Vrije **U**niversiteit **B**russel
Faculteit Wetenschappen - Departement Informatica
Academiejaar 1999 - 2000

Combining Components using Control Flow Components

Demaecker Philippe

*Proefschrift ingediend met het oog
op het behalen van de graad van
Licentiaat in de Informatica*

Promotor: Prof. Dr. Theo D'Hondt

Abstract

The implementation of distributed or parallel programs has always had a *dirty* edge: communication with other components, possibly located on remote machines or other processors, is mostly *hard* coded into the program source. Programs written in this way are often difficult to understand because they include calls to these low level procedures amidst cleaner code.

This thesis introduces several ideas that try to make the writing of distributed applications easier. We analyze some distributed programs and try to make them more flexible. We then discuss how the meta level of programming languages could be extended to increase reusability of distributed programs. Afterwards, we introduce components that make communication between unchangeable parts of programs possible.

Finally, we demonstrate the use of these new components with examples. A conclusion is then made based on the acquired experiences.

Acknowledgement

I would like to thank **Prof. Theo D'Hondt**, for allowing me to do my thesis at his lab. Secondly, my greatest gratitude goes out to **Werner Van Belle**, without whom this thesis would probably never have been in the first place. His practical aid and many deep discussions on the subject were invaluable; his numerous suggestions after proof reading drafts certainly improved the quality of my work. Many thanks go out to **Johan Fabry** as well, for correcting my not quite perfect English.

Many feelings of love go out to my girlfriend **Ilse** which pulled me over the finish.

I would also want to thank all the people of the PROG lab, for providing me pointers to interesting lecture.

Contents

List of Figures

Chapter 1

Introduction

Distributed Systems, almost everyone has heard about them. Many use them without knowing it. The popularization of the Internet resulted in operating systems that integrated distribution transparently into our desktop environments: the world was brought onto our screens. But what exactly *is* a distributed system? Coul[1] defines a distributed system as:

> *a collection of autonomous computers linked by a network, with software designed to produce an integrated computing facility.*

Because a distributed system contains a network, the interconnected computers have the ability to send information to each other. More importantly, by using the right software and protocols, these machines can *communicate* with each other. If they can communicate, they can also *cooperate*. This means that programs, located on different computers, can be implemented, such that those programs can operate with each other. A global problem can then usually be solved, by dividing the needed workload amongst several processors. This results in a more efficient and faster computation.

Another issue concerning distribution concerns the fact that data can be remotely accessed. This possibility resulted in the development of dedicated software that made remote access and interchange of data intuitively easy. We all know the results: internet browsers, mailing systems, internet games and so on. All those applications use the same properties of distributed systems: a server offers certain services, which can be used by several clients.

In this thesis we are not concerned with the type of applications that can be implemented. The goal here is take a look at the lowest levels of those applications: the programming and the meta level. Some problems that arise here are addressed and several solutions are proposed.

1.1 Problem setting

The use of multiple processors in one system resulted in the creation of specially adapted programming languages. In the parallel processing area, we can mention languages such as occam[2] and CSP[3]. At that time, in the non-parallel field, no really new languages were invented, rather, existing programming languages were adapted so that it would be possible to handle distribution, more or less separately. For the most part, the need for facilities, that made distribution possible, was met by providing additional primitives or libraries. Although these extensions allowed to address a subset of distributed problems, they lacked the level of freedom that currently existing languages posses.

Let us illustrate one of the shortcomings (at the distribution level) of the C programming language: C is a single-threaded programming language (we'll see more about threading later on). Although some facilities are provided by libraries, most of the work that handle communication has to be implemented by the programmer himself. For example, if we want to implement some kind of distributed program, we have to provide sockets and set up communication manually and we have to describe the port used and type of desired communication. All this additional work makes the source code rather ugly to read and more importantly, drastically slows down the development of our software.

This problem (and others, not mentioned here) arises from the fact that C was never really meant for the development of distributed applications. It was merely *extended* that some distributed features could be supported. Another example of this can be pointed out: C is a sequence based programming language. Such languages can restrict the potential power of distributed systems. A programmer has to take into account that calls to remote programs can be delayed. These delays are often not predictable, and thus, it can not be predicted how much time a sequence based program will have to wait for the result of such a call.

A step beyond C (and for that matter, C++) is Java, which was developed by *Sun microsystems*. Java was primarily implemented such that programs written in this language could be used on any possible platform that could run the Java Virtual Machine. Aside of this feature, a significantly internet-oriented background was always kept in mind during development. As a result Java[1] offers more support for distribution. These additional features facilitate the development of distributed programs, but, as will be shown in this thesis, there are still a few improvements to be done.

The general goal of this thesis results from all the problems mentioned above: how can we make the writing and maintenance of distributed applications easier? Can we reuse certain components in other programs? What

[1]more information on Java can be found at http://java.sun.com

types of applications can be simplified? We try to find answers to these questions in this study.

1.2 Goal

The approach we wanted to use would try to solve most of these problems all at once. We try to *extract the control flow* out of components which make up a program. A program consists of variables and constants (functions for example), and solves a specific task by calling these functions. The order functions are called determine the *control flow* of that program.

In distributed systems, the control can move to a process which is located on a remote computer, e.g. when a call is made to a remote program. Because the reference to this computer is often hard-coded in the program, *the control flow is fixed.*

We wanted to study if the control flow could be described by an external entity: references would be determined by this entity, instead of being hard-coded in the program. In addition, we wanted to reuse the control flow, which the entity described, with other components. We elaborate on these two problems.

Extraction of Control Flow

By extraction of the control flow of a component, we mean that the declaration of remote references is moved outside that component. In practice, this means that targets of remote procedure calls are not hard-coded in the component. Rather, an external entity tells a function where it's result should be sent. Sending this result to another part of the program can be performed by different functions. Therefore, we also want the external entity to determine the function that should be used to transport this data.

This separation results in components which aren't restricted to the program itself: we are able to *reuse* these components in other applications, if needed. In addition, maintenance and reusability of the components is improved. Of course, we don't want to remove the *entire* control flow from these programs. Calling a function or sending a message to an object automatically gives the control to that function or object. To extract the entire control flow we would have to empty the object of all its methods. This would result in inert objects, i.e. objects with no behaviour.

The entity which determines *where* data must be send and *how* it must be sent, totally determines the control flow between remote parts of a program. Therefore, we name this entity a *control flow component* (or CFC's) from now on.

Reuse of the control flow

We wanted to reuse CFC's in other programs: a CFC would take several remote components as arguments and link them together into a working program. The components which a CFC combines are reused: they are part of already existing programs. Because these components originate from different programs, they are not necessarily compatible with each other. We could fix this incompatibility by translating the functions of the components. This translation could then be kept in the CFC.

It should be clear that such translations can't be reused in other applications. This is a drawback, since we wanted to reuse the CFC's. In fact, this problem is so complicated that it became the primary research goal in this thesis. Because of this complexity, our original goal (the extraction of control flow) was not achieved.

To summarize, let us restate our main goal:

> *to develop* reusable *glue components, which can combine static components, in distributed programs.*

1.3 Method: general approach

In this part the general approach to and progress of our research is explained. Recall that we wanted to extract part of the control flow from distributed applications. The extracted code would be poured into a new component that could be reused.

To try to achieve this goal we started to study some distributed programs. This was necessary to analyze the problems that are inherently associated to distributed programming. Several conclusions could be drawn from these analyzes and a foundation could be made for further research. This research addressed the reusability of the CFC's and the stripped components.

The analyzed programs were written in an experimental programming environment, which was implemented at the lab. As a result improvements that encouraged or facilitated distributed programming could be easily added to the environment.

The following approach for control flow extraction was used

Exploration:

1. Conceptually separate the different components of the program

2. For each component, analyze the source to detect the code that make the several components cooperate with each other.

Amelioration: (reengeneering)

1. Extract the control flow: move responsibility of setting up references between components to an external entity.

2. Research other applications that could use the same CFC's. Look for applications that might share a communication behaviour that is similar to the previously addressed programs.

At this point we started to research possibilities, to make the reuse of the CFC's possible.

1.4 How to read this thesis

In the first part of this thesis we take a closer look at the simpler examples. We first start with some kind of lexicon that explains some terminology that is commonly used in distribution. This is necessary to clarify some problems we might encounter in the programs we study. After that we discuss the chosen programming environment and argue why we chose it above other, maybe similar environments.

In the second part of the thesis, we analyze distributed programs and their disadvantages, restrictions or flaws. We then present new implementations of these programs and discuss why the resulting programs are still too restricted. We discuss how the meta level of our language should be adapted to make distributed programming more easy and flexible.

Afterwards, we show how the changes of the meta level are used to create glue components. We also illustrate these components with examples. This report is then closed by a final conclusion where some possible future work is also proposed.

To make reading easier, only the essential parts of discussed source code will be included. In addition, small pieces of code will be formatted in sans serif. Big chunks of code will be included in figures.

Part I

Preliminaries

Chapter 2

Properties of Distributed Systems

2.1 Introduction

In this chapter we give an explanatory introduction of distributed systems. First the term "distributed system" is explained and the importance of such systems is made clear by discussing a complex application. Terms that are often used in the distribution field are discussed. The terms are classified in 3 sections: *communication, processes and paradigms.* In the communication section we show by what means communication between machines or components is set up. In the process part we take a look at the impact of distribution on processes. Finally, we discuss what type of paradigms are known in our area of interest.

2.2 What is a Distributed System?

To give an overview of what a distributed system is and what the main properties are, the excellent book from Coul[1] will be followed. Let's start with a definition of what a distributed system is:

> *a collection of autonomous computers linked by a network, with software designed to produce an integrated computing facility.*

Distributed system software (or a distributed operating system) enables computers to coordinate their activities and to share the resources of the system: hardware, software and data. Users of a well-designed distributed system should perceive a single integrated computing facility even though it may be implemented on many computers in different locations. Let's illustrate this with some everyday life examples. Most computer scientist are familiar with Windows NT or Linux. These are instances of the so-called distributed system software. Like a conventional operating system, a distributed operating system is a collection of software components that simplifies the task of programming and supports a wide possible range of applications. Unlike conventional operating systems, distributed operating systems are modular and extensible. New components may be added to such a system in response to new application needs. In addition, those systems allow us to share printer and scanners (the hardware), client-server applications such as web browsers (the software) and databases (the data) all at once, in a transparent manner: it behaves as if, for example, a printer is directly connected to the computer in front of us.

A More Complex Example

To illustrate the power of such systems we will give a more complex application of distribution, depicted in figure 2.1: a commercial application. Many commercial data processing and information systems involve data communication. Consider, as examples, the systems used by airlines for seat reservations and ticketing, the networks operated by banks to operate cash dispensers and systems operated by supermarkets to coordinate stock control, delivery and sales check-out.

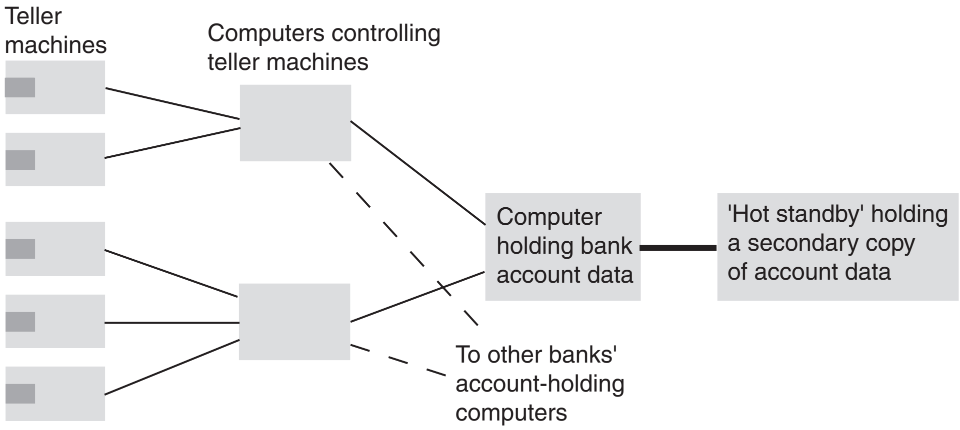

Figure 2.1: An automatic teller machine network

Complex distributed applications need various requirements: a high level of reliability, security against external interference and privacy of the information that the systems holds. These distributed systems must also be able to provide concurrent access to databases by many users, guaranteed response times and service access points that are geographically widely distributed.

Although, in this study, we are not really concerned with real-time transactions or reliability, we do want to make a point about the fact that writing such complex applications can become a real hassle, without the help of dedicated programming environments.

2.3 Communication

In this section we will try to explain some commonly used terms in the field of distribution. We take a closer look at how programs interact; the primitives and constraints that are needed to make such interaction possible are reviewed.

2.3.1 How Components Communicate

The components of a distributed system are both logically and physically separate; they must communicate in order to interact. Distributed systems and applications are composed of separate software components that interact in order to perform some tasks. We shall assume that all of the components that require or provide access to resources in distributed systems are implemented as processes. In client-server systems (more about this later in this chapter) a client process must interact with a server process whenever it requires access to a resource that it does not directly control – hardware, software and data.

Communication between a pair of processes involves operations in the sending and receiving processes that together result in:

a. *the transfer of data* from the environment of the sending process to the environment of the receiving process;

b. in some communication operations, the *synchronization* of the receiving activity with the sending activity, so that the sending or receiving process is prevented from continuing until the other process makes an action that frees it.

a. Transfer of data isn't a trivial thing. First, you have to know to which machine you are talking to. Second, a connection channel has to be set up. When this is done, the transfer can start using a predetermined communication protocol. This protocol receives serialized data from some upper layer and sends it through the network. Serialization consists of transforming the code into data that can be sent through a network[1] The receiver then de-serializes the data, so that it can be used by the expecting program. Fortunately, most of the time this happens transparently.

b. is implicit in the operation of all programming primitives for communication. The basic programming constructs take the form of programming primitives *Send* and *Receive*. These two primitives deserve a section of their own.

[1] This is necessary because you can't just send a piece of code through a cable. Compare it with the our (physical) mailing system. You first have to put your letter in an envelope before putting it into the posting box.

2.3.2 Communication Primitives: Send and Receive

Single message passing can be supported by two message communication operations, *Send* and *Receive*, defined in terms of destinations and messages. In order for one process to communicate with another, it sends a message to the other process. This activity involves the communication of data from the sending process to the receiving process and may involve the *synchronization* of the two processes.

Together, these primitives perform *message-passing* actions between a pair of processes. Each message-passing action involves the transmission by the sending process of a set of data values (a message) through a specified communication mechanism (a channel or port) and the acceptance by the receiving process of the message. The mechanism may be *synchronous* or blocking, or it may be *asynchronous* or non-blocking. Let's clarify these two new terms.

2.3.3 Synchronous and Asynchronous Communication

A queue is associated with each message destination. Sending processes add messages to queues and receiving processes remove messages from queues. Communication between the sending and receiving processes may either be synchronous or asynchronous. In the *synchronous* form of communication, the sending and receiving processes synchronize at every message. In this case, both the *Send* and *Receive* are *blocking* operations. Whenever a *Send* is issued, the sending process is blocked until the corresponding *Receive* is issued. Whenever a *Receive* is issued, the process blocks until a message arrives.

In the *asynchronous* form of communication, the use of the *Send* operation is *non-blocking* in that the sending process is allowed to proceed as soon as the message has been copied to a local buffer and the transmission of the message proceeds in parallel with the sending process.

In the asynchronous form of communication the *Receive* operation can have blocking and non-blocking variants. In the non-blocking variant, the receiving process proceeds with its program after issuing a *Receive* operation which provides a buffer to be filled in the background, but it must receive notification that its buffer has been filled, by polling or interrupts. The following table summarizes the variations of synchronous and asynchronous communication:

Communication Type	Blocking Send	Blocking Receive
Synchronous	Yes	Yes
Asynchronous	No	Yes
Asynchronous	No	No

In a system environment that supports multiple threads in a single process, the blocking *Receive* has few disadvantages. It can be issued by one thread while other threads in the process remain active. The simplicity of synchronizing the receiving threads with the incoming message is also a substantial advantage. Non-blocking communication appears to be more efficient, but this is not entirely true: it involves extra complexity in the receiving process associated with the need to acquire the incoming message out of its flow of control. In addition, because most of the operating system's idle time is used by the polling loop, the system performance is degraded.

