A Study Of Communication Models For Mobile Multi-Agent Systems.

Karsten Verelst

February 8, 2001

Acknowledgment

First of all I would like to thank Werner Van Belle for his guidance and his help. For the many hours he has spent working with me. Introducing me in the domain of software agents, and helping me finding my way. Listening to my ideas and giving me feedback. Without Werner, this document wouldn't even be worthy of the name thesis. Thank you.

My dad also deserves a big Thank You for the many nightly hours he has spent, correcting errors and reading this document over and over again. So, if you find any spelling mistakes, blame him :-)

I would also like to thank professor Theo D'Hondt for the trust he vested in me and helping me out when I needed it most. Wolfgang De Meuter and Tom Tourwe for proofreading. My mom for caring and nagging that I should go back to work.

I would also like to thank all students, assistants and profs for making these last four years worth remembering. For all the fun and all the friends I have made.

And last, but not least, I would like to thank myself. If it wasn't for me, this thesis would have never existed in the first place. So, Thank me Karsten! :-))

Contents

Chapter 1

Introduction

1.1 What Are Agents

"Agent" is a very widely used concept. It involves many research disciplines and has many different definitions. Some people use the concept agent for genetic algorithms. Others even consider computer-controlled players in computer games as an agent. And also personal assistants are considered agents. The word agent is currently a much used buzz-word and a definition of the concept is really necessary.

In this document we will define agents as dynamic autonomous active software components. Autonomous reflects that agents must be able to work alone, without interaction. They have to make their decisions themselves. Consider for example a Mars lander. It is impossible that the earth control center tells the robot what to do. It must make its decision autonomously. This example also demonstrates the use of the word active in the definition. The agent has a goal it wants to reach. We don't want a Mars lander to remain passive.

"Dynamic" in the definition points to the environment the agent must be able to work in. An agent should be able to adapt itself to changes to the environment it is interacting with. Again the Mars lander is a good example. It should be able to interact with all kinds of different environments. But the Mars lander is not a real agent, because we defined an agent as a software component.

So what are agent according to this definition? Well, the computer game bots can be considered agents. They act autonomously, in a dynamic changing environment. Some bots even learn from their opponent's behavior. Also some Java applets could be considered agents.

But the agents we are really referring to are the "Patti Maes" agents. Patti Maes, a MIT-researcher has developed a number of personal assistants. Her team has made an agent in the form of a web browser that automatically searches for pages with contents that might interest you. The agent learns what is interesting by observing its master's behavior. Similar to that web browser is a search engine that learns your preferences, compares it with other people that share your taste and therefore only returns pages it thinks to be of interest to you.

1.2 Mobile Multi-Agent Systems

In this thesis we deal with mobile agents. The word mobile can be interpreted in two different ways. We will not regard a mobile agent as an agent that moves around in a room. Robot's designed for descending vulcanoes or Mars landers are not the kind of mobile agents we are discussing.

When we refer to a mobile agent it will be an agent (computer program) that has the ability to migrate to a different computer. Java applets can therefore be considered as mobile programs since before their execution they are moved to the machine where they will be executed. The mobile agents we will be discussing will go a step further and will be able to migrate during there execution. They will even be able to initiate the migration themselves.

Consider for example a network of several different computers. One computer contains a huge database, the other has a powerfull processor and let us say that the third computer has a printer attached. A mobile agent can decide to migrate to the computer containing the database, whenever it wants to access it. From there it can decide to move on to the computer with the superfast CPU to detect some correlation in the data, and then finally it can migrate to the computer with the printer to output the results.

1.3 What Are They Used For

This is just a simple example but the number of possible applications are vast. We can for example create an agent setup that automatically balances the workload of all computers on the network. If one computer is "overworked" a number of its applications can be migrated to another computer.

But there are other more commercial applications too. Suppose that we want to fly from Vancouver (Canada) to Brussels or Amsterdam. To find the cheapest flight, we need to consider lots of different flights. It turns out for example that Vancouver-Frankfurt-Amsterdam is cheaper than Vancouver-London-Brussels. But if we conduct a better search, we can find that the flight Vancouver-Zurich makes a pitstop at Frankfurt and is cheaper than the flight Vancouver-Frankfurt. Thus we can better buy a ticket to Zurich. A friend of us actually made the trip and has spend three days on the internet comparing prices.

So the first thing we would like to do is define an agent that finds the cheapest flight for us. This can be rather easily implemented. A simple graph-search algorithm would do. But the number of flights this algorithm has to look up grows exponentially, and the internet isn't really known for its fast connect times. Therefore it might be better to create a mobile agent that moves to the different airports' computers and conducts the exponential search locally. After it has found the result it reports back home with the cheapest flight.

Conclusion is that the mobile agent can report the best result a whole lot quicker and more efficient (wasting less bandwidth) than an agent performing the computation at the local site.

1.4 Goals Of This Thesis

Mobile agent systems are a relative new area of research and much still needs to be done before a real mobile application can be written that will be able to roam the internet freely.

One of the most primary needs of an agent system is inter-agent communication. Communication has already been extensively researched, especially in the field of concurrent systems design. However the introduction of mobility in an agent system, poses a lot of problems. We will show that the traditional models can be easily adapted to continue to work in the presence of mobility. In fact they even form a good base for a communication model in a mobile agent system.

There are however some drawbacks to most of the current models. Indeed, a lot of models where borrowed from the concurrency research world, and focus on deadlocks and static networks. Other models, more specifically designed for agent systems are too low level or lack the proper communication directives.

Because most current models turn out to be inadequate, we have designed a new high level formalism (called KaSAAC), superbly suited for agent communication in mobile systems. We will prove that this communication model is really as usefull as we claim. And we will even show that KaSAAC provides at least the same power as any of the existing formalisms.

1.5 Structure Of This Document

1. First of all there is this chapter, giving a general overview of agents and mobile agent systems. It also sketches what we want to explore and what goals we want to reach.

2. Then a description of Borg[18] will be given. Borg presents an actual implementation of a mobile agent-system. From Borg we learn what's possible nowadays and what problems exist in mobile environments.

3. The next chapter zooms in at the special needs and problems mobility brings along.

4. It turns out that most of the problems deal with communication between agents. A good communication model would solve many problems. Therefore an overview of different properties of possible communication protocols is in place.

5. In the next chapter we go shopping and have a look at what is available today.

6. It turns out that most of the existing models can be readily unified with mobility. This will be proved by extending Borg with CSP (a communication model).

7. Although CSP, just like the other formalisms, can be unified with a mobile agent system, most of these models lack the proper communication primitives needed by an agent system (asynchronous communication, dynamic agent structure). Therefore we propose a new model (KaSAAC) that beautifully covers all required needs. To validate that KaSAAC really does what it promises, we express a few other formalisms (π-calculus[8], actor system and CSP) in this language. This not only proves that KaSAAC works (correctly, on a high level, and is easy to reason with), it also proves that it is at least as good as, if not better than, any other formalism.

8. We finish by giving a short overview of the conclusions that can be drawn from the research: it seems that existing communication models (possibly with some minor changes) remain valid in the presence of mobility. KaSAAC is at least as good as any other formalism and this new high-level formalism fits the communication needs of an agent system perfectly.

a guide to quickly browse through this document

If you would like to quickly browse through this document, we suggest that you read the introduction. Then you might want to have a glimpse at the existing models. Reading the conclusions of CSP, π-calculus and actor systems should be sufficient. We strongly encourage you to have a look at chapter 7, the KaSAAC model, and finish off by taking a peek at the conclusion.

Chapter 2

Borg

In the previous chapter we have seen what a mobile multi-agent system is, and for what it can be used. Nowadays however, mobile agents can't be straightforwardly expressed in a standard programming language because they lack the support for mobility, security and agentinteraction. In order to run an agent-system on a network we therefore need an environment capable of handling these special needs for these agents.

It is of course necessary to setup this environment on every computer on the network. If we didn't need to setup every computer, we could create a situation in which we could just invade someone's PC without him knowing. We do not have to tell you that this is a major security hazard. Furthermore it is also necessary to create this environment so a remote agent could start executing on this machine. No ordinary PC today, enables a remote program to just start accessing its CPU. The program code of the visiting agent must always be invoked by a local running program. A Unix rexec daemon in the simplest case. If you consider Java-applets for instance, they need their own Java-interpreter. No applet can invade your computer unless you have started your interpreter. Applets are not even a good example since you invite an applet on your computer, whilst agents invite themselves.

Today, several of these environments exist, but since research towards mobile multi-agent systems hasn't progressed very far, they all lack a uniform standard and they're usually used only for experimental purposes. Some of the more popular languages, like Obliq[4] and RMI will be briefly discussed in Chapter 5.

In this chapter we will present an existing environment in which these mobile multi-agents can interact, pointing out the special needs and problems that such a system creates. The system we will describe is called Borg. Borg is a programming language based on an imperative educational language called Pico (Echo is the OO-variant). Both languages have been developed by PROG. The reason we use Borg as a case is that it is representable. We mean that it points out the same problems and techniques used in other mobile multi-agent environments. The reason we use Borg specifically is that we have a background in Pico, and that the source-code of the system was easily available to us, so we could readily implement several communication models, without experiencing restrictions of languages who's source we couldn't obtain.

The reason we give an overview of how the Borg system works is to give you an idea of how mobility can be expressed in an agent-system, and how the syntax of such a language has to be modified. By this explanation we hope to give you a better understanding of the implementation of such an agent-system and the agent-systems that could be created.

If you are already familiar with Echo, the OO-variant of Pico, you might want to skip section 3.1. If you already know what a Borg-program looks like, you can easily skip this chapter.

2.1 Pico-Echo

Borg was based on Echo, a programming language devised by prof. Dr. Theo d'Hondt. The main goal of Pico, an imperative programming language and the predecessor of Echo, is to be an educational language. Easy to learn and easy to understand, so that it can be used to teach the fundamentals of programming to non-computer-science students. Although Pico's primary aim is to be simple, and the core language itself only contains about 60 primitives, it is a complete language equally expressive as every other imperative programming language.

Pico is strongly inspired by Scheme, a Lisp-derivate, but clarifies the syntax and simplifies Scheme even further. For example infix notation has been reintroduced in Pico to make a Pico program more readable by computer illiterates. Pico, just as Scheme, is interpreted, thereby offering more flexibility to the user than a compiled language. Knowing that the fundamental aim is simplicity, it should be evident that the Pico virtual machine is equipped with a garbage collector. Other features borrowed from Scheme include static variable binding and dynamic typing.

Because the first version of Pico didn't contain support for Objects, a second version was created which supports a prototype based approach to Object Orientation. In class based OO, new objects are created by instantiating them from some class definition. For example, a point is an instance of the class Point. In prototype based OO new objects are created by cloning existing objects. To create for example a bunch of rectangles, we write one rectangle including code to calculate the circumreference and the surface. All the other rectangles are then created by cloning this first rectangle and changing some parameters. If you aren't familiar with prototype based languages, we suggest you read on. It will probably become clear, once you have seen the syntax and some examples.

In this short overview of Pico it is impossible to describe Pico in depth. If you'd like to know more about this language, we suggest you take a look at [1].

2.1.1 First Introduction

Atomic Types Since Pico's syntax is so straightforward, we will start by presenting some examples. If you have some programming experience you should be able to grasp the intention of the fragments of code right away.

```
3
:3
3.00
:3.0
'boe'
:boe
"kiekeboe"
:kiekeboe
```

From these examples we can deduce that Pico contains 3 atomic data types: numbers, fractions and text. These are of course self evaluating. Numbers are whole numbers, positive or negative with an absolute value smaller than 2^{30}. They are similar to a C or pascal signed integer.

Fractions are just the same as reals or floats in a more traditional language. They can contain a decimal point and an exponent.

Finally there exists a primitive Text-type, similar to strings in other languages. A constant string can be any text-value enclosed in single or double quotes. It does not matter wether you use single or double quotes as long as the first and the last one are the same.

Native Functions

```
2*3
:6
+(2,3)
:5
5>3
:<native true>
size([1,2,3,4])
:4
```

It is very intuitively how to use a native function (native means a standard built-in function). In Pico there are about 60 natives, which can be found in most other programming languages. Pico supports natives for simple mathematics (+,*,-,/,...), comparing (=,<,>,...), conversion (trunc, abs, ...), advanced mathematics (sin, cos, sqrt, ...), identifying (is_number, ...), tables (size, tab), control-structures (if-then-else, while, for, ...). Something quite peculiar about Pico are its text-processing natives. There are 2: implode and explode. Explode converts a Text into a table of characters (Text containing only 1 character). Using implode you can convert a table of characters back to a text. All text-processing primitives can therefore be substituted by table manipulation.

```
explode('kiekeboe')
:[k i e k e b o e]
```

Primitive Data Structures: Tables

```
table:[1,2,3,4]
:[1 2 3 4]
table[2]
:2
size(table)
:4
```

Because tables are one of the simplest data structures to grasp, they are the only complex data structure supported by Pico. Since all other data structures can be constructed using tables there's no need to complicate the language by introducing extra data structures. As you can deduce from the example, a tables first index is 1

Constants: booleans, eoln, void

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

True, false eoln and void are special Pico constants. We suppose their names are selfexplicatory. eoln is the special end of line constant. It is just a newline character. true and false are implemented as church booleans (λ-calculus). That's why they are displayed as native functions.

2.1.2 Control Structures

```
{display(1); display(2)}
:12
begin(display(1),display(2))
:12
if(5>3,1,2)
:1
x:1;
while(x<10, {display(x); x:=x+1})
:123456789
x:1;
until(x>10, {display(x); x:=x+1})
:12345678910
for(x:0,x:=x+1,x<10,display(x))
:123456789
```

As every normal language, Pico contains a number of control structures: if, while, until, for and the instruction sequence. Several expressions can be sequentially executed by separating them by semicolons and enclosing them in braces. Again this notation should be familiar from more conventional languages like C, Pascal, ... Actually this form is just syntactic sugar and is replaced by the reader in its simpler form of a function call. Actually an expression sequence is just a call to a native begin that evaluates its arguments from left to right. Both forms are valid in Pico, although usually the first one is preferred by the programmer. The result of the evaluation of an expression sequence is the result of the last expression.

The next control structure is the if-then-else construct. The first thing you might notice about it, is that it is no special language construct but just a simple function call. As we will see a bit further, it is real easy to implement delayed argument evaluation in Pico, so all control structures can be easily implemented by a function. Even the user can add some extra control structures without any problem. Although the semantics of the if defers a little from the more conventional programming languages, you should be able to spot the analogy. Since booleans are implemented as church-booleans, the if can be borrowed from the λ-calculus

The while, until and for loop's semantics are analog to C,pascal but their syntax defers a little. The special syntactic form in conventional languages is replaced by a simple function call.

2.1.3 Definition, Declaration And Assignment

```
x:5
:5
x
```

```
:5
x:=x+1
:6

y::10
:10
y
:10
y:=y+1
  -> error
```

In the above example we have a definition, a declaration and 2 assignments in its simplest form. A definition is just defining a variable. It is just like int x=5 in C. A definition in Pico creates a new variable, and assigns it a value. We use the ":" to denote a definition.

A declaration corresponds to a constant declaration. A constant can only be defined once and never reassigned. The constant's value is initialized at creation; the value being the expression after the "::". Since functions and operators are also first class, they are also defined using the ":" and "::" syntax. Since most functions don't change over time, it is best to declare a function as a constant. As we will see later, this has a serious impact on object cloning.

Finally assignment is just analog to almost any other imperative language. We use ":=" to reassign a variable to a value.

Simple variables are not the only types that can be assigned. It is also possible to define tables and functions.

```
t[3]:5
:[5 5 5]
{i:0; s[5]::i:=i+1}
:[1 2 3 4 5]
s[3]
:3

fac(n)::if(n>1,n*fac(n-1),1)
:<function fac>
fac(5)
:120
```

Just like variables we can define and declare tables. The distinction between declaration and definition still remains the same: declaration declares a constant. A table is defined/declared by evaluating the argument n times, n being the size of the table. This way we can easily instantiate tables with a dynamic size.

Defining a function is just as easy as defining a variable. You just write the function name, and its parameters, on the left side of the ":" or "::" and then on the right side you put the function body. The head of a function declaration can take many different forms, each form resulting in a, sometimes subtle, difference in parameter binding. All these different forms will be explained in the next paragraph.

Invoking a function on an argument is equivalent to invoking a native on its arguments. You just write the function name, and then between parentheses the actual parameters separated by comma's. In the same way that we defined the functions above, we can define new operators, infix or prefix, unary or binary.

2.1.4 Functions, Arguments And The Apply-operator

```
id(x)::x
:<function id>
id(10)
:10
sum(x,y)::x+y
:<function sum>
sum(5,3)
:8
sum@[5,3]
:8
```

These are all just simple function definitions and function calls. Maybe you should take a closer look at the "@"-operator. This apply-operator applies a function to a table of arguments. Now in Pico you can also define functions that take a variable number of arguments using this operator.

```
varargs@args::for(i:1,i:=i+1,i<=size(args),display(args[i]))
:<function varargs>
varargs(1,2,3)
:123
varargs@[5,10,15]
:51015
```

From this example we learn that if we use the "@" (apply) operator in a function definition, all arguments will be collected in a table. With this technique we can easily write functions that take multiple arguments. Finally, in Pico a third, really innovating, way of parameter binding, called functional parameters, exists. A functional parameter is a parameter, that is bound to a function. The formal parameter is bound to a function, with the actual argument as function body. Some examples will probably help you understand this new kind of parameter binding.

```
afunction(f(),g(x)):f()+g(5)
:<function afunction>
afunction(5,8)
:13
f2(g())::{display('before');g();display('after')}
:<function f2>
f3(x)::{display('before');x;display('after')}
:<function f3>
f2(display(' Kiekeboe '))
:before Kiekeboe after
f3(display(' Kiekeboe '))
: Kiekeboe beforeafter
myif(cond,t(),e()):if(cond=true,t(),e())
:myif
myif(true, display(1), display(2))
:1
```

We hope these examples did help you understand functional parameters. What actually happens is that the functional formal parameter is bound to a function, with as function body the actual argument. Note that the actual parameter isn't evaluated yet. It is transformed to an unevaluated function body. If we then call this function, say 5 times, then the argument is also 5 times evaluated. Functional parameters therefore allow us to introduce lazy evaluation in a very simple may. Another interesting property of these functional parameters is that they make it possible to introduce some kind of dynamic scoping. When using a variable in the actual argument of a functional parameter, the variable is resolved in its original environment, resulting in some kind of dynamic scoping:

```
x:5
:5
f(g())::{x:10; g()}
:<function f>
f(display(x))
:5
```

2.1.5 Object Oriented Programming In Pico

Creating Objects The programming language Echo further extends Pico by introducing prototype based OO. In most object oriented languages, objects are created by instantiating some class. A class is just a prescription that describes a new instances' functionality and data. It is just a model you use to create objects.

Prototype based OO uses another approach. We just write the code for one object and then we clone the object, creating as many clones as we like. Say we would like to create a rectangle class, then we create a first rectangle, writing down all instance methods in this object. Then to create a second object we clone the first rectangle. This is opposed to class based OO where we would start by creating a Rectangle class, and then instantiating rectangles from this class. It is possible in prototype based OO to model inheritance, so prototype based OO, and class based OO are equivalent. In Pico prototype based OO was preferred, because it feels more natural to people without much computer experience.

```
createRect(width,height)::
{
   surface()::width*height;
   circumreference()::2*(width+height);
   clone()
}
:<function createRect>
rect1:createRect(5,3)
:<Dictionary>
rect1.surface()
:15
rect1.circumreference()
:16
```

Implementation Using Dictionaries You might have noticed that an object is actually represented as a dictionary. Perhaps this needs an explanation. A list of dictionaries is used as Pico's memory model. A normal memory model is nothing more than a list of associations between variable-names and their values, in short a dictionary. There is one special dictionary being the global environment containing all natives. Every time a function is called, the dictionary is extended: a new dictionary containing the functions' parameters and a link to the previous dictionary is created. Using this link, a stack of dictionaries evolves, thus enabling scoping. Since Pico is statically scoped, a dictionary's upper dictionary is the dictionary in which the function was defined. This way we get a chain of dictionaries, always ending at the global dictionary.

