for the username of the sender and the text being transmitted, as can be seen below:

```
package dmsg;
public class Message implements java.io.Serializable {
   public String username;
   public String text;

   public Message(String user, String thetext) {
      username = user; text = thetext;
   }
}
```

To send a message to a user, we need to put it in her `MailBox`. Therefore, the `MailBox` class is the representation of a user within the system. It contains a Vector of `Messages`, and provides a method `addMessage` which adds a new message to the Vector.

This class is executable, the `main` method will set up and run the application by creating a MailBox instance, using the Singleton pattern to obtain a reference to the `ContactManager`, registering the user, and creating a user interface, which we will describe in the next part.

The code for the `MailBox` class is quite simple, as can be seen below:

```
package dmsg;

import java.util.*;

public class MailBox {
   private Vector messages;
   private FancyUI ui;

   public MailBox() {
      messages = new Vector();
   }

   public void addMessage(Message msg) {
      messages.addElement(msg);  ui.newMessage();
   }

   public Vector getMessages() {
      return messages;
   }
```

```
   public void setUI(FancyUI theui) {
      ui = theui;
   }

   public void usersChanged() {
      if (ui != null)
         ui.usersChanged();
   }

   public static void main(String args[]) {
      if (args.length != 1) {
         System.out.println("Need user name as argument!");
         System.exit(1);
      }

      MailBox mb = new MailBox();
      ContactManager cm =
         (new ContactSingleton()).getInstance();

      if (!cm.addUser(args[0],mb)){
         System.out.println("User name already in use.");
         System.exit(1);
      }

      FancyUI ui = new FancyUI(cm,args[0],mb);
      mb.setUI(ui);  ui.usersChanged();
   }
}
```

FancyUI

We have also implemented a user interface for the system, in the `FancyUI` class. We will not include it here as it is quite large, while not containing much relevant code. Three methods are of interest: `usersChanged`, `sayError` and `sendMessage`.

The `usersChanged` method will display a new list of online users, based on the result of calling the `getUsers` method on the `ContactManager`. Displaying an error message is achieved by calling the static `sayError` method, which will print out this message for the user to see. The reason why this method is static will be discussed in 7.1.4. `sendMessage` sends a message

to a given user, the code for the method is straightforward, as can be seen below:

```
public void sendMessage(String username, String msg) {
   try {
      MailBox mb = cm.getMailBox(username);
      mb.addMessage(new Message(thisname, msg);
   }
   catch(UserNotFoundException ex) {
      sayError(ex.toString()); usersChanged();
   }
}
```

Having finished the base aspect, we can now concentrate on the distribution aspect.

7.1.3 The Distribution Aspect

With the code for the base functionality complete, we can now reason about which classes should be made remote, where these remote classes should be placed, and how to handle errors.

Dist and Serv

Deciding which classes should be remote is quite straightforward: We need the contact manager to be placed on a central server, but we need to take into account the singleton design pattern.

Therefore if we declare `ContactSingleton` to be remote, and locate it on a central server, the unique instance of the `ContactManager` will also be located on the server. To achieve this, we specify the following in the Dist file:

```
class dmsg.ContactSingleton tongariki:4242
```

We need not specify any other class as being remote; DistraC will automatically deduce which other classes should be made remote because of the parameter passing rules.

However, to illustrate the rules, we will discuss how they are applied to the `MailBox` and `Message` classes.

- `MailBox` will be made remote. This is because it will be given as a parameter to the `addUser` method of the `ContactManager` class, which is also remote. The last because it is given as return value of the `getInstance` method of `ContactSingleton`.
- `Message` will not be made remote. It is a parameter of a method of the `MailBox` class (which, as deduced above, is remote), but, because it implements the `Serializable` interface, it is not made remote.

Having specified a remote class which can be instanced remotely, we need to indicate the servers on which this class can reside. This is achieved by the following Serv file:

```
tongariki:4242 {
   dmsg.ContactSingleton;
}
```

With the above specifications, we can compile a working messaging system. However, this system will not be able to recover from any partial failures, such as, e.g. a users' machine failing.