The blocking form of *Receive* is suitable for use in servers that have nothing else to do when waiting for request messages from its clients. But in some programs, it is not appropriate that a process that has started a *Receive* operation should wait indefinitely, for example, in situations where the potential sending process has crashed or the expected message has been lost. In these cases, a *timeout* is useful. A *timeout* specifies an interval of time after which a certain operation will give up its action. We can now associate a timeout with *Receive* operations on certain ports, for example when a new port is created.

Distributed systems can be designed entirely in terms of message-passing, but there are certain patterns of communication that occur so frequently and are so useful that they can be regarded as an essential part of the support for the design and construction of a distributed system. In the last section we introduce the two patterns of communication most commonly used in the construction of distributed systems. These are the *client-server* communication model or communication between pairs of processes and the *group-multicast* communication model for communication between groups of cooperating processes.

2.4 Processes

The implications of distribution on processes are discussed in this section. We discuss some new, major, programming possibilities that distribution offers: concurrency, threading and migration.

2.4.1 Concurrency

When several processes reside in a single computer we say that they are executed concurrently[2]. If the computer is equipped with only a single central processor, this is achieved by interleaving the execution of portions of each process. If the computer has N processors, then up to N processes

[2]Since each process only exists for the duration of a program execution, coexistence implies concurrency of execution.

can be executed simultaneously (that is, in parallel) achieving an N-fold improvement in computational throughput.

In distributed systems there are many computers, each with one or more central processors. If there are M computers in a distributed system with one central processor each, then up to M processes can run in parallel, provided that the processes are located on different computers.

In a distributed system that is based on resource-sharing, opportunities for parallel execution can occur for two reasons:

1. Many users simultaneously invoke commands or interact with application programs.

2. Many server processes run concurrently, each responding to a different request from client processes.

Case 1 arises when one or more application processes are running on behalf of each active user. On most distributed system architectures, application processes run in a user's workstation and they do not compete for processing resources with the application processes of other users. If the workstation only has a single processor and there is more than one application running on the workstation, these are executed in an interleaved manner. Workstations with multiple processors enable users to perform several computations in parallel or to execute applications programs that exploit more than one processor.

Case 2 arises when there are multiple server processes for each type of resource. These resources normally run in additional computers, enabling each server process to proceed in parallel with other servers and with the processes running on workstations. Multi-processor computers are particularly effective for the implementation of heavily used servers, since an N-processor server has the potential to handle up to N requests from clients without processing delay. Request for access to resources from a given server are queued at the server and processed in sequence, or several request may be processed concurrently by multiple instances of the resource manager process. When several resource-using programs access the same resource concurrently, the server must synchronize their actions to make sure that they do not create conflicts. The synchronization must be carefully planned to ensure that the benefits of concurrency are not lost.

To summarize, concurrency and parallel execution arise naturally in distributed systems from the separate activities of users, the independence of resources and location of server processes in separate computers. The separation of these activities enables processing to proceed in parallel in separate computers. Concurrent access and updates to shared resources must be synchronized.

2.4.2 Threading

Campione[4] cleverly remarks that all programmers should be familiar with writing sequential programs. We've all written a program that displays "Hello World!", or sorts a list of names, or computes a list of prime numbers. These are sequential programs: each has a beginning, an end, a sequence, and at any given time during the runtime of the program there is a single point of execution.

A thread is similar to the sequential programs described above: a single thread also has a beginning, an end, a sequence, and at any given time during the runtime of the thread there is a single point of execution. However, a thread itself is not a program–it cannot run on its own–but runs within a program. The Figure 2.4.2 shows this relationship.

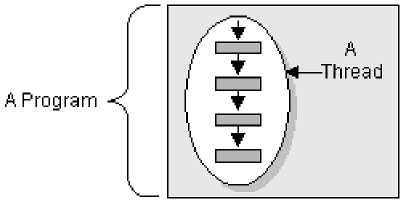

Figure 2.2: A thread within a process

A definition for a thread could be:

a thread is a single sequential flow of control within a program.

There is nothing new in the concept of a single thread. The real focus surrounding threads is not about a single sequential thread. Rather, it is about the use of multiple threads in a single program, all running at the same time and performing different tasks. In this way, performance is gained even in single processor systems. Figure 2.4.2 illustrates a multi-threaded program:

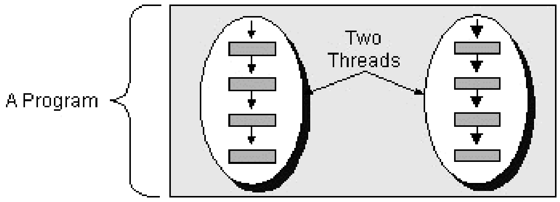

Figure 2.3: Multiple threads in one program

The HotJava browser is an example of a multi-threaded application; within the HotJava browser you can scroll a page while it's downloading an applet or image, play animation and sound concurrently, print a page in the background while you download a new page, or watch three sorting algorithms race to the finish.

Some texts use either the name lightweight process instead of thread. A thread is similar to a real process in that a thread and a running program are both a single sequential flow of control. However, a thread can be considered a lightweight process because it runs within the context of a full-blown program and takes advantage of the resources allocated for that program and the program's environment.

As in a sequential flow of control, a thread must carve out some of its own resources within a running program (it must have its own execution stack and program counter for example). The code running within the thread only works within that context. Thus, other texts use execution context as a synonym for thread.

The following example from Moreira [5], should clarify the concurrent nature of threads.

```
class TwoThreadsTest {
    public static void main(String[] args){
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start(); } }
class SimpleThread extends Thread {
    public SimpleThread(String str, String stringName) {
        super(str); }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + stringName);
            try {
                sleep((int)(Math.random() * 1000)); }
            catch (InterruptedException e) { }
        System.out.println("DONE! " + stringName); }} }
```

Outputs:

```
0 Jamaica
0 Fiji
1 Fiji
1 Jamaica
2 Jamaica
2 Fiji
...
8 Fiji
9 Fiji
8 Jamaica
DONE! Fiji
9 Jamaica
```

2.4.3 Migration

The concept of moving objects from one machine to another during runtime is called object migration. This can happen for a lot of reasons. For example, imagine that we have two objects that cooperate together. One of these objects could migrate closer to the other one, such that the communication between those objects is faster. Another reason could be migration to a more performant machine, such that computation would be faster. An object might want to migrate to a machine that offers services that the object requires. Another use could be fault-tolerance: an object is copied and migrated to another computer.

Migration can be implemented in several ways. Two approaches will be briefly discussed here:

1. **Weak Migration: The Looping Model**

   With this approach, the object has to keep track of it's own changes. For example, when the object has to migrate, it must save it's current state somewhere, migrate, and restart from where it was interrupted. It is from this stop $\rightarrow$ migrate $\rightarrow$ start loop that this model gets its name.

   To make an interruption possible, the run-time stack and the execution pointer have to be saved. However, most programming languages don't support run-time stack access. Because of this an object can only migrate when it has finished it's computation. It can then save it's result, move, and continue to work. Because this approach doesn't provide "immediate" migration, it is called *weak*.

2. **Strong Migration: Telescripting** Telescripting gets its name a scripting language that was extended with migration features. In this case the programming environment provides facilities for migration. More precisely, the saving of the stack an execution state is implemented in the language as a feature, and thus handled automatically. This allows the possibility to save an object's state during it's computation. When the save is finished, the object moves to another location, retrieves it's run-time stack, and continues it's computation from there on. Because this type of migration is issued immediately, it is called *strong migration*.

2.5 Paradigms

Distribution has been around for a while now, and some paradigms have become very popular. The well known client-server and group multicast are presented. The section is closed with the concept of callbacks.

2.5.1 Client-Server Communication

The client server communication model is oriented towards service provision. An exchange consists of:

1. transmission of a request from a client process to a server process;

2. execution of the request by the server;

3. transmission of a reply to the client.

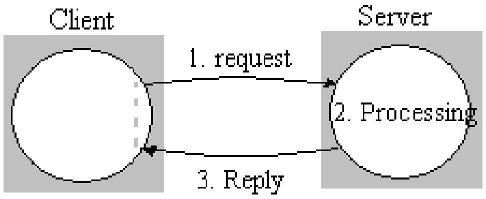

Figure 2.4: Client-server communication

This pattern of communication involves the transmission of two messages (in 1. and 3.) and a specific form of synchronization of the client and the server. The server process must become aware of the request message sent in step 1. as soon as it arrives, and the activity issuing the request in the client process must be suspended (*blocked*), causing it to wait after step 1. until the reply has been received in step 3.

The client-server pattern of communication can be implemented in terms of the basic message-passing operations, which were outlined above (*Send* and *Receive*). The client-server pattern is commonly presented at the language level as a *remote procedure calling* (or RPC) construct: This construct conceals the communication operations behind the familiar procedure-calling abstraction and is implemented in terms of a *request-reply protocol*, for which the underlying communication operations are optimized to the purpose at hand. Recently, these RPC's have been introduced in object oriented languages. In they context of OO, they are known as *Remote Object Invocations* (or RMI).

2.5.2 Group Multicast

In the group multicast pattern of communication, processes interact by a message passing, but in this case the target of a message is not a single process but a group of processes. To a single group *send* there corresponds a *receive* performed by each member of a process group, as illustrated in Figure

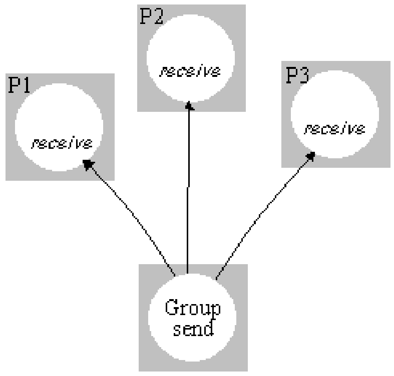

Figure 2.5: Multicasting to a process group

2.5. Sending a message to the members of a specified group of processes is known as *multicasting*.

There may or may not be hardware support for multicast communication in the underlying network. Without this support, a multicast message has to be sent sequentially by the communication system to the members of a process group. The group multicast is a structure concept for distributed system and is independent of the existence of a hardware support for multicasting.

2.5.3 Callbacks

In this final section of our chapter, we discuss a technique used in one of the lowest levels of an application: the source level. A work-around that is often used in distributed programming is explained here. This so-called work-around is known as a "callback".

Imagine that you have program pA, an a machine A, that calls a function from a program pB, that is located on another machine B. The only problem is that pB doesn't know where to return its results.

This problem can easily be solved by providing the address of the caller to the callee. The callee then knows where to return the result of the computation. A *call-back* extends this idea: in addition to directly returning the result of a computation, a the callee calls a function with the result as argument. The following example, that computes the determinant of a matrix, should clarify the general idea:

Machine A	Machine B
answer(b) ::display(b);	calcdet(matrix, target):: result = determinant; target.answer(result)
B.calcdet([[1,8],[5,6]], agentself())	

On machine A, a procedure answer(x)taking one parameter is defined. Here, the job of answer(x) is to display the passed argument, but it could do something else if needed. On B, a function calcdet(m, t) is defined: the computation is done here. In addition, we describe that the result of this function-call should be returned[3] to the caller of this function.

With these functions defined, B.calcdet(m, t) is issued: from machine A, calcdet(m, t) (which is located on B) is called. At this point, the control is at B, in procedure calcdet(m, t). The execution of calcdet(m, t) results in the calling of function answer(result) on machine A, all in a transparent manner. The following figure describes the control flow of this simple program:

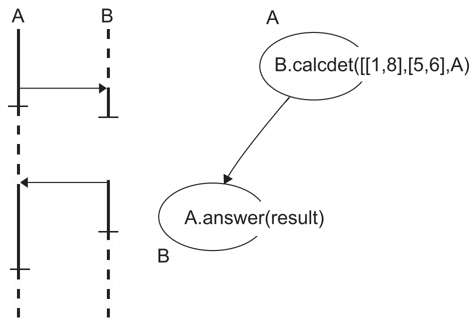

Figure 2.6: Control flow using a callback

2.6 Conclusion

The things that we discussed in this chapter form the building blocks of a distributed system. They are highly necessary when constructing such a system and as you could expect the Internet could not exist without them.

Those concepts have been implemented in most popular programming languages such as Java, C and C++. Especially the callbacks will play an important role in the remainder of this thesis. In the next chapter we will introduce CBorg, a programming language that offers an excellent support for distribution. Because of this, we have chosen CBorg to conduct our exploration of distribution.

[3]The dot notation is often used in object oriented languages. Here, its syntax is TargetObject.function.

Chapter 3

An Introduction to CBorg

3.1 Introduction

In this chapter an overview will be given of the programming language that was used during the study: CBorg. CBorg was originally designed for Mobile Multi-Agent Systems. These systems are introduced. CBorg's main features and workings are discussed, and some explanation is given how agents can be created using CBorg. We show how OOP related objects can be managed as well. Some distribution issues that were explained in the previous chapter are explained and mapped to the CBorg context. To close this chapter, a comparison between CBorg and other programming languages that support distribution is given.

3.2 CBorg: an Experimental Programming Environment

The CBorg (pronounced *see-borg*) programming language was originally designed to provide support for the implementation of Mobile Multi-Agent Systems. The reason for designing a new programming language was that existing languages didn't offer enough facilities, transparently for distribution, like migration. Programming languages like Java, which take care of e.g. serialization, were first considered but could not be used to migrate a process to another processor.

CBorg is based on an already existing language, namely Pico[6]. Pico is an educational language developed by Professor Theo D'Hondt at the Vrije Universiteit Brussel. It was designed to be as simple as possible, so that it could be taught to non IT students, and thus provides a very simple syntax. Pico tries to retain the essence of the Scheme[1] language, and with it, it's power. CBorg is an extension of Pico: new native functions that support mo-

[1] which is a LISP variant

bile multi-agent systems and therefore distributed programming were added. It is worthwhile to inform the reader that, since CBorg (and similarly, Pico) is an *experimental* environment, it made programming a though job: unsuspected bugs and crashes were observed at the most unexpected moments, which made the coding part of this thesis a rather frustrating job. Fortunately, the incredible flexibility of this program kept the motivation high.

3.3 Mobile Multi-Agent Systems or MMASs

Research about mobile multi-agent systems[7] is done at the Vrije Universiteit Brussel in Belgium. Van Belle and Verelst define the following things:

agent :

An agent can be described as a single threaded program with an in-queue and an out-queue, respectively used to send and receive messages. The agent can be referred to through it's "agent" name and has it's own code and data space. An agent contains a program which determines it's behaviour[2]. This behaviour can only be altered by changing messages to the agent.

mobile multi-agent :

This is a stand alone agent which is able to communicate with other agents and which can migrate to other places (processors).

mobile multi-agent system :

Such a system is the architecture (or operating system) which executes these mobile agents. This system delivers messages between agents. The sheduling of agents and the serializing of them, as well as transmitting the messages between them, is taken care of by this system.

The artificial intelligence community also uses the term "agent", but with a slightly different significance. In the AI field, an agent is a piece of software with some kind of implemented *intelligence*. It might be able to learn from experiments and adapt it's behaviour according to these experiments. These agents may act autonomously and might have the possibility to migrate.

A mobile multi-agent system can easily support AI agents, because an agent's intelligence is contained in the program, and migration is by definition possible in a *mobile* multi agent system.

We will now take a look at how concepts such as "agents" and "objects" are represented in CBorg.

[2]In this context an agent can be viewed as *autonomous*.

3.4 CBorg Concepts

In this section, we explain how CBorg creates agents *internally*. We then takes a look at how *objects* and *agents* can be created in CBorg.

3.4.1 Agents in CBorg

When an agent has to be created, a chunk of memory is allocated for that agent. This memory, which can be dynamically resized, is divided in two parts: the data space, which contains the agent's variables, and the source space, which contains the source that implements the agent. It is important to note that the data and source space can't be accessed by other agents.

Further on, an in-queue, which contains incoming messages, is associated with each agent. A stack, which is divided in data and code continuations, is also associated with the agent. This stack can be used by the single thread that is contained by the agent.

Because in CBorg, objects are part of agents, we show how objects can be created in this language. Afterwards we discuss two methods to create agents.