So the dictionary that is returned, by cloning an object is just a piece of Pico's memory model containing all data private to that object. Actually it is a bit more complicated. For performance reasons only the private variable part is cloned. The constants are shared by all equivalent objects.

Self, Super When we want to use inheritance, this can easily be done by defining an object within its parents scope. This way, the object's upper dictionary is the parents dictionary, easily allowing a child to access its parents methods, and creating the possibility of method overloading. Since an object is a dictionary, it is obvious that self, is the objects own dictionary. The super-object can be obtained by taking the upper dictionary. In Echo the special constants self and super are defined, always pointing to the correct self-object and the corresponding super-object.

```
CreateSquare(width)::
{
   surface()::width*width;
   circumreference()::4*width;
   CreateRect(height)::
   {
      surface()::width*height;
      circumreference()::2*(width+height);
      clone()
   };
   clone()
}
:<function CreateSquare>
s:CreateSquare(5)
:<Dictionary>
r:s.CreateRect(3)
:<Dictionary>
s.surface()
:25
r.surface()
:15
```

2.2 Borg: Agents, RDCs, Syntax, Setup

As we have mentioned before, Borg is an extension of Echo, enabling the creation and migration of agents. The reason we have taken Echo as base of our agent system, instead of a more popular language like for example Java or C is flexibility. At the KUL, Bert Robben and his team have developed a mobile agent system, Correlate[**?**], as an extension to C++ and Java. The major problem with this approach is that the existing languages pose too rigid constraints on the system, resulting in lots of syntax-extensions, precompilers, awkward constraints, . . .

2.2.1 Pico versus Java

Before choosing for Echo, we have considered using Java to express mobile agent-systems. The Java language already offers support for threads and serialization.

If you haven't any experience with Java, threads are concurrent running Java processes in the same dataspace. Java enables you to run several pieces of code concurrently. For example if you use the Java AWT, there will be always one invisible thread checking what the user is doing, and invoking the correct listeners. There is always a little program, running concurrently with your program, checking for mouseclicks and keystrokes and acting on them.

If we want to implement a system of concurrently running agents, threads come in handy. However, threads alone do not permit us to migrate code between several machines. To be able to take an agent, and migrate it we need to serialize this agent at one end of the network, move it through the network to its destination and deserialize it at the other side. Serializing and deserializing are one of the most difficult topics in agent-migration. We need to capture the agent's runtime-stack and its memory, and convert it into a format that can be sent over a network, and turn it back into an agent at the other end. This process of serializing is virtually impossible in Java. In Java it is impossible to get information about the runtime stack, without implementing your own virtual machine. This is not what we want to do. So although Java seems pretty appealing at first, an a bit more thorough investigation shows that quickly a lot of problems arise.

We have chosen for Echo because Echo is an interpreter. Its implementation in C uses an explicit stack with continuations. The Pico interpreter is implemented as a stack machine, pushing and popping "thunks" on the stack. Therefore in Echo it is no problem to access the stack, and send it over a network. Furthermore Echo containes already some notion of parallelism. In the Echo-environment you could open up to 10 windows, running a different program in each window. To achieve this, Echo has a basic process scheduler built in, further simplifying the implementation of agents. There is however one giant problem with these concurrent processes: they cannot interact. In an agent-system, especially a mobile one, we need to be able to let different agents communicate with each other.

2.2.2 Agent Creation

As we have just explained, Borg is based on Echo. We have also explained that Echo already contains support for several concurrent, non-interacting processes. When Echo was extended to an agent-language, it was evident to extend these processes to agents.

In Echo we created a new process by clicking the new-window icon. This creates a new window, where we can enter text, evaluate programs and output results. With each window, a different process is internally associated. Each process has its own stack and has its own private part of memory, separate of all other processes, not allowing process-interaction.

Borg has extended these processes to agents. Now when you click the new-window icon, a new window pops up, and internally a new agent is created in stead of a process. The major difference between an agent and a process is its ability to communicate. As we will see later, agent communication is based on asynchronous messages. Therefore every agent needs a queue containing the messages it has received. Thus an agent's main characteristics are: a stack, a memory containing variables and functions, and a queue containing messages.

Borg's primary idea of communication is based on the remote function call paradigm. The remote function call is based on a communication method that's popular for distributed OO-systems. OO is all about sending messages between objects. If we distribute these objects, we would have to send messages to objects on the other side of the network, resulting in something similar to a remote method invocation. Borg extends this idea even further to a remote function call. Every agent can contain a lot of functions and a lot of objects. A remote function call simply is a way to start the execution of a function an a different agent. To simplify, any agent can just say to any other agent: please run this function for me with these parameters.

agent1	agent2
	sum(x,y)::display(x+y)
	:<function sum>
agent2.sum(5,8)	
:<void>	:13

If we create a new agent, pushing the "new window button", this agent is automatically initialized to check its queue when its idle. If the queue contains any messages, these messages are executed. If the queue is empty the agent is suspended, until a new message arrives. If a message arrives, when the agent is already evaluating another request, the message is queued. Even when the user enters a program in the agent's window and request evaluation, a message is put in the queue, waiting for the agent to complete its current computation.

Pressing the "new window button" to create a new agent isn't very dynamic. Using this technique no agent can spawn a new agent on its own. This is a major drawback but it has been solved by adding the agent-clone primitive to Borg. The agent-clone operator however is quite complex to explain. It will therefore be explained in a few paragraphs, once we have a better understanding of Borg. Creating agents this way has one big advantage: every agent has its own window. If every agent has a window, the systems output is nicely sorted per agent, making it easier to debug applications.

2.2.3 The Borg Environment

From the above section it should be clear how to create several agents on a single machine. Borg is more powerful and can be setup on several machines on a network. Agents are then free to roam all machines that have the Borg environment installed. We can even setup several Borg environments on a single machine. This proves usefull during the development phase of an mobile agent system, or on systems that lack network support. With this technique we can also simulate an agent system with more nodes than there are machines in the network. If we run several environments on a single machine, concurrency will be provided by the operating system. This has its drawbacks because some operating systems have a real poor performance when it comes to taskswitching.

In Borg every machine is supposed to have a name. This name can be arbitrarily chosen and has absolutely no connection with network names, IP-addresses or any other machine-dependend properties. Borg works with names like "Hello" and "Kiekeboe" for machine names. Because of the independence between Borg machine-names and the actual machines, Borg systems are portable. We can develop an agent system on our own local network, maybe even on our own computer, using the little trick of spawning different environments on our machine, and then we can just port the entire system to a different network, having absolutely no connection with our system whatsoever. To obtain this level of portability, Borg works with configuration files. When we install the Borg environment on a computer we have to adapt a little configuration file. This file specifies the name we wish to give to this environment.

A sample configuration file might look like this:

```
name(elephant)
```

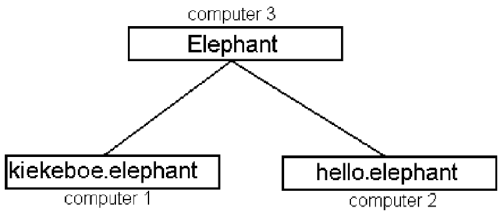
This file is a the most basic configuration possible, configuring our network for only one host. This file is valid in the case where we only have 1 machine, running 1 environment. The file specifies that this machine will be referenced to as *elephant*. If we create agents in this system they will be called something like elephant/ses1 and elephant/ses2.

The configuration file is also used to find other computers in the network that are willing to take part in the agent-system. Because Borg machine-names are independent of machine-specific properties, Borg has to find its own way of routing messages through the system. To do so, Borg creates its own hierarchy of nodes, implementing routing on top of this hierarchy. Every node has a top node or a forwarding node. If we originate a message from the current machine, the current machine will try to deliver the message itself. If the current machine can't satisfy that request, it will send this message to its forwarding node. This forwarding machine, will detect if the message is destined for any machine under its "domain" and if so forward the message to the correct host. If the forwarding node can't deliver the message either, the message is sent up one level, to the forwarding host's forwarder. This might sound a bit abstract but it will certainly become clear in the section about routing.

What you need to know now is that the machines, willing to participate must be placed in some kind of hierarchy. This hierarchy is specified in the configuration file using the forward keyword.

echo.machine1	echo.machine2	echo.machine3
name(kiekeboe) forward('elephant's IP')	name(hello) forward('elephant IP')	name(elephant)

This series of configuration files specifies the following network structure:



In this little network, we have 3 machines: kiekeboe.elephant, hello.elephant and elephant itself. Both kiekeboe.elephant and hello.elephant forward their messages to elephant.

Finally their is the possibility to create several environments on a single machine. This is done by running each environment on a different port. This port can also be specified in the configuration file.

These configuration files are quite flexible and allow the user to easily adapt his agent system to any network he likes. Even when suddenly an extra machine becomes available on a network, it can be added to a running agent system, by adapting that machine's configuration file and starting up the Borg environment.

2.2.4 Remote Dictionaries

It should be clear how to setup an agent-environment and how to create agents. The next step we need to take before we can migrate agents and let them communicate, is to create references to agents. The original idea behind creating references to agents is based on distributed OO languages.

In a distributed OO system we start from the observation that objects are encapsulated data structures, containing within them all functions that operate on them. Since objects are so nicely encapsulated, only admitting interagation through message passing, they are ideally suited to be distributed, by putting each object on a different server. Communication is performed by remote method invocation. The only difference in a distributed OO system, compared to a traditional, single-machine-program, is that a reference to an object can be a reference to an object on a different machine.

For Borg this idea of distribution forms the base for the language. In Borg an object is internally represented as a dictionary, containing all the objects private variables and methods. Now to make this system distributed, we need the notion of a remote object. Since an object in Borg is actually a dictionary, distributed objects are therefore referenced as a remote dictionary (RDC).

2.2.5 Communication

Remote Method Invocation Now that we can create references to objects, even on other machines, it might be usefull to have a look at communication between these objects. Communication is realized by remote method invocation. We send a message to a remote object in the same way as we send it to a local one.

```
localobject
:<dictionary>
remoteobject
:<remote dictionary>
localobject.method(param1, param2);
:<result>
remoteobject.method(param1, param2);
:<void>
```

The syntax for remote method invocation is identical to the local variant. Also, the main goal, execution of some code, is the same. And of course, both forms of invocation, can take a number of arguments. There is however a major distinction. As we gather from the example, a local method invocation returns the result of the method whereas the remote variant does not wait for a return value. It simply returns void. We thus have asynchronous message passing.

This has some advantages. We don't need additional constructs to provide parallelism for example. Since message passing is asynchronous, we can send a message to 2 different machines, letting them both work at the same time. Using synchronous message passing, our computer would hang, until a result is received. Only then, can the second machine be sent its message. Thus, using synchronous message passing would require an additional language construct. Something like a par-instruction for example.

Another major advantage of working asynchronous is that our local machine does not become dependend on another machine. If the other machine crashes, our local machine can keep working, whilst it would hang, waiting for a result in the synchronous case.

Callbacks There are however also a few drawbacks to this approach. First of all, and maybe this is the system's major flaw, returning the result of a method can be quite difficult. Suppose we send a message to another machine, requesting for example a difficult computation. When the other machine has completed its computation, we would like to receive the result somehow. We don't know when the other machine has finished, and even if we do there is no clear way how to get the result. We use callbacks to get the result back. A callback is the technique where we send, together with the RMI, a parameter that specifies a method that has to be called with the obtained result. When the remote method has found the result of its computation it will call the callback function with the result as parameter. This way we can get the result back to the machine that originated the request. This is quite a lot of overhead but the only way to get the result back to the originator.

```
CreateObject()::
{
  start()::
  { Matrix : ...;
    ...
    anotherObject.determinant(Matrix, self.receiveResult);
    ...
  }
  receiveResult(determinant)
  {
    display('The matrixes determinant is ');
    display(determinant);
  }
}

CreateAnotherObject()::
{
   determinant(Matrix, sendresult)::
   {
      determinant= ...;
      sendresult(determinant);
   }
```

```
}
```

Although this technique works, we think it is rather cumbersome. For every method we invoke, we need a different result receiver. Of course this causes a lot of overhead. Further it turns out to be quite difficult to just pass a value to an agent. People in favor of this communication model claim that these problems never occur if the application is well designed.

Argument Passing We have just demonstrated that it is possible to call functions on another machine. Of course we can pass arguments with a remote function call. This is where problems might arise. Normal argument passing is done by the call by value mechanism. The arguments are first evaluated on the client site and the result of this evaluation is sent as actual parameter to the remote function.

This mechanism has been used for many years in many languages. However it poses problems in distributed OO systems. The problems occur when we pass objects with inner state as a parameter. There are 2 possible options: we send a copy of the object to the remote function, or we create a reference to the current object and send this reference to the remote site. The first approach has as major drawback that after a while we have a lot of copies of the same object, each with its own inner state. Furthermore all these copies are out of sync. They can have a different inner state and the program might not behave like it should. Maybe to clarify this, a small example. Suppose we have an object that registers how often a certain method has been called. When we pass this object as a parameter a copy is created. The remote site will work with the copy, the local site with the original one. As you can see, after a while, neither of the counters inside the object or the copy is correct.

The other alternative, passing a reference, solves this problem but introduces a couple of new ones. First of all, we have the performance issue. Every call to this reference is translated to a remote function call to the original machine. Furthermore their might rise some concurrency problems. We have an object with inner state, that can be manipulated by several threads at the same time. Then of course we can run into trouble. Just imagine two processes who update the same instance variable at the same time. A solution to this problem can be provided by queuing all messages to the object, but then again we might run into infinite loops and deadlocks. Another solution could exist in supplying the programmer with the necessary deadlock-preventing mechanisms extending the language and making programs unnecessarily complicated.

2.2.6 Cloning Agents

Creation As explained, the way to create a new agent is to tap the “new agent”-button. This method is not powerfull or dynamic enough. We have mentioned that there's another way but it requires some more detailed knowledge about Borg. Well, now that we have seen remote dictionaries, we wil explain how new agents can be dynamically created.

```
CreateSomeObject::
{
   instvar1:...;
   instvar2:...;
   method1()::...;
   method2()::...;
   doit()::display(done);
   clone();
}
<closure CreateSomeObject>
anObject:CreateSomeObject()
<Dictionary>
agentclone(anObject,'tecra/elephant');
<Dictionary>
c:remotedict('tecra/elephant');
<remote dictionary>
c.doit()
tecra/elephant: done
```

The principle behind creating new agents is based on the same principle we use to create new objects: cloning. We take an existing agent and we clone it. Then we have 2 identical agents, which can operate independent of each other. We can invoke messages on both agents. To start a computation on a different computer, we clone the current agent, move the clone to a different machine and send it a message to start its evaluation.

There still is the subtle matter of where output of the clone should be sent. If the clone displays or accepts user information, where should it be sent? Well, in the current release of Borg, the output of the clone is sent to the window of the original agent. When we start migrating the agent, the output rules for migrated agents apply. The output is then sent to the debug window of the new host. After an agent has been cloned, both agents further behave normally.

Termination

```
agentdie()
<Now the agent does not respond to anything you type any more.>
```

```
CreateSomeObject::
{
   instvar1:...;
   instvar2:...;
   method1()::...;
   method2()::...;
   doit():: { display(done); agentdie() }
   clone();
}
<closure CreateSomeObject>
anObject:CreateSomeObject()
<Dictionary>
agentclone(anObject,'tecra/elephant');
<Dictionary>
c:remotedict('tecra/elephant');
<remote dictionary>
c.doit()
tecra/elephant: done
<The newly created clone has just been killed. It does not accept any messages any more>
```

Another functionality needed by a dynamic agent system is the ability to destroy an agent. This is done by the Borg native "agentdie". This function simply destroys an agent, killing its current computation and making it no longer possible to communicate with it. The process then no longer consumes CPU-time. It is just erased from the computer's memory. There is however a subtle problem with this implementation: it is possible that other agents still have a reference to the agent that died. This reference is not changed in any way. When killing an agent the other agents, don't know that it is gone. If the programmer wants to notify any other agents before the agent dies, he should do so manually, by sending a message.

Because of security reasons, an agent can only kill itself. Usually we don't want to have agents, running around the network killing all other agents, claiming all resources for itself. Well, maybe we want to have some kind of supervisor agent, limiting the other agents CPU-time, or memory requirements, but then we could run into a possible security hazard. If we want to be able to implement these kind of agents we need a special security model giving permission to these special agents to do so.

The problem of a secure mobile multi-agent systems is far from solved. Some implementations today pose too rigid constraints, like for example Java applets, who restrict almost everything, leaving only a small subset that may be used. Other environments often implement security by giving each agent a name. We can only communicate with an agent through this name. So no-one can just start communicating with an agent before receiving this agents name. Security then automatically follows. An agent is not even aware of other agents on the same machine, with exception of the ones who told him their name (normally an agent only knows the agents he's supposed to interact with)

2.3 Migration

So far we have described a simple agent language, with the ability to dynamically create and destroy agents, create references and let all agents communicate with each other. However we have said nothing about Borg's most interesting capability: agent migration. In Borg we can migrate agents from one server to another.

```
display("before moving");
agentmove(remotedict("otherplace.elephant"));
display("after moving") }
```

This is a very simple example that migrates half way its execution to another machine. When we execute the instruction agentmove(remotedict("otherplace.elephant")), the computation is resumed at the machine otherplace in the domain elephant.

Borg even surpasses most existing systems because it can move agents at any time during their evaluation, and not at certain specified points. This technique is called telescripting. There are two different mechanisms for migrating agents. The first technique only allows agents to be moved at certain specified points in the code (for example when the agent is created). The other method, called telescripting allows the agent to be migrated at any point in the program. Of course telescripting is much more interesting and dynamical than the other method, and it allows the migration aspect to be handled my a management layer. We need telescripting when we want to create an automatic loadbalancer for our network.

When a running program encounters the agentmove instruction, it moves itself to the new machine, taking with it, its entire computational state: its stack and its memory environment, together with some less important variables.

All this state is first serialized on the local machine. This means that the entire stack and memory structure is transformed into something that fits on a network cable. Usually a dictionary (Borg's memory model) contains references to itself, thereby creating circular structures. Then this entire state is moved over the network to the new machine. There the agent is deserialized: it is transformed again into a valid agent, with which the interpreter can work. When the agent is deserialized it just continues its computation, allbeit it now on a different machine.

The agentmove instruction can be invoked from really anywhere in the code. There have even been experiments, where agents themselves were made multithreaded. One thread was performing some kind of computation while the other suddenly decided to move. This worked perfectly. The computation was neatly transformed to the other machine.

There are still a few loose ends waiting to be tied concerning the moving of agents. The main concern being the problem of where to put user output (and of course gather user input). When we create an agent by tapping the "new agent"-button, a new window is popped up. It is of course to this window that output is written to. When we create an agent by cloning an existing one, the parents window is shared by the parent and the clone. Now when we move an agent around, where should the output go to? Displaying the output on the remote machine is not what we really want. We want to see all messages on our local window, independent of the location of our agent. This has been achieved by seperating the UI-layer from the agent. A single agent consists now of 2 agents, one that performs the computation, and one that puts all output on screen (a so called UI-agent). When we migrate the agent only the computation migrates, but the UI-agent retains its position. As a result all output is always written to the local screen but the computation can take place anywhere.

