Vrije Universiteit Brussel
Faculty of Sciences
2001-2002

Distributed Serving of Applications Using Agents

A thesis submitted in fulfilment of the requirements for the degree in Computer Science.

By: Koen Bailleul

Promotor: Prof. Dr. Theo D'Hondt
Advisor: Werner Van Belle

Abstract

Security and copy-protection is of growing concern these last years. Our question is: is it possible to enchance security and copy-protection by dividing an existing application into agents? The process should be transparant for both user and programmer.

Contents

Preface

Organization of the Thesis

In a nutshell this document tries to go deeper into a fairly new concept. We try to attack the lack of copy-protection in existing applications. Many programs get copied because of this lack of security. It is known that protecting the application with a key does not work.

Some preliminary information on the chapters to come will now be provided. So readers who already know most of the stuff described in the first chapters may of course skip these chapters.

Chapter 1, is mainly an introductory chapter. It tries to give answers to a few possible questions, what are agents, what is a Mobile Multi Agent System and how do the previous relate to Borg and applicationservers?

Chapter 2, tries to introduce us to the world of the mobile multi agent system Borg and its predecessor Pico. For people who have never heard of Pico or Borg, this chapters give a small introduction on the syntax and grammatics of these programming languages. This introductionary chapter gives the reader the opportunity to understand the examples which are presented throughout this document. Readers who have previously visited this world, may skip this chapter.

Chapter 3, summarizes the problems we may encounter when programming for a mobile multi agent system (or other distributed systems). In this chapter we may learn that latency and bandwidth are not the only problems we have to face. Readers with a more than fair knowledge of distributed systems and their associated possible problems, can of course skip this entire chapter.

Chapter 4, tries to explain what we are going to do with applications. This chapter contains a short overview of the steps that will be taken from a normal (local-)written application in Borg to an agent based (distributed) program. In this chapter the 4 steps of the transformation process get a short introduction.

Chapter 5, describes how the first program transformation, the abstract distributed memory layer, is added to Borg functions. Thanks to this memory, layer functions are able to share variables.

Chapter 6, in order to cope with the fact that agents use an asynchronous way of communicating, functions, that were written without knowledge of agents, need continuations.

Chapter 7, this small chapter tells us why we need a control flow graph in order to be able to divide an application.

Chapter 8, tries to explain how we actually divide an application into agents. This chapter uses the thechniques discussed in the three previous chapters.

Chapter 9, in this chapter some references to possible future work are given. There are several possible performance enhancements possible for the implementation given in this document.

Chapter 10, presents a conclusion on the idea presented in this thesis.

Notations

Algorithms will be written like this

```
function([parameters]*)::
{
   <expression>*
}
```

References to functions or methods from the Pico and/or Borg language are noted as *function()*, that is: we do not mention which parameters they possibly could have, because most of the times these parameters do not matter; if it does matter we will of course mention it.

Acknowledgments

First of all I would like to thank my dearest sweetie-π Kristel.

Many thanks to my mother for supporting me all these long years and for constantly nagging about the progress of my thesis.

Dirk van Deun receives many thanks for the great and loveful care for Tux, the penguin.

Of course, I wish to thank the creators of the various tools I used during the last years. Especially the creator of TEX and the authors of LYX.

1 Introduction

1.1 A Short Introduction to Agents

Agents are everywhere, and there are widely varying definitions of the term ”agent”. Some refer to agents when they think of a piece of software which monitors the state of a computer others think of agents when they try to defeat the ultimate incarnation of evil in a computer game. Others see the agents crawling on the Internet trying to retrieve and process data. As one can imagine, the term agent has become more and more a rather popular buzz-word and is a very widely used concept, because the examples stated above are nothing but the top of the ice berg.

A dictionary lookup of the term ”agent” follows:

> "An entity authorized to act on another's behalf."

This definition comes closer to describing the type of agent we will refer to in this thesis, in this document we will define agents as dynamic autonomous active software components. The dynamic stands for the fact that an agent should decide for itself what to do depending on the environment it resides in. For example, consider an agent crawling on the world wide web looking for information, not every web-page has been designed the same way. Some use databases to publish data, others use CGI scripts, etc.

The word autonomous tells us that the agents should be able to take initiative and exercise a non-trivial degree of control over its own actions. For more on the attributes of agents see [4].

1.2 Mobile Multi Agent Systems

Now, imagine a lot of these agents cuddling together, interacting with each other. Imagine that several of the bunch want to migrate to other computers because they feel it gets overcrowded. Well, that is a very rough description of a Mobile Multi Agent System. Mobile agents are computer programs that can move (or migrate) to different computers. The mobile agents we will use in this thesis, will decide for themselves when and where to migrate. They are even able to migrate to other computer in the middle of the execution of a function.

For example, consider an agent which has the duty to collect data on the Internet, it moves itself to a computer it wants to examine. After completing the designated task, it migrates to another computer, and when the right criteria are met, it returns to the computer of departure. The reader may point out, that this is the behaviour of a virus roaming around on the Internet, but a virus simply copies itself to other computers, it does not migrate.

1.3 Borg

Having explained what agents are and what a Mobile Multi Agent System (MMAS) is, I will now introduce a MMAS which is being developed at the VUB. The environment is called Borg.

A more thorough description about the syntax and semantics of Borg will be given in the next chapter. The origin of Borg is Pico [1], a programming language invented and implemented by Theo D'Hondt. Pico is an interpreter for a simple and clear language, it is used to teach non-computer science students a few basics of what it is to be a computer science student [2].

Pico is easy to learn and to understand, not only because the core language only contains about 60 primitives. It is a complete language, as expressive as any other imperative language. The name of the language, Pico (as in very small), refers to the idea of creating a small language with a very general impact. With Pico many computer science students of the VUB were introduced to meta-programming. Pico is interpreted, which offers more flexibility to the user than a compiled language.

The first enhancement to Pico was the addition of several features which allow programmers to work in object oriented way. Thus, Pico was mixed with prototype based OO. Prototype based OO was chosen over class based OO, because non-computer science students would have less troubles understanding this kind of OO.

After Werner Van Belle finished his thesis, he started to expand the abilities of Echo, a few years of hard work later the result was a very promising looking mobile multi agent system, called Borg. With Borg a programmer could work with agents and write distributed programs in a lean and easy way. Everything programmed in Borg is an agent, so if there is a new editor window created, internally there is a new agent and another new agent, which takes care of the output of the first mentioned agent. This way we are able to send the output of the execution of a program to the display of another computer.

The main purpose of Borg was to create a simple Java alternative, an alternative that allowed the creation of agents which had the ability of strong migration and a asynchronous RMI. Java was not suited for this purpose.

Borg allows us the create agents, which can decide for themselves if they want or have to move to another computer. Also agents are able to clone itself, for example we could make a program which computes fractals in a distributed way. The program spawns one agent, which immediately moves itself to another (more powerful) computer. Because fractals are ideal to be computed in a distributed way, the agent which computes the fractal could spawn itself and thus use the processing power of more computers.

But Borg is still in a constant fase of development. Every year students of the VUB try to add lots of interesting features. For example, Borg now runs on Linux, Windows, Dos and Palm.

More information on the syntax and semantics of Borg can be found in chapter 2 on page 6.

1.4 Application servers

As the reader might suspect, application servers are servers which provide programs to users. Although, not in the “file-server” sense of the word. When a user wants to use a program like a word processor, a spreadsheet or anything else, the program will be downloaded and executed locally on the workstation. That way, the workstations have become more than a mindless terminal. This kind of software-serving is a rather new concept. It allows software vendors to use some of the now unknown capabilities of the Internet. Customers of a software firm would not have to buy a software package, but rather lease it. In return for a (hopefully) small fee, they should get access via the Internet to the software. Because the software is still kept on servers close to the company which created it, updates are easy to apply (it is known that most users don’t apply the latest security patches because they don’t have the time for it). Even if a new version of an application becomes available, it should be instantaneously available on the servers for every user.

This technique of software leasing has other advantages: users who need to be on the road a lot, should be able to run programs without having to connect to the company network via VPN; also, this kind of software should occupy less disk space on the local hard-drive.

Unfortunately, this kind of application serving has some disadvantages: it requires a rather fast connection between the server and the user(s); also, because of the fact that most networks are a shared medium, it could be easy for users to copy-and-store a program that is being transmitted. In this thesis we mainly focus on the concept of improving copy-protection for this kind of application serving. In chapter 9 I’ll present some ideas on performance enhancements.

Pirate copies of commercial software has been a growing concern for software makers around the world. We try to alleviate this burden from the software firms by dividing applications and serve them as agents to the users.

1.4.1 Why are we dividing applications?

In an ideal world, there would be servers which distributed applications like word processors, spreadsheets, etc. without any security. Unfortunately, this world is far from ideal.

There are several existing techniques that circumvent users from copying software, ranging from a simple serial number to the new way of registering windows software (enter a serial number, get some output, call Microsoft and provide them this output, then they will provide another key of plus-minus 50 characters which allows the users to work on his very expensive paid Windows XP... which only allows a user to change hardware 6 times before having to renew the license key); or the usage of dongles on the serial port of a computer to server-based authentification (for example: Mathematica).

In this thesis I'll try to explain a different approach to the copy-protection concept.

When I went to my advisor for some advise on the topic of my coming thesis, he threw several interesting ideas at me. One of these ideas was trying to secure a program by dividing it. and offering only parts of the code to the end-user packaged in agents.

Since, apparently, nobody has done this before, we will investigate the possibilities of creating a such a copy-protection system. Also what are the advantages and disadvantages of implementing this kind of copy-protection?

Now, the idea: every time a user wants to use an application from the application-server, the application has to be divided 'on the fly' into several agents. Some of these agents are to be sent to the user's computer, other have to remain on the server. Since there are a lot of smart hackers who find it intriguing to crack copy-protections simply for the fun of it, we should make it a little bit harder for them. Therefore, every time an application is requested, the application should be divided in other parts then the previous time. Also, upon each execution of a program there should be a percentage of agents residing on the server. This way, we make it harder to create a complete copy of the program. The larger the program, the harder to crack, because statistical there are more possible options to divide the application.

1.5 Related Work

Although the concept of dividing existing applications into several parts and serving these parts to users is a pretty new, most of the concepts and methods used to come to this end-result are not. Before I started writing this thesis, I received several pointers from friends which might be interesting.

Chapter 4 contains a more in depth explanation of the method I will use to transform existing applications written in the Pico language to an agent based divided application[1].

In the first stage of the transformation process we will perform a rather "dumb" transformation which transparently extends existing Pico code with a new memory model. The second stage is already more interesting; we will add call-back functions to the code, these call-back functions are necessary to allow the functions of an application - originally written for local use only - to still receive results when they are transformed into agents. The third stage then constructs a control flow graph of the application, which will be used in the fourth stage to transform the application into several agents.

To summarize; in order to create this thesis, a more than fair knowledge of the following topics is needed. Since an introduction to all this topics is too long to mention in this document, I would like to direct the reader to the bibliography section of this document.

- Several simple code transformations on the abstract grammar of a Pico program
- Continuation-passing style (CPS) transformations
- Building a control flow graph of a program
- Program Slicing (using a Control Flow graph)

[1] There is no need to worry if most of the terms used in this subchapter might appear unfamiliar, most of them will be explained in depth in chapter 2

2 Borg

In the first chapter the reader got a fast introduction to what we conceive as a mobile multi agent system (MMAS). In this chapter there will be more information on the internals, syntax and semantics of our mobile multi agent system, Borg.

To be honest, most of this chapter has been inspired by the excellent introduction Karsten Verelst gave in his thesis [3], he has written a very thorough introduction on this subject. In this chapter I tried so summarize the chapter in Karsten Verelst's thesis, and only the things relevant to this document will be described.

For even more information, I can point the reader to the the web-pages of Pico and Borg [1, 2]. Readers who are already familiar with Pico, Echo or Borg can skip this chapter.

2.1 Why Borg

Why did we not use Java or any other existing language to implement agents? Because most of the current languages are not at all developed with the concept of agents and distributed computing in mind. At first glance, Java looks very promising, it is platform independent and has the Java RMI and automatic serialization of objects. Java also supports threads[1], these threads should allow us to run several agents concurrently in Java. Of course there is a downside; Java does not allow us to easily migrate threads to another computer. Also, a major drawback of extending a current existing and well used language is that it results in lots of syntax-extensions and troublesome constraints.