3.4.2 Objects in CBorg

Although CBorg was not specifically designed for object oriented programming, it can be used to implement objects[3] in a very simple manner. To illustrate this simplicity, consider the following example. The createpoint program is an abstraction of a point in a two-dimensional space and is therefore entirely defined by it's two coordinates: 'x' and 'y'. The following program implements this abstraction, with it's set of *get* and *set* functions. We link a name (createpoint) to an environment that contains the object's variables and the clone() primitive makes an object out of this chunk of data. (clone() is discussed more in detail after the example). As a result we have an object that we can manipulate by sending it messages to.

```
createpoint(x, y) :: {
      getx():: x;
      gety():: y;
      setx(nx):: x:=nx;
      sety(ny):: y:=ny;
      clone() }
:<closure createpoint>

a : createpoint(1,2)
:<dictionary>
```

[3]These objects encapsulate data, just like OO objects.

b.getx()
:1

b.setx(8)
:8

b.getx()
:8

a.getx()
:1

Cloning

As mentioned, the clone() primitive makes a copy of the dictionary contained in it's scope and returns it; it makes an *object* out of a chunk of data. Let's illustrate the this primitive with the following example:

```
makePoint(x,y):: {
    makeSquare(w):: {
        area()::w*w;
        clone()
    }
    area()::0;
    clone()
}
```

When we analyze the next diagram, we can see several environments: the root environment, which is the top rectangle, and the other environments, which are "attached" to the root environment.

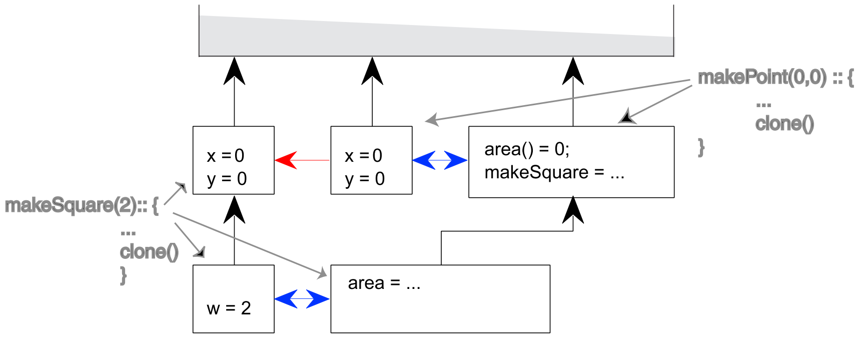

Figure 3.1: Clone() internals

When a point is created using makePoint(0, 0), the environment containing this object's code is copied. This environment contains two lists of associations: a declaration list (with constants area and makeSquare) and a definition list (with variables x and y), as can be seen in the picture. The cloned environment is attached under the root environment.

When makeSquare is called, a new dictionary is created (because it is also cloned) and attached beneath the makePoint environment. This environment is attached beneath makePoint, because we have a clone() inside another one[4]. Again, this new dictionary contains constants and variables. The new object can access the x and y by reference (like a shallow copy).

In contrast with clone(), there is also a primitive called agentclone(). The agentclone() function makes an *agent* of an object; the object is made active, and possesses it's *own* source and data space.

	clone()	agentclone(anObject, name)
Input	chunk of data	an object
Output	object	agent

3.4.3 The Creation of Agents

There are two ways to create agents in CBorg: a new *session* can be started or the native function agentclone can be used.

Starting a New Session

When CBorg is started, two windows are opened: the debug window, which displays information about CBorg communication, and a *session* window. In the session-window we can write and evaluate programs. A session in CBorg is in fact equivalent to an *agent*. Therefore, a session can be referred to, from other CBorg sessions. Every agent can be referred to by it's *name* and additional sessions or agents can be started at any time.

Let's show how a reference to another session is made and how information is sent. Assume that on our local machine "tecra", we have two active sessions: tecra/ses1 and tecra/ses2. In CBorg ":" is used for first time variables declarations.

tecra/ses1	tecr/ses2
refToSes2:remotedict(tecra/ses2)	
<dictionary>	
refToSes2.display("Hi!")	:Hi!
<void>	

In this program we first make a reference to the second session. Then we tell this session to run the display function. This example only implements a

[4] If MakeSquare's clone() had been declared outside makePoint, it's environment would hang under the root environment as well.

simple remote display, but we could call functions that we defined ourselves on tecra/ses2 as well.

Another way to create new agents is through the native function agentclone, which is presented now.

Using agentclone

Agents can created through agentclone(object, agentName). This native takes an *object* and a *name* as argument. The *object* will be transformed to an agent, which will be identified as *agentName*. Great care has to be taken when using this native, as the code can become inconsistent. Conside the following example.

Take a look at figure 3.2. In a CBorg session "tecra/ses1", we first declare a function hello(x), that displays: "Hello <*filled in parameter x*>". Second, we make an object Display() (using clone()) which contains a method show(x). Note that this method uses Hello(x), declared outside of the object's scope, to do its computation. This would work, but suppose that we make an agent of this object, moving all the data into this new agent. Since this agent has it's own source and data space, and since Hello(x) was not defined in the scope of Display(), Hello(x) cannot be reached from our new agent tecra/ses1/newdict!!

Now that we discussed how to make objects and agents in CBorg, we will take a look at the way CBorg handles distribution.

3.5 Distribution Issues in CBorg

In this section we address some issues we discussed in the previous chapter and take a look at how they are implemented in CBorg.

3.5.1 Communication in CBorg

Two types of communication were reviewed in the last chapter: synchronous and asynchronous communication. Both these are supported by CBorg[10] as we will now show.

3.5.2 Asynchronous Communication

The primitives that are defined in CBorg for asynchronous communication are *send(targetAgent, data)* and *recv(targetAgent, data)*.

- **send**

  When such a send operation is issued, the data is transparently serialized and sent to the targetSession. The program that made this function call doesn't wait for a response from targetSession; it just

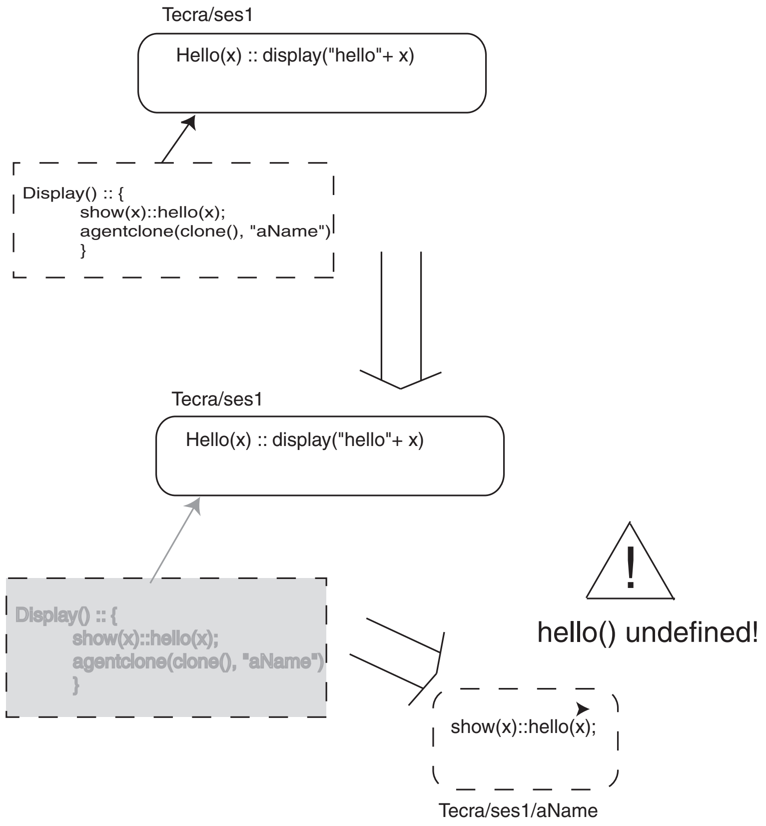

Figure 3.2: The agentclone() problem

continues it's computation. A send returns <void> to the calling session as a response. On the side of the receiver, the sent data is kept in a queue.

- **recv**

  When a recv primitive is reached during a computation, the data is fetched from the receive-queue, such that the needed data can be used in the current (or other) computation.

- **function calls**

  Communication with another agent can be achieved through function calls. Assume that agent A contains a function foo(). This function can not only be called from within the containing agent, but from another agent as well.

  The syntax to do this is, is as follows:

  We first create a reference to that agent ...

  refToAgent:remotedict(A)

and we then simply call that function.

refToAgent.foo()

Note that such a function call does not return a result value to the calling agent. If we want the result to be returned, we have to use call-backs, which we discussed in the previous chapter.

3.5.3 Synchronous Communication

The primitive that is used for synchronous communication in CBorg is called *sync(Agent)*, and synchronizes two or more agents. This primitive can be used to simulate synchronous sends and receives. They are defined as *ssend(to, data)* and *ssend(from, data)*.

The following table gives an example of use of a sync.

Agent 1	Agent2
sync(agent2)	...
... *Waiting for sync* ...	...
...	sync(agent1)
Execution Continues	*Execution Continues*

- **sync**

  When an agent wants to synchronize with another agent, it issues a sync. The sync command takes the other agent as parameter. A second optional argument can passed: this argument is a pattern, that puts restrictions on what agents may synchronize. This way we can choose our synchronee by other means than it's name. As stated in the previous chapter, when a sync is called in an agent, this agent pauses until the other agent responds to this function by calling a corresponding sync.

- **ssend**

  The idea of a synchronous send is the following: an agent *waits* for some kind of response from the agent that received data, that was sent using a ssync. As long as the receiving agent is not ready to synchronize, the sending agent has to wait before it can resume it's computation. Therefore, synchronous sends and receives are also known as *blocking*, as was described in the previous chapter.

  An synchronous send can be simulated in CBorg using the following self explanatory program:

```
ssend(to, message): {
    send(to, message);
    sync(to, message)
        'message is used as pattern for the sync
    }
```

- **srecv**

  The srecv primitive is the complementary answer to ssend. Whenever an agent expects data from another one (using srecv), it synchronizes with that agent. If this second agent can synchronize, it means the needed data has been sent by that agent (because a ssend first sends data and then synchronizes). Thus, if the synchronization succeeded, the data is available and can be fetched from the message queue by the requesting agent. After the synchronization, the computation can then be resumed by both agents.

  The srecv is implemented in CBorg as follows:

```
srecv(from, pattern): {
    sync(from, pattern)
    recv(from, pattern);
    }
```

3.5.4 Threading

In CBorg, an agent contains only one thread. Because of this, only the concurrency between different agents has to be taken care of: if an agent would contain more than one thread, CBorg would first have to synchronize the threads inside of an agent and second, synchronize the threads between agents. To be able to synchronize these inter-agent threads, additional primitives would have to be implemented. This is an unwanted difficulty and not really necessary: if you need another thread, just start another agent.

Each one of these agents different processor. This is a consequence of the fact that agents have their own source and data space (see the definition of an agent): it is an easy matter to move al this data to another processor or machine. This fact makes it possible to migrate agents as well.

3.5.5 CBorg's Naming Procedure

We will now see how CBorg determines a name for a newly created agent (when not using agentclone), so that they can be referenced to, by other agents.

When CBorg initializes, it reads in a file that contains the name of your computer. This name is taken as "root-name". The name of the current

CBorg session is than concatenated to this root name. In addition, the name of the environment with which the agent is associated is invisibly added after these two names.

CBorg organizes several agent-systems in an hierarchical manner, forming a tree. One root agent-system server is started, and every other agent-system has a link to a parent. This parent is declared in CBorg's configuration file, named "CBorg.*cpu-id*.ini".

Let's illustrate this naming with an example. We start the first mobile multi-agent system called *tecra*. We then start two other mobile multi-agent systems, *nema* and *kasbah* on other machines. Since the configuration files of these two agent link them to *tecra*, (using the forward(target-ip) setting), these two MM-AS's become children of *tecra*.

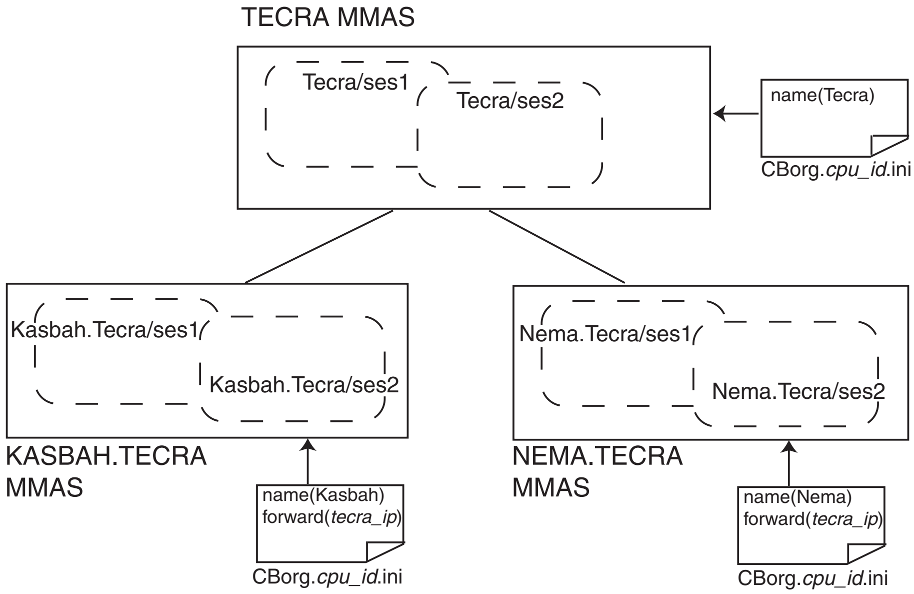

Figure 3.3: A CBorg naming hierarchy

It is important to note that when an agent migrates, it's name doesn't change. When an agent moves to another location, it has to move through the hierarchy tree, until the target location is reached. Each node (which is in fact a MMAS), that is passed by the moving agent, keeps track of the next node where the agent moved to. When a message is sent to a migrated agent, that message follows the path the agent followed, arriving to the previously moved agent.

Example: assume that agent *Kasbah.Tecra/ses1* migrated to the *Nema* MMAS. To get there, it had to pass through it's parent *Tecra* first.

Kasbah.Tecra recorded that the agent moved to *Tecra*, *Tecra* recorded the the agent then moved to *Nema.Tecra*. Now, when a message has to be sent to *Kasbah.Tecra/ses1*, the tree is traversed, following this previously recorded path, delivering the message to the agent.

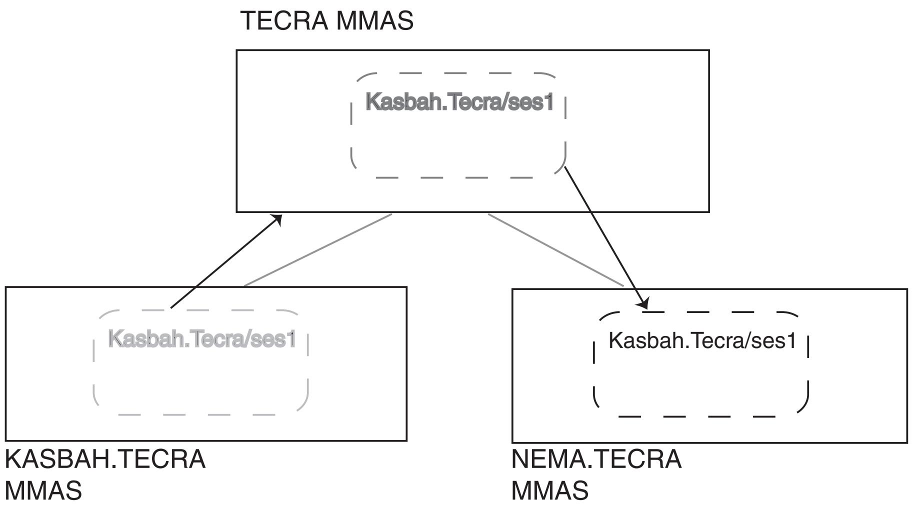

Figure 3.4: An agent is traced when it migrates

3.5.6 Migration

CBorg supports strong migration[11]: it's interpreter is capable of serializing the data store and has the ability to reify and serialize the computational state. These two features can easily be used to implement strong migration. Even if an agent process is in the midst of being evaluated, the process is simply interrupted, the state of the entire computation serialized and sent to another location while the process is removed from interpreter control. At the receiving end the agent is deserialized and a process gets started which uses the freshly deserialized computational state.

The above steps are taken at a moment at which the stack is consistent. No continuation thunk must be executing, no unsaved (in the data store) global variables should be present.