2.4 Internal Architecture

Knowing the syntax of a correct Borg-program, it might be usefull to know how Borg is actually implemented. This is also necessary to fully understand Borg's routing system and serialization procedure.

Borg is based on Pico, a simple programming language. It evaluates programs by the simple concept of the read-eval-print loop, exactly like in Scheme. Borg works internally like a stack machine. It contains 2 huge stacks. One stack contains continuations(thunks), the other stack holds values and arguments needed for the evaluation of these thunks. However in the rest of the explanation we will sometimes refer to Borg's stack. With this we mean the combination of both stacks. In the C-implementation these 2 stacks are implemented as 1 table anyway.

When a program or instruction is entered in the Borg environment, it is first run through a scanner and a parser. These 2 components translate the Borg-program into an abstract grammar, a treelike structure that represents your program in such a way that the computer can easily work with it. Then we start with the evaluation of this abstract grammar. To evaluate this tree, we start pushing and popping thunks on the stack. When for example we want to evaluate an addition of two expressions, we will first have to evaluate both arguments, and then add the two results. In a stack machine this is done by first pushing a thunk "eval-+", then a thunk to evaluate the first argument and a thunk to evaluate the second argument. On the expression-stack (the other stack containing values) the two arguments are pushed.

After Borg has evaluated the +-thunk, the result of the sum will be pushed on the expression-stack after the two arguments have been removed. If you don't grasp this idea at once, don't panic. It is the topic of an entire course, and so it just can't be explained in a few lines. Just remember that Pico works internally with a stack. This stack contains everything the interpreter should still do and thus is a snapshot of the evaluation. With this snapshot we can at all times restart the evaluation.

As you can see, it is this stack that gives Borg the power to move agents over a network. With this stack, we know at all times what we still have to do in order to evaluate the expression. This is also the reason why this mobile multi-agent system was implemented in Pico and not in a language like Java. In Pico we have easy access to the stack making it easier to move it around. We know the computation at all times. In Java however, we haven't got a clue of what's on the stack. The stack itself is managed internally by Java and it cannot be accessed, let alone be moved around.

Apart from this stack, Borg also contains another important data structure, being a dictionary. This dictionary is used as Borg's memory model. When we look at a memory model real abstract what does it contain? Some bindings between variable names and values. Now if we hear the word binding, we automatically think dictionary. Since Borg was developed as an educational language, and simplicity is rated more important than performance by its designers, Borg's memory model has been implemented as a dictionary.

Actually when we look at the C-code it is a bit more subtle. The dictionary containing all variable bindings has been replaced by two dictionaries. One dictionary containing all constant declarations (variables declared using the "::"-operator), and the other dictionary containing all variables (defined using ":"). The only reason for this second dictionary is to prevent the programmer from changing a constant-variable's value.

Now we have almost grasped Borg's memory model. There's only one tiny difference with the real implementation left. Borg keeps a list of dictionaries. Every node in this list represents its own little environment, needed by function calls and objects. When we just start Borg, our dictionary only contains some native functions (the global dictionary). If we then call a function, an extra environment is added to the list of dictionaries, ensuring static scoping.

Finally there's just one extra bit of state we need to capture: the message queue. Borg uses a method of asynchronous message passing to communicate between agents. To implement this kind of communication we need a message queue, that is able to hold all messages that have already been sent to the agent, but haven't yet been received. When using the model of remote method invocation, the queue contains all messages that still have to be invoked on the object.

One of the more intriguing aspects of Borg is its delicate balance with its meta level. All the data structures, that have just been mentioned: the stack, the dictionary and the queue, are actually implemented in Borg. The stack and the queue are both implemented as a Borg table. The dictionary is a Borg list. This might seem awkward at first, but it has proven usefull on many occasions. It are already Borg data structures anyway. This technique has also proven usefull on other occasions, because we don't have to make any exceptions to garbage collect (gc) the stack, queue or dictionary. Actually these structures even serve as root to the gc. Even agents are implemented as a Borg data structure.

This almost sums up a Borg agent. All we need to do when we want to take a snapshot of an agent, is copy its stack, its memory model and its queue. Now if we want to migrate agents, we only have to serialize the stack, dictionary and queue, send them over the underlying network, receive them at the other side, and deserialize them again. Then the agent can continue its computation, albeit on a different machine.

2.5 Router

As we have mentioned before, Borg implements its own router system on top of the underlying network routers. Implementing this router is essential if we want performant network communications in a mobile agent environment. Because agentnames are independent of machine-specific properties, Borg has to implement its own way to find agents across the network. This can again be elegantly solved by the router system.

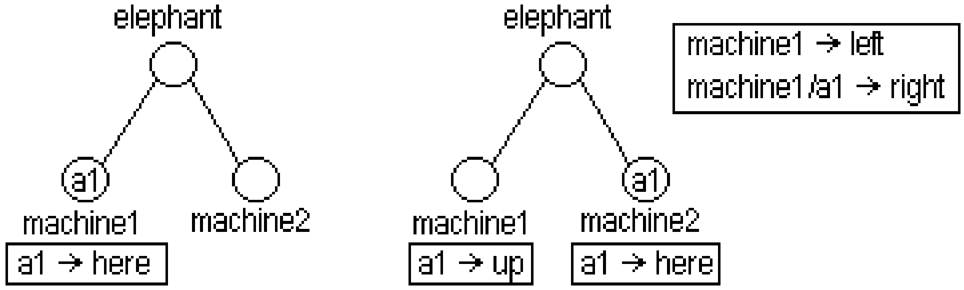
The router is set up using a configuration file. By adapting this configuration file, we can easily make our agent system portable (independent of the network we're using). Adding a machine involves creating an new configuration file. The configuration file contains a mapping between agent names and machines, and further specifies the hierarchy which is used to route messages. Machines running Borg are interconnected in a treelike structure. To route a message to its destination, there will first be checked if the message is for the local machine. If not, it is sent up one node in the tree. This node will examine the message, and determine if its destination is one of its children. If so it is forwarded there, if not, it is again forwarded one node up the tree.

The routing of messages in a mobile agent-system becomes especially difficult if we know that the agent "elephant/ses1" might have moved to, say for example, machine "Hello". To solve this problem, it is possible to create stubs in the original place where the agent was created. These stubs should then forward every message to the correct place where the agent resides. When we start migrating an agent, we could create a linked list of stubs, each forwarding the message until it has reached his destination. This has a huge performance overhead, increasing network traffic, and slowing down communication speeds.

A better option is to create only one stub and update this every time the agent moves, so that it always points to the correct place. This does not leave huge trails of forwarders but, implementation is very difficult and could result in message loss if not carefully implemented. If an agent wants to move and updates his stub then it could be that the agent has already moved to his new location, before all messages en route have arrived at the old location, resulting in the loss of several messages.

Using a self implemented router system, this problem can easily be solved. When we migrate an agent, we update every router it passes on its way to its new location. Because routing a message results in following a path through a tree, all messages will arrive correctly, since the tree is correctly updated.

updating the routing tables when a1 moves from machine1 to machine2



agentmove(remotedict("machine2.elephant"))

Although implementing an entire router and message passing system looks quite a useless overhead at first, it really proves usefull when we examine the system a bit further. Not only does a self implemented router, handle the independence between agent names and machine specific properties quite nicely, it further proves to be the ideal mechanism to get messages to an agent that might be roaming the network.

2.6 Serialization

Serialization is a complicated process that is responsible for transforming Borg expressions into a format that can be sent over the network. Deserialization is the inverse process. The process of serialization/deserialization is used every time a Borg expression has to be sent over the network. This can be to send a simple message to another computer or even to transfer an entire agent over the network.

Normal Borg-programs are represented at run-time by some kind of graph. Serialization is the process of transforming this graph into a flat structure (a table or a string) that can be sent over the network. Deserialization is the inverse process: we take a serialized flat structure and kind of inflate it again. After inflation we'd like to have the original structure again.

Serialization is used in different parts of the Borg interpreter. First of all there is the migration of agents across the network. In order to migrate an agent it must be serialized and deserialized. Further it is also used for simple message passing. Borg supports remote method invocation as communication standard. This involves sending a message name and some parameters over the network. It are especially these arguments that have to be serialized. Borg has also been extended with other communication protocols (CSP-like send and receives for example). These forms of communication also require serialization.

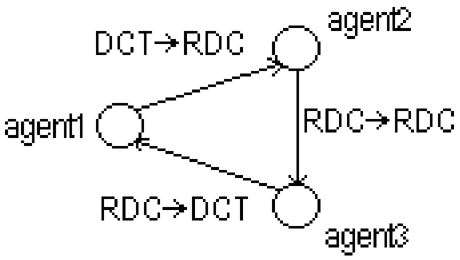
Now as we have mentioned in the section about Borg's internals, almost everything is represented as a Borg data structure. Even the most basic data structures used in the language's implementation have been absorbed. This has the major advantage that all network transit can be performed by serializing nothing but Borg expressions. Even the serialization of an entire agent, with its stack, queue and dictionary comes down to serializing some Borg expressions.

Since all objects that have to be serialized are stored in Borg memory, serialization is done by taking a memory dump, and sending this over the network. This is just a coarse view. Since the memory is shared by different agents a finer approach is required. Serialization is done by first running over all the agent's data structures: stack, dictionary, queue,... and tagging them. Then we loop over Borg's entire memory, copying every tagged cell to a table. This table is then sent over the network. Deserialization is performed in a similar way. The table is copied back to memory. Now problems arise, all data structures aren't copied to the same memory address as on the first computer, thus resulting in dangling pointers. Pointers in these data structures still point to their reference on the original computer. The new place could be almost anywhere in the new computers memory. So we need to loop over the structure once more, and update all pointers so they point to their corresponding location again. To find the new address the pointer should reference, another table is needed. When serializing, a table, containing all the addresses of all the individual cells, is created. Then this table is sorted. Since the memory is run over from left to right, the nth cell's address is contained in the nth index of this table. So if we need to find the new position of a pointer, the old address can be looked up in this table and we find the index of the cell. This can then be mapped on this cell's location in the new memory.

Using this technique of serializing data, we have no problem sending circular list or graphs with loops over the network. There are however a few subtleties. When we deserialize, a number of expressions need to be recreated. Borg strings for example need to be recreated, since they have to be defined in the stringpool. This is so for almost all types of expressions. There is however one special type of expression that needs extra attention. When sending dictionaries over the network we need to do some extra work.

Dictionaries are used to represent objects. When we send an object to a different computer, we don't want to copy the entire object (synchronization problems). We want to create a reference to this object., a RDC (remote dictionary). So when we need to serialize a dictionary it is transformed into a RDC. A reference to the original object.

The inverse does not always hold. When we send an RDC to another machine it should only sometimes be changed to a dictionary. The following pictures sums it all up. This what we get when we pass agent1 as a parameter around



When we send an RDC (reference to some object) to another machine (any machine not "holding" this object) it remains an RDC. It stays a reference to a remote object. Only when we send an RDC back to the agent, that the object is referring to, we need to change it into a dictionary. This makes sense because when we send a reference to a remote object to this remote object, it should be changed into a local reference. So in short: a dictionary is always transformed into an RDC. An RDC is only changed to a dictionary if it is sent to the agent that it is referring to.

Finally there's one optimization we haven't mentioned yet. When moving an agent the global dictionary is not serialized. The global dictionary only contains all natives known to Borg. They are the same in all Borg environments independent of the machine it is running on. So an optimization is performed by not sending the global dictionary over the network.

2.7 Disadvantages Of Borg

Borg is a good example of a mobile agent environment. From Borg we can learn how a mobile system can be implemented and what its weaknesses and advantages are.

Advantages are of course mobility. Borg offers a way to create and destroy agents and let them migrate all over the network. Borg is still a prototype but the first results are promising.

Borg still contains a number of disadvantages. The main drawback is Borg's communication model. The RMI (remote method invocation) works really well in theory, but we believe that the programmer has to make a lot of sacrifices when designing the application. RMI is based on the principle that an object is a nicely encapsulated structure, that only sends messages. Thus objects seem a good agent. But we don't think that this is true. In the airplane application, we need to send more then one class to the other side. Also RMI does not take into account that objects that send loads of messages to each other, should be grouped in one agent. Also inheritance might present problems, and especially class based OO is difficult to model in a mobile environment since we have to keep track of where our class definition is located.

RMI also causes several other drawbacks. The communication is totally asynchronous, and it is really difficult to perform a remote procedure call, that is, call a procedure and wait for the result to be returned. The only mechanism provided are callbacks, but these result in lots of unwanted little methods and instance variables.

In Borg we also have some minor problems with concurrency. Suppose we have a class that has two different methods. One method is very complicated while the other uses almost no CPU time and both methods are independent of each other. When we have an agent that calls the complicated function, and a bit a later another agent calls the simple function, it must wait for the complicated function to finish. This introduces very big unwanted delays. It would be nice if these two methods could be run concurently.

Borg still has a few drawbacks but they are all situated in the communication model. If we could extend Borg with a more "standard" (read send/recv) kind of communication model, many drawbacks would be solved.

2.8 Conclusion

In this chapter we have presented Borg. Borg is a mobile agent environment created by Werner Van Belle[17] in service of PROG. Borg is created by extending Pico (an imperative language similar to Scheme) with mobility and communication primitives. This chapter quickly sketched the syntax and semantics of Pico and Borg.

Special about Borg is that new agents can be created, cloned and terminated. These agents can sent messages to each other and they can even move to a different location on the network. Borg's communication model is based on RMI (remote method invocation). Every agent is an object and communication takes place by invoking methods on that agent. The agent we are sending messages to can be located on the local machine or on a remote host.

Borg also has the ability to move an agent at any time. (telescripting). We can at any point in the code even, in the middle of a while loop if we like, move the agent to another machine. The agent will continue its computation where it left of. Telescripting has the big advantage that the initiative to migrate can be taken by any agent, at any place in the code. We can even create a manager that directs which agents are to be placed at what location.

In this chapter a short overview of the implementation of Borg is given. Every agent consist of a message queue, containing messages waiting to be received. The message delivery system works asynchronous and just places the new messages in the queue. To find the way to an agent in a mobile system, Borg implements its own routing service. We create a hierarchy independent of any machine property. When delivering a message to a host, it is routed through this hierarchy. When agents migrate, all router nodes that it passes are updated, so that all paths to the moved agent are correct again. This is important because, we now have an independence between communication and migration. We can implement any type of communication on top of mobility.

Chapter 3

Problems Encountered Due To Mobility

Although mobility can be a very powerfull property, it also causes many problems. In this chapter some of these problems will be described. Note that not all these problems will always occur, some will be solved by correct implementations and others can be solved by avoiding the problem. (e.g. do not implement a garbage collector)

3.1 Multithreading And Concurrency

This property is not really mobility related, but we will have to deal with it anyway. In a system with different agents, that can reside on different machines, it is also possible that two agents reside on the same machine. Every host must therefore implement some kind of multithreading mechanism. Multithreading means that we can evaluate several agents at the same time.

Since these agents can interact we will soon run into concurrency problems. One agent can be waiting on the second, while the second agent waits for the first. The classical scenario of a deadlock.

We can navigate around this problem by using only asynchronous communication, like RMI. But deadlocks are a well-researched problem, especially in concurrent system design, and we can borrow techniques from those research groups to solve this problem.

3.2 Where / Who Is My Communication Partner

This is one of the main problems in mobile systems. Communication partners can hop all over the network, and at implementation level we still need to specify the physical machine (IP-address or equivalent) where the communication partner resides.

Keeping all agents updated all the time is an almost impossible job. It would result in a huge overhead of messages, and it would delay the system incredibly. So a more clever approach is needed. A centralized name server could do the job, but this is in practice not feasible. Soon the nameserver would become overloaded. And it would still not provide a solution for the delays.

So keeping track of all agents is not simple. We will always need to find some clever mechanism to solve this problem. Borg has for example solved this by introducing its own message routing system.

3.3 Creation Of Agents

This might seem easy at first sight. Just spawn a new one. But it is a bit more tedious than this. Not only do we need to specify what the new agent should do, we should also find a way to get all necessary information to it. We need to get function definitions, objects, variables and parameters to the new agent. Sometimes it might even be usefull to create some sort of shared scope that parent and child share. But then again, what happens when the child moves, what if the parent moves, and how about both agents updating the same shared variables? A big mess that certainly needs to be straightened out in a real world implementation.

Another problem concerning agent creation is that we need some dynamic way to spawn new agents. Static methods (like the CSP par command) will not do because they are not usefull in a real world application.

The agent creation problem is not such a big problem. It has been solved in many models. Borg for example uses the *agentclone* primitive. Other languages have solved this problem differently, and their solutions will be presented throughout this document (take a look at some of the models in chapter 5 and 7).

3.4 Security

This is a major problem, that is still unsolved to today . Agent security works in two ways: we need to protect the host from the program and vice versa.

When a program is downloaded from the internet we would appreciate some kind of prove that the application is safe, that is that it will not mess up our computer or steal private information, in short, virus detection. This has been partially solved by Java applets. Java applets restrict all kinds of hazardous functions. Although this is a good idea, some work still needs to be done. First of all there exist hazardous Java applets that crash a computer, while on the other hand, some restrictions are too strict making applets unusable.

On the other hand agents need to be protected from their interpreter. An interpreter can see all internal data an agent is carrying. Encoding this data is no solution, since if the agent wants to access the data itself, it needs to carry the key, which in turn is then visible to the interpreter.

Actually the situation is even worse. A malicious host could even brainwash our agent. Take again the airport visiting agent, mentioned in the introduction, that is visiting airports, looking for the cheapest flight to a certain destination. Suppose it visits a malicious airport office that desperately wants to sell its flight to us. This airport could reprogram the agent and corrupt all data gathered so far, making all other airports' flights more expensive. Up to today no general solution has been found for this problem.

This problem is rather difficult and requires lots of research and will therefore not be further addressed in this document.

3.5 Robustness

Especially when creating big agent systems that are to roam the internet, robustness is an important property. When one computer ceases to work, we don't want our entire application to crash. It should provide some sort of backup mechanism. For example, if we know that the computer is going down, we could move all agents to another machine.

Again no practical solution has been found for this problem. However since we are still far away from big agent systems that need a considerable uptime, this problem is in general, at least for the moment, less important.

3.6 GC

Another nuisance is garbage collection. Many systems these days, especially since Java conquered the world, use garbage collectors. Garbage collectors are really helpfull in a single machine application, but a distributed garbage collect is much more difficult, especially if we use all RAM in the network transparently. We have spent several months looking into this problem but no solution has come up so far. A standard stop and copy garbage collector does not work. Imaging that we suddenly shut down the internet to garbage collect.

A possibility is to garbage collect all local nodes, however this again created problems. We don't know wether there still are external references to our data. After about a month of brainstorming we have found a solution for this problem but it does not work with circular data structures. With this we mean a circular data structure smeared out over different machines.

Some tricks have already been published to facilitate garbage collection but no real solution has been found so far. One of the better systems works by rearaging memory to be on the same machine as the process, that access it mostly. This way trains are created, but they still don't provide a real solution to garbage collection.

Borg also offers garbage collection, but Borg does not provide real shared memory. We can only create references to other agents but we cannot address a cell in another computers memory, thus the entire problem is avoided.

3.7 Typesystem (Where Is My Class Definition)

This problem has already been mentioned before. When we use standard class based OO in a distributed system we might again run into trouble. Imaging we create a new agent that in turn instantiates an object of a certain class. To create this object, we need the class definition but this might already have been moved over the network. Things get even worse when we use class variables. Class variables not only pose the problem of finding them on the network, they also create concurrency problems. When we have two objects of the same class, updating the same class variable at the same time, we are certainly heading for trouble.