In a mobile multi agent system we need the possibility of migrating an agent to another computer. By migration we mean the movement of an agent to another location in the network and transparent continuation at the point before the migration occurred. If we want to migrate an agent, we need to serialize this agent, transfer it to the other side, deserialize it and put it back into an active state. Serializing and deserializing is not as simple as it sounds, it is one of the most difficult topics in agent-migration.

To serialize a running application, we need to be able to completely copy the runtime stack, save the execution pointer and copy the memory used by the application. The copy of this data then needs to be reformatted so it can be sent over a network (this is

[1] Threads in Java are concurrent running Java processes, they share the same dataspace.

called serializing). Upon arrival on the destination computer, the serialized data has to be deserialized, so it can be put back into an executable state.

Because the transfer of all the aspects of an agent is rather difficult and expensive, we distinguish between strong and weak migration. Strong migration means the transfer of the agent's code and its complete state, whereas weak migration can be defined as every migration that is not strong.

This technique of transparent migration is difficult in Java, it demands several extensions to the Java virtual machine because only a limited access to internal and native information is available (as Java classes are being interpreted and executed securely by the virtual machine). So although Java is being promoted as an allround language, we can see from some further investigations that Java is not really suited to be an MMAS (although some brave people try to convince us otherwise [8]).

On the other hand, Borg has become a mobile multi agent system that gives us strong migration and asynchronous message sending, view this together with the fact that Borg allows us to work on the abstract grammar of a program in an easy manner, so Borg is a perfect tool for working with agents.

2.2 Pico and Echo

In Pico every new window created, creates an extra process in the Pico interpreter, thus enabling a kind of multitasking. This means that we can run several Pico processes concurrently.

Another of the internal features, is the internal garbage collector; this means we don't have to bother students which are learning this easy language, with the fact that in other languages variables have to be freed after usage in order to reduce memory leaks.

2.2.1 Syntax

Before we go deeper into the syntax and semantics of Pico, Echo and Borg, I will give an explanation of the notation used. If you see code that looks like this

```
<expression>
:<value>
```

This means: we have an expression *<expression>* and the return value after having evaluated this expression is *<value>*.

As most current programming languages Pico uses the infix notation. Although some native functions in Pico as +, -, etc are also defined as functions, so they actually can be used in both notations:

```
+(3, 4)
:7
3+4
:7
```

2.2.2 Atomic types And Primitive Data Structures

Atomic types

Pico has three atomic data types: numbers, floating points (or fractions) and text.

```
“this is text”
:this is text
42
:42
3.1415
:3.1415
```

Variables in Pico are not typed, because this is easier to grasp for people who have little or no programming experience. The downside is that the programmer always has to remember what has been assigned to which variable. To positive side is that a programmer is able to assign everything to a variable, even functions.

Primitive Data Structures

Tables are the only primitive data structures in Pico, because they are easy to understand for computer illiterate. Since all other data structures, such as graphs or linked list can be implemented using tables, there is no need to complicate the language.

```
t:[1,2,3]
:[1 2 3]
size(t)
: 3
```

Special Constants

Every language has some special constant variables like true, false, EOF, etc. In some languages these constants are definitions which the pre-compiler replaces with the actual values during the first stage of compiling. Naturally these constants are most of the time a mere extra abstraction layer between the programmer and the actual values used in the language and compiler.

Pico has not many of these built-in constants:

```
void
:<void>
eoln
:
:
:true
:<native true>
false
<native false>
```

These constants need not much explanation, the *void* constants is mostly used in the same way as NULL in C/C++, and *eoln* is the special end-of-line constant, as endl used in C++. As one can notice in the code-examples above, the evaluation of true and false tells us that they are native functions. This is because true and false are implemented as church booleans (λ-calculus).

2.2.3 Native Functions And Control Structures

Most of the native functions a programmer has access to in other imperative languages, also exist in Pico. There is support for the simple mathematical functions (+,*,/,...), advanced mathematical functions (sin, ^, ...), comparing (=, <, >, ...) and the usual support for control functions (while, until, if; then; else, for).

Because the variables in Pico are not typed, sometimes it comes in handy to determine what type the value of a variable has. In Pico there are lot of identifying functions (is_number, is_function, ...). When Werner Van Belle extended Echo to Borg, he made the identifying functions of Pico more consistent and added a few functions so that a programmer could identify the extensions made to Borg.

The code-transformations which will be used later on in this thesis use a lot of identifying functions (in fact, all of these are used).

2.2.4 Declaration, Definition And Assignment

There are two different ways to declare a variable in Borg:

```
x:1
:1

y::2
:2
```

```
x:=5
:5
y:=x+1
 -> <error>
```

In this example the x is a normal variable and y is defined as a constant. If a variable is defined as a constant with "::", it can never be reassigned. The value is initialized at creation; the value being the expression following the "::". Most of the time, constant variables are called declarations, whereas normal variables are called definitions.

As in any other normal programming language, there are 2 operations possible on variables (and one on declarations). The first operation is a fairly trivial one, namely the reference and the other is the assignment ":=". Clearly, one cannot change the value of a constant, so the only valid operation on a constant is to make a reference to it. Because Borg has no typing, it is also possible to assign tables and functions to a variable.

Functions are also defined using the ":" or "::" syntax. Because most functions don't change during execution, it is advised to declare functions as constant.

When declaring a table, the difference between definition and declaration remains. There is one catch, if a table is defined or declared, the argument is evaluated n times (with n being the size of the table). This way we are able to create tables with a dynamic size.

```
tmp[4]:void
:[void void void void]

{
   n:0;
   t[7]: n:=n+1
}
```

Since Pico reads from the back to the front, the initialization will return the following:

```
[1 2 3 4 5 6 7]
```

One can reference into a table as follows:

```
t[3]
:3
```

2.2.5 The Memory Model

Now a quick word on the memory model of Pico; every function in Pico has an environment containing a list of variables names and their respectively values. This list is suitably called a dictionary. There is only one special dictionary, the global environment, which contains a list of all the natives. The memory model is basicly a linked list of dictionaries, with the root being the global environment.

Every time a function call gets executed, a dictionary is added, this dictionary contains a link to the previous dictionary and the functions' parameters. Obviously we get a stack of dictionaries when we follow the links in the dictionaries (this stack is the scoping). When we follow the links upwards, we always end in the global environment. In Pico we have a static scope, where the upper dictionary of a dictionary is the scope in which the function has been defined.

2.2.6 Object Oriented Extensions to Pico: Echo

Class Based versus Prototype Based Languages

In class based languages are classes the fundamental building blocks of objects. A class itself doesn't do anything, it is a blueprint, a prescription for objects created from that class. After instantiating from a class, you get objects. Objects vary in what values their variables have, this is also called the object's state. What variables and methods the objects have, is defined by their class. Classes are sometimes called prototypes for objects, but we reserve that word for another paradigm in the object-oriented paradigm. In brief, a class-based language is based on two fundamental entities: classes and instances.

Prototypes are the constructs of the new prototype-based languages, sometimes also called object-based. In these languages we simply write the code for an object, this object then can be cloned to have another concrete, fully functional object. These languages no longer have classes, so obviously the objects cannot be subclassed. Briefly put, in prototype-based languages there are only objects. This simplifies the language a lot.

An example; say we have a database which contains some information on persons. In a class based language we would create a person-class. This class would have several variables, which altogether contain some interesting information about a person. Every time we add data about a new person in the database, we create an instance of the person-class, the object gets stuffed with the new information and then is added to the database.

In a prototype-based language we write code for one object by writing down all the instance methods. If we want to add a person to the database we simply clone this object, then fill it with the new information and add it to the database.

It is possible in the prototype based object-oriented programming paradigm to model inheritance, so prototype based OO is equivalent with class based OO. The explanation presented in this section is of course a simplified overview of the two concepts [6, 7]. A more in depth introduction does not really belong here and there already a lot of books and papers written about OOP.

Echo

Because we live in an era where large program become unmanageable if they are not written with a object-oriented programming paradigm in mind, Pico needed to be extended so that it could work with objects. The follow-up of Pico, Echo, was created to extend the Pico language with prototype based OO. Prototype based OO was chosen, because it is easier to understand than class based OO for people without much programming experience.

```
person_data(name, age, sex)::
{
   Name():: name;
   Age() :: age;
   Sex() :: sex;
   clone()
}
:<function person_data>
person:person_data(''Jos'', 19, ''female'')
:<dictionary>
person.Name()
:Jos
```

Notice that the the definition of the variable *person* results in a dictionary. This is the part of Pico's memory which contains all the private data for that object. The possible constants defined in the object are shared between the object and all of its clones, thus making a little memory-optimization.

2.3 The Agent Environment

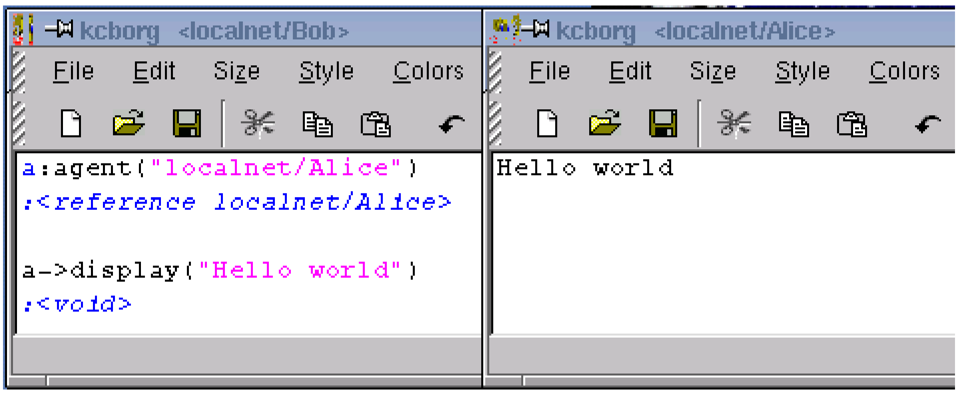
2.3.1 Agent Internals

Echo was chosen to become the base for our agent system Borg. As explained at the top of this chapter, every time a user opens a window, internally a new process (thread) is created and associated with this window. This process has its own stack and private part of the Pico memory. Two down sides of this approach are: there is no shared memory between those processes, neither is there any inter process communication.

In Borg these processes were expanded to agents, thus allowing us communication between these agents. The communication between agents happens in an asynchronous way, which creates the need for message-queues in the agents. We are now able to define the main characteristics of a Borg agent; every agent contains a stack, a local dictionary which contains the variables and the functions, and a message-queue. Also, an agent is contained in one single thread, this simplifies the agents a lot because we don't have to take care of concurrency issues in agents.

Every new created agent has a message-queue, this queue is checked at a regular interval. If the message-queue contains a message, the agent automaticly starts executing this message. If the agent receives a message while still executing another message, the freshly arrived message is stored in the message-queue for later evaluation. Borg's message system is based on the remote function call paradigm. Programmers familiar with OO-languages know that objects send messages to other object in order to let them execute methods in the latter object. If the objects are distributed over a network, we have to send messages to objects over this network, this is something similar to a remote method invocation. In Borg this idea has been extended to a remote function call. We can regard a Borg agent as a set which contains functions and objects. When one agent makes a remote function call to another agent, the other agent perceives this as an order to start the execution of a function.

An example grabbed from the tutorial part of the website of Borg [1]:



In this example we first created a reference to the second agent, after which we are able to make a remote function call to the second agent and we tell it to display the famous words *"Hello World"*.

2.3.2 The World Of Borg Agents

Names

Before we go further into the world of agents, we first need an explanation on how the naming structure of Borg agents works. This is important because we cannot simply use IP-numbers (or analogous in other types of networks) to identify an agent, this would create a dependency involving the network and we want Borg to be as portable as possible.

Borg allows us to create multiple agents on a single computer, and if the Borg environment has been set up on other computers on a network, agents are able to traverse between these environments. If a programmer wants to develop on a single computer because it lacks a network interface or he has only a few computers at his disposal, he can execute several Borg environments on one single computer. This allows us to simulate an agent system with more nodes than there are computers in the network.