The example below illustrates how migration can be used. We see that we can call the agentmove native function at any time. When the agent arrives at its new location it will print "after moving" without the need for restarting it at its main entry point.

```
Calcul(pars):: {
   <some calculation>;
   display("before moving");
   agentmove(remotedict("otherplace.tecra"));
   display("after moving")
   }
```

3.6 CBorg vs Other Distributed Environments

As you can see, CBorg allows us to create agents in a very straightforward manner. Communication can, in general, be easily implemented, more easily than in other programming languages. A comparison with other distributed programming languages will be made: we will see how much code we have to write in these languages, just to link components together, so that messages can be sent to these components.

3.6.1 Corba

As the OMG[8] simply states, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB).

The ORB is the middleware that establishes the client-server relationships between objects. Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its method, and return the results. The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object's interface. In so doing, the ORB provides inter-operability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

With an ORB, the protocol is defined through the application interfaces via a single implementation language-independent specification, the IDL. ORBs provide flexibility: they let programmers choose the most appropriate operating system, execution environment and even programming language to use for each component of a system under construction. More importantly, they allow the integration of existing components. In an CORBA-based solution, developers model the legacy component using the same IDL they use for creating new objects, then write "wrapper" code that translates between the standardized bus and the legacy interfaces.

This results in having to write several pieces of code. These components have to be implemented to write a simple client-server program:

- The IDL, that describes the interfaces to the object or objects that will be used or implemented.
- The server.

- The client.

In addition, some extra work has to be done to link all these components together: the IDL has to be compiled, using an IDL compiler provided by the particular ORB and all the generated code and the application code have to be compiled.

Inside of these components, a lot of work has to be done to make them work together. If you want to make a simple program that displays “Hello”, you will have a program of more or less 50 lines of code. Fortunately, this difference decreases when bigger programs are implemented. The fact that CORBA works on language independent platforms is of course interesting, but it makes implementation needlessly complicated. And because this feature was not needed in our research, another programming paradigm was looked upon: Java RMI.

3.6.2 Java RMI

According to jGuru[9], Java RMI is an architecture that supports communication between objects that are located on different virtual machines, just like CORBA. Unlike CORBA, this communication is possible only between Java objects. RMI can thus easily be used for the implementation of simple applications. The communication itself originates from a client process towards a server object. This client retrieves, from a registry, the name of a certain object. This registry returns a so-called stub object. Every call to this stub will be converted, by the RMI architecture to a call to the server. The server contains a thread that dispatches the incoming calls to the right target object. Once this object finishes the method, the answer is returned to the client, so that the client can continue its computation. Without further going into detail, we will describe the some components of a working RMI system, as described by jGuru:

- Implementations of the remote services
- A server to host the remote services
- A client program that needs the remote services

For each of these components, a piece of code has to be written. Like in CORBA, communication aspects have to be implemented by the programmer inside of the components.

CBorg

In CBorg we take care of remote services or communication more easily, as was seen in the `makePoint` example, using `remotedict`. In practice, we only have to implement the server and the client, which can be seen as agents.

Communication to an agent is simply achieved by sending messages using the name by which that agent is referenced.

3.7 Conclusion

In this chapter we introduced CBorg and the concept it was originally designed for. We have taken a closer look at components and an implementation of objects (in the end, an object is a component) in this language. We discussed the distribution aspect of CBorg and made a comparison with other distributed programming languages. In this comparison we argued why CBorg was better suited for our study.

CBorg offers an easier interface, and allow us to do complicated things very clearly. In the subsequent chapters, this will come in handy when describing the control flow. In the next chapter a review will be made of other approaches to our main problem: the separation of distribution from the functional aspect of programs.

Part II

Combining Unchangeable Components

Chapter 4

Improving the Reusability of an Agent

4.1 Introduction

In this chapter we take a first look at the results of this study. We take a look at two distributed programs which are built of agents. We study how these agents can be changed, so that they are more flexible, and as a result, more reusable. These changes are then implemented and demonstrated. We then analyze the impact of these changes on distribution and reusability in general.

The examples that we present are very similar. It would have been more interesting to demonstrate the increase of flexibility on *different* programs, but because of a lack of time, other examples could not be included.

The presented programs are similar because we initially wanted to extract the control flow out of these programs, as was mentioned in the introduction. The extracted control flow would have been poured into a control flow component (CFC). The purpose of this extraction was to have a control flow abstraction that could be reused with other programs: the CFC would combine parts of already existing programs together.

As a result, a CFC would have been presented in the last section of this chapter. The use of this CFC would have been illustrated, by reusing this component with the two presented programs.

Unfortunately, combining components was more difficult than expected and as a result, a time consuming study was needed. This study teached us that to combine incompatible agents together, we had to make these agents more modular. Therefore, instead of presenting a CFC for the presented examples, we analyze how these programs can be made more modular.

Note that, although the examples that are presented look very alike, the problems which are discussed apply to distributed programs in general.

4.2 The Pipeline Component

A concept called *pipeline* is now presented. The idea of the pipeline is as follows: several identical components are linked together, in a one-by-one fashion. Some kind of data generator sends data to the first component. This component sends the processed data to the second one, and so on. The last component sends the data to a collector, which functions as terminator of the pipeline. This concept is illustrated in figure 4.1.

Figure 4.1: The pipeline concept

The advantage of using a pipeline is that processing of data in different components can happen concurrently: while the first component starts working on the second input (that was received from the generator), the second component works on the data that was received from the first component[1]. Although the pipeline is not fairly load-balanced, the amount of time gained is more or less proportional to the amount of used components.

In the next section we present two distributed programs that are based on this pipeline. The first program that gets analyzed is the parallel version of the *Sieve of Erastosthenes.* The second program is a parallel version of a *Fast Fourier Transform.*

4.3 Sieve Of Erastosthenes

This method (discovered in ca. 240 BC) is used to find all prime numbers below a given maximum. The algorithm works by a process of elimination: it removes, in parallel, all multiples of some integer. A number is prime if it can only be divided by itself and one. As a result, a number is prime if it cannot be divided by any prime that is smaller than that number.

If the first n primes smaller that a given number are to be found, n stages are needed. In the following program, stages are implemented as components. Two additional components are also used: a *generator* and a *collector.* The generator generates the amount of numbers until the provided maximum is reached. The collector collects the numbers that passed all the components; it functions as a terminator of the pipeline. This concept is illustrated in figure 4.2.

[1] To achieve full concurrency, these components have to run on n different processors. In practice, these components ran on only one processor. Each component had it's own thread and thus we had several threads running on one machine.

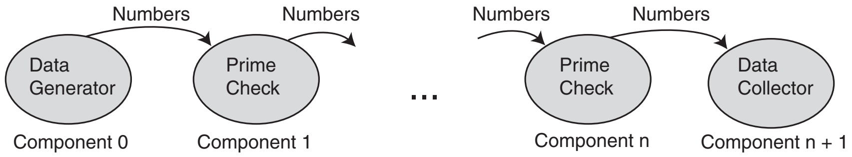

Figure 4.2: The sieve of Erastosthenes

The algorithm itself works as follows: the generator generates numbers and sends it to the first component. When a component receives a number, there are two possibilities:

1. if this component contains no number, the received one is a prime and it is kept locally (and possibly stored in a file).
2. If the component already contains a prime, the received number is checked by the component:
   (a) if division of that number with that prime yields an integer, that number is not a prime and is dropped form the pipeline.
   (b) If it does not yield a integer, it might be prime and must be checked by the next component.
3. The component is the collector and therefore, the number is discarded.

Let's now take a look at how the Sieve was originally implemented.

4.3.1 Original Implementation

We first analyze the generator. The generator's source code can be seen in figure 4.3.

```
CreateStartAgent(name, max, nextAgent):: {
  Start():: {
      for(t:2, t := t + 1, t < max,
            nextAgent.checkPrime(t))};

  agentclone(clone(),name)
}
```

Figure 4.3: A generator implementation

As can be deduced from the code, the creation of this agent requires a *name*, which is needed for the creation of this agent, and a number, which is denoted as *max*. This number corresponds with the upper bound, under which all prime numbers will have to be found. Logically, the agent to which the generated numbers are sent is given as parameter to the agent as well: this is the *nextAgent* argument.

This agent generates all the integers under the given maximum, and sends it to it's successor, by invoking the checkPrime method. The generator's successor is an agent CreatePrimeCheckAgent. This agent checks if the received number is a prime, according to the algorithm that we described before. The CreatePrimeCheckAgent source is given in figure 4.4.

```
CreatePrimeCheckAgent(name,displayTo,nextAgent):: {
   localPrime:0;

   checkPrime(proposedNumber):: {
     if (localPrime = 0, { localPrime := proposedNumber;
                                displayTo.display(localPrime)},
                         if ((not((proposedPrime \\ localPrime) = 0)),
                              nextAgent.checkPrime(proposedPrime),
                              void))
    };
  agentclone(clone(),name)
  }
```

Figure 4.4: The prime checking component

CreatePrimeCheckAgent takes a name and a successor as arguments. In addition, a second agent has to be provided: this second agent is used to display the resulting prime numbers. If we provide the same agent to each component, all the results can be displayed by the same agent.

CreatePrimeCheckAgent analyzes the given number according to the theory that we just described: if the number is the first received number, then that number is a prime. Otherwise, the number has to be checked, and possibly forwarded to the next agent. The successor of CreatePrimeCheckAgent is again a CreatePrimeCheckAgent, until the desired length (i.e. the amount of stages) is reached. The pipeline is then terminated by the Collector, depicted in figure 4.5

```
CreateCollectorAgent(name):: {
   checkPrime(collectedNumber)::void;

   agentclone(clone(),name)
}
```

Figure 4.5: The collector

We then initialize the several agents and run the program by asking the generator to Start(). To illustrate the use of the program, we create four components. This is depicted in figure 4.6. The given maximum is 20, so the first four primes under this maximum will be computed by our program.

```
pipebegin:CreateStartAgent("start",20,
  CreatePrimeCheckAgent("p1",agentself(),
  CreatePrimeCheckAgent("p2",agentself(),
  CreatePrimeCheckAgent("p3",agentself(),
  CreatePrimeCheckAgent("p4",agentself(),
  CreateCollectorAgent("Collector")))))
pipeBegin.Start();
```

Figure 4.6: Running the program

4.3.2 Analysis of the Original Implementation

Although this program works fine, it contains a few limitations.

1. The references to the successors, (nextAgent) in our example, are fixed. This is because the CreatePrimeCheckAgent doesn't contain methods to change the variables that hold these references. This means that if we need a pipeline with one additional component, we will have to create a totally new pipeline, with this extra component added.

2. The name of function calls, for example checkPrime in our program, are fixed (i.e. hard coded). This means that if we would like to insert a new component between two prime checkers, we would have to adapt the code of this component, to understand the checkPrime message.

3. The amount of parameters of a method is fixed: no extra parameters can be passed on the fly by the function calls if needed. Assume for example, that the collector needs additional information from the generator: e.g. a signal that indicates the end of the generation. This signal could be sent along with the last generated number. This would require an additional parameter in the checkPrime method. In the presented example, this is not really a problem, because we only make use of CreatePrimeCheckAgent components, and thus only the changes in the checkPrime method are required. But if we have a program with many different components, every component will have to be changed.

Problem 1 could be solved by keeping references (the agents where result of a method are to be sent) in local variables, together with get and set functions. These variables could then be changed at any time.

This could work well for small programs that only contain a few methods, but assume that an agent contains many methods. This agent would then need to contain a such a "reference-variable" for each method, again with their respective get and set operations. This can make the code difficult to read and has a large code overhead.

Problem 2 is even more annoying: we want to add new components between the already existing ones. These new components will have to be adapted, so that they understand the `checkPrime` method. This is of course not desirable; we would like to be able to tell somewhere *what* methods have to be called, and *where* they should be called.

Problem 3 explains that we would like to make communication between agents, which use methods with a different amount of arguments, possible.

We are now going to see how the new implementation looks like if we adapt our code to address those problems.

4.3.3 New Implementation

The needed changes that were conducted are the following:

1. Change of method names.

   If we want to add a new agent to the pipeline and it's source code is not available (because it is located on a remote machine), then we will have to adapt our own agents such that they can work together with this new one. We assume that we are in possession of the interface of the new agent. The other components are adapted to be compatible with this interface, so that communication between the old agents and the new one is possible.

   As an example we assume that the new agent understands the method `receive`. All the other agents are altered so that they can understand this method as well: we assume that every possible agent implements `receive`.

2. References to the next agent are moved.

   An agent has to know it's successor: instead of giving the successor to the agent as argument, we pass the successor as an argument to the agent's called method. The result of this method will then be sent to this argument. With this approach, it is the agent that calls a method that determines the method's receiver.

3. Extra parameter.

   To be able to transport extra information over the pipeline, the methods are extended with an extra parameter. In this case, the extra information is a boolean that indicates the end of the transmission to the collector.

   Another approach could keep all additional variables in a data structure. The components could then pick out the variables they need to perform their computations. In the following chapters we will discuss a method that handles the problem of additional parameters.

Let's now observe the changes that were made; the parts that differ from the original code are highlighted in light gray. We start with the generator, depicted in figure 4.7

```
CreateStartAgent(name, max):: {
   Start(nextAgent):: {
        for(t:2, t := t + 1, t < max - 1,
                nextAgent. receive(t, false))

        nextAgent. receive(t, true)};

   agentclone(clone(),name)
}
```

Figure 4.7: CreateStartAgent revisited

As can be observed from this new implementation, the agent's successor is given as argument to the Start method. This successor receives the generated numbers using the new receive method. Recall that the successor was originally given as parameter to the agent, at initialization time. Also, boolean parameter is added to checkPrime, which is set to true when the last number has been sent.

The changes to the CreateCollector are shown in figure 4.8.

```
CreateCollectorAgent(name):: {
        receive(collectedNumber, isLastNumber, continue) ::
                checkPrime(collectedNumber, isLastNumber);


        checkPrime(collectedNumber, isLastNumber) :: {
                if(isLastNumber, display.("Job Finished!"), void)};

        agentclone(clone(),name)
}
```

Figure 4.8: The Collector revisited

As can be seen, the collector has been extended by a receive method. In addition, the collector's checkPrime method now takes an extra parameter which signals to the collector, if the received number was the last from the generator. In this case the collector displays "Job finished!".

The CreatePrimeCheckAgent component underwent a lot of changes, as can be seen in figure 4.9.

```
CreatePrimeCheckAgent(name,displayTo):: {
   localPrime:0;

   receive(proposedNumber, additionalPar, returnAgent) ::
        checkPrime(proposedNumber, additionalPar, returnAgent)

   checkPrime(proposedNumber, additionalPar, returnAgent):: {
     if (localPrime = 0, { localPrime := proposedNumber;
                                        displayTo.display(localPrime)},
                          if ((not((proposedPrime \\ localPrime) = 0)),
                                 returnAgent.receive(proposedPrime,
                                                            additionalPar),
                                 void))
   };
  agentclone(clone(),name)
  }
```

Figure 4.9: CreatePrimeCheckAgent revisited

The agents that do the prime checking have been extended with a new method receive, so that communication with an agent of different type is possible (recall that receive is the only method that the additional agent understands). This method invokes checkPrime so that the original functionality remains the same. This checkPrime function takes two additional parameters: additionalPar, which transports the additional data (a boolean in this case), and returnAgent. Remember that the returnAgent variable is the agent where the result of checkPrime is to be sent (returnAgent is in fact the equivalent of the nextAgent in the original program, renamed to a more logical name).

If we take a closer look at this program, we can see that the receive method which is declared, takes a different amount of argument than the receive method, which is invoked in the message-send at the end of the checkPrime method. This is a consequence of the inclusion of the return agent in the method. We will show how this can be fixed, when we create a pipeline with these revised components.

We now setup a pipeline. As an example, consider the source in figure 4.10.

We initialize the different components of the constructed program. We now have to set up the references between the agents manually. This is because we don't provide the references at initialization time anymore. To set up these references, we need an extra component that links the several

```
S:CreateStartAgent("start",20);
P1:CreatePrimeCheck("First Check");
P2:CreatePrimeCheck("Second Check");
P3:CreatePrimeCheck("Third Check");
P4:CreatePrimeCheck("Fourth Check");

MakeSendTrough(primeChecker,continue) :: {
        receive(value, xtra):: primeChecker.receive(value, xtra, continue)}

s.start(makeSendTrough(p1,
        makeSendTrough(p2,
        makeSendTroughj(p3,
        makeSendTroughj(p3,
                CreateCollectorAgent("The collector")))
```

Figure 4.10: Initializing the program

parts of the program together. In the example in figure 4.10, this "link-component" is named MakeSendTrough. As we can see, this component fixes the problem that we mentioned about the receive methods.