3.8 References

Because we don't know where our communication partner is, it is also difficult to create a reference to it. We cannot take any machine specific property as reference because the agent will just move anyway and then our reference isn't valid any more. On the other hand, the property must be unique for every agent so the agent can be identified.

This problem is closely related with who/where is my communication partner because eventually we must be able to transform these references to a physical address that specifies where our partner is located. Borg uses its own router to get the messages to their destination and it uses remote dictionaries as a reference (machine independent, but specific for every agent) .

3.9 Conclusion

This chapter contains an overview of some of the most common problems that are associated with mobile agents. Many of these problems are dependent of the communication model actually used, and sometimes they can be avoided. For some of these problems general solutions have been found, but most of the time they must be solved with a clever, implementation specific, trick.

Chapter 4

Communication Models Overview

4.1 Introduction

This chapter defines some properties specific for communication models. The terms synchronous and asynchronous communication mentioned before will be explained here. We will also explain what channels are and discuss the difference between client server and peer to peer communication . Apart from these, a few other communication specific properties will be introduced.

4.2 Synchronous / Asynchronous Communication

The difference between these two is that synchronous communication waits for the message to be delivered. As long as the message is not yet received on the other side, the sender will wait. Consider the following two programs:

```
A: send(B, 'value');
display('done');

B: recv(A);
```

When we execute these two little programs in an environment that supports asynchronous communication, then process A will show the message 'done' immediately after it is started . When however we use synchronous communication, the message will not appear until the receiver has actually received the message.

Maybe synchronous communication seems silly because this means that the program has to wait. In practice however, synchronous message calls are easier to understand, and a simple mechanism to synchronize two different programs.

4.3 rendez-vous

A rendez-vous is an occurrence that can only take place in the presence of synchronous communication. When two programs synchronize, they will, for a brief period of time, both know what they are doing. A rendez-vous is a construction, first defined in ADA, that lets the programmer execute code during this synchronized state. Thus when two programs synchronize, they run simultaneous during the time when the code is executed, and only then do both programs continue their own business.

A rendez-vous is executing code during the synchronized state of two programs.

4.4 Remote Function Calls

Just does what is says. When we have two agents or programs running on different computers then a remote function call is that one computer calls a function that resides on the other computer. A remote function call is synchronous in that it waits for the remote function to return its result. It is actually quite similar to a normal function call, with the only difference that the function might be located on a different machine.

Remote Method Invocation (RMI) is very similar. Only this time we do not put functions on the remote machine but entire objects. Then we can invoke methods on this object. There is however a big difference between RMI and RPC (remote procedure call). RMI does not wait for the result to be returned. A remote method invocation is therefore asynchronous.

In these little example we can already see some problems of synchronous communication. Suppose we have a mutual recursive function. This means that two functions P and Q reside on different machines. Using RPC, P calls Q and waits for the result. Next Q calls P, but P is waiting and cannot execute the request. Of course there are workarounds for this problem, but remember that with synchronous communication, deadlocks are never far away.

4.5 Channels and Links

When we go back to send/recv couples we see that somehow we must specify the machine we want to send the message to. Another technique has been developed that does not use a machine or process name, but a totally different name, a so called channel or link. Only when two processes send/recv a message over/from the same channel, will communication occur. The difference between channels and process names is that a single process can use different channels to communicate through or that different processes can share a single channel. So a channel can be regarded as a more flexible way of specifying your communication partner

A good example is a bankaccount. A bankaccount is only modelled by one process but it can use two channels, one for deposits and one for withdrawels.

4.6 Peer-To-Peer versus Client/Server

Peer-to-peer and client/server specifies the machine that initiates the communication. A client/server model is the simplest, and also the most used of the two. We have a server (e.g. a PC bankingservice), and we have many clients (all the people who use PC-banking). In the client/server model, the server waits for the client to make a connection and make a request that the server will execute. It is therefore the client that initiates the communication. In our banking systems it is the client that dials the number and requests to deposit a sum of money. The server only executes the request.

In a peer-to-peer both parties are equivalent and each of them may initiate the communication. An example of peer-to-peer communication could be a talk program. Each of the users can initiate to give his friend a ring.

4.7 Conclusion

There are many different properties a communication model can have. Therefore, before choosing a communication model for an application, we need to know what communication needs the applications impose. Only then a good model can be chosen.

An important characteristic of Borg, and of mobile agent systems in general is asynchronous communication. If we plan to let our system roam the internet, it is impossible to use synchronous communication because delays would slow down the system to much. In Borg we have opted not to choose for channel communication. The mobility constraint is to difficult to unify with channels. It is already hard enough to get our communication to the correct place on the network. If we're planning to use channels, we have to keep the channels updated all the time, or use a centralized system. Neither of these options is very appealing. Therefore agent based communication is desirable.

Mobile agents use a mix of client/server and peer-to-peer communication. When two agents meet they can both initiate the communication (peer-to-peer), but when an agents migrates close to a database, the communication with the database will be client/server based.

Chapter 5

Existing Models

In this chapter we will take a look at some of the models that have been proposed so far to solve the problem of agent creation and agent communication. The research into mobile multi-agent systems has only gained much interest in the last couple of years, but many techniques can be borrowed from the field of parallel programming.

The models used for concurrent programming form, although they're rather old, and not really finetuned for agents, are a good base to start from. The first model we have researched is CSP.

In this chapter we will give an overview of models developed so far. First some formal models will be discussed. Then some examples of communication in existing languages will be presented. Since agent creation and communication have a great deal in common with process creation and communication in concurrent languages, the first models to be presented will be borrowed from concurrent programming.

First we will present some of the currently used models like CSP, π-calculus and actor systems. These are all formal models and to show what is used in practice , we will present Java RMI. A very popular communication model. We will also describe another mobile agent language, Obliq, so that we don't have Borg as our only reference.

5.1 CSP

5.1.1 Introduction

CSP is a formalism invented by C.A.R. Hoare, at The Queen's University of Belfast. CSP is an acronym for Communication Sequential Processes. In this formalism Hoare suggests that input and output (communication with other processes) are to be incorporated as primitive functions into an imperative programming language. When we extend this programming language further with structures able to run several processes concurrently, and we add Dijckstra's guarded command, a solid basis is provided for concurrent programming. CSP itself has been primarily adapted for concurrent programming: speeding up programs, by spreading the computation over several processors. As we will show, this formalism still is up to date, and it even forms a good basis for agent communication.

Traditional programming languages all have several constructs in common. Almost every traditional programming language has a repetitive construct (while), an alternative construct (if) and a normal sequential program composition. CSP is an extension to this traditional languages. In fact CSP only contains about 7 primitives. 3 of them are described above: repetition, alternative command, and sequential command. CSP further extends this set with primitives for input and output (communication and synchronization) and a parallel command.

5.1.2 Semantics

As we have just explained, CSP is a simple formal language based on very few language constructs. Here we will present an overview of CSP semantics.

1. CSP contains some kind of guarded commands (based on Dijkstra's guarded commands). A guarded command is an instruction consisting of 2 parts. The first part is a guard (a predicate indicating when the second part may be evaluated. The second part is just an instruction. It are these guarded commands that introduce and control nondeterminism[1] .

2. Next CSP introduces a parallel command. A parallel command runs its constituent sequential commands simultaneous. It is this command that introduces parallelism in CSP. Only with this operator new processes can be spawned. The par-command will finish only when all its constituent sequential components have finished.

3. Simple forms of input and output are also introduced. They are the mechanism used for communication between processes.

4. Communication can only occur when an input and an output process are both ready. In order to successfully communicate, both processes must indicate each other as a receiver. Thus communication will only occur when one process evaluates an output, and the other process an input command (with each other as recipients). CSP provides no buffering! This means that both processes must synchronize in order to communicate. All communication in CSP is therefore synchronous.

5. Input commands may appear in guards. When an input command appears in a guard, this guard will only fire when the corresponding

process is ready to output. If several guards become true, a random guard will be chosen (nondeterminism). The others have not effect.

6. A repetitive command may have input guards. If all sources named by this guard have terminated, then the repetitive command also terminates.

7. Finally CSP contains a simple pattern-matching feature. This pattern-matcher is used to select for example the correct input guard. With these pattern-matcher we can for example easily tag messages .

8. Finally CSP contains, like most other languages, the concept of assignment. This is just the same as in any other language, we assign the value of an expression to a variable.

These semantics might appear quite abstract, but actually they're rather intuitive. Once CSP's syntax has been explained, some examples of CSP-code will be presented. They are very simple and don't need further explanantion

5.1.3 Syntax

```
<command> ::= <simple command> | <structured command>
<simple command> ::= <null command> | <assignment command>
     | <input command> | <output command>
<structured command> ::= <alternative command>
     | <repetitive command> | <parallel command>
<null command> ::= skip
<command list> ::= {<declaration>; | <command>;} <command>
```

So far this syntax only introduces all the language constructs: input, output, assignment, alternative command, repetitive command, parallel command and sequential commands. 7 Basic primitives form the basis for this language. The null command does nothing. The command list denotes simple sequential commands. All commands in the list are to be evaluated from left to right. The declaration part is only used to introduce variables.

Parallel Command

```
<parallel command> ::= [<process> {|| <process>}]
<process> ::= <process label> <command list>
```

[1]Nondeterminism occurs rather frequently in concurrent systems. Sice we don't know anything about the relative speed of these processors, we don't know anymore when which instruction will be evaluated.

A parallel command takes the form of a number of processes separated by '||'. These processes will be executed simultaneously. We have omitted the syntax of the process label, since we find its syntax notation rather complicated. A couple of examples should make it clear. A process label, in its simplest form is empty. In its next simplest form it is an identifier. If we want to we can even give indexes to a label, making it easy to spawn for example 100 identical processes. This has the following syntax: "label(i:0..4)::"

```
[cardreader?cardimage || lineprinter!lineimage][2]
[west::DISASSEMBLE || X::SQUASH || east::ASSEMBLE]
[room::ROOM || fork(i:0..4)::FORK || phil(i:0..4)::PHIL]
```

This last syntax defines in total 11 agents. 5 forks (fork(0) till fork(4)), 5 phil's and 1 room. The variable i is bound in each of the processes fork and phil, so each process knows which fork/phil it is.

Assignment Command Again we will skip the formal syntax description. If you really want to have a look at it, take a look at [2]. It is in the assignment command in CSP that the pattern matcher is hidden. This makes assignment a rather powerfull operator. After you have seen a couple of examples, it should be clear how an assignment in CSP works.

x:=x+1	Afterwards x has his value incremented by 1.
(x,y):=(y,x)	exchanges the values of x and y.
x:=cons(left,right)	constructs a structured value and assigns it to x.
cons(left,right):=x	Fails if x does not have the form cons(y,z). If it fails left is assigned to y, and right to z.
insert(n):=insert(2*x+1)	equivalent to n:=2*x+1
c:=P()	c is assigned to a structured value P().
p():=c	Fails unless c is bound to P(), then it has no effect.
insert(n):=has(n)	always fails.

Input And Output Commands

```
<input command> ::= <source>?<target variable>
<output command> ::= <destination>!<expression>
<source> ::= <process name>
<destination> ::= <process name>
<process name> ::= <identifier>|<identifier>(<subscripts>)
```

[2] "?" means receive, "!" send.

Input and output commands specify communication between two concurrently operating sequential processes. Communication occurs between 2 processes of a parallel command whenever:

1. an input command in one process specifies as its source the name of the other process;

2. an output command in the other process specifies as its destination the process name of the first process; and

3. the target variable of the input command matches the value denoted by the expression in the output command. On these conditions the input and output commands are said to correspond. Commands which correspond are executed simultaneously, and their combined effect is to assign the value of the expression of the output command to the target variable of the input command.

An input command fails if its source is terminated. An output command fails if its destination is terminated or if its expression is undefined. Both processes always synchronize at the point of communication. This usually means that one process has to wait for the other. One should be carefull when using in- and output commands. Because they wait for their corresponding communication partner deadlocks may occur. Also note that process names may have subscripts. This makes it easy to define and manipulate arrays of processes.

process1	process2	
process2?value		<process1 waits for input>
	...	
	process1!5	<process2 sends message>
...	...	Synchronization. Both processes continue

As you see, process1 waits for process2. When process2 sends its value, both processes will synchronize. At this point, we know for sure, that both processes have reached a corresponding input/output-pair. After synchronization both processes continue their own evaluation. Since we know absolutely nothing about the relative speeds of both processes, we haven't got a clue of what's happening, until of course another synchronization occurs.

cardreader?cardimage	receive something from a process called cardreader and store it in the variable cardimage.
lineprinter!lineimage	Send the value of the variable lineimage to the lineprinter process.
X?(x,y)	receive 2 values from process X
DIV!(3*a+b,13)	send the 2 values to a process called DIV.
console(i)?c	From the i'th element of an array of consoles, input a value and assign it to c
console(j-1)!"A"	to the (j-1)th console, output character "A"
X(i)?V()	from the ith element of an array of process X, input a signal v(); refuse to input any other signal.
sem!P()	to sem, output a signal P().

Alternative And Repetitive Commands

```
<repetitive command> ::= *<alternative command>
<alternative command> ::= [<guarded command>
     {□  <guarded command>}]
<guarded command> ::= <guard> → <command list>
     | (range)<guard> → <command list>
<guard> ::= <guard list> | <guard list>;<input command>
     | <input command>
<guard list> ::= <guard element>{;<guard element>}
<guard element> ::= <boolean expression> | <declaration>
```

As we can see from the syntax definition, a repetitive command is just an alternative command repeatedly executed. So actually the guard is the loop's stop condition. An alternative command is just a series of guarded commands, separated by □. And a guarded command finally is simply a guard followed by an → and an expression. Only if the guard becomes true, the expression is evaluated. When in the alternative command several guards become true, one true guarded command is chosen at random (source of nondeterminism in CSP) Just as with the parallel command "||", a guard can contain range subscripts. For example (i:1..10)P→CL. This predicate will be evaluated for all i's ranging from 1 to 10. Actually this command is equivalent with: $P_1 \rightarrow CL_1 \square P_2 \rightarrow CL_2 \square ... \square P_{10} \rightarrow CL_{10}$. Note further that if an input command appears in the guard, it should be the last command.

examples:

- [x≥y → m:=x □ y≥x → m:=y]
  if x≥y assign x to m; if y ≥x assign y to m; if x=y either assignment can be executed.

- i:=0;*[i<size; content(i)≠n→i:=i+1]
  The repetitive command scans the element's content(i), for i=0,1,..., until either i≥size, or a value equal to n is found.

- *[c:character; west?c→east!c]
  This reads all characters output by west, and outputs them one by one to east. The repetition terminates when process west terminates.

- *[(i:1..10)continue(i);console(i)?c →X!(i,c); console(i)!ack(); continue(i):=(c≠sign off)]
  This command inputs repeatedly from any of ten consoles, provided that the corresponding element of the boolean array continue is true. The bound variable i identifies the originating console. Its value, together with the character just input, is output to X, and an acknowledgment is sent back to the original console. If the character indicates 'sign off', continue(i) is set to false, to prevent further input from that console. The repetitive command terminates when all ten elements of continue are false. (An implementation should ensure that no console that is ready to provide input, will be ignored unreasonably often).

Examples Even though CSP is a high-level formal model to describe concurrent algorithms, it still retains its simplicity. When you just see a CSP-program, it should be clear what is does, without knowing any lowlevel details of CSP. All you have to know is the meaning of the main 6 primitive functions:

?	input
!	output
||	parallel
→	guard
□	alternative command
*	repetitive command

Factorial recursive

```
[fac(i:1..limit)::
  *[n:integer; fac(i-1)?n →
     [n=0 → fac(i-1)!1 □
      n>0 → fac(i+1)!n-1;
              r:integer; fac(i+1)?r; fac(i-1)!(n*r)
     ]]
|| fac(0)::USER
]
```

This simple program demonstrates how an iterative array of processes can be created. This is just a simple recursive factorial, where each recursive step is performed by a different process. Actually this is a silly example in terms of gaining performance, but it shows how CSP can elegantly be used.

Dining philosophers

The Dining philosopher problem is about 5 philosophers who do nothing else but eating and thinking all day long. In the dining room is one circular table with 5 forks and 5 plates. When a philosopher gets hungry he goes to the dining room, picks up his left fork, then his right fork and starts eating. Then he returns to the library to think. Because there are only 5 forks, all forks are shared by 2 philosophers.

```
PHIL=*[... during ith lifetime... →
       THINK;
       room!enter();
       fork(i)!pickup(); fork((i+1) mod 5)!pickup();
       EAT;
       fork(i)!putdown(); fork((i+1) mod 5)!putdown();
       room!exit()
      ]

FORK=*[phil(i)?pickup()→phil(i)?putdown() □
       phil((i-1)mod5)?pickup()→phil((i-1)mod5)?putdown()
      ]

ROOM=occupancy:integer; occupancy:=0;
     *[(i:0..4)phil(i)?enter()→occupancy:=occupancy+1 □
       (i:0..4)phil(i)?exit() →occupancy:=occupancy-1
      ]

[room::ROOM || fork(i:0..4)::FORK || phil(i:0..4)::PHIL]
```

This program should again be straightforward to read and understand. There is however still one problem with this program. When all philosophers enter the room at the same time, and pick up their left fork at the same time, no philosopher will be able to get its right fork (since another philosopher is holding it). This can be solved by limiting the number of philosophers in the dining room to maximum 4.

The sieve of Eratosthenes

The sieve of Eratosthenes is an algorithm for computing prime numbers. The parallel algorithm works as follows. Use an array of processes, SIEVE, in which each process inputs a prime from its predecessor and prints it. The process then inputs an ascending stream of numbers from its predecessor and passes them on to its successor, suppressing any that are multiples of the original prime.

```
[SIEVE(i:1..100)::
    p:integer;
    SIEVE(i-1)?p;
    print!p;
    *[m:integer, SIEVE(i-1)?m →
        [(m mod p)=0 → skip □
         (m mod p)!=0 → SIEVE(i+1)!m]]
|| SIEVE(0)::print(2); n:integer; n:=3;              // start
    *[n<10000 → SIEVE(1)!n; n:=n+2]
|| SIEVE(101)::*[n:integer; SIEVE(100)?n → print!n] // end
|| print::*[(i:0..101) n:integer; SIEVE(i)?n→...]   // print
]
```

5.1.4 Conclusion

Even though CSP was developed quite a while ago, and was primarily designed for programming concurrent programs, it still is a good candidate for a communication model for mobile multi-agent systems. We especially like CSP's simplicity. It is based on high level language constructs, which in turn means easy and understandable programs. Furthermore CSP's design resembles that of a lot of standard imperative languages, like C, Pascal,...

There are however a few drawbacks to CSP. Especially if we want to use it as the communication standard for a mobile multi-agent system. First of all, process creation is static. We create new processes using the par-command because it is statically encoded. We cannot create a couple of new processes at runtime. We think that this is a major drawback. Especially in mobile agent systems, where we would like a more dynamic method for creating and destroying agents.

The fact that CSP uses synchronous message passing is another drawback. Maybe in the future, we will have mobile agents roaming the internet. It then is undesirable that 2 agents (for example one in Europe, the other in Australia) use synchronous communication to talk to each other. When delays on the internet easily reach several seconds, then two agents, definitely don't want to wait on synchronization.