Fix

To add failure recovery to the distributed system, we have to specify a number of exception handlers in a Fix file.

Recall that the server was specified to be on a machine with high availability. This implies that communications from the clients to the server will rarely fail, and if there is a server failure, this will be quickly resolved. Therefore, in case there is a communication failure, we can let the clients retry until the communication completes successfully.

This strategy is realized by giving the following Fix specifications for the `ContactSingleton` and the `ContactManager` class:

```
class dmsg.ContactSingleton
{
   catch(Exception ex)
   switch {
      FancyUI.sayError("Server down, retrying.");
      return "tongariki:4242";
   }
}
() {
```

```
   catch(java.rmi.RemoteException ex)
   switch {
      distrac_switch_instance(new ContactSingleton());}
}
getInstance() {
   catch(java.rmi.RemoteException ex)
   switch {
      distrac_switch_instance(new ContactSingleton());}
}

class dmsg.ContactManager
extends {
   private void retry() {
      FancyUI.sayError("Server down, retrying.");
      ContactManager newcm =
         (new ContactSingleton()).getInstance();
      distrac_switch_instance(newcm);
   }
}
addUser(String name, MailBox box) {
   catch(java.rmi.RemoteException ex)
   switch {retry();}
}
getMailBox(String name) {
   catch(java.rmi.RemoteException ex)
   switch {retry();}
}
getUsers() {
   catch(java.rmi.RemoteException ex)
   switch {retry();}
}
removeUser(String name) {
   catch(java.rmi.RemoteException ex)
   switch {retry();}
}
```

The above specifies that whenever an exception is thrown while contacting the server, first an error is printed out, and second the instance is switched over to a new instance. This 'new' instance will be the singleton, located on the same server. However, the server may still be inoperational,

therefore the error-handlers will be re-invoked, which implies that the server will be re-contacted, et cetera ... In other words, while the server is down, the client will retry until the server is contacted successfully.

As stated in the specifications, clients are more error-prone. If a message can not be delivered because a client has failed, the message may be lost, but an error-message has to be printed out. This can be easily handled by the following Fix code:

```
class dmsg.MailBox
addMessage(Message msg) {
   catch(java.rmi.RemoteException ex) {
      FancyUI.sayError("could not deliver message.");
   }
}
```

7.1.4 Discussion

This experiment showed us how we can easily create a distributed system using DistraC.

When programming the base functionality we did not need to concern ourselves with the distributed nature of the system. All the functionality relevant to the distribution aspect has been specified solely in the aspect languages created for that purpose. Using these separate declarations, it was easy to specify the distributed nature of the system, which included error-handling.

An interesting point in this example is the use of the Singleton design pattern. In distributed systems services are usually made available by binding them to a name in the binder. Clients wishing to use the service get a reference to it, using the binder service. DistraC does not provide this feature, however this does not mean that services made available by a server can not be used by clients, as this is exactly what happens in this experiment. Using the singleton allows us to easily connect to an existing service delivered by a server. This is because the semantics of a singleton in a non-distributed program is highly similar to the semantics of a service delivered by a server in a distributed system: there is one instance of the service available, and clients can easily get a reference to that instance.

Note that if static method calls were correctly handled by DistraC, we could have implemented the singleton without needing to create an extra class. However we feel that the overhead of creating this extra class for the singleton is not significant, as it aids to the legibility of the program.

Also notable in this example was a problem which arose when writing the error-handling code. Concretely: whenever we wanted to print out an error message, we had to call a method on the user interface. However, the proxy objects for both remote classes did not have a reference to the user interface object, and therefore no instance methods of the user interface could be called. We solved this by changing the code of the user interface which displays the error; we added a static method that prints out the error. However, when doing this, we changed the code for the base aspect, which means that the error-handling has not been separated out completely.