Let's overview the implications of this new implementation.

4.3.4 Analysis of the New Implementation

The changes that we looked at have some consequences; some are positive, some negative.

Advantages

- A method's return value is sent to another agent. This receiving agent is given as parameter to this method. The use of a parameter implies that the receiver of the method's return value is not fixed anymore. This means that a method can have a different receiver for each call.
- With these methods, no new initialization is needed; only the references have to be set up again. In fact, the setup of these references is done by one specific entity. As a result, the setup is centralized and clearer.
- The assumption that every component understands a receive method allows us to make communication between different agents possible. As a result, different agents can be used in a pipeline.

Disadvantages

- We still have to mess around in the component's source code.

- We need a construction to link the different parts together (such as MakeSendTrough), which is rather difficult.

- We have to dispatch of methods by ourselves, using the previous construction.

In addition, we encountered problems which originated from the desire to include "target agents" into methods. Recall that we wanted to make components compatible with each other by renaming methods to "receive". This requires that the amount of arguments of these receive methods are equal. If we include "target agents" into methods, this equality doesn't hold anymore. This could be seen in the presented example.

Therefore, we drop the idea of including "target agents" to methods. Instead, we focus on how to make agents compatible.

We will now discuss our next example, a parallel Fast Fourier Transform implementation. As will be seen, the same problems will be encountered.

4.4 Fast Fourier Transform

Linear transforms, such as the Fourier transform, are widely used to solve problems in science and engineering: linear system analysis, optics, probability theory, digital signal processing, etc. The Fourier Transform is used to alter a mathematical problem into one that can be more easily solved.

The mathematical background of the Fourier transform is not discussed. The only thing that we need to know is that a Fourier transform takes an amount of numbers (coefficients) as input, and outputs (a vector A) the same amount of transformed numbers. The algorithm that performs this transformation is as follows, as given by Cormen [13]:

ITERATIVE-FFT(coefficients)
  BIT-REVERSE-COPY(coefficients, A)
  $n \leftarrow length[a]$ n is a power of 2
  for $s \leftarrow$ **to** $lg(n)$
    do $m \leftarrow 2^s$
      $\omega_m \leftarrow e^{2\pi i/m}$
      $\omega \leftarrow 1$
      for $j \leftarrow 0$ **to** $m/2$
        do for $k \leftarrow j$ **to** $n-1$ **by**m
          do $t \leftarrow \omega A[k+m/2]$
            $t \leftarrow A[k]$
            $A[k] \leftarrow u+t$
            $A[k+m/2] \leftarrow u-t$
          $\omega \leftarrow \omega\omega_m$
  return A

Where $\omega = e^{-i2\pi \frac{i}{n} xs}$, and A and $coefficents$ are given in coefficient form: $a = (a_0, a_1, ..., a_{m-1})$. The BIT-REVERSE-COPY subroutine is as follows:

BIT-REVERSE-COPY(a, A)
$n \leftarrow length[a]$
for $k \leftarrow 0$ **to** $n - 1$
do $A[rev(k)] \leftarrow a_k$

And finally, $rev(x)$ is the lg-n-bit integer formed by reversing the bits of the binary representation of x.

This algorithm has several concurrent variations: a *coarse grained* and a *fine grained* version. The coarse grained version takes several sequences of coefficients as input, which will be computed in parallel. In contrast, the fine grained approach computes coefficients of only one sequence in parallel.

We are going to take a look at the coarse grained parallel version of this algorithm. The working of this coarse grained FFT version is described in figure 4.11, as given by Cormen [13].

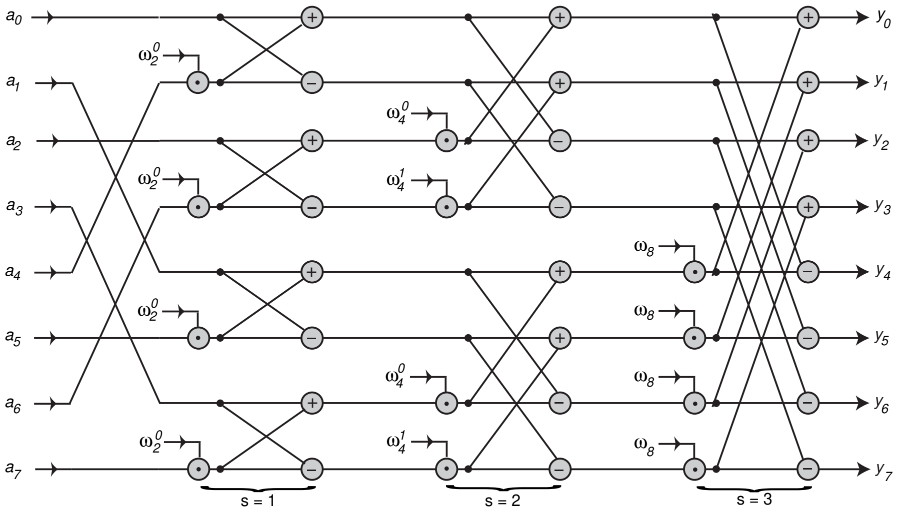

Figure 4.11: Overview of the coarse grained concurrent FFT

This combinational circuit begins with a bit-reverse permutation of the inputs, followed by lg n stages. Each stage consists of n/2 *butterfly* operations, executed in parallel. A butterfly operation takes two arguments x and y. The second argument is multiplied with another value z. A sum and difference are then returned: $x + zy$ and $x - zy$.The depth of the circuit is therefore $O(\lg n)$.

The leftmost part of the circuit performs a bit-reverse permutation, and the remainder mimics the following procedure:

BASE-FFT(coefficients)
  $n \leftarrow length[a]$ $\qquad$ n is a power of 2
  for $s \leftarrow$ **to** $lg(n)$
    do $m \leftarrow 2^s$
      $\omega_m \leftarrow e^{2\pi i/m}$
      for $k \leftarrow 0$ **to** $n - 1$
        do $\omega \leftarrow 1$
          for $j \leftarrow 0$ **to** $m/2 - 1$
          do $t \leftarrow \omega A[k + j + m/2]$
            $u \leftarrow A[k + j]$
            $A[k + j] \leftarrow u + t$
            $A[k + j + m/2] \leftarrow u - t$
          $\omega \leftarrow \omega\omega_m$
  return A

In practice, we use a setup which is similar to the setup of the sieve, as can be seen in figure 4.12. First, a generator provides the vectors of coefficients that have to be worked on. Per stage, an agent carries out the computations and sends it's results to the next stage.

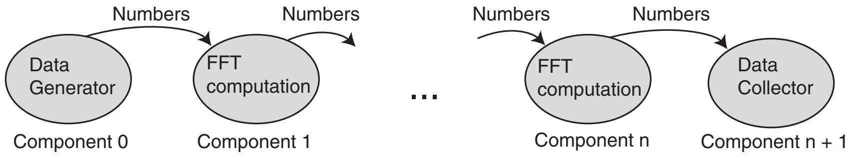

Figure 4.12: A concurrent FFT computation

We now take a look at the original source code. Note that the part of the algorithm that uses the ω's has been left out. This is because these variables make use of imaginary values, which can't be easily represented. This is alright though, because we only want to illustrate the distributed aspect of this algorithm.

4.4.1 Original Implementation

Recall from section 4.3 that the generator, that was used in sieve, generated the input numbers automatically. The Fast Fourier Transform takes *specific* values as input, instead of generated ones. Therefore, the generator of our FFT takes these values as arguments. This data is given to the generator, at initialization time, using the start method. This component is depicted in figure 4.13.

```
CreateStart(firstStage,name):: {
   start(inputArray):: {
     i:1;
    while(i < size(inputArray) +1, {
        firstStage.CalcFFT(inputArray[i]);
        i:=i+1})
   };
 agentclone(clone(),name)
 }
```

Figure 4.13: The generator, which provides the coefficients to the first stage

This CreateStart agent also takes a name and the first stage as argument. The first stage receives the coefficients from the CreateStart generator.

The collector, shown in figure 4.14, used in the FFT, is also slightly different from the one that was used with the sieve.

The collector takes a name and console as arguments. This console is an agent where the results from the FFT will be sent. This agent could then for example, display these results, or do some other computation.

The CreateFFTStage agent, which is depicted in figure 4.15, implements an FFT stage. It only takes a name as argument at creation time. It contains an initialization method init which sets the current stage number

```
CreateCollector(name, console):: {
   CalcFFT(resultArray, console):: {
      console.display(resultArray)
   };
 agentclone(clone(),name)
 }
```

Figure 4.14: The collector; similar to the sieve collector

(which is needed in the computation), and the next stage (which will receive the newly computed coefficients).

```
CreateFFTStage(name, stageNr) :: {
    nextStage : void;
    stageNr:0;

    Init(newNextStage, newStageNr) :: {
       nextStage := newNextStage;
       stageNr := newStageNr;
    };

    CalcFFT(coefficients) :: {
        n : size(coefficients);
        m : trunc(2 ^ stageNr);

        for (j : 1, j := j+1, not(j > (m // 2)),
           for (k : j, k := k + m, not(k > n), {
              temp1   : coefficients[k];
              temp2   : coefficients[k + (m // 2)];
              coefficients[k]   := temp1 + temp2;
              coefficients[k + (m // 2)]  := temp1 - temp2}));

     nextStage.receive(coefficients)
     };

    agentclone(clone(), name)
 } :
```

Figure 4.15: The implementation of an FFT stage

The program initializes as follows: the several agents are created and the stages are initialized. The generator is then started with a given input. As mentioned before, the amount of coordinates determines the amount needed stages.

We will now analyze this program.

```
st:CreateStart(s1, "Generator");
s1:CreateFFTStage("stage1");
s2:CreateFFTStage("stage2");
s3:CreateFFTStage("stage3");
console : remotedict("tecra/ses1")
c:CreateCollector("Collector", console);

s1.Init(s2, 1);          s2.Init(s3, 2);          s3.Init(c, 3);

st.start([[1,2,3,3,4,5,6,8], [2,6,4,5,8,7,9,3]])
```

Figure 4.16: Initialization of the program

4.4.2 Analysis of the Original Implementation

Within this program, references are set up using an initialization method. In this way, references are kept in local variables. Like we argued before, this can easily be done, but we have to keep a "target-variable" for each method: if we we have an agent with much methods, a lot of variables will have to be used. In addition, we have to provide another target, each time a method has to send its results to another target. This is not possible in this implementation.

The other two problems that we had with the sieve program are present in the FFT as well. We have to mingle in the source code if we wan't to add a new (different) agent or if we need an extra parameter in the agent's methods.

We again show an adapted version of the program to address these problems:

1. We pass the target-variables to the method themselves, instead of passing it to some initialization method.
2. We rename the methods to a more uniform name and we assume that every agent we want to add understands this (method)name.
3. We add a new parameter to the methods to support additional information.

4.4.3 New Implementation

The new implementation is now presented, changes are highlighted in grey. Like in the revised sieve, the target stage is now passed as argument to the start method and the coefficients are sent to the next stage, using the general receive method.

The CreateFFTStage has undergone the most changes. The initialization method Init has been removed: the stage number must now be passed at

```
CreateStart(name):: {
   start(inputArray, firstStage):: {
     i:1;
     while(i < size(inputArray) +1, {
         firstStage.receive(inputArray[i]);
         i:=i+1})
   };
  agentclone(clone(),name)
  }
```

Figure 4.17: The generator revisited.

creation time, and the next stage must be provided to another method. In analogy to the generator, information is now given to the next stage by the use of receive.

```
CreateFFTStage(name, stageNr) :: {

receive(coefficients, nextStage) :: CalcFFT(coefficients, nextStage);

CalcFFT(coefficients, nextStage) :: {
      n : size(coefficients);
      m : trunc(2 ^ stageNr);

      for (j : 1, j := j+1, not(j > (m // 2)),
         for (k : j, k := k + m, not(k > n), {
             temp1   : coefficients[k];
             temp2   : coefficients[k + (m // 2)];
             coefficients[k]   := temp1 + temp2;
             coefficients[k + (m // 2)]  := temp1 - temp2}));

    nextStage.receive(coefficients)
   };

    agentclone(clone(), name)
 }
```

Figure 4.18: The adapted stage implementation.

The collector has undergone a slight change as well. It's new implementation is depicted in figure 4.19. As can be seen, this agent now implements the new method receive.

As in the sieve, we now need to glue the different parts of the program together. We can reuse the MakeSendThrough component to do this linkage. Again, we can note that things become more complex if we want more functionality.

```
CreateCollector(name, console):: {

   receive(coefficients, nextAgent) :: CalcFFT(coefficients);

   CalcFFT(resultArray):: {
      console.display(resultArray)
    };
 agentclone(clone(),name)
 }
```

Figure 4.19: The new collector.

```
st:CreateStart(s1, "Generator");
s1:CreateFFTStage("stage1");
s2:CreateFFTStage("stage2");
s3:CreateFFTStage("stage3");
console : remotedict("tecra/ses1")
c:CreateCollector("Collector", console);

MakeSendTrough(fftStage,continue) :: {
        receive(value):: fftStage.receive(value,continue)}

st.start([[1,2,3,3,4,5,6,8], [2,6,4,5,8,7,9,3]],
        makeSendTrough(s1,
        makeSendTrough(s2,
        makeSendTroughj(s3,
                CreateCollector("The collector", console)))
```

Figure 4.20: Initialization of the program.

We could make another analysis from this revised program, but this analysis would be identical to the one we made with the revised sieve. Therefore we will immediately jump to the conclusions that follow from the used programming methods.

4.5 Conclusion

These extensions we implemented were mostly work-arounds: renaming a method and assuming that every added agent understands this method is not really realistic. Also, including a "target agent" into a method doesn't make things really easier. The need to use an extra component, that links the components together, makes the program harder to write than before. Implementing such "link components" can get even harder. In the given examples, the agents used only one method, so the linkage was relatively simple. Linking agents of complex programs (i.e. using agents that contain more than one method) together, will be much more difficult.

Therefore, we will focus on possibilities to make agents understand other agents methods, without altering them. This means that we must translate method invocations to other method invocations. This translation could be done by a special component.

In practice, we would need to tell this component what methods must be invoked. We must therefore be able to give those methods as input to this component. We represent these methods in the most logical form: strings. These strings are method names that must be translated to messages or method calls by the components we want to implement. To perform these translations, we need to use of the *meta object protocol* (or *MOP*) of CBorg. This protocol deserves a chapter of it's own.

Chapter 5

Metaobject Protocols and Distributed Systems

5.1 Introduction

In the last chapter we described some tricks to make agents that were part of a distributed system more reusable. One of these tricks was to use a separate component that controlled the communication between the several agents. We deducted that, to make this component reusable, we needed more control over the creation of message sends. This control can be attained by using CBorg's meta-level.

In this chapter we study metaobject protocols. We describe how Java's MOP can be used. An extensive analysis of the MOP of CBorg is given. We then look at the flaws of metaobject protocols of distributed programs, and study what the requirements for a good distributed MOP should be. We then use CBorg's meta-level to illustrate these requirements with some examples. These examples show how, given a good MOP, a higher degree of reusability can be attained.

5.2 What is MOP?

According to Stroud and Wu[14], metaobject protocols are interfaces to a system, that give users the ability to incrementally modify the system's behaviour and implementation. Reflection plays a critical role in the metaobject protocol approach. Reflection makes it possible to open up a system's implementation without revealing unnecessary implementation details.

Metaobject protocols are implemented by *metaobjects*. In this case, a metaobject protocol is implemented by a system with metaobjects, which are accessible to users. These metaobjects can contain an object's name, an object's method list, etc. The user can add new methods to an object or ask an object's name, if he desires to. Other object elements can be generally be changed, but since we don't need these, we wont discuss them.

We now demonstrate a use of Java's MOP.

5.2.1 Java's MOP

We will give an example of an MOP-feature of Java that allows a programmer to transform a string into a method invocation. The following piece of code illustrates this. The program retrieves a method with name mName and arguments mArgs, from an object fromObject. Once the method is retrieved, it can be invoked with m.invoke(fromObject, mArgs).