In Borg every machine has a name, this name can be arbitrarily chosen and has no connection whatsoever with the networknames, IP-numbers or other properties of the computer on which the environment has been executed. Borg works with names without spaces as *Alice*, *Bob, Dirk*. This independence between names for computers with the Borg environment running and the actual machines, clarifies why Borg systems are portable. We are able to develop an agent system on a local network (even on our own computer, using the trick of multiple environments explained above), and then just install the end-result on a different network, having no connection with our own development network whatsoever.

The Beginning

It would be rather shortsighted if the only possible way to create new agents was via the editor's "New Window"-command. An agent has to be able to have complete control over its functions; an agent should be able to move itself to other computer or it should have the possibility of terminating itself after having completed a designated task.

For this purpose a few extra natives were added to the language. The first three are *agentclone()* and *agentnew()* and *new*(), both *agentclone()* and *agentnew()* are functions defined in Borg and *new()* is the native used by the two previous mentioned functions.

The first, *agentclone(object, agentName)*, is used in almost the same manner as the *clone()* native mentioned earlier this chapter. This native takes an object and a name as argument. The object will be transformed to an agent, which will be identified as *agentName*. An agent is able to dynamicly create create new agents, this is useful if, for example, an agent decides it needs more computation power. Borg has a built-in function for this: *agentclone()*; it has the same behaviour as the *clone()*, the *agentclone()*

function makes an agent of an object; the object is then made activeand possesses its own source and data space. We could also use an existing agent to clone, it then results in an identical duplicate of the original which can operate independent of the original.

Then we also have *agentnew()* which is used to create new agents (in fact, it also used by *agentclone()*). These newly created agents are like empty sheets of paper: they have no functionality yet, they only have a name. In order to give these newly created agents a meaningful live, the programmer needs to extend the dictionary of the agent. Since nobody actually uses this *agentnew()* function, because it is mainly used in internal functions (as the *agentclone()* function) , it won't be discussed any further. For more information see the Borg webpage [1].

Nobody Lives Forever

There are two ways an Borg agent can stop with its execution (and cease to exist): first we could close the window the agent is currently connected to; secondly we could use the *agentdie()* native. This *agentdie()* native is important if we want to have a dynamic agent system, it allows the agent to terminate itself after having performed a computation. Other agents are no longer able to communicate with a terminated agent, the process has been completely removed from the system's memory.

It is obvious, for security reasons, that an agent can only terminate itself. It would not be nice, if a virus-agent started to kill other agents at random. While we are on the topic: security for mobile multi agent systems is still an unsolved problem. Especially Borg has not much built-in security measures. It is possible to setup a computer which captures agents, modifies them into mallicious agents and put them back on the network. In this document I will assume that nobody will interfere with the created agents. The topic of securing mobile multi agents systems still needs a lot of research, which could fill a lot of books.

Unfortunately there is a small problem: when an agent decides to terminate itself, there is a possibility that other agents still have reference to the agent that ceased existing. There is only one way that the other agents will know when one of their fellow-agents has been terminated: the programmer has to add notification code so that before an agent makes a call to the *agentdie()* native, he notifies all other agents which have a reference to him.

2.3.3 Communication Between Agents

Agents can communicate with each other in three different ways. The most used way of communication is the remote function call (or remote method invocation as it is called more recently), lesser used are the a- and synchronous send and receive communication primitives. Since almost nobody uses these last two kinds of communication in Borg, they will not be not explained or mentioned in this document. For more information on the (a)synchronous primitives see [1].

Remote Method Invocation

Remote Method Invocation is the most commonly used method of communication between Borg agents. Since Borg tries to unify the way of communication between local and distributed objects, sending a message to a remote object does not differ from sending a message to a local object. While the underlying mechanisms may differ depending on the location of the objects, these mechanisms stay hidden to the programmer whom writes exactly the same code in order to send a message to an object. The syntax for the remote function calls is almost the same as their local variants. They only differ in one way, since the remote function calls are made asynchronous they do not wait for results. Remote functions calls always return *void* (a local function or method call returns, most of the times, a result of a computation). So a programmer has to create call-back methods.

This asynchronous way of method calls has the advantage of providing a way of concurrency, an agent (process) is able to instruct a number of agents to compute something at the same time. While in the case of synchronous message calls, the agent (process) would simply hang after having made a remote function call, waiting for a result and thus not be able to instruct the other agents to do something.

Working asynchronously has another advantage: there is no single point of failure. Which means that a computer does not depend on another, it is able to continue working (though results may differ). In the synchronous case, when an agent makes a remote function call to a distant relative, which crashes after the receival of the message, the agent from which the message originated would also hang, waiting indefinitely for the other to return a result.

Unfortunately, as allmost all good things have some down sides, the asynchronous way of the remote function call introduces a problem: returning the result of a remote function call can be rather tricky; if an agent instructs another agent to perform a large computation, it would like to receive the end-result of that computation. Since we can't simply poll for results, we have to use call-back methods. Which means that for a each remote method we want to get results from, an extra parameter is added. This extra parameter then contains the method to be used to return the result to the function from which the method is executed.

The following example, which calculates a factorial, should clarify the general idea:

```
Agent localnet/Alice:

calculateFac(i, result)::
```

```
{
   if (i>0, calculateFac(i-1, result*i), result)
};
Factorial(i, callBackAgent)::
{
   res:calculateFac(i, 1);
   callBackAgent->handleResult(res)
}

Agent localnet/Bob:

go()::
{
   remoteAgent:agent("localnet/Alice");
   remoteAgent->Factorial(10, agentself)
};
handleResult(res)::
   display('The result is: ', res);


go()
:<void>The result is: 3628800
```

Let me explain the last line of this example. When a remote method is executed it immediately returns <VOID>, the agentAlice then starts calculating and executes the call-back function on Bob which, at the end, prints the result.

2.3.4 Migration Of Agents

Until now, we have have discussed how Borg agents get their identifiers (their "names"), how they are able to create other agents and/or destroy themselves. Now some information on the topic for which Borg and other mobile multi agent systems were conceived: agent migration. This subsection contains a small introduction on how a programmer is able to order Borg agents to migrate to other Borg environments.

Although migration will be extensively used in the source code which appears later on in this document, this shallow introduction's main purpose is to give the reader a quick overview of what the syntax and semantics for the migration of Borg agents really are. Again, for a more in depth overview of these functions the reader is invited take a look on the Borg website [1].

Borg agents are not only able to migrate to other Borg environments, but it is also possible to do so in the middle of the execution of an agent. With this feature surpasses

Borg most of the other existing systems, which only allow migration at certain specified points. We call this technique of being able to migrate at any point during executing *telescripting*. There are actually two different techniques for the migration of agents:

1. **Weak Migration**
   This technique only allows agents to be moved at certain points of their execution. This because most programming languages do not allow run-time stack access. Therefore an agent can only migrate when it has finished it's computations. Then it is able to save the result, move to somewhere else and resume its work. Because this approach does not provide "immediate" migration, it is called *weak*.

2. **Strong Migration (or telescripting)**
   Telescripting got its name from a scripting language that was extended with migration features. In this case the programming environment provides facilities for migration. The saving of the stack and execution state is implemented in the environment as a feature, and thus handled automatically. This allows the possibility to save an agent's state during its computation. When the state has been saved, the object moves to another evironment, retrieves its run-time stack and continues its computation from there. Because this type of migration is issued immediately, it is called *strong migration* or *telescripting* [11].
   Of course, telescripting is much more interesting and dynamical than the first method, and it allows the migration aspect to be handled by a management layer.

So Borg supports strong migration, its interpreter is capable of serializing the data of agents and has the ability to reify[2] and serialize the computational state. These two features can easily be used to implement strong migration.

There are some natives at our disposal which allow Borg agents to perform migration in an easy manner, to name a few: *agentself*, *agentname*, *agentmove()*.

The *agentself* variable contains a reference to the dictionary of the agent itself (such as self() in Java). This is a convenient function when we try clone agents or if we have to pass a reference of a agent to a second agent in order to allow the second agent to be able to execute call-back functions on the first one.

Agentname contains a reference to the route plus the name of the agent.

Agentmove() is the main function used in a migration process. It can be called at any time during the execution process of the agent. When an agent calls this function, its execution gets suspended until it arrives at the other destination where is resumes its computation from the point before it migrated.

[2]More about reification in subsection 2.3.6

2.4 Borg's Meta Level and Abstract Grammar

Although it was not mentioned higher up in this chapter's part 2.2, most of the following explanation is also valid for Pico and Echo. But since most of the work done later on this thesis will require some knowledge of the abstract grammar of Borg, I thought it might be wise to skip the introduction of abstract grammar in Pico (and Echo) and immediately give an introduction for Borg. (Because some of the tags and other internal representations have changed, and we simply don't use Pico in this thesis).

2.4.1 Internal Representation of An Expression (or Reification of Borg code)

The abstract grammar of a Borg expression is the way the expression is internally represented. The abstract grammar is the end result after the source code has been passed through the scanner and reader. For example, the following statement

```
fun(a) :: b := a;
```

will, after having passed through the scanner and reader, result in the following table of tokens

```
[DEF [APL fun [TAB [REF a]]][ASS b [REF a]]]
```

In Borg, every expression is represented by a table which contains the abstract grammar tree of each type of expression. The following overview describes the structure of some of these expression:

Expression Type	Parameter 1	Parameter 2
Application	name	table of arguments
Message	target	method
Variable definition	reference	value
Reference	name	n/a
Function Definition	dictionary	n/a

The greatest part about Borg is that the language allows us to work directly on the abstract grammar of a function (or any other expression in Borg). We have complete control over the abstract grammar, because we are able to change the abstract grammar of an expression so that it is transformed into something completely different.

There are several ways to obtain the abstract grammar tree of an expression in Borg. One way is to use the *read()* native in Borg which allows a user to retrieve the abstract grammar tree which is associated with a given expression. The *read()* native takes an expression as a string as argument, for example consider next simple example in which we will retrieve the abstract grammar tree of the sum of two numbers:

```
gram: read("1+2")
:<application +>
```

The variable *gram* now contains the abstract grammar of the operation which computes the sum of the two given numbers.

If we take a look into the table above, we can see that an application expression takes two parameters. Consider the previous example in which the abstract grammar of the sum of two numbers was stored in the *gram* variable (as a Borg table). We are able to retrieve the name of the type of the application and its arguments as follows:

```
gram[1]
:<reference +>
gram[2]
:<table>
gram[2,1]
:1
gram[2,2]
:2
```

In order to include variables and/or constants of expressions obtained using the *read()* native, we have to evaluate them. This evaluation is done using the *eval()* native, which returns the result of the evaluated expression. For example, using the result from the previous two examples, we get:

```
eval(gram)
:3
```

Using this extra-ordinary option in Borg, we are able to perform transformations on code for which do not need the source-code file. Every type of expression can be read and evaluated using the meta-level of Borg this way. Such creation or manipulation of expressions is not possible in many other languages.

Of course it would be much more useful if there was to way to determine which expressions are stored in an abstract grammar tree without having to check it manually in the Borg environment. For this purpose we have a set of predefined functions in Borg which are able to check the tags of an expression. But first more about these tags.

Tags

The tags of expressions are an important part of the Pico and Borg interpreters. Every expression type is identified by a tag. One way to retrieve the tag of an expression is using the *tag()* native. This *tag()* native takes an expression as argument and returns the tag associated with this expression. In the next example we first retrieve the tag of a number, followed by the tag of a function after which we retrieve the tag of a native:

```
tag(2)
:32
tag(display)
:28
tag(+)
:27
```

An overview of the most common used expressions and their respective tags are shown in the next table[3]:

Expression	**Tag**
Function	28
Native	27
Assignment	7
Definition	6
Declaration	5
Reference	1

As mentioned in the previous subsection, we have a predefined set of functions at our disposal to automatically determine what the type of a specific expression is. Since their will be a lot of code transformations, which work directly on the abstract grammar of Borg expressions, in this document an example of these functions is required.