On a more abstract level this problem can be stated as follows: the error-handling code might need references to objects which the normal code does not need. Therefore, there can be cases, as above, where the error-handler can not access certain objects, because the base code which caused this error to occur does not contain a reference to the required object.

A possible solution for this would be to extend the error-handling aspect with the facility to insert code for error-handling in more locations than currently is the case. This code would then store references needed for error-handlers in a location which can be accessed by them in, for example, a static variable.

Concretely, we suggest that this code may be added to each method of each class in the system, and also that extra methods and variables may be added to all classes. This would allow object references to be stored whenever they would be necessary to handle errors later.

7.1.5 Conclusion

Our first experiment was a distributed messaging application, where each user can send a textual message to any other user logged in to the system.

We have seen how the base functionality can be specified without needing to keep in mind the distribution aspect, and how this distribution can easily be added later in a completely separate phase.

We have noted the similarity between the Singleton design pattern and the concept of clients connecting to a service provided by a server. We have shown that the singleton can be used to provide this feature to programs made with DistraC.

Also, we have encountered the problem that in some cases error-handlers may need to access objects which are not referenced in the corresponding base aspect code. We have proposed as a solution extending the error-handling aspect with the possibility to specify code to be executed in a wider variety of locations. This code would then store the needed references where

they would be accessible to the error-handlers.

7.2 The Distributed Library

The second experiment is a simplified simulation of a distributed library, where books can be searched for on different locations. First we will present a small overview, followed by a description of the base functionality and a description of the distribution aspect. We will conclude with a short discussion of the experiment.

7.2.1 Situation

A different kind of application from our first experiment, this experiment is a simple simulation of a distributed library. The library could be considered similar to a big university library, where the books are kept in multiple locations on campus. The main functionality which is required, is to be able to search for a book by name or by author.

Each location keeps a database of its books locally, and can perform the search locally. The overall application needs only to delegate the search to the different locations, and join the results.

The library we simulate will consist of two locations. One of these two locations contains a pair of servers in a mirrored configuration, i.e. if one of the two servers fail, all clients can immediately switch over to the other server without loss of data. The other location, however, contains only one server, and there is no guarantee on its availability.

Given this overall structure, we will now specify the base aspect code for this distributed library.

7.2.2 The Base Aspect

The code for the base aspect, which we will introduce here, contains four major classes: the `Library`, `Location`, `Booklist` and `Book` classes. We will now give an overview and provide the code for each of these classes.

Library

The `Library` class is the top-level representation of the distributed library; all interactions with the library are through this class.

A `Library` object contains an array of `Location` objects, in this case two; one for each location of books. When creating the library, the array is

filled with the `Location` objects, and finding a book is performed through the `findBook` method, which returns a list of found books.

This class also contains a `main` method, which runs a simple simulation which first looks up a book by title and prints out the results, and subsequently looks up a book by author and prints out the results.

The code for the `Library` class is quite simple, as can be seen below:

```
package library;

public class Library {
   private Location[] locales;

   public Library() {
      locales = new Location[2];
      locales[0] = new Location("\\library\\first","One");
      locales[1] = new Location("\\library\\secnd","Two");
   }

   public BookList findBook(Author auth) {
      BookList list = new BookList();
      for (int i=0; i<locales.length; i++) {
         BookList newlist = locales[i].findBook(auth);
         newlist.appendTo(list);
      }
      return list;
   }

   public BookList findBook(Title titl) {
      BookList list = new BookList();
      for (int i=0; i<locales.length; i++) {
         BookList newlist = locales[i].findBook(titl);
         newlist.appendTo(list);
      }
      return list;
   }

   public static void main(String args[]) {
      Library lib = new Library();
      System.out.println("Looking for title title2");
      System.out.println(lib.findBook(new Title("title2")));
```

```
      System.out.println("Looking for author author1");
      System.out.println(lib.findBook(new Author("author1")));
      System.out.println("done");
      System.exit(0);
   }
}
```

Location

As implied above, a physical location containing a number of books, is represented by a `Location` object, which has access to a database representing these books.