If a method with name mName is not present in fromObject, "no such method" is displayed.

```
public void invokeMethod(String mName, Object fromObject,
                                            Object mArgs) {
   try {
      Method m = fromObject.Class.getDeclaredMethod(mName, mArgs);
      <adapt mAgrs if needed>
      m.invoke(fromObject, mArgs);
   }
   catch (Exception ex) {
      System.err.println("no such method: "+mName);
   }
}
```

Note that a lot of work has to be done to translate a string to a method invocation. We now introduce CBorg's meta-level.

5.3 Accessing CBorg's Meta Level

Unlike Java, CBorg doesn't use objects to access the meta level. In CBorg, expressions can be directly modified: function names can be changed, arguments can be added, etc.

5.3.1 Reification

In CBorg, every expression is represented by an array which contains the abstract grammar tree of each type of expression. The following table describes the structure of the most relevant expressions.

Expression Type	Content 1	Content 2
Application	name	arg list
Message	target	Application
Variable definition	reference	value
Reference	name	n/a
Closure	function body	n/a
Function definition	dictionary	n/a

CBorg's `read` native allows the user to *fetch* the abstract grammar tree which is associated with a given expression. `read` takes this expression as a string as argument. As an example, consider the sum of two numbers. A sum in CBorg is an application. The result of `read“(5+3)”` returns the abstract grammar tree which represents the application of that computes the sum of the two given numbers. In this example, the grammar tree is stored in the `tree` variable.

```
tree:read("(5+3)")
:<application>
```

Using the previous table, the components that make up an expression can be retrieved. Consider the previous example where an application was read. The applications's name and arguments can be retrieved as follows:

```
tree[1]
:+

tree[2]
:[5,6]
```

To include the variables and constants of expressions in an environment, the expressions need to be evaluated. This evaluation is done using the `eval` native, which returns the result of the evaluated expression:

```
eval(tree)
:<application>
```

Using CBorg's meta-level, every type of expression can be read and evaluated in this way by the CBorg interpreter. It is even possible to create and manipulate expressions, which is not possible in most other languages.

Expressions in CBorg are identified by a *tag*, which we present now. Afterwards, an example of the creation and manipulation of expressions will be given.

Tags

In CBorg, every expression type is identified by a tag. This tag can be retrieved using the tag native. The tag native takes an expression as argument and returns its associated tag. The use of this native is now illustrated.

```
tag([1,2,3])
:0

tag("read(foo(x)"))
:10
```

As can be deducted from these examples, a tabular expression is associated with the identifier "0" and an application is associated with the identifier "10". An overview of the most relevant expressions and their respective i.d.'s is given in the following table.

Expression Type	Tag
Application	10
Message	20
Variable definition	14
Reference	8
Closure	4
Function definition	2

The unique number which identifies each type of expression can be used to create new expressions. This is explained in the following subsection.

5.3.2 Absorption

An expression can be created using the make native. This function takes an expression-tag as argument. A corresponding expression is created internally, with all its contained sub-expressions set to zero. These sub-expressions can be manually set by the user.

The make native is illustrated, again with an application-type expression. Assume that we want to make the following application: (sum(2,3)). Recall that an application consists of a name, which is a string, and an argument list, which is a table. Also, an application applies a function to arguments. This function has to be defined somewhere, or else the application will yield an error. Therefore, we define the sum function, before we make the application.

```
sum(x,y):: (x+y)
:<application>
```

Let's start with the application. We first create an expression, which we define to be an application using the appropriate tag.

```
appl: make(10)
:<application>
```

We can now set the name of the function the application will invoke: "sum".

```
appl[1]:= "sum"
:<application>
```

The argument list is set as follows:

```
appl[2]:= [2, 3]
:<application>
```

The application can now be evaluated, using eval:

```
eval(appl)
:5
```

Again, any CBorg expression can be constructed in this way: the only things you need to know are the tag of the expression and the expression's structure.

We discussed the meta level of the programming language that we used to perform our study. We will now discuss how current MOP's could be altered to offer more support for distributed programming.

5.4 Flaws of Distributed MOP's, and Possible Solutions

The restrictions of most metaobject protocols are now discussed. These restrictions are illustrated with examples. These examples demonstrate problems that we would like to solve with MOP.

- **Method invocations are not first class.** To make communication between components possible in distributed programming, we often need wrappers. These wrappers make it possible to translate "not understood" messages to an object to messages that the object *does* understand. To be able to do this, the wrapper must have the possibility to create method invocations.

  The manual creation of method invocations is not always possible. As an example, consider the invoke from Java. Java's MOP makes it possible for users to invoke a method, by using that method's name, as was illustrated by piece of program in section 5.2.1.

It would be much easier to let the programming environment handle all these things. For example, it would be much shorter and straightforward, if we could write targetObject.<mName>, where targetObject is an instance of an object, and mName a string. To tell the environment that this string is in fact a method's name, this string could be placed between a pair of special delimiters, like $<$ and $>$.

- **Message representation is not first class.** Consider for example a message "anAgent.doSomething(1, 2)". We can't change the value of anAgent either: this could be useful to redirect the receiver of the doSomething method (e.g. like a proxy). A second thing that we can't do is rearrange the order of the parameters, or add a third one. It could be useful to add another object as parameter, for example, an object where the result of doSomething is to be sent: anAgent.doSomething(1, 2, *returnAgent*).

- **No standard entry point** (such as "Message Not Understood") **for objects.** When a message is received by an agent, it is immediately invoked. There is no way that this message can be delayed by the receiving agent. Delaying the execution of this message could be useful: this message could then, for example, be forwarded to another agent.

  Note that smalltalk allows the user to adapt "Message Not Understood".

- **Fixed return mechanism** Usually, when a function is called, it perform some computation and *returns* a value of some kind. It would be interesting to tell a function to return it's value to another agent, *from outside* that function.

- In some case the user would like to do serialization by himself, but this is, again, often not possible.

- **Fixed error handling mechanism.** Assume that we have several remote agents that work together to solve a certain task. If one agent fails, then the other agents might not be aware of this failure. The only thing that might happen is an error message on the machine of the failing agent. This is a problem because all the agents are needed to solve the given problem: a "defective" agent might not have the possibility to respond to other messages anymore.

  It might therefore be interesting to adapt the error handling. This could be useful if an agent doesn't get a response from another agent. This defective agent could then return some kind of error report to the other agents, telling that the computation has failed.

Some of these meta level features were already present in Pico, other requirements were added. With a few extensions, Pico now offers support for the following issues:

- First class method invocations (already present).
- First class message sends (already present).
- A standard entry point for objects (added).
- A more flexible return mechanism (added).

In the next section we will present the MOP features that can be used to make components of distributed programs more flexible and reusable.

Recall the two programs from the last chapter, the Sieve and the FFT, which both used the pipeline design. We tried to make the agents that were part of these applications more reusable. We achieved this increased reusability by making the several agents more flexible. These agents were linked together by a special component.

5.5 Making Agents Understand Each Other

Assume we have a program built up from agents and we want to add a new agent to this program (e.g. add a new different component to the pipeline). Some of the agents will have to be changed so that they understand each other's messages. If neither the new agent's source can be altered, nor the source of the already present ones, then the agents wont be able to communicate with each other.

There are in fact *two* problems:

1. a new agent doesn't necessarily understand the messages it will get from other agents, and
2. this agent doesn't know what messages the other agents understand.

In the last previous chapter, we "solved" this problem using the assumption that every agent, old or new, understood some kind of universal method. We named this method "receive". This receive invoked the method that contained the functionality of the agents (the PrimeCheck or CalcFFT).

This approach is of course not realistic: an agent usually contains more than one method, and renaming every one of them to receive would lead to conflicts and inconsistency. In addition, we can't assume every agent understands receive. It's even worse: we can't alter the source code of an agent to implement this method, because we don't have access to this source code.

We will show that, if agents know the names of the methods they invoke, then this problem can be solved using CBorg's meta-level. To make communication between old and new agents possible, we *tell* agents what methods must be invoked, and what methods they should understand. By doing this, the "old" agents can send appropriate messages to the "new" agent and the "new" agent knows what messages the "old" agents understand.

We now address the problem of agents that don't know what messages other agents understand.

5.5.1 Creation of Methods to Invoke

To "tell" agents what methods they should invoke, we use strings. These strings represent the names of the methods which need to be called. These names are used to create messages, which invoke the appropriate methods. Messages can be created using CBorg's meta-level.

Assume that we would like to create a message "targetAgent.aMethodName(1, 2, 3)". From the first table in section 5.3.1 we can see that a message consists of two parts: a receiver name and an application.

We first create an expression of type "message":

```
message:make(20)
```

The receiver of the message is now set:

```
message[1]:targetAgent
```

The second part of the message, an application, is fetched and the application's name and arguments are set.

```
app:message[2]
app[1]:="aMethodName"
app[2]:=[1, 2, 3]
```

The message aMethodName will now be sent to the targetAgent, as soon as the created message is evaluated.

```
eval(message)
```

Message creation is not really easy to implement, without good knowledge of CBorg's abstract grammar representation. Therefore, we extended CBorg with a feature that creates and executes an invocation, given a method and/or an agent name. The syntax of such invocation is as follows:

```
targetAgent.<aMethodName>(argList)
<targetAgent>.<aMethodName>(argList)
```

<targetAgent>.aMethodName(argList)

CBorg detects if the "<" and ">" delimiters are used in a message, and automatically transforms the string between them to a reference to an agent or method.

We can now integrate these facilities into our examples. Assume that we know the name of the method that must be invoked. Let us look at the source code of the new agent.

```
CreatePrimeCheckAgent(name,displayTo, methodToInvoke):: {
  localPrime:0;

  checkPrime(proposedNumber, additionalPar, returnAgent):: {
    if (localPrime = 0, { localPrime := proposedNumber;
                          displayTo.display(localPrime)},
                    if ((not((proposedPrime \\ localPrime) = 0)),
                        returnAgent.<methodToInvoke>(proposedPrime,
                                                     additionalPar,
                                                     returnAgent),
                        void))
  };
 agentclone(clone(),name)
 }
```

Figure 5.1: The agent takes the method to invoke as argument.

As can be seen, the user can provide the name of the method that checkPrime will invoke. Assume that we want to add a new agent to a program. If we know the names of the messages that this new agent understand, we can give these method names as argument to CreatePrimeCheckAgent. As a result, CreatePrimeCheckAgent will send messages to the new agent, which the new agent will understand.

We could alter the source code of the new agent in an identical manner. This agent would then also be able to send messages that old agents understand. But if the new agent is located on a remote computer, or if it's source code is not at hand, we can't alter its source code. Therefore, we will show another way to make old agents understand messages from new agents.

5.5.2 Making Agents Understand Received Messages

Assume that some agent A sends a message aMethod(argList) to an agent B. B must have a method declaration for aMethod somewhere, to be able to understand this message. If there is no such declaration, B will generate a "Message not Understood" error, and report it to the user.

We will take a look at two methods to make the agent B accept received messages. The first method consists of the creation of method declaration.

The second method consists of overriding the "Message not Understood" error, which we will discuss later. Both these methods make use of CBorg's meta-level. We now take a look at how a method declaration can be created using CBorg's meta-level.

Method Declarations with CBorg's meta-level

We can make a method declaration in an agent B, given a method name "aMethod", using CBorg's meta-level. This agent is then able to understand the invocation aMethod(argList). But this newly created method doesn't know the body it must contain. To simplify, we make the declared method execute the body of another function. We now take a look at how CBorg can be used to implement such declarations.

In CBorg, a function declaration consists of two parts: a method name with an argument list, and the body of the declaration. The body of this declaration is an application. Like we said before, this application part is the body of another function. This function is a closure-expression, which consists of three parts: the name of the closure, references to its arguments, and its body, which is an application.

We now make the method declaration. To do this, we make an expression of type "declaration" which has tag 16.

```
decl:make(16)
```

First, we set the method name and argument list. Second, we set the body of the declaration. This body is an application, which we take from the closure doSomeThing[3].

```
decl[1]:= read(aMethod+"@argList")
decl[2] := doSomeThing[3]
```

When the declaration is evaluated, the method declaration is set up, and the object that contains this declaration will now understand method invocations with name aMethod.

```
eval(decl)
```

If we put such a declaration in a CreatePrimeCheck it would look like in figure 5.2:

With these changes, CreatePrimeCheckAgent is now able to understand invokedMethod messages. A few remarks can be made: first, the called method is required to have the same amount of arguments as the function whose body is used. This is a restriction, and therefore we would like the created declaration to take an arbitrary amount of arguments. Second, we are adding translation aspects into our agents. We want translation aspects to be separated from the functionality of the agents. Therefore, it is better

```
CreatePrimeCheckAgent(name,displayTo, methodToInvoke,
                                        invokedMethod):: {
  localPrime:0;

  checkPrime(proposedNumber, additionalPar, returnAgent):: {
                    [...]
  };

  decl: make(16);
  decl[1] := read(invokedMethod +"(proposedNumber, additionalPar,
                                              returnAgent)");
  decl[2] := checkPrime[3];
  eval(decl);

  agentclone(clone(),name)
}
```

Figure 5.2: The CreatePrimeCheck agent that can now understand invokedMethod messages

to move the communication aspects to another component, which we already used in our examples: the MakeSendThrough component.

5.5.3 Communication through MakeSendThrough

Recall from our examples that, once initialized MakeSendThrough sets up references between agents. The MakeSendThrough component places itself between the agents: when an agent sends a message, it is in fact sent to MakeSendThrough, which dispatches the message to the following agent. Because MakeSendThrough functions as a dispatcher, it has to know the messages agents understand and the messages agents can send.

Because the method that was invoked by agents was always named receive, catching and dispatching was easily implemented. We now show how MakeSendThrough can be adapted using the extended translation features from the two previous sections.

As can be seen from figure 5.3, MakeSendThrough now understands a method name invokedMethod. In addition, it sends messages <methodToInvoke>(argList, nextAgent) to the next CreatePrimeCheckAgent.

If we use this version of MakeSendThrough with the CreatePrimeCheckAgent agents, we need to give the name of the methods that are used to send and receive information as an argument. The result is shown in figure 5.4.

Again, the MakeSendThrough components link the several agents together. These components now take two additional arguments: invokedMethod, which is used to make a method declaration, and methodToInvoke,

```
MakeSendTrough(primeChecker, nextAgent, invokedMethod,
                                          methodToInvoke) :: {
  Dispatch@argList :: {
       primeChecker.<methodToInvoke>(argList, nextAgent)
  };

  decl: make(16);
  decl[1] := read(invokedMethod +"@argList");
  decl[2] := Dispatch[3];
  eval(decl)
}
```

Figure 5.3: The invoked message calls a dispatcher, which sends the data to the right agent, using the appropriate method.

```
S:CreateStartAgent("start",20);
P1:CreatePrimeCheck("First Check");
P2:CreatePrimeCheck("Second Check");
P3:CreatePrimeCheck("Third Check");
P4:CreatePrimeCheck("Fourth Check");

S.start(makeSendTrough(P1, "receive",  "receive")
        makeSendTrough(P2, "receive", "receive")
        makeSendTroughj(P3, "receive", "receive")
        makeSendTroughj(P4, "receive", "receive")
             CreateCollectorAgent("The collector")))
```

Figure 5.4: MakeSendThrough takes invokedMethod and methodToInvoke as arguments.

which is used to create a message.

We now show the second method, to make an agent accept methods that it does not understand.

5.5.4 Catching "Message Not understood"

In CBorg, every agent has an associated "Message Not Understood" error message. This message is automatically declared when an agent is created. This error message is invoked when the agent receives a message that it does not understand, i.e. a method that should be invoked by the receiving agent, but which is not declared in this agent. CBorg's meta-level allows us to override this error message. This feature can be used to forward messages, e.g. depending on the value of an argument, without executing them.

We will now show how MakeSendThrough overrides "Message Not Understood", so that received messages can be forwarded to other agents. Assume that MakeSendThrough receives the message wrongMethod(argList) from an agent. MakeSendThrough doesn't understand wrongMethod and therefore, the "Message Not Understood" method is invoked. This method is has the following parameters: the unrecognized method, the method's arguments and the agent that sent the unrecognized method. These arguments can be retrieved from "Message Not Understood" using CBorg's meta-level. These arguments can then be used to create new messages. In our example, wrongMethod and argList are used to create a message "aTargetAgent.<wrongMethod>(argList)". This new message is then evaluated and as a result, wrongMethod(argList) is dispatched to aTargetAgent. Figure 5.5 gives an example of the above.

```
MakeSendTrough(name, nextAgent) :: {

  MessageNotUnderstood(aMethod, argList, sender) :: {

        nextAgent.<aMethod>@argList
  };

  agentclone(name, clone())
}
```

Figure 5.5: MakeSendThrough overrides "Message Not Understood"

In this example, MakeSendThrough doesn't understand any message. Therefore, all messages which are sent to MakeSendThrough will be sent forth to the nextAgent parameter[1].