During the apprenticeship period a *PrettyPrint()* function was written. This function took an expression as argument and printed out the abstract grammar that was used in this expression. While this function is too large to write down here, a small code-snippet will certainly be sufficient to get the general idea.

```
PrettyPrintLight(Ag):: {
   if (is_ref(Ag), display(''Reference : ''+text(Ag)),
   if (is_tbl(Ag), {
     display(''Table     : '');
```

[3] A more detailed list can be found on the Borg webpage [1], just follow the links to the sources.

```
      PrintTable(Ag);
    },
    if (is_nbr(Ag), display(''Number     : ''+text(Ag)),
    if (is_def(Ag), display(''Definition: ''+text(Ag[1])+'':''+text(Ag[2])))))
}
```

(The *PrintTable()* in the code above is a function which recursively prints out all the members of the table which it has as argument.)

When performing the transformations. Every function, number, variable, object, ... has its own unique tag. Consequently these uniques tags can be used to identify everything in Borg. Some examples

2.4.2 Absorption

Unlike other languages, Borg does not use objects to access the meta level. In Borg expressions can be directly modified: function names can be changed, arguments can be added, etc. So we are able to change expressions directly, which means we don't have to let changed expression be validated by the Borg interpreter.

We now know that all entities in Borg have their respective unique identifiers (their tags). We are able to identify all parts of an expression using these tags, the next logical step will be: how are we able to create expressions using tags and the abstract grammar of Borg?

There are two ways to create expressions by working directly on the abstract grammar. The second possibility is simply an extension to the first, and has been added to newer versions of the Borg environment.

An expression can be created using the *make()* native. This function takes an expression-tag as argument, it then creates internally a corresponding expression with all its sub-expressions set to zero. These sub-expressions can be manually set by the user.

The *make()* native is illustrated here with an application type of expression. Assume that we want to make an application *sum(1,2)*, an application consists of a name (of type string), and an argument list (of type table).

The latest releases of the Borg environment have extra internal functions which allow us to create expressions in an easier way. Since these functions will be used extensively in the code that performs the transformations, some examples will be given.

Thanks to this meta level of the programming language Borg, transformation of code has become very simple.

2.4.3 The Vision of Unified Objects

A paper I read when researching on the Internet on the topic of distributed systems and the problems that go with these (it will be discussed in the next chapter), pointed out that there is an overall vision of distributed object-oriented computing in which, from the programmer's point of view there is no essential distinction between objects that share an address space and objects that are on two machines with different architectures located on different continents.

Quote:

> In such systems, an object, whether local or remote, is defined in terms of a set of interfaces declared in an interface definition language. The implementation of the object is independent of the interface and hidden from other objects. While the underlying mechanisms used to make a method call may differ depending on the location of the object, those mechanisms are hidden from the programmer who writes exactly the same code for either type of call and the system takes care of the delivery.

If we regard Borg this way, we see that this is exactly what Borg tries to do: make no distinction between a local function call and a remote function call.

Of course there are a few functions which can only performed on remote object (or agents), for example the command to move agents across the network. For the rest, agents are threaded as if they were simple local objects.

One of the most remarkable differences between a local and a remote function call is that a remote function call to a remote object (or agent) always returns *void*. This because of the asynchronous way of communicating.

2.5 Disadvantages of Borg

Unfortunately, as all goods things have their downsides, Borg is not the ultimate mobile multi agent system as we would like it to be. Borg has several advantages, but there are also some disadvantages.

Although Borg has strong migration features which allow agents to move at any time they see fit, but we can't force an agent to migrate (unless the programmer adds this functionality to the agent).

The fact that Borg executes remote methods asynchronously has the advantage that processes can be executed almost concurrently and that one process does not have to wait until the process on the other end of the line completes its task. when it performs a remote method invocation. The downside of the way this asynchronous behaviour has been implemented in Borg is the fact the that everytime a return value is expected from a remote method invocation, we have to provide the necessary call-back functions. If the agent or process on the other end of the line was not written with the distributed paradigm in mind, we have to create wrapper objects.

Another downside of Borg is the fact that it does not support connection oriented approaches. This way we can not be sure if an agent at a remote site has simply crashed or moved. It frequently happened during my apprenticeship period that an agent, which had executed a remote method on an other agent, was waiting for a response that never came because the other agent had crashed. Of course it is simple to implement some code which checks the other agent at regular intervals, so that the user could a least get a warning, but this extra codes has to be inserted manually at various points in the agent (or we had to implement a kind of scheduler in the Borg code of the agent). But it would have been better if agents had internal support, which could be turned off, so that they would know it when their remote friend disappears or can not be reached anymore.

2.6 Conclusion

One of the nice advantages of using Borg as a mobile multi agent system is that Borg features easy agent creation and control. There is almost no difference between sending a message to an agent or sending a message to a local object, there is no need to learn new methods when switching from regular programming in Borg (or Pico for that matter) to distributed programming in Borg.

Because Pico and Borg are interpreters, they allow the programmer to quickly develop and test programs. The downside is that large programs sometimes get evaluated a bit slowly.

Although Borg has its disadvantages, such as the problems inherent to the asynchronous nature of its remote method invocations, they do not outweigh the advantages of Borg, which are strong migration, the reification and absorption of code. Since my work in this thesis depends a lot on the latter, Borg is certainly an advantage.

The asynchronous communication method had both its advantages and disadvantages. Asynchronous communication enables us to perform tasks concurrently because an agent does not have to wait for the response of a remote method (some kind of threading). The disadvantage is that we need to create call-backs functions if we want to receive results, or in case the remote object has no know source we have to provide wrapper agents.

Also Borg has no real way of knowing it when a remote agent disappears or the connection between two agents vanishes. So an agent can be waiting indefinitely for the results of a remote computation.

All these advantages make of Borg our preferred MMAS, especially since, as discussed at the beginning of this chapter, Java does not meet our specific expectations; Java simply lacks the support for strong migration and the ability reification and absorption without starting to fiddle with Java byte-code. For the disadvantages, well.. we just have to live with them (or create work-arounds).

3 Problems Inherent to A Distributed System

Writing a program with the intention to make it work in a distributed way is not as easy as writing a program for ”local computing”. In this chapter I will try to delve a bit deeper into some of the problems we may have to face when programming in a distributed environment. There are several issues involved which, while developing in a clean room, don’t occur very fast, but when the program is being used in the ”real” world latency, concurrency, partial failure, etc could give programmer grey hairs.

In the real world a sudden severe latency has a significant chance of occurring; for example, say that one of the co-workers start downloading popular music or movies, thereby creating a severe latency on the company’s local network. So, the most obvious maybe latency related problems, but these are in fact the least fundamental.

We also have to take into account the fact of a different memory model, while creating distributed applications. People creating distributed applications know that garbage collectors (which are built-in into languages as Java and Borg) are a really hard nut to crack.

Because there is nothing really new on these previous mentioned topics, I will now summarize a bit from an excellent paper I found on these topics [5, 1994]. After each explanation of a problem, there is a short side note which tells us the relation of the problem to something I’m working on in this thesis.

3.1 Latency

Latency is the most obvious difference between a local function call (or a local object method invocation) and a remote function call (or, remote method invocation). Currently the difference between the two is between four and five orders of magnitude. It is this difference in latency that is often regarded as the essential difference between local and distributed computing. Ignoring this difference would lead to designs whose implementations are almost assured of having performance related problems because the design requires a large amount of communication between the agents that are in different address spaces and on different machines.

So, let's take a quick look at how latency could be related to our context; as briefly mentioned in chapter 1 and further explained in section 4.2, we will divide an application in several smaller parts which are then distributed as agents, so we have to take the latency problem into account because some of these parts could require a huge level of communication between them. For example, we should not divide a for-loop into several parts and then transmit these parts to various computers, this would tremendously slow down the function in which this for-loop is being used, to the verge of becoming completely unbearable to use.

3.2 Memory Model

More fundamental than the latency issue is the difference between the local and remote computing in case of memory access. If we want to have no distinction between local and remote memory access, the memory manager under the hood needs complete control over every memory access. This has some interesting prospects, but there are downsides of this unified memory model: for one, it does not really like garbage collectors. Since Borg uses a garbage collector (as do most interpreters) this could be a problem, fortunately every agent in Borg has its own data space and there is no shared memory between agents. A lot of research has gone into the subject of distributed garbage collectors, but it is not trivial answer.

Of course we could limit a garbage collect to all the the local Borg interpreters, but this would only create problems. For example, if an agent has some references to remote objects, we can not know for sure if they are still valid (and it would create a huge overhead and slow down the garbage collect horrible, if we would check each remote reference).

There are a lot of papers on distributed memory and garbage collector, which try to address the difficulties on this subject. But there is not one solution that solves all the problems. As noted before, Borg also has a garbage collector, but Borg does not has no shared memory between the agents, which means that we are only allowed to create references to other agents, we cannot address memory from the remote agent (neither can we address memory of the remote interpreter). This way Borgs dodges the problem.

From the introductionary chapter and the chapter following this one, we read that we need a distributed shared memory model between several agents (which all have parts of one and the same function, so they obviously also need to share the data used in that function). Since there is no possibility (yet) of a fully working distributed garbage collector for the distributed shared memory model, I simply decided not to use one and simply rely on the garbage collector already in place in Borg. Although there is no real need for a garbage collector now (because the code was up till now only used for small tests), in a later stage we certainly will need one.

3.3 Partial Failure

What happens if an agent is computing something and the host-computer crashes or what happens if an agent has been sent to another part on the network to do some calculations, while the computer where the agent departed is waiting for answers, and the network connection between them breaks down?

Both local and distributed systems contain parts which are subject to failure. If a part of a local application crashes there is real problem, the whole program subsequently crashes (that is, a total failure). But if a component of a distributed program crashes (for example the machine on which the program currently resides or the network), the rest of the program may be unaware of this problem for a long time and could simply hang. Partial failure is central reality of distributed computing. Not only is the failure of the distributed components independent, but there is no common agent that is able to determine what component has failed and inform the other components of that failure, no global state that can be examined that allows determination of exactly what error has occurred. In a distributed system, the failure of a network link is indistinguishable from the failure of a processor on the other side of that link.

These sorts of failures are not the same as mere exception raising or the inability to complete a task, which may occur in the case of local computing. This type of failure is caused when a machine crashes during the execution of a method invocation or a network link goes down, occurrences that cause the target object to simply disappear rather than return control to the caller. A central problem in distributed computing is insuring that the state of the whole system is consistent after such a failure; this is a problem that simply does not occur in local computing.

There should definitely be a system which detects crashes of agents or broken network links and all the other problem which arise when working with agents on a real-world basis.

3.4 Concurrency

Almost every multi threaded system or network eventually has some concurrency problems. Most of this problems in a multi threaded system can be avoided if the system has a good design. Although concurrency de facto has nothing to do with the mobility of the agents, but rather with the fact that if agents are not properly designed they could create a deadlock or a race condition with incorrect data as a result.

In a mobile multi agent system agents are not only allowed to work on remote computer, but of course also concurrently on the local computer, so the environment which takes care for the execution has to allow the concurrent execution of the agent locally. As mentioned in chapter 2 Borg (and even Pico) has an internal scheduler which allows the execution of agents in parallel. We all know that programs running concurrently (or agents, to stay in the scope of this thesis) are prune to the fact that some of these agents could create a deadlock. In Borg we try to avoid the concurrency problem by using asynchronous calls between the agents.

3.5 Remote References

How does an agent keeps track of the fact that another agent it is communicating with has suddenly migrated to another location? This is an annoying problem of mobile systems, an agent can migrate in the middle of an execution.

We really can't update all agents every time one of their companions has migrated, because this would create a communication overhead which will grow enormous if more and more agents are added with a tendency of migration. Not only this would delay the other communication beyond usability, but if the system grows too large, it would start to hinder other applications as well.

A quick look at the internet and the TCP/IP protocol learns us that a decentralized nameserver could be an answer. But we have the same problem as above, as more and more agents are being deployed, the communication between the agents on the move and the nameserver would still be large. Not only are agents able to move several times per minute, but the agents who are not on the move need to send lookups for every remote function call they want to execute. A simple calculation shows us that this leads to a heavier load on the DNS server than some heavy webbrowsing would do. It still would not provide a solution for the delays in the messages.

But because Borg is not being used aside from inside our local university network we do not expect a lot of message traffic, so we opted for the nameserver solution for the moment. Unless we start to have problems or if Borg will be wider spread, we will keep on using this nameserver.