A `Location` object will keep these books in a Java Vector, which it will fill in when the object is instantiated. (We have deliberately not included this code below, as it is quite large while not being relevant to the discussion). Looking for a book is then accomplished by a simple linear search through the list, which will return a `BookList` containing the found books. Note that we do not stop when a book is found, as a location may contain multiple books with the same author or title.

As can be seen, code for the `Location` class is straightforward:

```
package library;
import java.util.*;
import java.io.*;

public class Location {
   private String name;
   private Vector books;

   public Location(String filename, String locname) {
      name = locname;
      books = new Vector();
      //Read stuff from file filename and put in books
      //CODE DELETED
   }

   public void addBook(Book abook) {
      books.addElement(abook);
   }
```

```
   public BookList findBook(Author auth) {
      BookList list = new BookList();
      for(int i=0; i<books.size();i++) {
         Book abook = (Book) books.elementAt(i);
            if (abook.hasAuthor(auth))
               list.addBook(abook);
      }
      return list;
   }

   public BookList findBook(Title titl) {
      BookList list = new BookList();
      for(int i=0; i<books.size();i++) {
         Book abook = (Book) books.elementAt(i);
            if (abook.hasTitle(titl))
               list.addBook(abook);
      }
      return list;
   }
}
```

BookList and Book

The `Booklist` class is a simple implementation of a list of books. Books can be added to the list, one list can be appended to another list, and the entire list can be printed out.

The code for `BookList` is given below:

```
package library;
import java.util.*;

public class BookList implements java.io.Serializable {
   private Vector thelist;

   public BookList() {
      thelist = new Vector();
   }

   public void addBook(Book b) {
      thelist.addElement(b);
```

```
   }

   public void appendTo(BookList other) {
      for(int i=0; i<thelist.size(); i++)
         other.addBook((Book)thelist.elementAt(i));
   }

   public String toString() {
      String result = "--\n";
      for(int i=0; i<thelist.size();i++)
         result += thelist.elementAt(i).toString()+"\n--\n";
      return result;
   }
}
```

Book

A book in the library is represented by the `Book` class. A `Book` contains a title and a Vector of authors. `Author` and `Title` are themselves classes, each a simple wrapper around a String. As these classes are extremely simple, we will not give their code here. The only remarkable about them is that they are serializable, which is required, as they are contained in a `Book` which also implements the `Serializable` interface.

The `Book` code is included below:

```
package library;
import java.util.*;

public class Book implements java.io.Serializable
{
   private Vector authors;
   private Title thetitle;

   public Book(Title tit, Author auth) {
      thetitle = tit;
      authors = new Vector();
      authors.addElement(auth);
   }

   public void addAuthor(Author auth) {
```

```
      authors.addElement(auth);
   }

   public Title getTitle() {
      return thetitle;
   }

   public boolean hasAuthor(Author auth) {
      for(int i=0; i<authors.size(); i++)
         if (auth.equals((Author)authors.elementAt(i)))
            return true;
      return false;
   }

   public boolean hasTitle(Title title) {
      return title.equals(thetitle);
   }

   public String toString() {
      String theauthors ="";
      for(int i=0; i<authors.size(); i++)
         theauthors +=
            ((Author)authors.elementAt(i)).getName() +" ";
      return "Book title:"
         + thetitle.getName()
         + "\n     auths:" + theauthors;
   }
}
```

Given the above code and descriptions we can compile and run the simulation. The resulting program will not be distributed, i.e. all will be run in one Java VM. To make the program a distributed system, we need to specify the distribution aspect, which will be done in the following subsection.

7.2.3 The Distribution Aspect

Defining the distribution aspect consists of two main parts: specifying locations of remote classes, using Dist and Serv, and specifying error-handlers, using Fix.