Dispatching messages to several agents is a little more difficult, because we don't know what messages have to be sent to what agents. Therefore, it would be interesting to dispatch a message whose receiver depends on the sending agent. Consider the following double dispatching like example. Assume that we have a MakeSendThrough component and three agents A, B and C. If MakeSendThrough receives a message from agent A, the message has to be dispatched to agent B, otherwise it has to be sent to agent C.

This leads us to the fact that a receiving agent would like to known it's caller. This problem is discussed in the next section.

[1]Note that MakeSendThrough must be an agent to be able to override the "Message Not Understood" method, because only agents have such a method.

5.5.5 You want to know your caller

A receiver would be able to know its caller, if this caller included a self-reference inside its sent message. This reference could be an extra parameter, for example, anAgent.aMethod(argList, *self-ref*). This additional parameter is then fetched from the argument list by the receiving object. Depending on the value of this parameter, the received method is then sent forth to the appropriate agent.

If we add parameters at the base-level, all receiving agents have to be adapted to support this additional parameter. It is therefore more interesting to add these parameters at the meta-level. In this case we could send a message, where the method part is extended with this parameter, to a dispatcher. This dispatcher would receive this message and the added parameter would be read. This parameter is removed, to keep the message compatible with the declared methods in the other agents. Finally, the message is sent forth, depending on the value of the parameter.

Depending on this additional parameter, it would be interesting to influence the *return mechanism*. We propose a new primitive which is able to do this.

5.5.6 The "→" notation

The new primitive, "→", that we present here was originally created to describe the control flow of a distributed program in a straightforward manner.

The syntax of this primitive is defined as follows:

agentA. aMethod(argList1) → agentB. anotherMethod(arglist2).
This expression does the following:

1. aMethod is invoked at agentA with the given argument list argList1.

2. The result of aMethod is appended to arglist2.

3. anotherMethod is invoked at agentB with argument list argList2, which also contains the result of aMethod (because of 2.).

Using this notation, we are able to easily describe the control flow of a program. Consider, for example, the following program:

A. getData() → B. doCalc() → C. show() (1)

To achieve the same functionality without the arrow, we would have to write down:

data : A. getData()
result : B. doCalc(data)
C. show(result)

It is clear the first program is shorter. In addition, the control flow can be read and deducted more easily then in the second program.

To be able to use this arrow, we need an additional primitive function, which is presented now.

5.5.7 The **return** primitive

A few questions arise when using this arrow. For example, what happens if a method wants to send its result to another agent than the arrow defines. Consider, for example, the method `getData()`, depicted in figure 5.6.

```
CreateSomeAgent(name, targetAgent) :: {

  getData() :: {
        data : < read data from somewhere, e.g. a file >;
        targetAgent.receive(data)
  };

  agentclone(name, clone())
}
```

Figure 5.6: `getData()` wants to send its result somewhere else

In this example, `getData()` sends its result to `targetAgent`. After this message-send, `getData()` will return void. This means that, in program (1), B. doCalc() will be invoked with an empty argument list, yielding an error.

To deal with this, the arrow must be able to override such messages. Therefore, the arrow must have the ability to detect these message-sends. We try to solve this problem using yet another primitive: `return`. We use `return` by putting it in front of a message-send, as illustrated in figure 5.7.

```
CreateSomeAgent(name, targetAgent) :: {

  getData() :: {
        data : < read data from somewhere, e.g. a file >;
        return targetAgent.receive(data)
  };

  agentclone(name, clone())
}
```

1. If getData() was invoked by "->", "data" will be returned.

2. If getData() was invoked without "->", targetAgent.receive(data) is evaluated

Figure 5.7: Using `return`

The semantic of return is as follows. Consider a method which contains return agent.aMsg(data). If that method was invoked by an arrow, then return will return the arguments, contained in the message-send, to the arrow. The right part of the arrow can then be evaluated. If the method, which contains a return, was *not* invoked by an arrow (e.g. by a message-send), then return is ignored.

To be consequent, we define that the arrow can only get return values using return. We summarize this functionality in the following table. In this table, the arguments of the methods are in fact argument lists.

return	$\rightarrow$	**Effect**
val	$\rightarrow A.methodX(a)$	$A.methodX([a, val])$
$Y.methodY(a1)$	$\rightarrow A.methodX(a2)$	$\rightarrow A.methodX([a2, a1])$

As we have seen, the arrow notation can be used to describe the control flow in a straightforward way. It should be noted that the example we used to demonstrate this arrow was very simple. How the control flow of complicated programs looks like using the arrow, we don't know. This is because we stopped the research about control flow to focus on glue components.

The side effect of this notation is that we can override messages from outside an agent. This could prove useful, for example, to redirect data to other agents. Overriding messages might also be used to translate messages, but we didn't research this possibility: to use this arrow, we need to use return. If we reuse agents, we can't assume that these agents use return to return values or send messages.

We will use the arrow notation in the next chapter, but only because to make the written code shorter and easier to read. We use the arrow we assume that the called methods return results using return.

We now summarize what was done in this chapter.

5.6 Conclusion

This chapter introduced the concepts of MOP's, and CBorg's meta-level was extensively demonstrated. Some restrictions of existing MOP's were discussed and some ideas that could increase reusability in distributed systems were presented. We demonstrated that, to realize these ideas, we needed an extended MOP.

In the next chapter we use these idea's to create special components. The purpose of these components is to combine several agents into a working program. The idea is that the used agents are pieces of already existing programs which will therefore be reused.

Chapter 6

Glue Components

6.1 Introduction

Several agents are presented in this chapter: *glue components*. The purpose of these agents is to *glue* agents, which are reused from other programs, together, into a working program.

The glue agents use the meta level of the programming language to make communication between the glued agents possible. The meta level must be used, because we can't always alter the source code of agents: e.g. when agents might be located on other machines. The techniques that were presented in the previous chapter are used to implement the glue components.

The agents that are presented are the following: the Adapter, the Multicast, the Pipeline and the Dispatcher. Note that these agents are not in any way robust or foolproof. They only demonstrate the "glue" aspect.

To make these agents work we need a set of assumptions, which are presented in the following section.

6.2 Assumptions

- **The interface to agents is known.** Because we must be able to communicate with agents, we must now their interface. The interface of an agent is the set of methods, and associated parameters, of that agent. We also assume that we know the *meaning* of each method, i.e. what the methods compute and return.

- **Called methods are known.** As we just mentioned, agents must be able to communicate with each other. To control this communication, we also must know the names of methods that are invoked, so that we are able to catch these methods.

- **Methods possess a "target" parameter.** Recall from previous chapters that this "target" parameter is the agent, that will receive the result value of the called method. If we assume the existence of such a parameter, we are in full control of the control flow of agents. We can call a method and send it's result anywhere we like.

We now present several glue components.

6.3 The Adaptor

The adapter functions as message translator between two agents. Consider an agent A that needs to send data to an agent B. It is possible that B doesn't understand messages from A, as can be seen in figure 6.1.

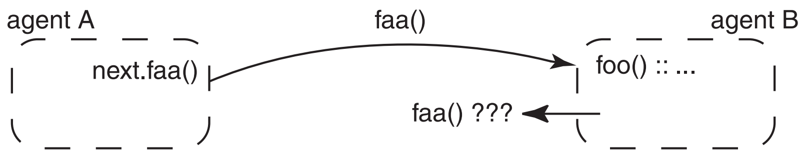

Figure 6.1: B doesn't understand messages from A

To make communication possible, we could change the source code of A or B, but this is not possible because we assume that the source code of the agents can't be changed. Therefore, we need to send outgoing calls from A to another agent that understands A's calls[1]. This other agent translates it's received calls to calls that B understands, and sends these calls to B.

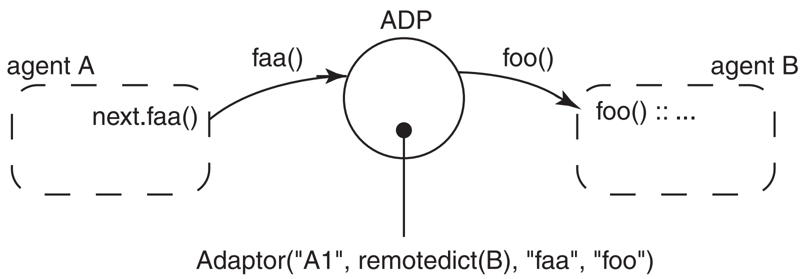

Figure 6.2: Messages from A are translated by the adaptor "ADP", and sent to B

This is exactly what an *adaptor* does. It translates methods, received from an agent, to calls to another agent. This translation is done, using a mapping, which totally determines the method $\rightarrow$ method transformation. The use of an adaptor is depicted in figure 6.2.

[1] Sending A's calls to another agent is always possible, since we assumed that methods take their receiver as argument.

The adaptor's source code is depicted in figure 6.3. Internally, CBorg's MOP is used to do the translation, just as we did in sections 5.5.1 and 5.5.2.

```
Adapter(name, targetAgent, mapping) :: {
   MessageNotUnderstood(methodName,argList):: {
       binding : Lookup(mapping, methodName);
       if (is_void(binding), error("message not understood"),
                               methodToInvoke:binding[2]);
       targetAgent.<methodToIinvoke>@(CreateArgList(bindings))
   };

   CreateArgList(bindings, argList):: {
       if (size(bindings) = 2, newArgList : argList,   {
               s : size(bindings[3]);
               newArgList[s]:void;
               for(i:1, i < s + 1, i := i + 1,  {              arg(nr)::clone()
                       a:argList[i];
                       if (is_arg(a) & (a.nr < size(argList) + 1,
                              newArgList[i] := argList[a.nr],
                              newArgList[i] := a)})
       };
       newArgList
   };

   Lookup(aMapping, aName) :: {<lookup and return binding>}

   SetBinding(aName, newBinding) :: {<lookup and replace binding>}

agentclone(name, clone())
}
```

Figure 6.3: The adaptor's source code

As can be seen, the adaptor needs three arguments for it's creation:

1. A *name*, which is needed for the agentclone function.
2. *targetAgent*: the agent where the translated calls are sent.
3. *mapping*: a list of associations. An association is of type [”invokedMethodName”, ”nameOfMethodToInvoke”]. An association can contain a third element, which contains information about the arguments. This information is only added if the arguments have to be reordered, or if new arguments have to be added. Consider, for example, a mapping: [[“boo”, “baa”, [remotedict(tecra/ses1), arg(2), arg(1)]]]. If this mapping is given to the adaptor, it will translate an invoked method “boo(6,9)” to a method “baa(remotedict(tecra/ses1), 9, 6)”

The adaptor we presented here only deals with one directional calls. Extending the adaptor to handle bi-directional calls is not difficult. We only have to provide references to *both* agents. When an incoming call is received, it has to be dispatched to the agent that did not sent the call.

We give now give an example of the adaptor's use. To make things easy, we don't consider the reordering of the arguments.

Example

Assume that we have two agents. The first agent is MatrixOperations, which computes operations on matrices and displays the results, using an agent. The second agent DataDisplayer, which pretty prints data, and We would like to perform an operation on a matrix and display the result of this operation, using the DataDisplayer. Note that MatrixOperations does not use DataDisplayer to display it's results. We assume that these two agents originate from two different applications.

An example of DisplayMatrix source code is depicted in figure 6.4. MatrixOperations is displayed in figure 6.5.

```
DataDisplayer(name) :: {
        displayMatrix(2DVector) :: <  pretty print the two-dimensional vector
                                      as a matrix>

        displayTree(nestedList)  :: < display the nested list as a tree >

        displayInt(integer) :: < display integer >;
          ⋮

agentclone(name, clone())
}
```

Figure 6.4: A possible implementation for DisplayMatrix

DisplayMatrix contains several methods. Consider, for example, displayMatrix which pretty prints a two-dimensional vector as a matrix. Another method that could be contained is displayTree which displays a nested list as a tree.

MatrixOperations contains methods that perform operations on matrices: e.g. the calculation of the determinant or inverse of a matrix.

As an example, we would like MatrixOperations to compute the inverse of a matrix and to display this result using DataDisplayer. The problem is that MatrixOperations displays a matrix using the showMatrix method: the DataDisplayer doesn't understand this method.

We can easily solve this incompatibility by using an adaptor (don't forget

```
MatrixOperations(name) :: {
        calcDet(aMatrix, target) :: {
                result : < calculate the determinant of aMatrix>;
                target.showInt(result) };

        calcInverse(aMatrix, target)  ::  {
                result : < display the nested list as a tree > ;
                target . showMatrix(result) };
        ⋮

agentclone(name, clone())
}
```

Figure 6.5: A possible implementation for MatrixOperations

we can't change the source of the agents). We just have to tell MatrixOperations to invoke target.showMatrix on the adaptor. As a result, the adaptor will invoke the correct displayMatrix method.

We show how this is done in figure 6.6.

```
d : DataDisplayer("aDisplayer");
mo : MatrixOperations("Computer");
adaptor : Adaptor ("a1", d, [ ["showMatrix", "displayMatrix"],
                              ["showInt", "displayInt"]]);

mo.calcInverse([[1, 2], [3, 4]], adaptor);
```

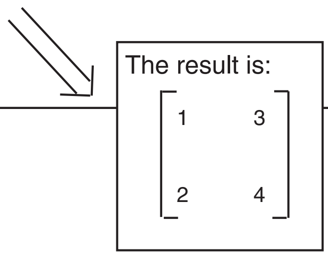

Figure 6.6: Using an adaptor ...

The adapter can be seen as glue between two agents and provides translations in a one-to-one manner. The next glue component that we present is the Multicast.

6.4 The Multicast

The *Multicast* is a glue agent that forwards a received message to a number of other agents. These agents send the results of the received message back to the Multicast agent. When all the results are returned to Multicast, they are returned back to the agent that sent the original message. The use of a *Multicast* agent is depicted in figure 6.7.

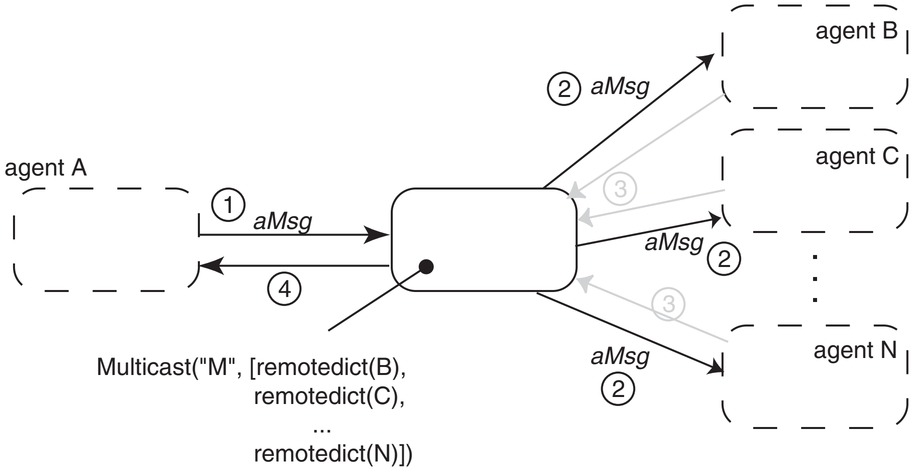

Figure 6.7: Control flow of the multicast

The Multicast agent overrides "Message Not Understood" to receive and send messages, as was demonstrated in section 5.5.4. The advantage of this approach is that *different* messages can be sent to the Multicast. If we didn't use "message not understood", we would have to write a method declaration for every method that could be received. The source code of Multicast is given in figure 6.8.

The multicast is different from the adaptor. The adaptor *translates* methods, the multicast only forwards them.

The Multicast can be used to provide concurrent computation. For example, assume that an agent A needs a an agent B to work on a set of data. To achieve this, A would have to send this data to B and B would have to return the computed data back to A. The computation that B performed could be computed concurrently if a Multicast was used. To do this, A would have to send the set of data to a Multicast agent. This component would then forward this data to other agents, which would fetch data from the received set. Each agent could work on it's own piece of data, and return it's results to the Multicast. These results could then be retrieved by the first agent.