3.6 Security

Weird enough, this was not mentioned in the paper I mainly used to construct this chapter. The need for security is far more greater with distributed systems, since the shared concept of the transmission medium creates easy opportunities for malicious persons. Not only the communications between distributed processes are vulnerable but also the roaming processes itself. Apparently agents are not yet widespread enough to need a kind of protection. Traveling agents are very vulnerable to various kinds of attacks. Especially when they roam on a shared medium as the Internet or even a LAN; it is easy to copy the agents to local memory in order to ”study” the contents and code of the agent.

Aside from the simple copy-and-disassemble method, more devious persons could easily change the code of an agent of the move, thus creating damage or change the results from a possibly expensive experiment. While there may be none of those problems at this moment, the moment agents are used outside an academic research facility the chance will rise that there will be attempts at cracking these entities.

The idea presented in this document does not take care of this security problem, we consider agents as secure, our goal is to enhance security and improve the copy-protecting of software by dividing it into separate components encapsulated in agents. If this concept is ever to be used outside the academic environment, there is certainly a need for more security in the agents. Preferably some security which is transparent to the programmer.

3.7 Conclusion

Several of the problems noted in this chapter were already discussed in the paper mentioned in the introduction of this chapter. These topics together with the others presented are of real importance once a mobile multi agent system is used outside the clean environment of a research laboratory. Especially the security of agents needs more research, the partial failure (or robustness of agents) concept has maybe a solution discussed in the thesis of one of my co-students.

On the distributed garbage collector topic, there is still research needed (if implemented incorrectly it can bring a severe performance problems).

We now also know that simply keeping track of the voyage of several agents could create a rather high latency for other agents or network dependent software (e.g. a remote filesystem). In Borg we temporarily solved this issue by using a dedicated nameserver, but if Borg gets more widespread this solution will not suffice!

Although the security and protection of agents is not a problem at this moment later on, when these agents leave the safe environment of the academic world, and are becoming more used, they will need some sort of protection against our more malicious fellow human.

4 Method

In the previous chapters the reader was introduced into the field of mobile multi agents systems and the possible problems associated with these. This chapter provides an overview of what we will try to accomplish, a rough overview of the various code transformations used in the next chapters will be given, so the reader can see the general picture. Briefly put, this chapter explains why these four transformation steps are used, the next four chapters explain us how these steps are implemented.

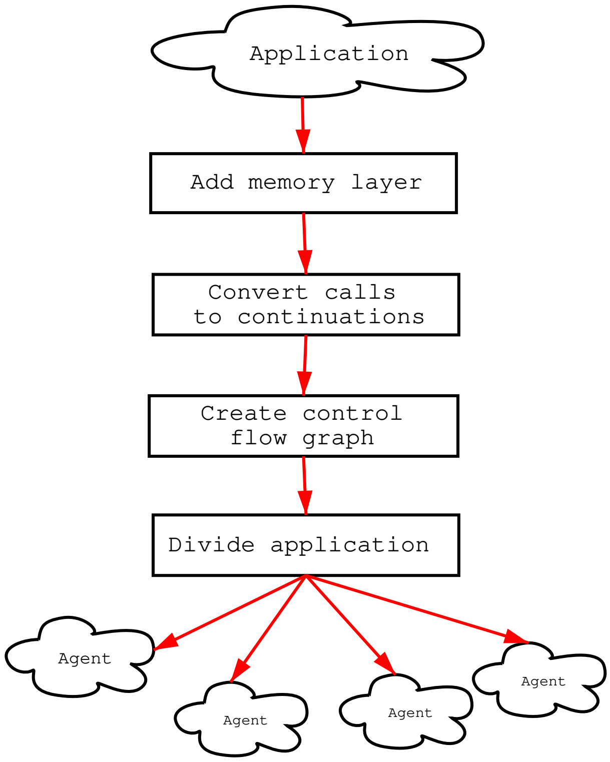
4.1 Introduction

The idea was that an application, written using the Borg language, could be transformed into a set of agents that would give the same results after execution as the original program. Unfortunately, due to a lack of time I was not able to implement all of the aspects of the Borg language; we know that Borg supports objects and has the ability to work with agents, but these two concepts are rather time consuming to incorporate in the transformation process. Especially when a program uses some agents, we would have to make the choice to either have these agents divided or not. If the agents are small, there would be not much of a problem; on the other hand if these agents would be large enough to have the need for division, then we would have to create a kind of super agent that operates as a bridge for messages.

So, because there was only limited to get a prototype running, a decision was made not to implement support for objects and agents; it has been put in the queue for future work. We will not go any further into this matter. This basicly means that we will transform applications written in the Pico language (as seen in chapter 2 are the objects and agents only extension to Pico).

4.2 Overview of The Steps Used in The Transformation Process

With the picture below as a guide, we now try to explain the steps in the transformation process



One is able to notice that the complete process consists of 4 steps, and although it is possible to merge some of the steps, we keep them separate in order to present the ideas to the reader in small doses, it also eases the development cycle.

Every of these steps is actually a small Borg interpreter on its own, these small interpreters all use the abstract grammar of a Borg program to do their respective jobs. The first two steps and the last all make changes to this abstract grammar and the third step uses the abstract grammar to construct a control flow of the program.

Now, let us take the first step.

4.2.1 Step 1: Adding An Abstract Distributed Memory Layer

When dividing functions into several parts, it is obvious that we need some kind of extra memory layer that allows us to pass around variables across the several parts of a function. There are two ways to implement this, we could simply pass the local variables plus their scope with the continuations or we could create an abstract memory layer that implements a kind of shared memory between the several parts of a function. They all have their advantages and their disadvantages. The first choice offers us fast local read and write capabilities since the memory is passed down as an object, but it can slow down the function calls because the memory object can grow quite large.

The other possibility has the advantage that it offers a shared memory across the entire application; also there is no need to pass the memory as a parameter with every function call. The downsides of this approach: although variables are stored locally with every co-instance of the shared memory (which allows us fast memory access), write access to a variable results in sending synchronization messages to every other instance of the memory layer. Also, if there are a lot of instance of this shared memory, the possibility that one agent, containing an instance of this memory layer, suddenly dies or is temporarily unavailable is quite large and it would lead to memory inconsistencies.

Since we wanted the advantage of fast communication between several parts of a divided function, while trying to have as few problems with inconsistencies as possible, a choice was made to mix the two options together. Between the functions of the application we pass an object containing the scope and between several parts of a function we implement a shared memory. The two memory enhancements will from now on referred to as "memory layer" unless unless more precision needed.

After finishing the entire transformation process each of the divided functions has to be able to work seamlessly together (as before when it was still undivided).

There are two ways to incorporate this memory layer

- In one way we change the Borg environment and enhance the current memory system. This way we only have to add a few extra functions to the Borg language and let the internal memory system take care of the rest. One downside is that we have to extend the internal memory structure of Borg, which is not easy; and we would still have to create a program transformation that adds the new functions to the program.
- In the other approach we add a layer between an existing program and the Borg memory manager. Because we already were conducting abstract grammar transformations on the code, it would simply add one extra transformation A positive side from this approach: we only need to change one thing: the program to transform.

Because extending the existing Borg memory system into a distributed memory system (with a distributed garbage collector) would be a very though job to do in one year and without a lot of experience with the internal Borg memory system the decision was simple: it would be the second proposition.

So the idea was to create an extra abstract memory layer, that implements a kind of distributed shared memory. Looking at the list of possible problems with distributed systems as noted in chapter 3, we know we have to create a very well considered design, because otherwise we could bring down the whole distributed application, for example by sending too many synchronization messages between the distributed shared memories of each divided function.

It is obvious that if we want to use a distributed shared memory, we need some restrictions in order to keep the performance of the end-result of the entire transformation process within limits. So, a decision was made to restrict the use of global variables. No global variables are allowed, because it would create a rather huge communication overhead if these variables were frequently used and changed. It would also increase the possibility that some agents are still using older values of the variables because of for example a temporary network failure. And using global variables is not a really good programming practice too.

Global constants on the other hand, would be no problem. Many programs written today have a lot of global constants, so it would be shortsighted to have no support for these constants; and support for global constants is not that hard to add to the memory model. In fact there is a possibility of optimization in the abstract distributed memory model, since these constants could be stored in the memory of every agent after the application has been divided in agents and they do not require any further communication in the shared memory. But no time was spent to implement this support for global constants, because it would not be hard to add at a later time and it was not really necessary.

Another issue might rise: how will the abstract shared memory cope with the large possibility that some parts of a divided function frequently use some variables and thus create a high load of messages for memory synchronization? Well, that is a good question, unfortunately there is no real answer. It is more a question that should be examined very thoroughly once all transformations are fully functional (the motto "First make it work, then make it beautiful" comes to mind) and in chapter 9 some possible enhancements will be offered that could allow us to make some improvements.

This was a simple overview of what the reader may expect in the next chapter, in which the abstract distributed shared memory will be explained in depth. Later we will see that thanks to this memory layer the transformation of a function into continuations becomes rather easy.

4.2.2 Step 2: Transforming Calls Into Continuations

Or, as they are sometimes called, call-back functions.

It was already mentioned in the second chapter, a problem associated with the way Borg executes methods on remote objects (agents) is that the functions always return *void* and results have to be delivered by means of continuations. The fact that a remote message always returns *void* is pretty useless except from the fact that we know that message has arrived safely at its destination.

If we are dividing existing applications, that are developed for local application and as such have no knowledge of how to handle remote objects and the need for call-back functions, we need to adjust these applications so that they could used be in a distributed fashion with Borg agents.

Because one picture tells more than a thousand words, so a piece of pseudo code should do no less, this sub-chapter contains some code snippets which will guide us.

Say we have two simple (pseudo code like) functions written in the Borg language:

```
f2(<some parameters>)::
{
   ...
   result
}
fun1()::
{
   b := 1;
   ...
   a := f2(<some parameters>);
   display(a+b)
}
```

Because we do not have a way to temporarily pause the execution of function (till new values have arrived) and if we want these functions to behave well in an agent environment, this means that these function should be able to handle the asynchronous calls to other functions, we have to add continuations.

This procedure consists of two adjustments to the original code:

1. extend the called function with a parameter and use that parameter to return the result
2. transform the caller function, divide it in two separate functions: one which starts the execution (and stops when the function call has been executed) and one that finishes the execution.

We will now transform the two functions from the previous code example:

```
f2(<some parameters>, return)::
{
   ...;
   return(result)
}
fun()::
{
   ...
   f2(<some parameters>,returna);
}
returna(result)::
```

```
{
   a := result;
   display(a+b)
}
```

In the code snippet above, we have added a continuation, the function *returna()*, this function continues the computation that *fun()* started.

From this example, the first problem shows up, after comparing careful we see that the transformation results in non-working code. The variable *b,* used in the original function, has moved to the call-back function which has no knowledge of this variable. There is no real solution for this using Borg natives or functions, but we will see in the sixth chapter that the abstract memory layer will help us solve this problem.

The previous sample transformation was only a simple one, we have a harder time when we try to add continuations to more complex functions, for example functions that use some kind of loop- or control expressions.

The funny part about adding this continuations is that we already divide some functions into several subparts. We could easily have stopped at this point and called it a success, but there a still a lot of functions that largely consist of expressions other that loop- or control-expressions.

4.2.3 Step 3: Creating a Control Flow Graph

In order to divide an application in smaller parts, we need a data structure that enables us to get a complete overview of the program's code. For that purpose we decided to construct a control flow graph of the application. A control flow graph has several properties that are of interest to us, after constructing this graph it shows us:

- which functions are most likely frequently used;
- which functions are called from where;
- where we can add continuations (more on that, see next subsection)

Using this graph we are trying to see which functions are most frequently referred to and which are rarely used. Although this may vary when the application is executed! We can not be sure that for example a function is referred to several times in the code, but is only used when the application starts. If a function is used from another's loop-expression, then we mark this too, this will indicate that there is a possibility that the function could be called in bursts.

This graph also shows us which functions are called from where.

Creating a control flow graph of the application is simple - which of course we may thank the abstract grammar for - we traverse the abstract grammar of the application and create for each function encountered in the application a node in the graph.

This node contains some meta-data about the function. When scanning the abstract grammar of a function, we add some statistics to its node: for example, whenever a function gets referred to from within another function, this information is added to the information stored in the node.