We finally have a last problem with CSP. We do not like the alt-command. Even though the alt-command is very powerfull, and provides a graceful method to communicate with several agents at the same time, we don't like its syntax. When you take a closer look at the syntax of the alt-command, in CSP, one will see all kinds of restrictions. No output commands can be used in an alt for example. In his book Hoare says that to evaluate an alt-command, it is a good way to just try all guards, and take one that becomes true. Actually this is the only way to implement this alt-command. Now this results in all kinds of unwanted effects. First of all, suppose that we have a destructive guard somewhere. A guard that assigns a variable some value, or a guard that performs an output-command. We will never know wether this guard is actually executed. If another guard happens to become true, before the destructive guard is tested, the destructive command is not executed. Another time, it might be executed. Depending on the implementation, half the guard might be executed. We find this an undesirable side effect of the alt. The alt actually poses too many restrictions, and requires too much indepth knowledge of the programmer. Thus it is not as intuitive as it appears to be on first sight.

Even despite these drawbacks, we still think CSP forms a good base for agent communication. Extend CSP with a dynamic process-creating command, and maybe provide means for asynchronous communication and we have a good basis for an agent system. As will be explained in a later chapter, we have done some experiments by adding CSP to Borg, and this actually turned out to be quite a usefull communication protocol.

5.2 π-calculus

π-calculus is a formal model in which we can easily express communication between dynamical processes. One of the special properties of this calculus is the ability to create random agent-systems, and dynamically change this system. The π-calculus is also one of the few models that can describe mobility (albeit in a rather obscure form, although this can be solved by introducing some syntactic sugar).

π-calculus has been based by extending the CCS process algebra with mobility and further simplifying this by making no distinction between variables and constants. In order to avoid this distinction, π-calculus introduces names. (compare to call by reference). When communicating between different processes, names are passed along the channel, not values. Further π-calculus introduces a special construct to create name-scopes. That is we now have the ability to define locale or private names. In this short overview of π-calculus we will finally give some examples and explain several notions of equivalence.

5.2.1 Syntax

In π-calculus, a process is defined using the following syntax:

$$
\begin{array}{lll}
P & ::= & \mathbf{0} \\
 & | & P_1+P_2 \\
 & | & \overline{y}x.P \\
 & | & y(x).P \\
 & | & \tau.P \\
 & | & P_1|P_2 \\
 & | & (x)P \\
 & | & [x=y]P \\
 & | & A(y_1, ..., y_n)
\end{array}
$$

5.2.2 Semantics

0 The empty process

P_1+P_2 P will behave like P_1 or P_2.

$\overline{y}$x.P output
Send the name x to channel y, and behave further like P. Since π-calculus presumes synchronous communication, P will only be executed when another process receives the name x over channel y.

y(x).P input.
The receive instruction will receive a value on channel y. The process will then behave like P$\{z/x\}$. (Substitution of z by x)

τ.P silent prefix.
τ is the silent prefix. It performs the silent action τ and then behaves further like P. You can think of τ as the empty communication, analogous to the empty word ε in other formal languages.

$P_1|P_2$ Composition
This construct executes the 2 process P_1 and P_2 in parallel. Compare with the PAR-command in CSP.

(x)P restriction (scope operator)
This agent's behavior is identical to that of P. The only restriction is that communication through the channels x and $\overline{x}$ is prohibited. If P consists of different components, it is allowed that these components use the channel x to communicate with each other, because this transaction is reduced to a τ-action of the entire process P. This operator can be best compared with a scope operator, that makes x a private variable of the process P.

[x=y]P match-operator

This agent behaves like P if and only if x and y are equal; otherwise it will behave like **0**. [x=y] corresponds to a predicate that must be true, before P can be executed.

A($\mathbf{y}_1$, ..., $\mathbf{y}_n$) Agent definition.

This construct allows to define agents. For every agent identifier there should be an equation of the following form: A(x_1, ..., x_n)=P, in which x_1, ..., x_n are the only free variables in P. A(y_1, ..., y_n) will behave like P{x_1/y_1, ..., x_n/y_n}. This form of definition allows the recursive definition of agents, because P may contain any agent identifier, including the agent A itself.

further definitions A prefix is a general name, used to reference all of the following structures: $\overline{y}$x.P, y(x).P, τ.P. We call $\overline{y}$x.P a negative prefix, y(x).P a positive prefix and τ.P the silent prefix.

Since π-calculus is a formal model, some more standard definitions will also occur here. First of all fn(P) is defined as the set of free variables in P. More specific these are the names that occur in P without occurring in a positive prefix or a restriction.

α-equality is defined identical to the λ-calculus. For those of you who have never heard this term, α-equality is only about equality of the syntax. 2 Programs are α-equal when these programs are the same on syntactic level, literally the same after substituting variables. For example (y)($\overline{y}$x.0 | y(x).0) is α-equal to (z)($\overline{z}$x.0 | z(x).0), because they are literally equal after substituting y by z.

P{y_1/x_1, ..., y_n/x_n} stands for simultaneous substitution of all free occurring names x_i in P by y_i. It could be that other substitutions have to take place in order not to bind any free names in P.

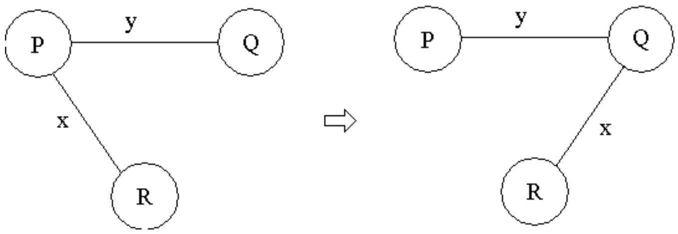
Order of operations:

$$\left.\begin{array}{l}\text{restriction}\\ \text{prefix}\\ \text{match}\end{array}\right\} > \text{composition} \;> \text{sum}$$

Of course parentheses may be introduced anywhere in order to change to order of evaluation. Normal evaluation occurs from left to right.

5.2.3 Examples

Link Passing



$$\overline{y}x.P' \,|\, y(z)Q' \,|\, R \rightarrow P' \,|\, Q'\{x/z\} \,|\, R$$

In π-calculus, all communication is done through channels. When communicating, we do not specify the process we want to correspond with. We specify the channel that the message has to use to get to its receiver. With the above program we can pass links between two processes. Note that process R knows nothing about all that has happened between P and Q. R can still use channel x to communicate even though the receiver has changed.

Molecular Actions

Suppose we have the following agents:

$$\begin{aligned} P &\equiv \overline{x}u.\overline{x}v.P' \\ Q &\equiv x(y).x(z).Q' \\ R &\equiv x(y).x(z).R' \end{aligned}$$

When we evaluate the program $P \,|\, Q \,|\, R$, we would expect that the values u and v from process P are sent to process Q or to process R. There might however arise a difficulty. When we send u down channel x, it might be received by Q. Now when we send v, this can be received by R. With this simple program, communication can be done in 4 different ways.

Q	R
u,v	
u	v
v	u
	u,v

This is not what we want since we want to communicate a pair of values: (u,v). So only modes 1 and 4 in the table are correct. In π-calculus there is no way to construct structured datatypes, so another solution has to be found.

$$\begin{aligned}P &\equiv (w)(\overline{x}w.P' \mid \overline{w}u.\overline{w}v.0)\\ Q &\equiv x(w).w(y).w(z).Q'\\ R &\equiv x(w).w(x).w(z).R'\end{aligned}$$

These agents solve the problem. We first introduce a new private channel w. This channel is unique, and will only be used to communicate the pair (u,v). Now the channel w is sent down channel x. This channel will be received by process Q or process R. When we communicate u and v over channel w, we can be sure that both values will be received by only 1 process, since only one process knows the channel w.

Process Passing

$Exec(x) \equiv x(y).\overline{y}\varepsilon$
$(x)((z)(\overline{x}z \mid z(\varepsilon).P) \mid Exec(x))$

The exec function can be thought of as a function that receives a link and activates this link. The other process is a bit more difficult to understand. What we actually do is prefix the process P. We just add a few communication cycles before executing P. These communication cycles are nicely resolved by the exec statement, so that the result of the entire program is just the execution of P. We can look at the program as the combination of two processes : P_1 that is only P with some prefixes, and another process P_2 that at a certain point in time executes P. What actually happens is process passing. We make it possible that P_2 receives P as a parameter and executes it. We pass a link (x) to process P_2 that in turn makes sure that P is executed. In fact we can show that the above program using link passing is equivalent to process passing.

By adapting π-calculus' syntax a bit, the above program is equivalent with $(x)(\overline{x}P.0 \mid x(p).p)$, or passing process P as a parameter. It turns out that link passing provides the same power as process passing in the presence of recursion.

5.2.4 λ-calculus

λ-calculus is a formal model on which today's functional languages are based. It has been developed around 1930, and many languages today (like LISP, Scheme,...) are still based on its principles. It is therefore reassuring to know that we can write a simulation for λ-calculus in π-calculus.

To refresh your memory, here's basic λ-calculus syntax:

$$M ::= x \mid (\lambda x M) \mid (M\,N)$$

X in this syntax defines a simple variable. $\lambda x M$ defines a function which takes one argument x and has M as body. $(M\,N)$ finally denotes function application. Here a function M is applied to its argument N.

In order to convert these basic expression to π-calculus we define a lazy reduction relation $\rightarrow$ such that if $(\lambda x M)N \rightarrow M\{N/x\}$, and if M→M' then MN →M'N. This reduction strategy is deterministic. We now introduce the following transformation:

[[λxM]]u=u(x)(v).[[M]]v u(x)(v) denotes passing 2 arguments as a molecular action

[[x]]u=$\overline{x}$u

[[MN]]u=(v)([[M]]v | (x)$\overline{v}$xu.x(w).[[N]]w) (x not free in N)

Using this transformation we can transform linear λ-calculus expressions to π-calculus. However we still run into trouble when we try to transform λ-expressions including the **Y**-recursion operator. In order to resolve this problem we introduce replication:

$\alpha * P = \alpha.(P \,|\, \alpha * P)$

When we change the last transformation rule to:

[[MN]]u = (v)([[M]]v | (x)$\overline{v}$xu.x(w)*[[N]]w) (x still not free in N),

then N can be called multiple times from M, since everytime a new replica of N is generated. Using this transformation we can simulate all λ-calculus expression in π-calculus.

5.2.5 Algebraic Theory

π-calculus defines several different equivalence relations. We can make a clear distinction between strong bisimilarity($\dot{\sim}$) and strong equivalence($\sim$). During the explanation of these equivalences we will also encounter SGE or strong ground equivalence.

Strong bisimilarity ($\dot{\sim}$) is the simplest form of equivalence. Its definition is based on some 16 simple axioms, each describing a basic transformation. Using these axioms we can evaluate a π-calculus program. These 16 axioms are sometimes referred to as SGE or strong ground equivalence. In fact when 2 programs P and Q are equal up to a certain number of transformations we write: SGE⊢P=Q. The terms “strong bisimilarity” and “SGE” can be used as synonyms. There are however a few drawbacks to strong bisimilarity. The equivalence does not hold under substition of free variables of P. It also turns out that strong bisimilarity is complete over finite agents, but that this property does not hold for infinite agents.

Therefore a new equivalence relation is introduced, called strong equivalence. Actually strong equivalence is just the same as strong ground equivalence, but a few new axioms have been added, in order to preserve equivalence under all substitutions. Strong equivalence is based on about 20 axioms. 4 axioms describing congruence have been added so that the transformation is substitution-invariant.

Finally a last form of equivalence is defined, called D-equivalence. The D stands for distinction, and a distinction is a symmetric irreflexive relation between names. A substitution σ respects a distinction D if $\forall(x, y) \in$

$D, x\sigma \neq y\sigma$. Two agents P and Q are said to be strongly D-equivalent, written P$\sim_D$Q, if P$\sigma\dot{\sim}$Qσ, for all substitutions σ respecting D. We can now simply define
$\dot{\sim}=\sim_{\mathcal{N}}$ and $\sim=\sim_{\emptyset}$. A theoretical beautiful property with alas little practical use.

5.2.6 Polyadic π-calculus

Although π-calculus is a really nice model, it is quite difficult to even write a simple program in it. We have demonstrated that ordinary λ-calculus can be simulated but doing this makes the programs extremely complicated to understand. So since its first publication, a few adjustments have been made to the π-calculus, making it a bit easier to write high level programs. Since then, the original calculus is referred to as monadic π-calculus. The new version is then known as polyadic π-calculus. The adjustments made to the calculus are only syntactic sugar, to replace some common high-level constructs.

A first step towards the simplification of π-calculus includes the ability to communicate multiple values over a single channel at the same time. In the previous section we have demonstrated how this can be achieved using molecular actions. The syntactic sugar introduced for this purpose is just an extension of the already known prefixes. To communicate a number of values $y_1, y_2, ..., y_n$ we use the following syntax:

polyadic notation	syntactic sugar for	comment
$x(y_1, y_2, ..., y_n)$	x(w).w(y$_1$).. . . .w(y$_n$)	Receive multiple values from a single channel.
$\overline{x}$y$_1$. . . y$_n$	(w)$\overline{x}$w.$\overline{w}$y$_1$.. . . .$\overline{w}$y$_n$	Send multiple values over a single channel

Next we introduce an operator for replication. This one comes in quite handy when writing recursive functions. We shall define !P≡P | P | P | P | P | P | !P. Thus !P spawns as many copies of P as we need, and then generates a new replicator. With this construct we can create an infinite number of copies of P.

Also polyadic π-calculus further extends the monadic model in that it introduces numerals, and some arithmetic operators. Again this is nothing special, as all numerals can be represented in traditional π-calculus, using an encoding similar to Church numbers in λ-calculus. A number n will be represented by taking n times the successor of zero. To create a number we therefore need two arguments: a successor and zero. We can then define a number

$$\underline{n}(sz) = \overline{x}.\overline{x}....\overline{x}.\overline{z}$$

Using this definition, we can construct agents that are able to add two numbers. This can be achieved by the following code:

$$(s_1 z_1 s_2 z_2)(\underline{n_1}(s_1 z_1) \mid \underline{n_2}(s_2 z_2) \mid Add(s_1 z_1, s_2 z_2, yw)) \approx \underline{n_1 + n_2}(yw)$$

Finally polyadic π-calculus introduces a concept called abstractions and concretions. Abstractions are another way of writing an input-command. Instead of writing $K(x_1,...,x_n)=P$, we can also write $K=(\lambda x_1...x_n)P$. This has its advantages when we want to chain a couple of processes.

Suppose we have a process P that receives input on a link x and sends it output on a link y. Now we want to link several of these processes in a chain. Using the λ abstraction this can be quite elegantly solved, without using complicated substitutions. A concretion is the output-equivalent of an abstraction and we define it as $\overline{x}y_1...y_n.P=\overline{x}[Y_1...y_n]P$. A result of the introduction of abstractions and concretions is that we can represent the truth-values true and false, and that we can write an if-then-else statement, which on itself isn't nothing new since we could already simulate λ-calculus in which these operations have been defined.

These 5 constructs are the only extensions made to the monadic π-calculus. They have only been defined in order to make programming a little bit easier and they don't really add new computing power to the calculus.

5.2.7 Conclusion

π-calculus is a formalism based on a handfull of simple primitives: send, receive, scopes, parallelism, and a few others. π-calculus is a simple calculus, easy to understand. When we see a π-calculus program, it is easy to figure out what it actually does. It is very uniform (in contrast with CSP) and all of its base commands are intuitive and easy to understand .

But as always there are a couple of problems with this calculus. First of all it is too low level. Imaging yourself writing a graph-traversal algorithm in this language. We can use the λ-calculus as a base to start from but the program would become too complicated to work with.

They have tried to solve this problem with the introduction of polyadic π-calculus. It helps a little, but even the polyadic variant is too low level to use in an actual programming language.

At the end of chapter 3 we have specified that we would like asynchronous communication for a mobile agent system. Again, this cannot be provided by π-calculus since it uses synchronous communication. Actually it can be provided by making the message queue explicit, but for any practical implementation this is too much overhead.

Furthermore they claim to be the first calculus that models mobility because they can simulate to pass a process as a parameter. This is however not the kind of mobility we are looking for. Moving agents over the internet is hardly the same as passing some instructions as a parameter to some function. Actually the agent is not even moved, but a reference is created.

A final comment is that π-calculus uses links (or channels) to communicate. As we explained, channels can be rather difficult to implement in a mobile agent language.

We hope you do not get us wrong, we think that π-calculus is a good model but it needs some big enhancements before it can actually be used in practice..

5.3 Actor-Systems

After having discussed CSP, and π-calculus, there is still one widely used formal model about which we have nothing said. This model is called actor computation, and the theory we will present has been developed by Gul Agha[3]. Actors are self-contained, concurrently interacting entities of a computer system. They communicate via message passing which is asynchronous and fair. Actors can be dynamically created and the topology of actor systems can change dynamically. The actor model is a primitive model of computation (based on λ-calculus), but nonetheless easily expresses a wide range of computational paradigms. It directly supports encapsulation and sharing, and provides a natural extension of both functional programming and object style data abstraction to concurrent open systems.

The semantics defined combine the message passing features of the primitive actor model with an applicative functional language for describing individual actor behaviors. This provides a simple yet expressive language with manageable formal semantics.

5.3.1 The Language

Actor systems are based on a simple functional language.Now when constructing a formal model, what language can better be used than simple λ-calculus. Gul Agha's actor system extends λ-calculus with 3 new primitives.

The first primitive that has been added to the language is the *send*-construct. send(a,v) creates a new message with receiver a and contents v and puts this message into the message delivery system. One major difference with the calculi seen so far is that the send function uses asynchronous communication. After performing the send instruction, the current computation carries on, not waiting for the message to be received at the other side.

What makes actor systems also very different from any other calculus for distributed or concurrent systems is that it provides no explicit receive function. This is solved by the basic λ-calculus *function application*. Consider the following example in freestyle λ-calculus:

$$\text{fac} = \lambda \text{n. if(n}<\text{2,1, n*fac(n-1))}$$

This function can then be called from within another actor using the send function:

send(fac, 5)

The result of this evaluation will be that the faculty of 5 is computed. The result of this computation is just trashed, since the send primitive is asynchronous.

Actor systems therefore have only a send-construct. The receive native has been replaced by a simple function application.

Like all other models, we need, besides communication primitives, also a way to create new actors. This is done by introducing another primitive into the language. *letactor{x:=b}e* creates a new actor b and assigns it to the variable x. This variable x can then be used in the expression e.

letactor{facactor:=fac}send(facactor, 5)

This construct can be compared to the Scheme letrec-function as it allows recursion and can be used to create 2 actors locally referencing each other. When we take a closer look at an actor-system, we will discover that the letactor is only syntactic sugar for two other primitives called newaddr and initbeh. This has only been done to keep the lambda as the only binding expression. This has however no implications in this simple introduction.