7.2.4 Dist and Serv

As said in the overview, we have two locations, each placed on their own server. This implies that we have to specify a Dist file for the `Location` class, which assures that the objects are instantiated in their respective server. A valid Dist file is the following:

```
class library.Location {
   serv = places[idx++];
   return serv;
}
extends {
   private static int idx=0;
   private static String[] places =
      {"tongariki:4242","akivi:4242"};
   private String serv;
}
```

As the `Booklist`, `Book`, `Title` and `Author` classes are serializable, and the `Library` class is never passed as a parameter to a method invocation, DistraC will not generate any other remote classes. The only remote class is the `Location` class, which can be instantiated remotely.

This implies that we must create a Serv file for the different servers used in the system. Note that there are not two, but three servers in the system; recall that one of the two locations contains two servers in a mirrored configuration. The two mirrored servers of that location are named `tongariki` and `vinapu`, the server of the second location is named `akivi`.

```
tongariki:4242 {
   library.Location;
}
vinapu:4242 {
   library.Location;
}
akivi:4242 {
   library.Location;
}
```

The system will now work in a distributed fashion. However there is no error-handling yet, this will be introduced when providing the Fix specifications.

Fix

As there is only one remote class in the system, the Fix file needs only provide a specification for this one class. However, the strategy for the exception handlers is twofold: For the location with the mirrored servers we need to switch over from one server to the other, and for the other location we need to implement an alternative error-handling strategy, for example: constructing and using the class locally.

Deciding which strategy to use is fairly simple; in the Serv file we deliberately saved the host name of the server the object should be on in the `serv` variable. We can use this saved name here to determine which strategy to use.

To implement switching over to another server, we specify a switch block in the boot catch, which according to the name contained in the `serv` variable, either switches over to a mirror server, or creates the class locally.

Now all the other error-handlers need to do, is switch over to a new instance, which will be placed according to the strategy described above.

The Fix code is fairly simple, as can be seen below:

```
class library.Location
extends {
   private String fil, loc;
   private void doSwitch() {
      Location newloc = new Location(fil, loc);
      distrac_switch_instance(newloc);
   }
}
new {
   fil = filename; loc = locname;
}
{
   catch(Exception ex)
      switch {
         if (serv.equals(places[1]))
            serv = null;
         else if (serv.equals(places[0]))
            serv = "vinapu:4242";
         else
            serv = places[0];
         return serv;
      }
```

```
}
(String filename, String locname){
   catch (java.rmi.RemoteException ex)
      switch {doSwitch();}
}
findBook(Title titl) {
   catch (java.rmi.RemoteException ex)
      switch {doSwitch();}
}
findBook(Author auth) {
   catch (java.rmi.RemoteException ex)
      switch {doSwitch();}
}
```

7.2.5 Discussion

In this experiment we have verified that, as in the first experiment, we can easily create a distributed system, thanks to the fact that the distribution aspect has been split out form the base functionality.

This split allowed us to concentrate fully first on implementing the base algorithm, without having to think about the distributed nature of the system. Second, we could handle the distribution completely separately from the base aspect, we did not need to make any modifications in the base aspect code.

However, this experiment revealed a second notable omission in DistraC, which is that a remote instantiation must either make an instance of that class remotely or make it locally. More concretely: for the location on the server `akivi` (which did not provide for a means of fault-tolerance), if an error ocurred, the only option we had was to place the `Location` object on the local host. Although we could pop up an error message to the user, the base algorithm required at all times that a `Location` object be contained at each index in its array of locations.

This implies that if we did not place this `Location` object locally, the program would have to be terminated, even though it would have been able to continue if it would somehow 'skip' this object.

A solution here is to let the remote instantiation also be able to return an instance of a subclass of this type. This subclass would then be designed specifically to rectify the problem that occurred. In our example, the `Location` subclass, would not search though a list of books, but solely return an empty `BookList` object.