4.2.4 Step 4: Division of The Application

Now we introduced the three first steps we only need an overview of the last step. Using the control flow graph we take the last step of the transformation process. This step traverses the control flow graph, constructed previously, using a few criteria, such as: if a function is often referred to by another function, we should try to keep these two close together; or when a function has a reference to another function from within a loop-expression, we could try to keep them together as well.

Before we start dividing the application a number of agents is created that will be used to contain the divided application. This number depends on the size of the application and the fact whether the application will be used on a local (fast) network or across a slow link (for example, when using it via the Internet).

Considering the fact that the more functions get divided the more we get a performance drop when executing the application, but on the other side, the more we divide the functions the harder it will get for users with malicious intentions to copy the entire application. So we have to take the granularity of the end product into account when dividing the application.

The actual division of a Borg program consists of two steps: a function that traverses the control flow graph of a program and randomly picks out functions that will be divided; and a last transformation step that divides the functions and add continuations so that the functions are still able to do their chores.

The function that decides where the application will be sliced, needs more intelligence than one would think at first sight. There are several issues that should be taken into account: we need to randomize the division process, so that a user almost never gets the same agents with the same parts of functions; also functions that have a possibility of being frequently called should remain close to their callers.

As one can imagine, programs with a lot of loops are hard to divide when we still want some performance. The implementation used in this document is very rough and certainly needs more thinking about the performance issues that are connected to these design decisions. In chapter 9, some future work suggestions on this topic are presented.

Then when a function is chosen to be divided, we to add additional continuations and create additional clones of the memory layer for that function. Afterwards the function is inserted in an agent, that has been defined at the start of this last step.

Note that during the division process, the functions added to the application in the previous steps of the transformation process are not divided! The memory layer for instance always remains locally to the function it is bound to.

4.3 Conclusion

I hope that the reader now has a more profound idea of what I am trying to accomplish. And, although it might seem easy to do on at first, there are a lot of problems we have to face. With performance being a not so minor one. Dividing an application introduces a lot of overhead which is necessary to ensure that the divided application performs its job the same way the local (undivided) application would do.

This chapter presented four methods that will be used to transform an application:first we will add an extra memory layer to the functions that will make it easy to keep the variables synchronized when these functions are divided. The second transformation transforms the local function calls into continuation based calls, these continuation based functions are then able to be used in an agent based environment (which uses asynchronous messages). The third step creates a control flow graph of the code that is the result of the first two steps on the application. The fourth step then performs the last transformation resulting in a agent based application. More information is presented in the next four chapters.

It is a pity Borg does not support the possibility of pausing the execution of a thread. Otherwise it would have been much simpler.

5 Variable Bindings And Lexical Scoping

One of the biggest problems when you slice functions in parts, is how we are able to still have access to the variables of the function when one agent with a part of the function is on one computer and another with the rest of the function resides on another computer. The reader can clearly see that there will have to be a way to give access to the variables across agents. Unfortunately Borg has no built-in methods which could give us a shared memory between agents (for performance reasons).

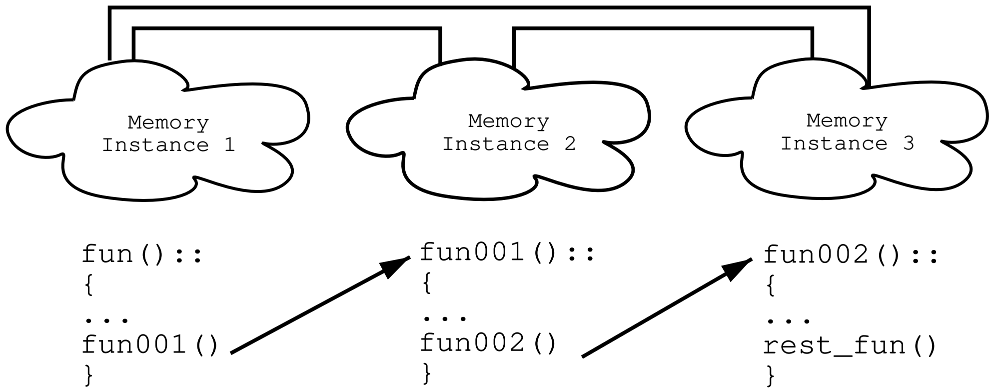
There are a few ideas: make a variable agent, that contains all the variables together with their scope. But if this agent has connection problems to some of the agents that depend on it the computation results could become incorrect. On the other hand, one could use an ADT for each function, the ADT keeps itself in sync across the several agents a function can be divided in. Thus creating a kind of rude distributed memory shared between agents that shall contain a part of a divided function.

Another solution - that will be used - is to have an extra abstraction added to the parts of a function. This abstraction then would take care of synchronizing the data between the co-instances of this abstraction. Sadly enough, this idea introduces another problem: if we follow this train of thoughts, we could also use the ADT for global variables. However after some arguing, my advisor and me thought it would be better, if we didn't. Because if there are a lot of global variables, the communication overhead would be enormous, when an agent would change a global variable, every variable ADT's should get sync-ed. A little note: even CSP doesn't use globals (or only read-only).

In the next subsections this memory layer will be explained in depth. This memory abstraction can be divided in two parts, the first part is the addition of an abstraction between existing code and its memory accesses; the other part - which implementation is completely invisible to the function its code - takes care of the data storage and the connections between several instances of the memory layer.

We first start with a simple implementation that has no knowledge of other instances, then we show how this memory layer is added to existing functions, and then the simple implementation is expanded so that the memory layer can communicate with other instances (we thus have a shared memory between two parts of a function).

For a better explanation, the reader could keep next picture in mind.



5.1 A Local Shared Memory

This preliminary version will only allow local access to its data. Since the memory layer will have an abstract interface the functions which use this memory layer will never know that they suddenly have a distributed shared memory.

We first need to define the interface between the memory layer and the program code. During my apprenticeship period this memory layer was constructed with the name *varADT*, since then it was never changed.

The memory layer *varADT* tries to emulate all the possible memory accesses that are available in the Borg language (This reflects the internal functions of the Borg VM). Which are:

- definition of a variable
- definition of a constant value (a declaration in Borg)
- referencing (read) of a variable or constant value
- changing the value of a variable

Here is an overview of the *varADT* interface:

```
varADT.define(name, expression)
varADT.defineTab(name, size, expression)
varADT.declare(name, expression)
varADT.declareTab(name, expression)
varADT.get(name)
varADT.getTab(name, index)
varADT.set(name, newvalue)
varADT.setTab(name, index, newvalue)
```

There are some special cases which were harder to implement: a table structure and a variable or constant which refer to function.

We took a decision not to include support for variables which can contain a function, this implementation of the first transformation step would only be a proof of concept and this functionality could later still be added. The problem was that we were not sure how these functions would be transmitted via the network to other agents.

In the case of variables containing tables, we had several options: we could only implement support for single dimension tables or we could fully support the multi-dimensional tables of Pico. In case of the multi-dimensional tables the code for referring into these tables can be quite troublesome. So - again - a decision was made to include support for single-dimensional tables only.

Since this memory layer will serve as a substitute for the Borg memory, we will need to implement the notion of a scope. This will allow the usage of the memory layer for loops and other kinds of iterations. We therefore add two more functions to the interface of the memory layer:

```
varADT.push()
varADT.pop()
varADT.merge(otherADT)
```

Internally this scope is implemented as a stack, if an instance of the memory layer object receives a message to create a new scope it creates a clone of itself which has as parent the current instance, it then returns the newly made clone to the function. Leaving the current scope is done by re-activating the parent scope (if available). This kind of behaviour is sometimes called a mixin method.

Some explanation of the scope interface functions, the *push()* function is called before we pass the memory instance to a function, this is the function that creates the new scope; the *pop()* function has to be called after the called function has finished, the *pop()* function re-activates the previous scope, thereby destroying the current. The *merge()* function is needed to merge a local memory layer instance with the one passed down as scope. Without the *merge()* function we would have to make a difference between function variables and variables from a previous scope.

The root scope is started from the ”main” function of the application.

We have now defined the interface to the memory layer, let us have a closer look at the internals of the memory layer object.

First of all, it is obvious that if one instance of the shared memory wants to synchronize with its co-instances, it must have a list of references to its co-instances. In this local case these references are the local objects.

In this local version of the shared memory layer, we only have one function that takes care of the synchronization process. The instance that wants to synchronize calls this function on all the co-instances. This functions has as parameters the name of the variable to change, the new value and the depth of the scope. Since the process of calling a local function is synchronous, no extra care is necessary to assure that the data has been well received.

Because we are working locally, all function calls are executed synchronously and thus when a function changes a variable it has to wait till the memory layer has synchronized and returned control to the function. The fact that agents send asynchronous messages will prove troublesome when we make this memory layer distributed.

5.2 The Code Transformation

Now, how can we add this memory layer to existing code? As previously mentioned, there are four distinct operations which work on variables. A variable can be defined or declared, a programmer is able to change the contents of a variable or he can refer to the contents of a variable (or constant).

In order to add the memory layer to an application, we have to add this memory layer to each function encountered in this application. The easiest way to achieve this is by traversing the abstract grammar of each function, replacing each occurrence of one of the four possible operations by its analogous counterpart defined in the memory layer object.

It should be noted that we do not want to add the memory layer to the application itself, thus not allowing any global variables or constant values. In the local case this would not really be a problem, but since this memory layer has is intended to be used in a distributed way, the use of global variables could potentially lead to heavy network usage. Therefore a decision was made not to include support for global values. Global constants would be no problem, since they would have been replicated across all instances of the memory layer (alas this is not implemented at the moment)

Adding the interface-functions of the *varADT* object to a function is pretty straightforward. By walking through the abstract grammar of the function we are able to determine what expressions are contained in the function. Every encounter of an expression which belongs to the set of possible operations on the memory is replaced by its respective counterpart in the memory layer object.

Consider this simple example:

```
foo(n)::
{
   a:0;
```

```
   b::n;
   tab[n]: 7;
   tab[1]:= a;
   display(tab[1]+b)
}
```

After execution of the transformation process on this function we get (ownADT is the memory layer that is associated with this function)

```
foo(varADT)::
{
   varADT.merge(ownADT);
   varADT.define(''a'', 0);
   varADT.declare(''b'', varADT.get(''n''));
   varADT.defineTab(''tab'', varADT.get(''n''), 7);
   varADT.setTab(''tab'', 1, varADT.get(''a''));
   display(varADT.getTab(''tab'', 1)+varADT.get(b))
}
```

Every memory access is replaced by a function which exhibits exact the same behaviour as the previous used operators. Only now, we use the memory layer.

There is another change needed in the function, we need the possibility to pass the local variables to other functions. In order to do this, we have to make a distinction between two kinds of functions: the native functions and the user defined functions. The native functions are built-in parts of the Borg interpreter and are therefore available to every agent we create. As a consequence we do not have to change these native functions to accept the memory object.

For the user defined functions in the application we have two options: in the first we change the functions to accept instances of the memory layer object or we do not change the parameters of a function, but then we have to add the parameters of a function, before the function actually starts computing.

We opted for the second method and within the function we add extra expressions that add the given variables to the memory layer. When calling a function we still pass the original parameter, which gets inserted before the function actually starts to do something. Envision this in the function mentioned above:

```
foo(n,varADT)::
{
   varADT.merge(ownADT);
   varADT.define(name(n), n);
```

```
    varADT.define(a, 0);
    varADT.defineTab(tab, varADT.get(n), a:=a+1);
    varADT.setTab(tab, 1, a);
}
```

Now an example with a *while* loop:

```
while (a<10, a:=a+1)
```

With loop operators we have to take care of the fact that most of the times a new block or scope is created for the expressions used in the body of the while loop. Therefore when transforming the while loop will create a new scope in the abstract memory layer. After transformation this will look like

```
while (varADT.get(a) < 0, {
    varADT.push();
    varADT.set(a) := varADT.get(a)+1;
    varADT.pop()
})
```

The other loop operators in Borg can be transformed the same way.

5.3 Make It Work Distributed

5.3.1 References To Co-instances

If a part of a function changes a variable, this change should be propagated almost instantaneously to the other instances of the memory layer. Therefore each instance of the memory layer should have a list references of its companions. This list is filed while completing the last step of the transformation process, before the agents are sent out. Actually list is a bad name, we create a table in the varADT object which holds the names of the agents together with the identifier of the instance of the memory object.