Actor systems finally have one more primitive function called *become.* The become primitive is a bit more tricky than the other two primitives seen so far. The become primitive is for changing behavior. *become(b)* creates a new anonymous actor that carries out the rest of the current computation, and alters the behavior of the current actor so it will execute b. An anonymous actor can spawn new actors and can send messages to other existing actors. It will never receive any message from another actor because its addres cannot possibly be known. This new primitive might seem a little strange at first but its presence is necessary in the language. It is used for history-sensitive behavior necessary to model asynchronous access to shared resources in an actor-system. Maybe an example will clear things up. The following little program is really simple. It first awaits a message, containing the address of an another actor. It will then send 5 to this actor.

$$\mathrm{b}_5 = \mathbf{rec}(\lambda \mathrm{b}.\lambda \mathrm{a}.\mathbf{seq}(\mathbf{send}(\mathrm{a},5),\ \mathbf{become}(\mathrm{b}))$$

Rec in this example means the λ-calculus fixpoint combinator, sometimes referred to as **Y**. It will just execute the lambda that follows repeatedly. During execution we can regard the b as the recursive function call. **Seq** is syntactic sugar for expressing sequential composition. Thus this program simply receives an actor address, and sends 5 to this address. After doing this, this program becomes itself again (it becomes ready to receive another actor address). Using this technique we can easily write a sink that trashes all its input and becomes itself again (kind of a UNIX /dev/null)

$$\text{sink} = \text{rec}(\lambda \text{b}.\lambda \text{m}.\text{become}(\text{b}))$$

5.3.2 Examples

Actor Cells

In this language it is easy to represent objects with local state. As an example of this an actor with an accessor and mutator function will be shown. The actor responds to two kinds of messages: get? and set!. A get? message takes the address of the actor to whom the answer should be send as a parameter. The set! message only takes the new value as a parameter. Actually the program takes only one message. A pair containing get? or set! together with their parameter. With the mkget and the mkset function, these pairs can be created.

```
B_cell = rec(λb.λc.λm.
          if(get?(m),
             seq(become(b(c)), send(cust(m),c))
             if(set?(m),
               become(b(contents(m))),
               become(b(c)))))
```

To execute this program we could write something like:

```
letactor{a:=B_cell}e  where
e = seq(send(a,mkset(3)), send(a,mkset(4)), send(a,mkget(a)))
```

Evaluating these lines results in the actor a receiving a message 0, 3 or 4, depending on the arrival order of the messages. Since we use asynchronous communication, no message arrival order can be guaranteed. This kind of object with internal state could have never been constructed without the use of the become primitive.

Multiplying All Leaves In A Tree

Another example of an actor system is a little program that calculates the values of all the leaves in a tree.

```
treeprod = rec(λf.λtree.
             if(isnat(tree),
                tree,
                f(left(tree))*f(right(tree))))
```

Simple but not very efficient. If we're using an actor system anyway, we might as well make use of the concurrency it provides. In order to do this we need a join continuation, which synchronizes the evaluation of the different arguments.

```
B_treeprod =
  rec(λb.λself.λm.
    seq(become(b(self)),
      if(isnat(tree(m)),
        send(cust(m), tree(m)),
        letactor{newcust:=B_joincont(cust(m),0,nil)}
          seq(send(self, pr(left(tree(m)), newcust)),
              send(self, pr(right(tree(m)), newcust))))))
```

```
B_joincont = rec(λb.λcust.λnargs.λfirstnum.λnum
              if(eq(nargs,0),
                become(b(cust,1,num)),
                seq(become(sink),
                  send(cust, firstnum*num))))
```

Maybe a little explanation is required. The second program is a simple multiplicator. It will receive two arguments, multiply them and send them to his customer. The program works as follows: for every node in the tree a multiplicator is created. When the multiplicator has received his left and his right argument, he multiplies them and sends them to his customer, his parent node.

The first program just traverses the entire tree, spawning multiplicators. Its syntax is however a bit tricky. First of all note that m is a pair containing (tree, cust). The next line: become(b(self)) spawns another actor with the same behavior. Everytime we "use" a treeprod-instance, a new one is generated. This way we have an unlimited supply of $B_{treeprod}$. This can be compared with $!B_{treeprod}$ in polyadic π-calculus. When we are traversing the tree, and we encounter a leaf, its value is send to the multiplicator. When we arrive at a node, we create a new multiplicator which will be the customer of the children of this node. The trick this program uses to work parallel consists of creating a second tree of multiplicators that is responsible for doing the entire computation, albeit now in parallel.

5.3.3 Conclusion

Actor systems are a rather new formalism. It is based on the idea of extending a programming language with some communication primitives. As base programming language, they have chosen λ-calculus, which in our opinion was a mistake because it imposes too many restriction. On the other hand λ-calculus can be quite nicely integrated with the communication primitives. The communication primitives that extend λ-calculus are: send, letactor and become.

Note that we have no receive command. This is because a send sends a message to a function (λ). We have therefore some kind of RPC, but the send is asynchronous. "Letactor" creates a new actor and the "become" primitive changes the behavior of the current actor.

Although actor systems seem difficult at first sight (could have something to do with the paper in which they are explained), they are really quite usefull. However it takes quite a while to discover its beauty. When we look at actor systems we should not concentrate on the calculus itself, but on its concepts because they turn out to be very usefull.

We think however that actor systems have a lot of drawbacks. First of all they are complicated. Very complicated. (We have spend an entire hour, trying to see how the tree-prod algorithm worked). The become operator has a lot to do with this. Changing procedures at runtime makes programs very difficult to understand and to maintain. A static analysis of the program makes no sense any more because at runtime all actors are replaced by others anyway.

We also have some questions about concurrency problems. There is no mechanism to intercept two actors sending messages to the same actor. Since all parameters to a function are curried[3], mixups can occur. We also have a grudge against the become operator. Extending a functional language with a destructive operator like become seems not very elegant. Equivalent with RMI it is quite difficult to receive the return value of a function. Again some sort of callbacks must be created.

An advantage of actor systems is that they use asynchronous communication. Exactly what we want in a mobile environment.

5.4 RMI

RMI stands for Remote Method Invocation and it is also the name Java has very pretensiously given to its communication package. Java already supports socket-communication but wants to go beyond this low-level communication model and therefore created RMI. Java RMI is a way for clients to execute a method on a remote server object. The client waits for the result of the method to be returned.

Java RMI is nothing new and can be simulated with socket calls[4]. Sockets

[3]Currying comes from λ-caluclus and is the phenomenom where we send our arguments to a function one by one.

[4]Every communication model can be written using only socket calls, just like every program can be written in assembler.

are a good method of communication for raw data, but they soon become a hassle because programmers have to come up with appropriete ways of encoding data and transmission protocols. RMI solves this problems by introducing a transport layer that handles all data encoding itself.

RMI is based on a client/server model. That is the client first creates a reference to a server (a remote object), then asks this server to execute a certain method and then waits for the result. The client/server connection thus only exists for a single method invocation. So to write an RMI client/server application we at least need a server-object and a client-object which may both reside at different locations. Both objects don't even have to know each other's address.

Only the client can initiate communication and needs some way of creating a reference to the server. This is done by a nameserver. Every RMI-application has at least one nameserver. Every server object first registers itself with this nameserver. Later clients can retreive a reference to a server by looking up its name. Java also needs a security manager and a couple of stubs and skeletons to work. A stub is a dummy object on the local site that contains the same methods exported by the server. The skeleton is the dual object residing at the server side. They both have a contemplary role. The stub intercepts all method invocations and marshalls them to the skeleton, while the skeleton delegates every marshalled message to the server object. Fortunately we do not have to write these stubs and skeletons ourselves since Java has created a tool that automatically generates them.

Here is an example of how Java RMI is actually used:

The remote interface:

```
public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

The server object:

```
public class HelloImpl extends UnicastRemoteObject implements Hello {
   public HelloImpl() throws RemoteException {
      super();
   }
   public String sayHello() {
      return "Hello World!";
   }

   public static void main(String args[]) {
      // Create and install a security manager
      if (System.getSecurityManager() == null) {
```

```
            System.setSecurityManager(new RMISecurityManager());
          }
          try {
             HelloImpl obj = new HelloImpl();
             // Bind this object instance to the name "HelloServer"
             Naming.rebind("//myhost/HelloServer", obj);
             System.out.println("HelloServer bound in registry");
          } catch (Exception e) {
             System.out.println("HelloImpl err: " + e.getMessage());
          }
          }
       }
```

and finally the client

```
    public class HelloApplet extends Applet {
       String message = "blank";
       Hello obj = null;         // server object

       public void init() {
          try {
             obj = (Hello)Naming.lookup("//" +
                   getCodeBase().getHost() + "/HelloServer");
             message = obj.sayHello();
          } catch (Exception e) {
             System.out.println("HelloApplet exception: ");
          }
       }

       public void paint(Graphics g) {
          g.drawString(message, 25, 50);
       }
    }
```

conclusion

Java RMI is an implementation of Remote Method Invocation, but just like almost everything in Java it is seriously overhyped. When we compare Java RMI, with the type of RMI that Borg supports, we must conclude that Borg's implementation is more user friendly and practical. In Java we have too much overhead to invoke just a simple message.

Furthermore, Java RMI uses synchronous communication in a client/server model which can not be used in an agent system. Actually the situation is even worse. The client can only send one message to the server. The server can not communicate with the client. This means that no communication protocols (e.g. FTP, POP, ...) can be implemented, since they use several communications cycles. Another drawback is the nameserver. A nameserver becomes useless in a mobile environment. It is ok to use a nameserver to look up the address of a server once, but in a mobile environment, all communication must go through this centralized server. It would soon become the bottleneck of the entire system.

5.5 Obliq

Obliq is another, popular mobile language. Although its principles are very similar to those of Borg we have included this language, not to restrict us to a single mobile environment.

Obliq is a lexically-scoped untyped interpreted language that supports distributed object-oriented communication. Obliq objects have state and are local to a site. Obliq computations however are free to roam the network while maintaining network connections.

Obliq is an Object Oriented language with a simple syntax. An example should make everything clear.

```
let o =
  {  x => 3,
     add => meth(self, y) self.x := self.x+y; self end,
     inc => meth(self) self.add(1).x end };

o.x        Select x from o.
o.x := 0   assign x to zero.
o.add(1)   invoke a method with parameters.
o.inc()    invoke a method without parameters.
o.inc := meth(self) clone(self).add(1).x end
           override inc so that it no longer updates its host object.
```

The Obliq syntax should be clear from this example. Obliq is however more than just another Object Oriented language. It is a distributed language. This means that we can have different objects at located at different sites. Objects must however be placed at their final destination when they are created. They cannot be moved once a location has been assigned.

Of course objects can invoke methods on other objects located at a different site (RMI). All kinds of values can be passed as a value, including an object. This does not mean that the object itself is transferred to the other site. Only a reference to the local object is created, and references are free to be moved around the network.

Obliq also works with a nameserver. The nameserver is referred to by its IP address. Like Java RMI, objects can register themselves with the nameserver. And other objects can ask for a reference. Once an object has obtained a reference, it can send messages to it, or pass the reference as a parameter to other objects.

```
net_export("obj", NameServer, site1Obj)
let remoteObj = net_import("obj", NameServer)
```

As already mentioned, Objects cannot be moved from their site, only references can. There exist however a technique to make migrate agents (computations in Obliq).

```
let agent = proc(state, arg) ... end;
atSite1(proc(arg) agent(copy(state), arg) end)
```

We assume that an agent is an entity dependant on some inner state, and some external arguments. When we want to migrate the agent we ask an available execution engine (atSite1 in the example) to evaluate a procedure that contains a copy of the state of the original agent. Thus we do not actually move the agent, we create a copy. Actually the programmer himself has to take the initiative not to use the old copy anymore.

With some good intent, we can view Obliq as a mobile language. However, Borg seems to handle mobility much more flexible. Obliq does not offer telescripting (migrations can only occur at certain specified points in the code) and we personally think that Obliq can be called a distributed language, but we don't think it is very mobile. We have presented the example here because Obliq is much referred to in literature and it demonstrates what current mobile agent technology looks like.

Chapter 6

CSP in Borg.

6.1 Extending Borg With A Communication Model

After looking at the problem of finding a good communication model for Borg, we have first turned our attention to the existing models.

Both π-calculus and actor-systems are a bit too low-level to implement in a high-level programming language. We especially dislike actor-systems because of their incomprehensible "become" instruction. Using this instruction, programs quickly become difficult to understand and maintain. Although we believe that the become-instruction could become redundant in an imperative programming language, we can not prove this. As a result we can conclude that actor systems are a poor candidate and difficult to integrate with the Borg architecture.

π-calculus on the other hand, is to low-level, uses channels to communicate and uses synchronous communication. On top of that its syntax is difficult to map on the existing Borg architecture. Another major nuisance is the scope operator. When we write for example $(x)(P_1 \mid P_2)$, then the variable x is both visible in process P_1 and P_2. Now in a mobile agent system, where we can start moving agents around, it is very difficult to have shared scope between agents on different machines.

Another formal model is CSP. This model is superbly suited to be integrated with Borg. First of all it is based on an imperative language (it has an assignment operator). Next CSP uses tables as structured datatype. This corresponds to Borg, which also has tables as only structured data type. CSP stands for communicating sequential processes. We take a bunch of sequential processes and execute them concurrently. Again this corresponds to Borg's idea of agents. An agent is a program written in a sequential way (Pico). The power of Borg comes from the idea that we can execute several of these agents at the same time and let them interact. Also CSP uses process names to communicate with other processes. Currently in Borg, this method is also used. We first create a remote dictionary (reference to a remote agent) and use this to communicate with it. So we actually specify the agent we want to talk to, we do not use channels or links. All this evidence makes CSP seem a perfect candidate for a communication model for Borg.

6.2 Implementation

To validate the usefullness of extending Borg with some CSP-like primitives, we have implemented CSP on top of Borg. At an abstract level this is relatively easy to achieve. CSP has only 7 primitive communication functions:

- send
- receive
- sequential
- parallel
- guard
- alternative
- repetition

The sequential command can be immediately removed from this list since it is already implemented in Borg. To execute a list of commands sequentially, just enclose them in braces and insert a ";" between each instruction.

Repetition is also implemented in Borg. We have at least three different kinds of loops (for, while, until), the user can define additional ones, and we can write recursive functions. Enough repetition we would gather. This then leaves the guarded command, only to be executed in combination with the alternative command. They can therefore be regarded as one command. After elimination of the trivial commands, there are only 4 commands left to implement. The new programming language which is the result of extending Borg with some CSP commands is jokingly called C-Borg (pronounced CyBorg), a suiting name for a mobile multi-agent system.

6.2.1 Send, Receive

IThe four commands we still need to implement are the send / receive mechanism, the parallel command and the alternative command. The send/receive is maybe the most important command to implement. We do not like Remote Method Invocation, since it makes returning computed values rather difficult, and requires to think about a problem in a sometimes rather awkward way.

In a first approach we have made the send-instruction asynchronous. It just puts the message in the router system, and continuous the current computation, not waiting on the corresponding receive command to be executed. The sending of CSP-messages can be quite easily integrated with the remote method invocation, used so far. The router is already capable of transferring messages and values. Only when the message arrives at its destination, all arguments are assembled in an application. We have added tags to the messages in the system. When the message is a RMI-message, an application will be created, otherwise, when it is a CSP message, the message will just be put in the message queue.

To execute a receive command the queue is traversed and any pending messages are retrieved. When no message has arrived yet (the corresponding send hasn't been executed) the program will be blocked till a new message arrives. This makes the receive synchronous. The result of the receive command is the value just received.

We now have a synchronous receive and an asynchronous send. This is a rather pleasant and new way to work with communication. It even feels quite natural. Actually it corresponds to the current methods of IO known today. A scanf or an input command will wait for input causing the program to block, while a corresponding printf or write command immediately continuous, without waiting. We have however adapted send's semantics to match that of true CSP. When a receive actually accepts the message, it will send a message indicating that the sending process may continue. The send process will, after it has put the message in the message system, change to a receive, and wait for an acknowledgment.

The send command takes as a parameter the machine to which to send the message and a value to send. If we want to send multiple values, we should put them in a table. The receive command is a bit more complicated. It can take one or two parameters. When we use receive with only one parameter, this parameter must specify the machine we wish to receive from. When we add a second parameter, this parameter will be interpreted as a pattern that will be matched with the pattern received. The Borg special keyword void is used to match anything. Using this parameter we can easily tag messages. For example: recv(someone, 'checkpoint1'). This instruction will only continue when it receives the message 'checkpoint1'.

6.2.2 Parallel Command

In CSP, the par command spawns a number of new processes. In Borg, this is of course slightly adapted. The par-command now spawns new agents. We have designed the par-command so that it creates a new agent with a partially shared read-only scope with its creator. This makes it a lot easier to define new agents, and specifying their behavior. Consider for example the following agent:

```
function1()::...
function2()::...


par(function1(), funtion2())
```

This little program defines 2 functions, and then spawns two agents who execute these functions in parallel. Now because we use the read-only shared scope, we have no problem letting the agents know what function they have to execute. Up till now, with only the agent-clone, and the "new-agent"-button, defining a new behavior for an agent could be quite tedious. Using the shared scope this problem is solved. Another advantage is that we can still use variables defined in the parent's scope, and we have absolutely no problem migrating agents, since there scope is read-only. It is just a snapshot of what it looked like before the agent was spawn.

There are however a few exceptions to the shared scope. Consider the following command:

```
par(Processes, i, 1, 10, some_process(i))
```

This instruction will spawn 10 new agents. Each agent will execute the function some_process(i). Now of course here we have an exception of the shared scope principle. Every agent has it own value for the variable i. The Processes variable in front is bound to a table containing the 10 agent's remote dictionaries. Because this variable is shared by all agents, we have no trouble communicating. Say for example that all these processes are linked together in a chain. When agent i now wants to send a message to its predecessor it can do so: send(Processes[i-1], value). "Processes" is shared, "i" is not.

Now the par-command goes even further. We can specify 30 identical processes but with a different head and tail.

```
par(chain, i, 2, 29,  node(i),
    chain, i, 1, 1,   start_node(),
    chain, i, 30, 30, end_node())
```

6.2.3 Alternative Command

The alt command is similar to the par command. It functions by trying to execute all branches that are available, just as is described in the book about CSP[2]. It then takes one branch of which the guard evaluated to true and executes this. The syntax is quite similar to that of the par-command.

```
alt(i, 1, 5, recv(phil[i], 'enter'), occupants:=occupants+1,
    i, 1, 5, recv(phil[i], 'leave'), occupants:=occupants-1)
```

Note that no new syntax forms were introduced . We could extend Borg's parser, so that the code above would resemble the original CSP model a bit more, but all functionality is included. After all this was a test to see if CSP could be united with Borg. Adding some syntactic sugar will not change the functionality.

6.3 Dining Philosophers In Borg

Just to show that all the commands we have introduced do really work, take a look at this code. It is the standard program of the dining philosophers based on the original CSP code.

```
think(): ...
eat(): ...

fphil(i):
  while(true,
   { think();
     send(room,'enter');
     send(fork[i],'pickup');
     send(fork[i\\5+1],'pickup');
     eat();
     send(fork[i],'putdown');
     send(fork[i\\5+1],'putdown');
     send(room,'exit');
   });

ffork(i):
  while(true,
   alt(0,0,0,recv(phil[i], 'pickup'),
             recv(phil[i], 'putdown'),
       0,0,0,recv(phil[if(i-1=0,5,i-1)], 'pickup'),
             recv(phil[if(i-1=0,5,i-1)], 'putdown')));

froom():
{ occupants:1;
  while(true,
   if (occupants<5,
     alt(i,1,5,recv(phil[i], 'enter'), occupants:=occupants+1,
         i,1,5,recv(phil[i], 'exit'), occupants:=occupants-1),
     alt(i,1,5,recv(phil[i], 'exit'), occupants:=occupants-1)))
}

par(room, void, void, void, froom(),
```

```
    fork, i   , 1   , 5   , ffork(i),
    phil, i   , 1   , 5   , fphil(i))
```

These dining philosophers have also been rewritten so that the library (thinking room) is on one computer and the dining room is on another. The philosopher agents actually move between the library and the dining room when they are hungry. The code for this program looks exactly like the program above, only a few lines containing agentmoves have been added. The concept of this program is of course rather foolish, but it proves that CSP and Borg are buddies. They have no problem working together.

6.4 Sieve Of Eratosthenes

Here is an example of the parallel sieve of Eratosthenes as it would be implemented in Borg.