The same functionality can also be added to the error-handlers of the method invocations, i.e. it should also be possible to switch to an instance of a subclass if errors occur at that point.

7.2.6 Conclusion

Our second experiment was a simulation of a library application, where a search for a book could be performed in a distributed fashion.

Again we have seen that the base functionality can be implemented without needing to consider the distributed nature of the final system. We showed how the distribution aspect can be easily implemented in a separate phase.

Also, we have noted a second deficiency in the error-handling capabilities of DistraC: we should be able to not switch to another instance of a remote class in case of errors, but we should also be able to switch to a subclass of that class, which would contain specific error-handling code which can not be placed in the current exception handlers, because it is needed at a later point in the execution of the program.

7.3 Conclusion

In this chapter we verified the claim that we can achieve a higher degree of distribution transparency through the use of AOP.

We developed two experiments: a messaging application, and a distributed library simulation. In both cases we could develop the base functionality without needing to concern ourselves with the distributed nature of the final application, and we could implement the distribution concern separately, in a later step.

This made the applications easy to develop, indeed leading to a higher degree of distribution transparency.

However, there were two negative points in the capability to handle errors: First it was shown that error-handlers may need access to objects which are not made available to them by the base aspect code at that point, and that therefore it should be made possible to include support code for the exception handler in a wide variety of locations. Second it was shown that in some cases it should be possible for an error-handler to switch the object to an object of another class, which contains specific error-handling code which will be used in the further execution of the program.

Chapter 8

Conclusions and Further Research

Time is a great teacher,
but unfortunately it kills all its pupils.
— Hector Berlioz

8.1 Summary

In a distributed system, a number of computers, connected to a common network, cooperate to achieve a common goal. An important issue when creating such systems, is how the concern of distribution is treated.

Ideally, the distribution should be transparent, i.e. when programming the core functionality of the application, the programmer should not have to concern herself with secondary requirements generated by the distributed nature of the system. It should be possible to treat the concern of distribution in a separate phase.

Sadly, current packages which facilitate the building of distributed systems do not provide this distribution transparency. A recurring symptom is the inability to reason separately about errors generated by the distributed nature of the application.

In this thesis we created a package which aids in the process of building a distributed systems and provides a much higher degree of distribution transparency. This high degree of transparency is achieved because it is now possible to implement the distribution concern completely separately from

the base functionality.

We first introduced a number of concepts specific to the field of distributed computing, such as distribution transparency. A widely used paradigm to achieve a certain degree of distribution transparency is the remote method call. We gave an overview of how a remote method call works, why it is fundamentally different from a normal method call, and what is required for a distribution package to be able to provide this feature.

To determine the degree of distribution transparency provided by current packages, we studied a number of packages providing support for building distributed systems in Java. In this study we found that the package which had the highest ease of use, and provided the highest degree of transparency, did this at a significant cost. Specifically: it was impossible to specify any error-handling code for exceptions thrown by the remote procedure call process.

We conclude that this impossibility to achieve distribution transparency, while keeping the control over such important parts as error-handling, is caused by the impossibility to decompose distribution using the same decomposition mechanism as is used for the base functionality.

Therefore, we must use a decomposition mechanism which does allow the programmer to specify the distribution concern using a different decomposition mechanism as the one used for the base functionality. One technique which allows this is Aspect-Oriented Programming. Using AOP, these special concerns, termed aspects, are specified separately, which implies a full separation of concerns. Having this separation of concerns will guarantee a higher degree of distribution transparency because of the fact that the distribution aspect is specified completely separate from the base functionality.

We used Aspect-Oriented Programming to achieve a high degree of distribution transparency in our thesis. We defined three special-purpose aspect languages: Dist, Serv and Fix. These languages are used, respectively, to define what classes are remote, i.e. possibly reside on a different computer, and on which computer they are placed, what computers contain instances of which remote classes, and what are the exception handlers for the remote method invocations used to access these remote classes.