5.3.2 Consistency

The big difference between the local and the distributed version of the memory layer, is that with the local version we don't have to worry about slow or unreliable network links, remote crashes of agents, etc...

For the distributed version of the memory layer we have to add additional checks for consistency of the data, time-outs while updating Also the way we will refer to the co-instances of a memory layer instance will differ, we will need additional information such as the name of the remote agent and the name of the function to execute on the remote agent (because in the end-result an agent could contain several parts of functions of the application, each with their own memory layer).

In order to insure the consistency of the data across the several instances of the memory layer, we implemented a four steps mechanism.

1. Send a lock-message to all co-instances
2. Transmit the changed values to all the co-instances
3. Wait for an acknowledgment from each co-instance
4. Send a release-lock message to all co-instances

When a memory layer instance receives a message to set a lock, it set the lock variable with a timestamp and the name of the originator of the lock-message. The lock has actually two purposes, first we avoid the possibility that another function tries to change a variable (this could happen if functions are executed concurrently); secondly we now have a way to detect if the synchronization process has been interrupted (for example, the memory layer instance which started the synchronization has crashed, or a network link has disappeared).

Because agents communicate via asynchronous messages, their is a need to keep the memory layer instance in control until all acknowledgments are received from its co-instances. The solution is not a very elegant one, a function *poll()* was created, in pseudo-Borg code it looks like:

```
poll()::while(true,
          if(allLocksReleased() or timeout(), break, wait(10)))
```

This *poll()* function is simply an loop which does not stop until all locks are released or a timeout is reached. The function *wait()* causes the memory layer instance to sleep during a given period (in ms).

Since the memory layer instance is always locally placed together with the part of the function it works for, the communication between these two happens synchronously. So there is no way that a function continues with the next expressions while the memory layer instances is still synchronizing with its co-instances.

There are other possible consistency checks:

- Create a checksum of the table which contains the data and transmit this when requesting a lock.

- Have a unique ID which increments for each function executed on the memory layer. This only applies to functions that make changes on the data of the memory layer instance.

The first is only a possible extension of the memory layer and has never been tested nor implemented. The other option has been implemented and tested. Every time, before a the lock-message is send to the co-instances, a unique identifier is incremented and send along with the lock message. The receiving end checks if the identifier is 1 larger than its own. If not, this could be a indication that there is an inconsistency.

5.3.3 Limiting Network Traffic[1]

If we would transmit the entire memory layer instance every time we make a remote function call (as is the case in the local version), we would transmit some redundant data to the other function. Because the memory layer will slow down the application already, we want to limit the size of the data transmitted.

For this reason an extra interface function was defined. This limits the data sent to the other function to only the data of the variables including the scope.

instead of

```
foo(varADT)
```

we now have

```
foo(varADT.data())
```

and on the receiving end:

```
foo(data)::
{
varADT.insertForeignData(data);
...
}
```

The memory layer instance of the other function creates a clone with the data received.

In the local version of the shared memory layer we have two functions that take care for the synchronization between the several instances. If a function changes the value of a variable an object with the name of the variable, the new value of the variable and the

[1] This feature was not yet fully implemented when this section was written.

scope-level of the variable are wrapped into an object which then is transmitted to all the fellow instances. For every other fellow instance of the memory layer object a flag is set when the synchronization message is sent. Every instances receives the synchronization message, and immediately sends an acknowledgment message back. If the originator object has received an acknowledgment from every other instance, the originator object returns control back to the function.

5.3.4 Changes To The transformation Process

Now we have a general idea of how the abstract memory layer communicates with its other instances, we will now review the changes to the transformation process.

Since the interface was designed from the start to be abstract, no changes are needed in the transformation process. the only change is the way the memory layer instances get their names. In a distributed environment these will have to be prepended with the name of the agent in which the memory layer is stored.

5.4 Problems Encountered With This Approach

Because this extra memory layer will enable a kind of shared memory between several agents spread across remote computers, this memory layer will have to deal with several of the problems mentioned in chapter 3 and some others.

First of all: too many updates between agents which shares the same memory, could severe limit the performance of the application. One of the problems of this implementation is that the more a function is divided - that is more separated agents - the more instances of the abstract memory layer need to be kept consistent. To keep the memory abstraction consistent at all moments, we generate a lot of overhead. Luckily on the write operations on the memory results in the sending of messages, the read operations are cached locally.

Another problem is the fact that although we try to create a scope for functions, we can not extend it to a global scope. This explains why no global variables are allowed in the application. If too many functions would use the same global variables the communication overhead between the memory layers between the Borg agents would be a real bandwidth hog.

Another problem may be latency, if due to severe latency a remote instance could not be updated properly, we could have an inconsistent memory layer, resulting in faulty results.

The code provided in this chapter tries to eliminate possible consistency problems, but since this code has only been tested on the universities network, an inconsistency has never occurred.

There is no support in Borg which would allow us to see if there is enough memory available, if the memory layer with the function is works for is on a computer with low memory it could crash if too much data was synchronized.

5.5 Conclusion

Since Borg has no methods which could provide us with a shared memory between some functions, we had to create this ourselves.

This extra memory layer enables us to share easily variables between functions which are on different computers interconnected by the means of a network. On the other hand it could create a huge drop in performance because every change to a variable of one memory layer object must but propagated to all its fellow instances. The more instances, the more traffic needed.

In order to obtain some performance we had to make compromises, a decision was made not to allow any global variables and thus avoiding the need for an memory layer for the entire application.

Adding the abstract memory layer objects to existing code is a relative straight forward matter. We use the abstract grammar of a function to replace every operator, which accesses the memory in some way, by its analogous counterpart defined in the memory layer object.

The code for the memory layer object presented in this chapter is not really intelligent and could have some improvements that reduce the size of data transmitted.

6 Transforming Calls to Continuations

Functions which are executed on agents only return the standard value of *void*, stating that the message was send successfully. From functions that are executed locally we expect them to return results other than *void* (most of the times). Since we are trying to transform a normal local application into a set of agents, we have to add functionality to these local functions that will allow them to be used in an agent based environment.

For example, the next example works perfectly when executed locally

```
do_something(var):: var+1;

fun(a):: display(do_something(a));
```

If the first mentioned function would reside in a remote agent, not only the way of calling it would differ, it would also return only *void* upon sending the remote method message. Within this chapter a new transformation algorithm will be presented which allows us to add continuations to the local function after which they should work when used in an agent-based way.

It is important to note that we will only transform user defined functions, Borg's internal functions and natives are, for obvious reasons, not transformed.

Again we introduce the algorithm by starting with easy examples. Since the complete algorithm easily spans several pages, only some examples in addition with their results shall be presented.

6.1 Simple Functions

The simplest function we can transform is

```
foo():: 5;

doIt()::
{
   a:= foo()
}
```

This is transformed to

```
foo(callBack):: callBack(5);

doIt()::
{
   foo(callBack001)
}
callBack001(val):: a:=val;
```

So the addition of return continuations always breaks up the original function and an extra function is added that takes care of the assigning the return value to the right variable. The function *callBack001()* is automaticly generated, hence the identifier 001 appended to the function name.

This transformation is easy to implement when we can use the abstract grammar of the Borg function. the only caveat is that we have to check whether the continuation function, that will be created, exists already.

There is another slight problem that arises[1], in the example above we had sheer luck that it worked without problems. Let us look at the next, somewhat more difficult, example.

```
foo(val):: val+5;

doIt()::
{
   a:0;
   a:= foo(a)
}
```

After we performed the transformation on this code, we get:

```
foo(val):: callBack002(val+5);

doIt()::
{
   a:0;
   foo(a, callBack002)
}

callBack002(val):: a:= val;
```

[1]A careful reader did, of course, notice this already

And now the previously mentioned problem shows more clearly. After execution of the new code, we will not get the same value in the variable a. We have two options, first we could move the variables outside of the function and introduce them as global variables or we could use the abstract memory layer that is presented in the previous chapter.

We are going to opt for the second choice, since we will already use this memory layer for other chores and it is an easy and simple way to make variables shared across several functions. In fact, the current order in which the transformations steps are performed on an application takes already care of the problem, we only have to include some extra code in this transformation step so that the memory layer of the original function gets cloned and added to the newly created continuation functions.

While we are mentioning the memory layer in combination with this transformation step, note that since the memory layer is always local to its function, we exclude the interface functions of the memory layer from the this transformation step.

6.2 More Complex Functions

This addition of continuations transformation seemed simple at first, unlike the following somewhat more complex functions. These more complex functions may contain one or more loop-expressions[2] or use one or more control structures that takes an action based on the return value of a function.

In this subsection the examples will be presented for the *while()*-loop only, the procedure is similar for the other loop expressions.

```
inc(nr):: nr+1;

fun()::
{
   nr:0;
   while(nr < 5, nr := inc(nr))
}
```

This transforms into

```
inc(nr, return001):: return(nr+1);

fun()::
{
   nr
```

[2]In the Borg language we have the *while*, *for* and *until* loops at our disposal.

6.3 clusion

Although at first this step in the transformation process seemed very easily and would prove no problems to implement, but then we had to implement some complexer functions which transformed functions containing a loop expression into several parts.

7 Creation of A Control Flow Graph

Creating a control flow graph of an application is a necessary step that allows us to intelligently divide applications. This graph offers us an overview of the program its code. It can show us which functions are called from where, it also can show us which functions have the possibility of being used a lot. This knowledge can be useful to keep the performance of the end-result of the transformation process from dropping beyond being unusable.

It not only allows us to keep frequently used functions together but it also allows us to add continuations more easily to the functions of the divided application.

The construction of this control flow graph will not slow down the transformation process, since this control flow graph can be constructed once and used many times during the division of the application, it is not a real overhead to the transformation process, rather an asset.

7.1 Overview

The control flow graph of a program is purely an internal data structure, every node of this graph contains information about a specific function of the application. The nodes are interconnected by a ”used-by” link, this means: when a function *foo()* is called from function *bar()*, then we have a ”used-by” link from the node of *foo()* to the node of function *bar()*.

The main purpose for using this graph, is because we can use it to get a fast overview of the application. The graph is also being used in the last transformation to store in which agents the functions are.

7.2 The Internal Datastructure

The graph consists of many nodes, each node contains some meta-date about the function and possible links to other nodes. Other than that, the node also contains some flags and data that help facilitate the last to perform its duty.

The most important parts of the node are:

- a table that stores references to the abstract grammar of the function (and its possible continuations obtained thru the algorithm from the previous step);
- a variable that has a reference to the memory layer object associated with the function;
- a variable that will be used in the next step to store the name of the agent this function will be stored in;
- a table containing the "used-by" links.

The link between the nodes is a "used by" link. Let me clarify this by giving an example:

```
function1(par1, par2) ::
{
   ...
   res: function2(a);
   display(res)
}
function2(par) ::
{
   do_something_with_par
}
```

The graph will contain two nodes: one for *function1()* and one for *function2()*, the node of *function2()* contains a "used by" link to the node of *function1()*.

7.3 Create A Control Flow Graph (or function dependency graph)

This graph is only constructed once on the server side (unless the application changes). Later on, we use this graph to determine which functions will be divided.

Using type inferencing we avoid that Borg's internal functions and natives are also added to the control graph, because these functions should not get divided when the application is divided and they are available to all agents anyway.

If a function has already roughly been divided in the second step, we need to store the continuation functions together with their associated function. So it is important that the continuations are defined beneath their associated function, otherwise we would not be able to recognise them as continuations.

For obvious reasons, the $varADT$ references will be excluded from the control flow graph as well. Though a reference will be added to the node of the function that they belong to.

7.3.1 Caveats

There are some problems that need to be solved before we are able to construct this control flow graph.

We need to pay attention that several functions could be referrering to each other, and thus creating a loop. This problem is easily avoided by keeping checking whether a node has been created for a function before entering this function.

Another problem is: how are able to recognize the fact that a scanned function is in fact a continuation function associated with another function? This problem can be solved if we give the continuation functions identifiable names.

7.4 Conclusion

This small chapter only introduced the control flow graph we create from the code of an application, this graph is intensively used by the last step of the transformation process. The graph contains meta-data of the functions of an application, this meta-data contains references to the memory layer object and the possible continuations of a function.

Since an application does not change often we can save the created graph and re-use it every time the application is executed.