```
fsieve():
{
  prime:recv(sieve[i-1]);
  display(prime);
  while(true,
  {
    num:recv(sieve[i-1]);
    if( (not ((num\\prime)=0)),
        send(sieve[i+1],num),
        void)
  })
};


startsieve():
{ for(t:2,t:=t+1,t<100,
    send(sieve[2], t))
};

collector():
  { while(true, recv(sieve[i-1])) }

par(sieve, i, 31, 31, collector(),
    sieve, i,  2, 30, fsieve(),
    sieve, i,  1,  1, startsieve())

tecra/ses2.sieve[2]: 2
tecra/ses2.sieve[3]: 3
```

```
tecra/ses2.sieve[4]: 5
tecra/ses2.sieve[5]: 7
...
```

6.5 CSP To Prevent Deadlocks

Throughout this document it has been pointed out many times that Borg's creators are not very happy with RMI as their base communication model.They would like a more send/recv kind of model to support their communication. Of course if all that had to be done was add a send and a receive to Borg, this would have been done a long time ago. The big problem when adding synchronous sends and receives to Borg is that we will almost certainly run into deadlocks. Consider the example of two mutual recursive procedures. Each procedure resides on a different machine. When the first procedure calls the second, it will wait for the result to be returned. So when the second procedure calls the first again, nothing will happen, because the first is still waiting for the second to return the result. We have a deadlock.

Consider also an object with two methods. One is a simple method, but the other is a time consuming one that takes for example 2 hours to complete its computation. When we use RMI to invoke the difficult function, and a bit later call the simple function, we have to wait two hours before both results are returned. In principle we could return the result of the easy method almost right away, if we could run it in parallel with the second method. If one object has different parallel threads, we will almost certainly encounter problems since both methods could update the same variable at the same time.

If we still want to go through with this technique we need to have a good communication model that can resolve all conflicts between two threads within the same agent. We have chosen a CSP-clone for this model. This gives a very flexible language. We have concurrent running agents, and every agent can consist of several processes running in parallel. This model has been implemented and some very interesting results could be observed.

In our implementation we have CSP's main primitives: par, alt, send, recv. The seq native and the repetition command could be simulated using the Borg primitives begin and while. The send and receive are synchronous, although we have also experimented with an asynchronous send. We could always synchronize by confirming the receive.

The par command was slightly modified. It just executes several commands in parallel. Every command has its own variable environment, but both can access the common parent environment (read and write).

```
a: remotedict('tecra/ses2');
par(while(true, display(1)),
    { recv(a); agent_move('tecra.duh') })
```

This simple example shows a nice example of telescripting and the integration of CSP and Borg. This program just starts spawning 1's until another agent send it a message. Then suddenly another machine in the network will start spawning 1's. Very impressive to see.

Because the shared scope of the branches of a par command could cause variable update problems we have introduced an indiv native (indivisible). This native treats the following expression as a single expression and will not task-switch during the execution.

```
t:0;
par(for(i:1, i:=i+1, i<100, t:=t+1),
    for(i:1, i:=i+1, i<100, t:=t+1));
t
: 160
```

After this program has run, the variable t will have a value of 160. Another time t might be 182. When we place the t:=t+1 expressions within an indiv-clause, the program will work correctly.

```
t:0;
par(for(i:1, i:=i+1, i<100, indiv(t:=t+1)),
    for(i:1, i:=i+1, i<100, indiv(t:=t+1)));
t
: 200
```

We have also included an alt command for completeness, but no very spectacular results have been achieved with this command.

Using a CSP-like model to create several processes within one agent, is a very good idea. It adds a lot of power and flexibility to the original language.

6.6 Traditional Models Still Remain Valid In The Presence Of Mobility

By implementing CSP in Borg, we have shown that it is possible to reconcile CSP with the mobility constraint. Not only have we shown that CSP can be implemented in a mobile agent language, it actually extends the language giving it more flexibility. Admitted, we had to adapt CSP a little bit to get the most out of this partnership, but CSP is a good base to start from.

We claim that not only CSP can be integrated with Borg, but that all communication models remain valid in a mobile environment. π-calculus and actor systems can also be implemented in Borg, and any other mobile language. We will prove this claim for actor systems and π-calculus in the next chapter but we will start here by presenting a theoretical argument.

Because all current communication models have successfully withstand the test of time, and thereby have proven there usefullness, they must be a good place to start from if we need a communication model for any application. So if we can show that all communication models for normal agent systems remain valid in the presence of mobility, we can reuse a lot of work done in the past.

Actually it is no surprise that these models remain valid. No communication model specifies its messaging system. They all start from the idea that there is some kind of messaging service that makes sure that when one program executes a send, and another a recv it will work. Actually most of these programs are initially designed or implemented to be used on a single machine. But since the messaging system is not specified, no one can stop us to put different agents, processes or actors on different machines. As long as we can create a messaging service that transmits a value from a send to a receive, everything keeps working. Therefore there is no problem when the agents start moving around the network. The calculi don't care, as long as their messages are delivered.

If there exists a messaging system for a single machine, it can be easily transported to a network. All we have to do is create a single centralized node that implements the messaging system. This node will map process names to physical machine addresses. When we introduce mobility, all we have to do is to keep track of which agent is on what machine. When an agent moves to another machine, it will notify the messaging center of its new address.

The idea of a centralized messaging center is a rather simple approach and is not suited to be used in a real world application, but it demonstrates that the traditional models are still valid in the presence of mobility.

This is however a theoretical prove, not suited for practical applications. In the next chapter we will present some more evidence, and show that all calculi mentioned in this document (CSP, π-calculus and actor systems) can be implemented in Borg.

Chapter 7

The KaSAAC model for communication

7.1 Definition

7.1.1 Introduction

In the previous section we have demonstrated that the traditional communication models can still be applied to mobile agent systems. Of course all these models need some slight adaptations but the core idea remains.

However when we want to equip Borg with these communication models, all models seem to have their drawbacks. First of all CSP and π-calculus use synchronous communication. This is not acceptable in a mobile agent-system. Furthermore CSP does not provide a mechanism for dynamic agent creation. π-calculus is too low-level to be usefull in a high-level language like Borg. Only actor-systems remain as a suitable candidate but we really detest the become-operator in actor systems. It makes programs hard to understand and maintain. Apart from this operator, functional actors might present some problems in the presence of an assignment operator.

Because the current models aren't really ideally suited, we have developed a new communication model. Karsten's Synchronized Asynchronous Agent Communication (KaSAAC), is actually based on actor systems. The idea for this model came from the idea to extend actor systems with an imperative language (like CSP) instead of λ-calculus. We were hoping that the assignment operator could replace the annoying become. Apart from this we have looked at the send/recv pair. The synchronous send can be easily implemented by an asynchronous followed by a synchronization. For the receive this is a bit more difficult. How do we perform an asynchronous receive? Well if there's a message present, the receive message returns it. If there are no messages, the asynchronous receive will return a unique value: *"nomesg"* indicating that no communication can be performed. Thus we can rewrite the synchronous send and receive as an asynchronous send/receive coupled with a synchronization. Our communication model will therefore exist of the following primitives

- sync
- send
- receive
- new

This seems simple enough, however we will prove that the combination of these commands is as powerfull as any of the other formal models.

One of our major concerns during the development of this model was to be able to receive communication from different agents simultaniously. Suppose we have a multiplicator agent that can receive input from two different processes. In CSP and π-calculus this is achieved by using the alt-construct. In our model this can be achieved by the following construct:

```
sync();      //wait until someone is ready to communicate
if ((mesg:=recv(agent1)) != nomesg)
  { // do something with message }
else if ((mesg:=recv(agent2)) != nomesg)
  { // do something with message }
```

Since recv is asynchronous, it will return the unique *nomesg*-identifier when no message has arrived. So we can easily model different communication requests.

What this little program really does is make the alt-construct explicit to the programmer. The programmer can make fancier constructs if he wants to. Another advantage is transparancy. We now know what's going on; we have got rid of the nondeterminism of the alt-construct. It is now possible to write assignments and output-commands in a guard, since we know at all times what's happening.

7.1.2 Base Imperative Language

As we have mentioned, this communication model is based on 4 primitives that are added on top of an imperative language. So in order to fully describe this new model, we need to choose the model of imperative language. As base language we have chosen Pico. Pico is a simple language, especially designed to be educational. It includes no difficult constructs and can be easily formalized. Furthermore, this makes it easy to extend Borg with our communication model since Borg is based on Pico. Using Pico does not constrain our model to much because almost all of Pico's syntax and semantics can be found in any other imperative language.

Basically Pico consist of few native types like numbers, text, functions,... It has one structured expression being a table. As basic commands we have definition, assignment and function application. A full overview of Pico is given in section 3.1 where Pico's syntax and semantics are fully explained. Pico has a few features (like functional parameters and the apply-operator) not found in most languages. But these features are not explicitly needed by our communication model and can be left out.

Pico has 4 special values being void, eoln, true and false. To make Pico a basis for our communication model, we will add one more special value: *nomesg*. This value will be returned by the recv-command, when no communication is available. Since we are dealing with agents, we need the concept of an agent, or at least a reference to an agent. Because they are not natively present in Pico, we add a reference to an agent to Pico's grammar.

Thus we present a simple language, containing numbers, text, functions, tables, agents, definition, assignment and function application. This language resembles the base language for CSP remarkably well, and thus the choice of using Pico as the base for our communication model didn't pose any special restrictions or drawbacks.

7.1.3 Send

All send-instructions in the different formal models, look similar and this send will not deviate from this standard. The send command takes two parameters. The first parameter specifies the agent, you're wishing to send a value to, the second parameter contains the value.

The send is asynchronous. Thus when evaluating it, the message will immediately be put in the message delivery system, and the send will not wait for the message's delivery.

```
send(agent2, 'someValue')
```

7.1.4 Receive

The receive takes two parameters too. The first one is the agent one wants to receive a message from. This argument can also be void indicating that you're not interested who the sender of the message is. It may be anyone.

The second argument specifies a pattern. When a message is received, it will first be compared to this pattern. A pattern can only contain numbers, text, tables, agents and the special identifiers: void, eoln, true and false. Using a function as a pattern has absolutely no meaning since it is not clear how to compare two functions. Especially when these functions reside in different agents on different machines, we have no way of telling wether two functions are the same. The special void-keyword is used as a wildcard. It matches any argument.

The receive command will return the value of the received message, only if a message has arrived originating from the specified agent, and if the message matches the specified pattern. When one of these conditions is not fulfilled, recv will return the *nomesg* identifier. Simply said: if there's a message, it is returned. If there's no message, the result of the recv instruction is *nomesg*. The recv command is therefore, just like the send, asynchronous.

agent1	agent2
	send(agent1, 15)
recv(agent2, void)	:<void>
:15	

agent1	agent2
recv(agent2, void)	
:<nomesg>	

agent1	agent2
	send(agent1, ['tag', 15])
	:<void>
recv(agent2, ['tag', void])	
:['tag', 15]	

Maybe the patternmatcher can be extended some more. Void is not a very good symbol for a wildcard. Though a real patternmatcher, in which variables are automatically bound to their corresponding values is rather difficult to create. In CSP this also poses restrictions.

7.1.5 Sync

The sync primitive shows some resemblance to the recv command. It takes two parameters similar to that of recv. The first parameter is the name of the agent you want to synchronize with. It can also be a table of agents, indicating you want to synchronize with any of these agents or it can be void. Void is a wildcard; it basically says that you're not interested in who's trying to synchronize with you. A sync instruction will block until a corresponding sync has been found. The corresponding sync is a sync-command that specifies the first agent as a synchronee.

To be able to implement the sync-command in a normal agent-environment, we impose the restriction that at least one agent in a sync-pair will have to call the other agent by name. This means that two sync(void,...) instructions will not synchronize. In actual programs this poses no problem, since a sync-pair is almost always used in combination with a send/recv-pair. The send-command has to specify its communication partner by name, so the sync can use this same name.

The sync command’s second parameter contains a pattern. When two corresponding sync’s have been found, their patterns will be matched. Only when they match, will both processes synchronize.

agent 1	agent 2
sync(agent2, ’tag’) <*process waits*> 	 sync(agent1, ’tag’)
...	...

agent1	agent2
sync(agent2, ’tag’) <process waits> 	 sync(void, void)
...	...

agent1	agent2
sync(void, void) <process waits> 	 sync(void, void) <process waits>

7.1.6 New

The new-command creates a new agent. The new-command takes two parameters. The first is the name of the agent. This is analogous to the CSP par-command, in which we also have to specify the name of the process we wish to create. The second parameter specifies the instruction that has to be carried out on the other computer.

When evaluating this command, a new agent is created with a name given by the first argument. This new agent will then evaluate the expression denoted by the second parameter.

```
new('hello', display('Hello World'))
```

This code creates a new agent called ’hello’ that shows the text ’Hello World’ on the screen.

```
fac(n):...
new('agent1', display(fac(5)))
```

This will spawn a new agent that computes the faculty of 5. It is here that we see the first practical problems arise. How does the second agent know the definition of the fac-function. In our model we will assume that a copy of the entire environment is made. In practice this is a bit more problematic. Taking entire copies consumes lots of memory, increases the used bandwidth when migrating agents and would even mean a security hazard. In practice this problem can be solved by making the programmer explicitly export all variables he wants to copy, or by just asking a copy of the variable/function when it is first needed.

7.1.7 Implementation

When designing this formalism we have always made sure that the formalism could be implemented. We claim that this is possible, and here we will see how we can implement KaSAAC in Borg.

Pico is the base language, and Borg is derived from Pico, so implementing the base language is a trivial matter which will not be discussed. But again not to restrict ourselves, Pico is a simple language, and a language like pascal or Scheme already has more functionality then Pico. So implementing the base language is not to difficult.

KaSAAC has four communication primitives: send, recv, sync and new. These can all very simply be implemented in Borg. We have implemented CSP, and the implementation of KaSAAC is very similar. Borg has a message queue and a router system to deliver messages. These enable us to asynchronously deliver messages. Since Borg already knows how to serialize and deserialize a message, a send just results in putting the message into the messaging system. This works asynchronous, and thus the send is asynchronous. To evaluate a receive command, we scan the message queue. If a message with a pattern that matches with that of the recv is found, it is dequeued and returned to the programmer. If no such message is available, the nomesg identifier is returned.

The sync command is a bit more difficult, but we have implemented it in CSP already and we can just borrow it. We have specified that at least one sync in a sync pair must specify its partner by name. That sync sends a unique message to the other agent. The other agent will scan the queue every time a new message arrives, until a message matches with his parameters. If this happens, a confirmation is send back, and both agents can continue because they have synchronized.

The new command can also be easily implemented in Borg. We just create a new agent, create a copy of the current environment and insert it into this agent. Then the expression to be evaluated is pushed on this agent's stack and the new agent is activated.

We can conclude that KaSAAC is easily implementable in a real agent language. We must however admit that some extra primitives might be usefull. In a formal model it is acceptable to poll for messages in a while loop. In a real application this would take up to many resources. This can be solved with some queuepolling primitive. The primitive "poll" can also be easily implemented. It is the same as a sync. It waits for a message to arrive in the queue, and resumes execution when this message has arrived. In the next sections we will see that a send and receive are often accompanied by a sync to perform a synchronized send. Therefore it might be usefull to include a ssend and a srecv primitive (synchronous send/recv).

As mentioned it might also be usefull in a real application to specify together with the new command what variables should be exported. Or we could use some convention. The ::-operator might be a good candidate for this. We export all constants.

That KaSAAC is high level and easy to implement is an important property. Until today no high level formal communication models existed. Many application-programmers therefore had to fall back on their imagination or on socket calls if they needed communication. As a result many languages today support a big range of communication operators. While other languages barely get past socket-calls. This could soon change because KaSAAC is a high-level formal communication model.

7.2 Validation

For validation of this proposed communication model, we have created a transformation that maps a π-calculus program onto an equivalent program that uses our communication protocol. By implementing this transformation we are able to prove two important properties. We demonstrate that the communication model works and that it can be used to implement real applications.

Because this transformation allows us to simulate any π-calculus program, we have proven that our communication model provides at least the same power as π-calculus. We have also found transformations that map CSP and actor-systems on our model. This means that our little communication model is at least as powerfull as any of the current formal models.

Until today we haven't found any proves of equivalence between any of these formal models. In our opinion this is because no-one has yet found an appropriate transformation yet. We think it is so difficult to find a transformation between any of these models, because they are all to low-level, and too difficult to design a real application in.

So in short: for validation, we will implement π-calculus, CSP and actorsystems, showing that KaSAAC is a high level language in which complicated applications can easily be modelled and that it has at least the combined power of all current communication models.

7.2.1 π-calculus

Just to refresh. This is what π-calculus looks like.

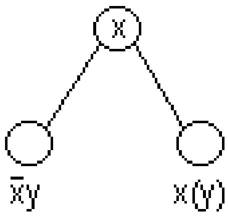
$$
\begin{array}{lll}
P & ::= & \mathbf{0} \\
 & | & P_1+P_2 \\
 & | & \overline{y}x.P \\
 & | & y(x).P \\
 & | & \tau.P \\
 & | & P_1|P_2 \\
 & | & (x)P \\
 & | & [x=y]P \\
 & | & A(y_1, ..., y_n)
\end{array}
$$

The transformation is rather difficult. Especially since KaSAAC communicates by specifying its partner by name, whilst π-calculus works with channels. In order to solve this, we will define our own message delivery system. Our program will look like a tree of little agents. Only the leaves will perform the actual computation. The nodes are only there for communication. They will specify the channels used. All channels will be defined by a restriction operator. Communication will then happen by sending the message up the tree, until it receives the agent (restriction) that implements the channel to be used. There it will be stored. When two messages: a send and a receive accumulate in the channel, they will be connected and the communication will occur.

The simple π-calculus program:

$(x)(\overline{x}y.0 \mid x(y).0)$

that just communicates the value y over channel x will be transformed in a tree looking like this:



When P_1 wishes to send y over link x it will send y up the hierarchy, until it reaches the point where the channel x was declared. There it will be stored. Every channel is a sort of container. It holds all messages that are currently on it. When process P_2 is ready to receive a value over channel x, it will also send a message up the hierarchy. When channel x receives the receive-request, it will send the message y to the recipient, and it will send the sender a message instructing him to continue the computation. After the value y has been received, a new node will be inserted, making sure that all messages for channel y are forwarded to the sender. This way we have no problem with two phenomena called scope intrusion and scope extrusion.

0

We will start very simply. The equivalent of the π-calculus **0** process is of course *void*.

τ

The result of transform(τ.P) is equivalent with transform(P). The τ process does not do anything.

$\mathbf{P}_1+\mathbf{P}_2$

The next transformation is already a bit more complicated. We will first discuss P_1+P_2. This evaluates either to P_1 or P_2. We decide by the means of a coin toss.

```
if(((random()\\2) = 0),
  transform(P1),
  transform(P2))
```

In the presentation we haven't been explicit about the existence of the if-command. Pico's true and false are the Church-booleans, borrowed from λ-calculus. So the if is only syntactic sugar for some juggling with true and false. We will also make no problem of loops. Pico supports recursion, and we can rewrite every loop as a recursive function call[1].

$\mathbf{P}_1 \mid \mathbf{P}_2$

The next transformation we will define is the parallel process. $P_1 \mid P_2$. As we can see in the example above, all that needs to be done is split the tree, thus creating two new agents that each calculate a subprocess. There is however a little catch. All communication that occurs in either of the two new agents must be forwarded upward. So we need to install a message forwarder in the current agent.

```
agent1: new(child(P1));       // create P1
send(agent1, self);           // let P1 know its parent
agent2: new(child(P2));       // create P2
send(agent2, self);           // let P2 know its parent
while(true,                   // become forwarder
{ sync(void, void);           // wait for message
```

[1]In smalltalk the method whiletrue has been implemented this way.