After defining these languages, we introduced DistraC, the tool which combines these specifications with the base functionality, written in Java, to form executable code. More specifically, we described how input code is transformed, according to the specifications in the aspect languages.

To validate the claim that through Aspect-Oriented Programming we achieved a higher degree of distribution transparency, we performed a number of experiments, of which we discussed two: a messaging application,

and a simulation of a distributed library. In these experiments we verified that it is now indeed possible to specify the distribution concern separately, including the distribution-specific error-handling.

Therefore we feel that we can state that we have obtained a significantly higher degree of distribution transparency, through the use of Aspect-Oriented Programming.

8.2 Further Research

An important area of research within Aspect-Oriented Programming is how to construct weavers which integrate a large number of aspects into executable code. We feel it should be quite straightforward to integrate replication and concurrency control aspects into DistraC, further increasing the distribution transparency by also including replication and concurrency transparency.

The justification for this statement is threefold:

1. In our previous work [3, 4, 5], dealing with replication transparency, we transformed each variable reference to a remote method call to a replication server, which handled the specifics relevant to the replication algorithm. This same transformation can be applied to the intermediate Java code generated by DistraC. It is clear that this will not interfere with the already existing code, as our previous work had a high degree of replication transparency, and there is no interference between the replication aspect and the distribution aspect.

   Concretely, adding replication transparency only implies that some extra remote method invocations, to some extra classes, with their own, specific error-handlers, are added to the code.

2. We previously studied AspectJ, an Aspect-Oriented Programming extension to Java, which includes the concurrency aspect. In this study, we found that the actions of the weaver basically consisted of adding variables to the class, code which changed the values of these variables and guard code at the beginning of methods, and code changing the values of the variables at the end of the methods. Now, instead of adding this code to a local class, this code could be added to the remote part of a remote class, ensuring that the concurrency control strategy for that class is applied on that server. This is quite straightforward, we could, for example, add this code to the class before it is

transformed into a remote class, which implies that it will be included in the remote part of the class after the transformation.

3. We do not see a need at this time for the variables of the concurrency aspect code to be replicated, but it is of course possible that the concurrency aspect code will need access to variables of the base functionality. However, as the variables of the base aspect, if so specified, will be replicated totally transparently by the replication aspect there is no interference between the concurrency and replication aspect.

As for our AOP system, some extra work can be performed on the aspect languages and to DistraC, most importantly the error-handling capabilities which we found missing in 7.1.4 and 7.2.6 should be added to the system. Also, some of the features we did not implement due to lack of time, such as automatically generating wrapper classes for system classes which are not serializable and are passed as parameters to remote method calls, can be added.

Having successfully tackled what is defined by many the defining problem of distribution transparency [26, 7], we can add transparency features to our system, so full distribution transparency can be achieved. Note that some other distribution packages, such as JavaParty [18], already include a large number of these features, indicating that it should not be a major problem to achieve full distribution transparency.

Specifically, if we have a replication system in place, we could use its features for correctly forwarding direct variable accesses to the remote class, by using the variable reference transformation mechanism described above. Also, with replication of static variables provided, our main problem in correctly handling static method invocations will be eliminated, and we could easily provide correct forwarding of static method invocations. This would transform our system into a system providing distributed shared memory for the programs which are created using it.

8.3 Conclusions

In this dissertation we created an Aspect-Oriented Programming system which aids in the development of distributed systems by letting the distribution concern be specified separately from the base functionality.

Due to the fact that the distribution concern can be specified separately, we have obtained a significantly higher degree of distribution transparency. Using our system, a programmer, when working on the base functionality,

does not need to keep in mind a large number of the secondary requirements generated by the distributed nature of the program. This is because these secondary requirements, belonging to the distribution aspect, can be treated in a totally different phase of development.

Appendix A

The Aspect Languages

This appendix contains the grammars for the three aspect languages we defined: Dist, Fix and Serv.

Note that all aspect files can also contain Java-like comments, and all whitespaces (spaces, tabs, newlines) are treated as if they were one space.