8 The Division of The Application

We now have a partially transformed application and a control flow graph. The last step of the transformation process uses the control flow graph to divide the application and put the parts in agents. The largest part of the resulting agents are then transmitted to the users' computer, leaving some on the server, this kind of placement of the agents provides with the necessary copy-protection.

Since three out of the four steps are already performed and saved, this step is the only that is performed when a user makes a request to execute the application.

The last step of the transformation process has the following (roughly stated) control flow:

- Create the agents
- A graph traverse algorithm that randomly selects functions
- A function that divides the resulting functions and stores them into agents
- Transmit the agents

To avoid the confusion which may arise: if we mention the term algorithm in this chapter, we actually speak of the algorithm (or part thereof) used in this last step of the transformation process; if we mention the term function(s) on the other hand, we refer to the functions of the application that will get divided.

8.1 Preliminary

Before we actually start dividing the application, we first create the agents that will be used to contain the divided functions., because it easier to prepare the agents (and store their names), then to create them afterwards and have a second pass through all the functions in order adjust them with new references.

The number of agents created depends on the job for which they are used. We have to keep in mind that more agents equals more communication between them, so if the application is used on a local (fast) network we can divide the application into more agents then when the application is used across a slow network link (for example, the Internet). This granularity thus heavily depends on the bandwidth and latency between the server and the client. The server which serves the applications can make a decision based on network speed of how many agents should be deployed.

8.2 Select Functions Using The Control Flow Graph

This *select()* algorithm does not simply select functions at random, there are a few criteria to meet. This criteria are specified in the hope to improve performance or copy-protection. One of these criteria is: if we select a function and see that it is referred to a lot, we try to accompany this function by the function that refers to it the most.

When a function is properly selected (by selecting its node in the graph), we retrieve the code of the function by using the table of "pointers" store in the node. As explained in the previous chapter, the node a table which stores the functions (and its possible continuations obtained by the algorithm of step 2) and a variable that stores the memory layer for that function. These two values - the table with the function code and the memory layer - are passed to the function that takes care of the rest of the process (this algorithm is discussed in the next subsection).

Every function that has been selected and divided is marked by a flag stored in the node of that function in the control flow graph. This way we make sure that interfering duplicates are avoided.

Not every function will get divided, there will be some functions that are added to agents without being divided. To decide whether a function will be divided we use a random number generator, this generator will say true in, for example, 90% of the cases.

In order to give the reader a clearer overview of the algorithm discussed here, we wrote it down in the following pseudo-code:

```
select(graph)::
{
   for every node in graph
      if (divideMe?()) {
         agent : divideAndStore(node.getFunctionAGTable(), node.getFunctionVarADT()
         node.markFlag();
         node.functionResidesIn(agent);
      } else {
         agent : store(node.getFunctionAGTable(), node.getFunctionVarADT());
         node.markFlag();
         node.functionResidesIn(agent);
      }
}
```

8.3 Dividing The Functions

This algorithm takes two values as arguments, as stated in the subsection above. For the moment we are only interested in the argument that has the table with the function to divide (and its possible continuations).

A function contains a number of expressions, we made a decision not to divide the simple expressions, since this would only harm the performance. More complex expressions such as loop- and control-structures, that aren't already divided by step 2[1] of the transformation process, are also left in one piece.

Add Continuations

Every function that passes through this algorithm receives an extra parameter in its function header„ this extra parameter is required to allow continuations. These continuations allow a function, after finishing its task, to give control to the function specified in the continuation.

The first thing this algorithm checks is if the function is large enough to justify division, if a function only contains a few expressions (if it only contains up to five expressions the function is considered small) we do not divide it. In the case of a small function, we only add the continuation parameter to the function and store it in a random agent.

If a function is large enough we divide it by traversing its abstract grammar expression per expression. Using a random number generator that decides after how many expressions we slice the function. We then store this part of a function temporarily in a table, because we need to fill in the continuations in after the complete function has been sliced. A new random number is then generated together with the name of the continuation. This procedure is repeated until the end of the function is reached.

Now we have a table starting with the first part of the function, followed by one or more continuations functions. Every one of these continuation functions has a parameter that takes care of the end continuation passed down from the function.

We certainly need an example to clarify this, say we have a function:

```
foo()::
{
   <do something 0>;
   <do something 1>;
   <do something 2>
}
```

[1]See chapter 6.

If we use the algorithm stated above, we get:

```
foo(continuation):: { <do something 0>; foo001(continuation) }
foo001(continuation):: { <do something 1>; foo002(continuation) }
foo002(continuation):: { <do something 2>; eval(continuation) }
```

The parameter continuation is the continuation parameter passed down till the end of the function, the other continuations *foo001()* and *foo002()* continue the execution of the function *foo()*. The algorithm could also have changed the example to:

```
foo(continuation) ::
{
   <do something 0>;
   <do something 1>;
   <do something 2>;
   eval(continuation)
}
```

This resulting function has no internal continuations because it was not divided, it does have the continuation parameter.

Memory Layer

For every continuation function we create, we have to create a unique clone of the memory layer associated with the original function. These memory layers receive a unique name. The necessary functions that handle the memory layer are added to the abstract grammar of the function. And the both the memory layers' co-instance tables are updated with the their respective references. Such a reference is actually the name of the name, plus the unique name for the memory layer instances.

To The Agents

We now have added the necessary continuations, the result is now ready to be stored in agents. We accomplish this by inserting the abstract grammar of the function plus its own continuations in random agents. The procedure to insert the function and its continuations written in pseudo code is

```
for i=length(table of function) to 0 do {
   agent = chooseRandomAgent();
   insertIntoAgent(table of funtion[i], agent);
   if (i>0, adjustFunctionContinuationToAgent(agent));
}
```

The function *adjustFunctionContinuationToAgent()* changes the abstract grammar of the function and its continuations (except the last one, which uses the continuation parameter) in such manner that they know in which agent the continuation resides.

The agent in which the first entry of the table is stored (the head of the function) is the return value of the *divide()* function.

Caveats

When dividing a function that has already continuations, the ones created in step 2 of the transformation process, we have to take care to bundle these continuations in the same agent except for the first and last because these do not belong to the loop- or control-structure. We still need to adjust the continuations used in the functions so that they know which agent to continue to.

8.4 Finishing Touch

The application has now been divided, it is up to the server to transmit a large number of the created agents to the clients' computer. Some of the agents remain on the server to prevent that they get copied. The others are transmitted to the client's computer.

8.5 Conclusion

This last step was merely a quick overview of how the last transformation should be performed, there was no mention of ways to improve the performance of this last step. This last step could performs very bad because a lot of functions can be transformed. Especially the *divideAndStore()* is very un-performant, we go through the abstract grammar several times in order to make all the adjustments.

The division of functions is harder than imagined. Although there is a simple *store()* that only adjusts a few parameters in the abstract grammar before the function is committed to an agent. But if the *select()* decided to divide a function, a lot of adjustments to the abstract grammar are made: the memory layer is clone for all the continuations created, the list of co-instances for the memory layer is extended with the new unique names. The abstract grammar of the function itself is traversed several times: we have to added the continuations, we added the agent names, we add a continuation parameter, etc.

Every function that has been processed is marked by a flag stored in the node of that function in the control flow graph. This way we make sure that interfering duplicates are avoided.

9 Future Work and Possible Enhancements

9.1 Experiments and Remaining Work

Due to a lack of time and some other limiting factors, I could not complete the last two steps of the transformation process, and thus I could not test-drive the complete transformation. Especially the method described in chapter 8 is only a sketch.

But the first two algorithms were tested without many problems. Some of these tests are used as examples in their respective chapters.

Since I was unable to test all the transformation steps on a case, the fact whether this approach to improve copy-protection is feasible still remains, but I have good hopes that it will work if some more work is devoted to it.

While writing this thesis, my advisor came with the idea of neglecting the work done already and instead creating a distributed interpreter. On one side this would simplify the transformation process a lot (for example, no need for continuations, because the interpreter would take care of that); on the other side, a distributed interpreter is very hard to implement (for example, a well performing distributed garbage collector is still not feasible).

Several abilities of the Borg language were neglected, such as objects, agents, global variables/constants, etc. Since this implementation was only meant as a proof of concept this was not really an issue. For example, adding the ability to use global constants would not be hard to implement and they would not even be a burden to the application because they can be cached locally using the memory layer.

9.2 Some Thoughts About Performance Issues

It is obvious that the implementation of the extra memory layer described in chapter 5 is not a very performant one. If an application is divided badly, the variable updates between two instances might create a lot of network traffic and thus slow down the complete application.

Several enhancements can be implemented; a few possible enhancements are listed:

We could implement a data flow analyzer, this would enable us to reduce the number of updates between the memory layer instances. If for example a variable is only used by two function parts out of many, we could limit the updates to this variable to only the two memory layer instances associated with these function parts.

Another option that could reduce the bad effect of a slow network is to install proxy servers. The servers then could be used for proxying the agents that should normally remain on a distant server; the proxy server could also be used for heavy computations and thus reducing the workload on the workstation.

Some optimization could be achieved by eliminating unused continuations, in fact it would be better if step 2 (addition of continuations) was moved to the last step. This would reduce the number of unused continuations. On the other hand, since this transformation process was created in order to improve the copy-protection of applications, the more continuations the harder to copy the application.

Improving performance can also be achieved by introducing redundancy, if we would notice that a certain number of functions always use a function we could duplicate this function and store it locally with every referrer.

9.3 Conclusion

This chapter presented several ideas which - in case the idea is ever going to receive more research - could enhance the performance of the transformation process as well as the end-result. In order to achieve a better performance, every little extra counts! There are enhancements for nearly every step of the transformation process, it would even be better that some steps were merged with others.

10 Conclusion

The lack of time at the end was very unfortunate because I really wanted to see it work. But at end the I learned a great deal while working on this thesis.

As said before, the project in this thesis was mainly made for educational purposes only. There was no intent to make a full fledged industrial usable framework. And unfortunately I was never able to pour all the ideas of this thesis into working code, but since all of the steps described, were based upon older ideas which are already proved, so I don't think the idea of application-servers is an impossible to implement.

The first transformation step was fairly trivial to implement. The scoping rules were a bit fuzzy but at the end, most of the possible Borg expressions can be stored in the memory layer.

The second transformation step proved to be the hardest, the addition of call-back functions to loop- or control structures was not easy and most of the time divided the function in several sub-functions.

The third and fourth step were never implement, so I hope the algorithms are of a good design. The only problem with the construction of the control flow is that we are not really able to determine whether a function is a function of the application or a newly added continuation (from step 2). The fourth step will require some attention because I think the algorithms will be hard to implement. There are a lot of properties that need to be taken care of (for example, with the algorithm mentioned in chapter 8 we traverse various times through the abstract grammar of the function and the memory layer).

There is certainly still a lot of work to do. The bunch of agents resulting from the transformation process are all but performant. The extra memory layer needs more attention in the direction of fault-tolerance and performance.

I can not really provide an answer to the question whether the end-result of this transformation process would be usable to the end-user.

Bibliography

[1] http://borg.rave.org

[2] http://pico.vub.ac.be

[3] Thesis Karsten Verelst

[4] O. Etzioni and Dan Weld, appearing in IEEE Expert, July 1995

[5] Jim Waldo, Geoff Wyant, Ann Wollrath and Sam Kendall. "A Note On Distributed Computing", November 1994, Sun Microsystems Laboraties, Inc.

[6] Theo D'Hondt, http://prog.vub.ac.be/~tjdhondt/POOL/HTM.dir/introduction.htm

[7] Craig, Iain. "The Interpretation of Object-Oriented Programming Languages (Second edition)", 2001, Springer Verlag (ISBN 1-85233-547-5)

[8] Torsten Illmann, Michael Weber, Frank Kargl, Tilmann Krüger. "Migration of Mobile Agents in Java: Problems, Classification and Solutions"

[9] George Coulours, Jean Dollimore, Tim Kindberg. "Distributed Systems: Concepts and Design (Second edition)", 1996, Addison-Wesley (ISBN 0-201-62433-8)

[10] Olivier Danvy, Lasse R. Nielsen. "On One-Pass CPS Transformations", January 2001

[11] Werner Van Belle, Theo D'Hondt. "Agent Mobility and Reification of Computational State", March 2000, Programming Technology Lab, Vrije Universiteit Brussel