Multicasts are often used in distributed systems. Applications which use this paradigm are, for example: whiteboards, internet games, and so on.

```
Multicast(name, targetList) :: {
    returnAgent : void;
    nrOfTargets : size(targetList);
    results[nrOfTargets] : void;
    nrOfReturns : 0;

    SetResult(idx, value) :: {
        results[idx] := value;
        nrOfReturns := nrOfReturns + 1;
        if (nrOfReturns = nrOfTargets, returnAgent.SetResults(results), void)
    };

    AllAnswersReceived :: nrOfReturns = nrOfTargets;

    SendToTargets(methodToInvoke, argList) :: {

        for(i : 1, i := i + 1, i < size(argList) + 1, {
                currentTarget : argList[i];
                currentTarget.<methodToInvoke>@argList
                        -> agentself.SetResult(i)
                })
    };

    MessageNotUnderstood(method, argList, sender) :: {
        returnAgent := sender:
        SendToTargets(method, argList)
    };

agentclone(name, clone())
}
```

Figure 6.8: The Multicast source code

We now give an example of use of the Multicast

Example

The example we present demonstrates the ease of use of the Multicast, together with the arrow notation from the previous chapter.

Assume that we want an agent X to perform a computation and that the results of this computation have to be displayed by agents A, B and C.

This can easily be done, as can be seen in figure 6.9.

Note that in this example we don't exploit the Multicast's possibility to collect the results of the invoked methods.

We now present the Dispatcher.

```
A: remotedict("/tecra/ses1");
B: remotedict("/tecra/ses2");
C: remotedict("/tecra/ses3");

MC : Multicast("mc", [A, B, C]);

X.doCalc(123, 456) -> MC.display();
```

Figure 6.9: An example of use of the Multicast

6.5 The Dispatcher

The dispatcher is a component which takes care of, as its name states, dispatching. The concept of dispatching we use is the following. An agent announces itself to the dispatcher with its name, self-reference and method name. The dispatcher associates the agent's name with the reference and method name. A request can then be made to the dispatcher, given a name and data. The name is looked up, and as a result, the associated agent and method name are retrieved. The provided data is then sent to the retrieved agent, using the associated method.

The source code of this agent is presented in figure 6.10.

```
Dispatcher(name) :: {
    Announce(agent, name, mName) :: {
        addAssoc(agent, name, Mname)
    }

    MessageNotUnderstood(method, argList, sender) :: {
        assocName : argList[1];

        if (not(is_string(assocName)), display("Message Not Understood"), {
                binding : lookup(assocName);
                if(is_void(binding)), display("Message Not Understood"), {
                        targetAgent : binding[1];
                        methodToInvoke : binding[2];
                        data : RetrieveData(argList);

                        targetAgent . <methodToInvoke>@data
                )}
        })
    };

agentclone(name, clone())
}
```

Figure 6.10: The source code of the dispatcher

An agent has to announce itself to the dispatcher, using the Announce method, as can be seen from the source code. Note that an agent might want to announce itself using another method. In this case an adaptor can be used to translate this method to Announce.

When the dispatcher receives a message that is different from announce, it assumes that this message must be dispatched. Furthermore, the dispatcher assumes that, if a message is received, the name it has to lookup is the first argument of that received message. If this is not the case, or if the name is not associated with anything, then the dispatcher will report "Message Not Understood".

Unfortunately, the application of the dispatcher, a simply web server, is not given, due to a lack of time. We do have an example for the last presented glue component, the Dispatcher, which is presented now.

6.6 The Pipeline

In chapter 4 we gave an example of a program that used a concept named pipeline. Recall that a generator sends data to an agent. This agent performs a computation on this data and sends the results of this computation to another agent. The latter agent also performs a computation and sends it again to another agent. This goes on until a receiving agent in the pipeline is a collector. This collector terminates the pipeline and, for example, displays the received results.

Communication between agents worked fine if all the agents in the pipeline were the same, because the agents knew what methods had to be invoked. A problem arose when we wanted to add other agents to this pipeline: these new agents didn't use the same methods. As a result the "old" agents would send messages to the "new" agents, which the "new" agents didn't understand. Therefore, to make communication possible, we assumed that all agents understood a "universal" receive method. The pipeline we present circumvents this problem by taking the method names that have to be invoked, as arguments. Consider, for example,
[agentA, ["primeCheck", "display"]]. With this information at hand, we can implement that agentA is called with a method named "primeCheck", and that the result of this method needs to be returned using the "display" method.

This is exactly what the CreateReceiver method does in the pipeline's source code, presented in figure 6.11.

CreateReceiver takes three arguments: a target, a receiver and a binding of two method names. When a message is sent to this agent, the first method of the binding is invoked on the target, which was provided as argument. This invocation yields a result. The second method name of the given binding is then used to invoke a method on the provided receiver argument, with the previous result as argument.

```
CreatePipeline(agentsAndBindings) :: {
  links[size(agents)] : void;
  bindings[size(agents)] : getBindings(agentsAndBindings);

  CreateReceiver(target, nextReceiver, currentBinding) :: {
      MessageNotUnderstood(method,argList) ::
            target .< currentBinding[1] > @ argList
                  -> nextReceiver. < currentBinding[2] >;

  agentclone(clone(),na)
  };

  MessageNotUnderstood(method,argList) :: {
      currentBinding : bindings[i];
      checkBinding(currentBinding, method);
      agents[1] .< currentBinding[1] > @ argList
                  -> links[1] . <  currentBinding[2] >
  };

 addAgent(agent, binding, idx) :: {
      < create new CreateReceiver >;
      < add the agent at position idx>
      < set up the right reference>
 }

  for(i:size(agents)-1, i > 0, i := i - 1,
       links[i] := CreateReceive(agents[i+1], links[i+1], bindings[i]));

 agentclone(clone(),name)
 }
```

Figure 6.11: The Pipeline source code

The MessageNotUnderstood method is used to the start the generator (agents[1]), again using a binding to invoke the correct methods.

The Pipeline can also add new components at run time. Again, we have to provide a binding which is needed by CreateReceiver to invoke the right methods.

The addAgent method is not implemented: we simply have to add a new element in a linked list structure.

The remaining for loop creates a CreateReceiver agent for each agent.

We now give an example of use of the pipeline.

Example

An example of the creation of a pipeline is given. Recall that in chapter 4 we had to use the following code in figure 6.12 to create a pipeline.

```
console : remotedict("tecra/ses1");
c : CreateCollectorAgent("c")
two : CreatePrimeCheckAgent("2", console, c);
one : CreatePrimeCheckAgent("1", console, two);
g : CreateStartAgent("g", 5, one);
```

Figure 6.12: Making a pipeline without the CreatePipeline agent

We remarked that adding a new different component to this pipeline was impossible without changing the agents' source code. In addition, we had to re-initialize all the components to set up the references again. With the pipeline that we presented we don't need to the all this anymore.

Adding an agent can be done at run time. In addition, we can add any agent (that fits in the pipeline concept) we like, because we provide names of methods to make the communication compatible with the original agents. We give an example of this in figure 6.13.

In this example we put an agent which displays the passing numbers between the first and the second prime checker.

```
p : CreatePipeline([CreateGenerator("g" )],
        [CreatePrimeCheckAgent("1", console, console), ["checkPrime",
                                                        "checkPrime"]],
        [CreatePrimeCheckAgent("2", console, console), ["checkPrime",
                                                        "checkPrime"]],
        [CreateCollectorAgent("g"),  ["checkPrime", "none"]])

p.addBinding([DisplayAgent("displayer"), ["display", "checkPrime"]], 3)
```

Figure 6.13: Using the pipeline

This pipeline can also be reused with the FFT pipeline from chapter 4.

We give a conclusion for the work in this chapter.

6.7 Conclusion

In this chapter we demonstrated how CBorg's meta level can be used to create reusable glue components.

The glue components we presented were:

- The adaptor, which translates methods to other methods, using strings.

- The multicast, which forwards every message it receives
  to other agents.
- The dispatcher, which dispatches messages to agents depending on a given key.
- The pipeline, which implements an abstraction that can be used with different agents[2].

We also showed how these components could be used through simple examples. From these examples it should be clear that these glue components can be reused.

We now discuss the achievements of this thesis, together with possible future work.

[2] It should be noted that this pipeline was not explained deeply enough, due to a lack of time. For the same reason, an error or inconsistency might be present in this agent.

Chapter 7

Conclusion and Future Work

7.1 Introduction

The initial goal of this thesis was to extract the control flow out of distributed programs. We wanted to move the control flow between processes in distributed programs to the application level. This extracted control flow would then be described by Control Flow Components. In fact, we wanted to specify at the application level which components interact with each other, without having to take care of bothersome implementation details: the CFC would glue those processes or components back together in a simple way. Because control flow aspects were the main focus, this thesis was initially titled "Control Flow Components in Distributed Programs".

Unfortunately, we didn't reach our goal because there were much difficulties in glueing even simple components together. This imposed a serious delay to our research. Nevertheless, we were able to deliver a thesis which shows what kind of features are missing in distributed programming languages, to easily combine components together.

We explain why a good adapted Meta Object Protocol (MOP), or Meta Level Architecture is needed in distributed systems. Furthermore, we explain what kind of features this new Meta Level Architecture may need and finally we show how this Meta Level Architecture can be used to write glue components. Note that this thesis is not concerned about error handling and fault tolerance. Another problem this thesis doesn't want to tackle is concurrency problems.

7.2 Summary

Part 1 of the thesis are the preliminaries. We first introduce some general concepts which should be understood before proceeding to the remainder of the thesis. We discuss processes, communication between processes and currently used paradigms. Afterwards, the experimental environment CBorg is presented. In this language we can easily implement communication between processes. As a result, not much time had to be spent writing communication code. We also explain that an agent in CBorg can be seen as a process, which is able to communicate with other processes or agents in an asynchronous way. We used asynchronous communication because this solves a number of concurrency problems and it is a good simulation of wide area networks.

After the foundations of the thesis have been set, the second part of this thesis begins. We introduce some programs in chapter 4. We analyze these program and explain why they can not be easily reused. We try to make them reusable, but it turns out this is rather difficult to do. We want, for example, to reuse a component in different programs, without the need to change that component's source code. We conclude that to achieve a high degree of reusability, we have to make use of the meta level of a programming language.

In chapter 5 we discuss the flaws of distributed meta object protocols. Afterwards, we extend CBorg's meta level with some features and primitives, needed to implement easily reusable agents. Recall that we wanted to pass extra arguments to functions, without the function knowing this. We wanted to be able to know the callers of functions. We also wanted to catch "message not understood"s and the ability to add return parameters to functions. Finally, we wanted to pass continuations to other processes. We summarize this now:

- **Parameterized application and message sends**

  These feature allows us to adapt and create messages. For example, we can create a message from a given string. This string is the name of the method that the message invokes.

- **Hidden parameters**

  Hidden parameters can be used to pass a function's caller. These parameters can also be used to pass arguments to a process which is called further in the computation. Because the other processes don't see these additional parameters, their signature don't have to be changed.

- **Overriding "MessageNotUnderstood"**

  This feature allows an agent to catch incoming calls. These calls can then be redirected to another agent or translated to other messages. Without this feature, it's almost impossible to write glue components in a simple manner.

- **The arrow and return primitives**

The arrow can be seen as passing a continuation: a message is sent to an agent and the result of this message is passed to another agent. We also added a return parameter which worked together with the arrow. Unfortunately, we didn't have time to implement this arrow exactly as a continuation because we ran out of time. Nevertheless, we presented the idea and showed how it could be used.

With these extensions to the meta level architecture we were able to easily write glue components. A few glue components are presented in chapter 6: an Adapter component, a Pipeline component, a Dispatcher component and a Multicast component. The clue of this chapter is that these glue components can combine components that can't be altered. In addition, these glue components can easily be reused in other applications. It should be clear that this reusability couldn't have been attained without a good meta level architecture.

7.3 Future Work

The changes we proposed to the meta level architecture are not as well defined as we would have liked. The arrow and its associated `return` operator, for example, have not been researched deeply enough.

If we had to rewrite this thesis from scratch, we would focus more on small but rigid and powerful changes to the meta level architecture. The resulting primitives would tightly interact with the existing meta level architecture.

Another worthwhile path to explore is how we can move or change synchronicity out of the code. In this thesis we always worked always with asynchronous calls, which made glueing easier. If communication between components would have been synchronous, i.e. components which lock and wait whenever they do a call, the combination of components would have been much more difficult.

Also we would have liked to investigate our initial goal, control flow extraction of distributed programs, a little deeper. Recall that a CFC should have the following features:

1. they have to glue unchangeable components together,

2. they have to be reusable,

3. they should determine the control flow of a program.

A solution for problem 3. could unfortunately not be found, because much time was needed to solve the first two problems. It would therefore be interesting to investigate problem 3., and for that matter the whole CFC concept, more deeply.

7.4 Related Work

We discuss related work, concerned with the extraction of control flow, and concerned with glue components.

7.4.1 Aspect Oriented Programming

Extracting the control flow out of a distributed program may sound similar to aspect oriented programming. There are indeed some similarities between AOP and extraction of the control flow out of distributed programs. For example, the cross cutting concerns of error handling and synchronization sounds more like aspects than modules. Nevertheless we didn't want to think in AOP terms, for the following reasons.

1. We are talking about *active* components/processes. Something which aspect oriented programming almost never tackles.
2. Components might be online services which we cannot change in any way. We don't have the source of these components and as a result, we can't alter their runtime behaviour.
3. Aspect weaving is mainly considered to be a compile time process. Interfacing between processes is a dynamic process which cannot be compiled and definitely not weaved beforehand.

7.4.2 Metaobject protocols and atomic data types.

R.J. Stroud and Z. Wu try to separate application code and synchronization code. They achieve this by implementing synchronization with the meta objects. Changing synchronization aspects of an object is then done by associating an object with another meta object. No other information about distributed communication was provided, so this model was only interesting as general knowledge.

7.4.3 Composition Filters

Composition filters [15] are wrappers around objects. These wrappers handle all incoming and outgoing traffic, and can be used to translate calls to other calls. In this way, communication between "incompatible" objects can be made possible. This is also what we have been researching in this thesis. The applications domain of composition filters is different from our's though.

Composition filters try to increase an object's reusability, e.g. by allowing multiple views of objects, by making persistent objects, by allowing transactions and so forth. Our goal is to glue components, which are processes in distributed applications, together.

The composition filter object model doesn't take distribution and active components into account. Therefore it wasn't the ideal model for our research.

Bibliography

[1] G. Couloris, J. Dollimore, T. Kindberg: *Distributed Systems, Concepts and Design, 2nd Ed.*, Addison - Wesley, 1994.

[2] G. Jones: *Programming in occam*, Prentice-Hall, 1987.

[3] M.G. Hinchey, S.A. Jarvis: *Concurrent Systems: Formal Development in CSP*, McGraw-Hill International Series in Software Engineering, 1995.

[4] M. Campione, K. Walrath: *The Java Tutorial Second Edition: Object-Oriented Programming for the Internet*, Addison - Wesley, 1998.

[5] D. Moreira: *Distributed Programming using Java*,
http://java.icmc.sc.usp.br/java_courses.

[6] T. Hondt: *The Pico page*
http://pico.vub.ac.be

[7] W. Van Belle: *Mobile Multi-Agent Systems*,
http://progwww.vub.ac.be/poolresearch/cborg/.

[8] The OMG: *What is CORBA?*
http://www.omg.org/corba/whatiscorba.html

[9] jGuru: *Fundamentals of RMI*,
http://developer.java.sun.com/developer/...
... onlineTraining/rmi/RMI.html

[10] K. Verelst, W. Van Belle, T. D'Hondt: *Synchronization and Communication in Mobile Multi Agent Systems*, Vrije Universiteit Brussel, 1999.

[11] W. Van Belle, T. D'Hondt: *Agent Mobility and Reification of Computational State*, Vrije Universiteit Brussel, 2000.

[12] , F.M. Hoffman, *An introduction to Fourier Theory*

[13] , T. Cormen, C.H. Leiserson, R.L. Rivest, *Introduction to algorithms*, The MIT Press, 1999.

[14] , R.J. Stroud and Z. Wu, *Using Metaobject Protocols to Implement Atomic Data Types*, University of Newcastle, 1995.

[15] , L.M.J. Bergmans, *The Composition Object Model*, University of Twente, 1994.