A.1 Dist Grammar

```
Distfile:
   (Classdist)*

Classdist:
   "class" FullyQualdName [Location] [Extends]

FullyQualdName:
   <IDENTIFIER>

Location:
   <HOSTNAME>":"<PORTNUMBER> | Block

Extends:
   "extends" ClassBody

New:
   "new" Block
```

A.2 Fix Grammar

```
Fixfile:
   (Classfix)*

Classfix:
   "class" FullyQualdName [Extends][New][Bootstrap]
   [Creates][Invokes]

Bootstrap:
   BootCatchBlocks

BootCatchBlocks:
   "{"(BootCatchBlock)*"}"

BootCatchBlock:
   "catch" "(" <TYPENAME> <PARAMNAME> ")"
   (Block | BootBreak | BootSwitch)

BootBreak:
   "break" [Block]

BootSwitch:
   "switch" [Block]

Creates:
   (Create)+

Create:
   FormalParameters CatchBlocks

Invokes:
   (Invoke)+

Invoke:
   MethodDeclarator CatchBlocks

CatchBlocks:
   "{" (CatchBlock)* "}"
```

```
CatchBlock:
   "catch" "(" <TYPENAME> <PARAMNAME> ")"
   (Block | Break | Switch)

Break:
   "break" Block

Switch:
   "switch" Block

Extends:
   "extends" ClassBody

New:
   "new" Block
```

A.3 Serv Grammar

```
Servfile:
   (Server)*

Server:
   <HOSTNAME> ":" <PORTNUMBER>
   "{" BootClasses "}"

BootClasses:
   (FullyQualdName ";")*

FullyQualdName:
   <IDENTIFIER>
```

Bibliography

[1] Jonathan Aldrich, James Dooley, Scott Mandelsohn, and Adam Rifkin. Providing easier access to remote objects in client-server systems. http://www.ugcs.caltech.edu/~jedi.

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

[3] Johan Fabry. A framework for replication using Aspect-oriented Programming. Licentiaatsthesis, Vrije Universiteit Brussel, Faculteit Wetenschappen - Departement Informatica, 1998.

[4] Johan Fabry. Replication as an aspect. In *Ecoop '98 Workshop Reader*, number 1543 in LNCS. Springer-Verlag, 1998.

[5] Johan Fabry. Replication as an aspect - the naming problem. In *Ecoop '98 Workshop Reader*, number 1543 in LNCS. Springer-Verlag, 1998.

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

[7] Rachid Guerraoi and Mohamed E. Fayad. OO Distributed programming is not Distributed OO Programming. *Communications of the ACM*, 42(4), April 1999.

[8] Per Brinch Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34(4), April 1999.

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

[10] Objectspace Inc. Objectspace: Products - Voyager. http://www.objectspace.com/products/voyager1.htm.

[11] Objectspace Inc. *Objectspace Voyager Core Technology 2.0 User Guide*. Included in Voyager Core Technology software package.

[12] Sun Microsystems Inc. Enterprise JavaBeans home page. http://java.sun.com/products/ejb.

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

[14] Gregor Kiczales et al. AspectJ home page. http://www.parc.xerox.com/spl/projects/aop/aspectj/.

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

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

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

[18] Michael Philippsen and Matthias Zenger. JavaParty — transparent remote objects in java. *Concurrency: Practice and Experience*, 9(11), 1997.

[19] Eric S. Raymond et al. The jargon file, version 4.0.0. http://www.wins.uva.nl/%7Emes/jargon/t/Top.html.

[20] SETI@home. SETI@home: Search for extraterrestrial intelligence at home. http://setiathome.ssl.berkeley.edu.

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

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

[23] Sun Microsystems, Inc. RMI documentation. http://java.sun.com/products/jdk/1.1/docs/guide/rmi.

[24] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of International Conference on Software Engineering (ICSE'99)*, 1999.

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

[26] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing, November 1994.