```
    mesg: recv(void, void);    // recv message from either agent
    send(parent, mesg);        // forward message
    sync(parent, void)

  })


  child(P):
  {
    while((parent:recv(void, void)) = invalid), void);
    // let my parent send me its name.
    transform(P)               // evaluate the child process
  }
```

The child process first receives its parents name. We haven't used the sync primitive here because we found it too much overhead. The message should arrive almost immediately, making a sync wasted communication.

(x)P

Now comes the difficult part: the scope operator: (x)P.

we will first assume that we have a queue available. This can be easily implemented since we have tables. In this queue we store messages. A message has the following components: ['send', channel, sender, content] or ['recv', channel, receiver]. Now how is a scope evaluated? It first spawns its child responsible for the rest of the computation, and then waits for messages that are to be sent over this channel. When a send/recv couple arrives at this channel, the communication is performed.

There is also an extra message that needs to be taken care of. The comparison. We will later see how this works, for now all we need to know is that the format is ['comp', channel, sender].

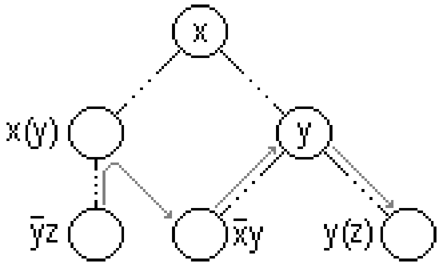
```
  agent1: new(child(P));             // create subprocess
  send(agent1, self);                // let P know its parent
  sendq: createQ();                  // create the message Queues
  recvq: createQ();
  compares: createSet();
  myChannelName: x;

  while(true,                        // receive loop.
  {
    sync(void, void);                // wait for communication.
    mesg: recv(void, void);
```

```
    if((mesg[2]!=myChannelName),      // if the message isn't for me
      {
        send(parent, mesg);           // forward it
        sync(parent, void)
      },
      if((mesg[1]='recv'),            // receiver ready.
        if((emptyQ(sendq)),           // but no sender
          appendQ(recvq, mesg),       // queue message
          {                           // send/recv pair found
            sender: getfirst(sendq);
            send(mesg[3],             // send content to receiver
                 [sender[3], sender[4]]);
            sync(mesg[3], void);
            sync(sender[3], void)     // instruct sender to continue
          }),                         // synchronous send
      if((mesg[1]='send')
        if(emptyQ(recvq),             // received message is send
          appendQ(sendq, mesg),       // analogous to  recv...
          {
            receiver: getfirst(recvq);
            send(receiver[3], [mesg[3], mesg[4]]);
            sync(receiver[3], void);
            sync(mesg[3], void)
          }))
      if((mesg[1]='comp'),
        if((mesg[3]∈compares),
           {
             send(mesg[3], true);
             sync(mesg[3], void)
           },
           {
             send(mesg[3], false);
             sync(mesg[3], void);
             insert(compares, mesg[3])
           })))))
})
```

$\overline{x}$y.P

The output command is again rather simple. We have explained the forwarding system, and all that a send should do is put the message into this system, and wait for a receiver. Now it is a little bit trickier than this. Suppose that we send a channel y to another process, maybe outside the scope y. Now when the receiver process starts sending messages over the y link they should arrive at the correct place, that is they should follow the tree starting at the senders node. This process is called scope extrusion.

scope extrusion



x(y).$\overline{y}$z | (x)(y)($\overline{x}$y | y(z))

```
send(parent,                // deliver message
    ['send', x, self, y]);
sync(parent, void);
sync(void, void);           // wait for message to be received
agent1: new(child(P));      // create child
send(agent1, self);         // let child know its parent
while(true,                 // become forwarder
{ sync(void, void);         // wait for message
  mesg: recv(void, void);   // recv message from child or remote
  send(parent, mesg);       // forward message
  sync(parent, void)

})
```

x(z).P

And finally the receive command. Which should be straightforward now. The only little detail we should bear in mind is link renaming. x(z).P | x(y).Q The link z should be renamed to y when it is forwarded.

```
send(parent, ['recv', x, self]);
sync(parent, void);
sync(void, void);
mesg: recv(void, void);       // mesg=[sender, contents]
sender: mesg[1];
value: mesg[2];
agent1: new(child(P));        // create P
send(agent1, P);              // let child know its parent

while(true,                   // become forwarder
```

```
{ sync(void, void);          // wait for message
  mesg: recv(void, void);    // recv message
  if((mesg[2]=z),
    {
      mesg[2]=content;
      send(sender, mesg);    // forward to sender
      sync(sender, void)
    },
    {
      send(parent, mesg);    // forward to  parent
      sync(parent, void)
    })
})
```

[x=y].P

The comparison operator isn't as straightforward as it seems at first sight. Consider the following example

$$(x)((\overline{x}y.\overline{x}y.P) \mid x(u).x(v).[u{=}v].Q)$$

We can see that just comparing the names is not enough. We will have to follow the link, and see if the initial declaration is the same. We achieve this by sending two 'comp'-messages. One for the u-link, one for the v-link. When both messages arrive at the same node, they are equal. If they arrive in different nodes, they are not equal. To know if they arrive in the same node, each node will remember the name of the agent that sent the message. When a node receives two message from the same agent, both links are the same. When it only receives one, they are different.

```
send(parent, ['comp', u, self]);
sync(parent, void);
send(parent, ['comp', v, self]);
sync(parent, void);
sync(void, void);
recv(void, void);           // the first answer is always false
sync(void, void);
if(recv(void, void),        // if links are the same
   {
     child: new(child(P));  // links the same, evaluate P
     send(child, self)
   },
   {
     child: new(child(0));  // links different, do 0
     send(child, self)
```

```
    });
  while(true,                // become forwarder
  { sync(void, void);
    mesg: recv(void, void);
    send(parent, mesg);
    sync(parent, void)
  })
```

A(x_1,...,x_n)

For every agent identifier A used, there must be a unique defining equation A(x_1,...,x_n)=P, where the names x_1,..., x_n are distinct and are the only names that occur free in P. A(y_1,...,y_n) behaves like P{y_1/x_1,...,y_n/x_n}. The expression A(...)=P is transformed into a Pico definition A:[vars,P]; vars is a table containing the formal parameters. We can transform an agent call into a new agent P, preceded by a node that transforms every y_i into x_i.

```
agent: new(child(P));          // create agent
send(agent, self);             // including the child

while(true,                    // become transforming forwarder
{ sync(void, void);
  mesg: recv(void, void);
  if((mesg[2]=xi),              // transform Xi into yi
     mesg[2]:=yi,
     void)
  send(parent, mesg);
  sync(parent, void)
})
```

Actually there should be a loop that iterates over all i's and performs the comparison for every i. We have omitted this to make the code more readable.

tying up the loose ends

globals There is still a major problem with global variables. Take a look at the following piece of code:

$$\overline{x}\text{y.P} \mid \text{x(z).Q}$$

The link x is nowhere defined. Thus when we send a message through channel x, the message will go up the tree, until it reaches the top-node and then the program will give an error. In order to make our program work we still need a special top-node. There are two possible solutions. The first one is not to accept any π-calculus program with free variables in it. This is no hard restriction since we can introduce a scope operator for every free variable.

$$(x)((y)\overline{x}y.P \mid x(z).Q)$$

The other solution is to create a special top node that behaves like a scope-node, but then for every unknown variable it receives.

end There is still one minor problem with this program. It never stops. This again can be easily solved. The **0**-process will be redefined to send a special 'quit' message to its parent. Every parent is just waiting to receive a message and can easily check to see if it is a quit. If it is a quit message it stops calculating. Only the parallel node $P_1 \mid P_2$ must receive two quits before it may stop.

transform And finally to make the program complete, We will present the definition of the transform function:

```
transform(P):
{
  if(empty_process(P),
     <code for empty process>,
  if(summation(P),
     <code for summation>,
  if(parallel(P),
     <code for parallel>,
  if(scope(P),
     <code for scope>,
  if(send(P),
     <code for send>
  if(recv(P),
     <code for receive>
  if(agent_call(P),
     <code for agent-call>
  if(tau(P),
     <code for tau>,
  if(comparison(P),
     <code for comparison>,
     ...))
}
```

7.2.2 CSP

The next model we will simulate is CSP. Fortunately this is not as complicated as π-calculus. This is mainly because KaSAAC resembles CSP quite closely. CSP has only a few commands. First of all there's the language it is based upon. We will not prove that our base-language (Pico) is equivalent with CSP's underlying imperative language. It should be clear that they have the same functionality.

The key to simulating CSP processes is representing each process by an agent. Since CSP, just as KaSAAC uses process names to specify their correspondent, communication can be modeled quite easily.

send

Send can be easily implemented using a sync and a send. For example p_1!(value1, value2) can be simulated as

```
send(p1, [value1, value2]);
sync(p1, [value1, value2]);
```

The first sync is to notify the other process that there's a message waiting. The second sync will do the actual synchronization. Later, in the alt-construction, we will see why this is necessary.

recv

Receive is analog, although we have to make a difference between tags and variables. In CSP we can write something like p_1?('tag_1', ..., tag_n, x_1, ..., x_m).

This translates to:

```
sync(p1, ['tag1',..., 'tagn', void, ..., void]);
mesg: recv(p1, ['tag1',..., 'tagn', void, ..., void]);
x1:mesg[n+1];
...
xn:mesg[n+m];
```

seq

This native is incorporated in the base language. seq(P_1,P_2) is equivalent with {P_1; P_2}

par

The par-command can be easily simulated by the KaSAAC "new" command.

```
t1(i:1..n)::process1 || t2(j:1..m)::process2
```

There is still a little catch since both variables t1 and t2 are known by both processes and the i's are local to every process. This can be achieved by a few extra communication cycles.

```
t1[n]:void;
t2[m]:void;
for(counter:1,counter<n,counter:=counter+1,
  {
    t1[counter]:=new(
      {
        sync(void,void);
        parent:recv(void,void);
        i:recv(void,void);
        t1:recv(void,void);
        t2:recv(void,void);
        process1;
        sync(parent,void)
      })
  }
<idem for process2>
for(counter:1,counter<n,counter:=counter+1,
  {
    send(t1[counter], self);
    send(t1[counter], counter);
    send(t1[counter], t1);
    send(t1[counter], t2);
    sync(t1[counter], void);
  }
<idem for process2>

for(counter:1,counter<n+m,counter:=counter+1,
    sync(void,void));
// wait for all processes to terminate.
```

alternative command

To evaluate an alt-command, we spawn an agent for every guard. When these guards are evaluated they send their value to the master process. When the master process receives a positive message, it instructs the agent not to send any more messages. There is however a problem. If communication happens inside the guard, it cannot be detected, since the sender of the message sends the message to the parent (the alt-construct) and not to the children. Thus when we reach an input command, we forward it to the parent, instructing the parent to receive the message. In order to do this we can redefine the recv-command inside the children of the alt command. If the parent receives the message, it is then forwarded to the child.

The syntax of the alt, in its most complicated form, looks like: (i:1..n) guard → command

```
children[n]:void;
patterns[n]:void;

create_child(guard, i):
{
  agent: new(child(guard));    // create child
  send(agent, self);           // let child know parent
  send(agent, i);              // let child know i
  children[i]:=agent;
};

for(counter:1;counter<n;counter++,
    create_child(guard[i],i));

stop:false;
winner:0;
while(!stop,
  sync(void, void);     // wait for a message to arrive.
  for(i:1;(i<n) && (!stop),i++,
  {
    if(((m:recv(children[i], ['result', void]))!=nomesg),
       if((m[2]=true), {winner=i; stop=true}, void),
    if(((m:recv(children[i], ['recv', void]))!=nomesg),
      patterns[i]:=m[2],  void)
    if((patterns[i]!=void),
      if(((m:recv(patterns[i][1], patterns[i][2]))!=nomesg),
        {
          sync(patterns[i][1], m);            //mesg recv'ed
          send(children[i], ['recvd', m]);    //forward to child
          sync(children[i], void)
        },  void), void)))
  })
)

// kill all children but the winner.
for(i:1,i<n,i++,
  if(i!=winner,
    send(children[i], ['stop', void]), void))
```

```
// the command has to be executed by the guard process
// since it might have bound some variables.
if(winner=0,
   void,
   send(children[i], ['exec', command[i]]));

// this means that I have to become a proxy
< become proxy>
```

The alt works by spawning a child for every guard. If a child has performed its calculation it informs the parent of the result. The parent will then stop all children. There is however a big problem. Inside guards, an input command may occur. The child however will never receive any input since its address is unknown to all senders. In order to solve this problem, we let all communication take place through the parent. The parent will poll to see if someone has sent a message yet.

If a guard evaluates to true, the parent will send a kill-message to all the children except the "winner". The winner has to execute the command. This must be done by the child since variables can have been updated (if we have m:recv(...) then m should be known inside the command) There is no simple way to transfer all updated variables to the parent, so the command must be executed by the child. This again poses problems. Again communication cannot be done. In order to communicate, we send all our requests to the parent, who will do the communication for the child (parent becomes proxy).

Again there arise problems. Suppose that again an alt command is executed. As a result another proxy is created. Thus in order to communicate, we must follow a line of proxies. All middle proxies are simple, they just forward.

```
sync(child, void);
request: recv(child, void);
send(parent, request);
sync(parent, request);

sync(parent, void);
mesg: recv(parent, void);
send(child, mesg);
sync(child, mesg);
```

The proxies at the end, will perform the actual communication. They have to do something like

```
sync(child, void);
mesg: recv(child, void);
```

```
case(mesg[1]='recv' ->
          { m:pico_recv(mesg[2], mesg[3]);
            send(child, m);
            sync(child, m)
          },
     mesg[1]='send' ->
          { pico_send(mesg[2], mesg[3]);
            send(child, 'done');
            sync(child, 'done')
          }
     mesg[1]='poll' ->
          {
            if(((m:recv(mesg[2], mesg[3]))!=nomesg),
               sync(mesg[2], m),       // accept message
               void)
            send(child, m);            //forward message to child
            sync(child, m)
          })
```

All proxies, children and alt can know when they need to use the proxy. We can therefore always use the correct form of communication.

All there rests to do is define the child. All the child should do is execute the guard and do something special when an input command is reached. It is quite equivalent with the above alt-command.

repetitive command

The repetitive command is quite similar to the alt command. The code for the repetitive command is borrowed from the alt and slightly adapted. Only when the winner variable equals 0 we stop the repetitive command. In case a guard becomes true, we send it not only the command to execute, but also another instance of the repetitive command. This way, the repetitive command will be recursively executed.

tying up the loose ends

There is one feature in CSP we haven't addressed yet. In CSP an input/output command fails when the communication partner does not exist any more. If we would have a predicate that would inform us if an agent was still active we could adapt our model to correctly behave in this case. A recv that would implement this behavior would use polling.

```
m:nomesg;
while(m=nomesg,
{
```

```
    if(agent-alive(P),
       m:recv(P, pattern);
       <commit suicide>);
    delay(1)
  }
```

7.2.3 Actor-Systems

Also actor systems can be simulated with KaSAAC. Fortunately this can be done quite a lot easier than the previous two models. First of all we will assume that λ-calculus can be simulated. The asynchronous send and the letactor functions should also be relatively easy to implement. Only the "become" can give problems. How do we transfer a running computation to an anonymous agent? Once you know the trick it is easy: we make a dummy agent that has the correct name. The actual computation will be carried out by an anonymous agent. The dummy agent will send all its communication to the anonymous one. When we want to execute a become statement, we create a new anonymous agent, and we update the dummy agent to forward all messages to the new agent. We wouldn't need the dummy agent in the presence of a rename or rebind operator. Coincidentely this operator has recently been implemented in Borg. But because KaSAAC does not support this operator, the prove presented here will use the dummy agent approach.

λx.P

```
sync(proxy, void);
send(proxy, ['gimme']);
sync(proxy, void);
x: recv(proxy, void);
act_on(P);
```

send(a, v)

```
send(a, ['send', v]);
sync(a, ['send', v]);    // a is proxy -> asynchronous
```

(M)N

```
send(proxy, ['send' N]);
sync(proxy, ['send' N]);
act_on(M)
```

(M)N is transformed into a send. We send the message N to the process M. We send the message to the proxy. M will retrieve it a few instruction later.

rec(λr.Q)

Since Pico does not have a fixpoint operator, this function will be introduced too.

```
r:rec(λr.Q);
act_on(Q)
```

This is all that has to be done in order to implement recursion. If we call r, the function will be evaluated once again. Note that Q is only a string or ADT representing some λ-calculus. It is no computation or value.

become(P)

```
agent: new(
    { sync(void, void);        // not necessary. new makes copy
      proxy:recv(void, void);  // and proxy remains the same.
      act_on(P)
    });
send(agent, proxy);
sync(agent);
send(proxy, ['update', agent])
sync(proxy);
```

letactor(x:=b)e

```
x: new(proxy());
agent: new(
    { sync(void, void);
      proxy:recv(void, void);
      x:proxy;                     // letactor=letrec.
      act_on(b)
    });
send(proxy, ['update', agent]);
act_on(e)
```

proxy

```
child: void;
mqueue: createQ();
while(true,
  sync(void, void);
  m:recv(void, void);
  case(m[1]='send' -> appendQ(mqueue, m[2]),
       m[1]='gimme' ->
           {
```

```
            if(emptyQ(mqueue),
              {                          //Q empty, wait for msg
                sync(void, void);
                mesg:recv(void, void);
              },
              mesg: serveQ(mqueue));
            send(child, mesg);
            sync(child, mesg)
          },
      m[1]='update' -> child:=m[2]))
```

7.2.4 RPC

We have also implemented Remote Procedure Calls. Just to prove it is possible and that KaSAAC is usefull.

Every agent that is capable of executing a remote procedure call must receive some sort of message and transform it to a procedure call. The transformation would then look like this:

```
ExecuteRPC(sender, procname, args):
{
  result: case(
    procname='proc_1' -> proc_1(args[1], ..., args[n]),
    procname='proc_2' -> proc_2(args[1], ..., args[n]),
    ...
    procname='proc_n' -> proc_n(args[1], ..., args[n]));
  send(sender, ['RPCResult', result]);
  sync(sender, ['RPCResult', result])
}
```

This function exports all procedures that may be remotely called. Actually in Pico we could write: eval(procname)@args and this would have the same result as the function above. But in the beginning we promised we would not use any of Pico's special operators, so we don't.

The RPC-engine itself would then look like this:

```
while(true,
  {
    sync(void, ['RPCCall', void]);
    m:recv(void, ['RPCCAll', void]);
    ExecuteRPC(m[2][1], m[2][2], m[2][3]);
  }
```

And an RPC-call would look like this:

```
RPC(agent, proc, args):
{
  send(agent, ['RPCCall', [self, proc, args]]);
  sync(agent, ['RPCCall', [self, proc, args]]);
  m:nomesg;
  while((m:=recv(agent, ['RPCResult', void]))!=nomesg),
  {
    sync(void, void);
    m:recv(void, ['RPCCall', void]);
    ExecuteRPC(m[2][1], m[2][2], m[2][3]);
  }
  m[2]    //return result
}
```

The difficult whileloop inside the RPC-call is to be able to make mutual recursive remote procedure calls. Apart from the mutual recursive calls the entire program is pretty straightforward, proving once again, KaSAAC's simplicity combined with power.

7.3 Conclusion

We have designed a new communication model (KaSAAC) by extending an imperative language with 4 communication primitives. As base imperative language we have chosen Pico. It might have been any other language, but since Pico is so simple, it is ideally suited as a simple generic imperative language. The 4 extra primitives are send, recv, sync and new. Send and recv are asynchronous messaging primitives. To make the recv asynchronous, it returns the unique nomesg-identifier, when no message is available, or otherwise the message content. We have included the sync primitive so that agents can synchronize when they want to. The "new" primitive creates a new agent. To new agent will be evaluated in a copy of the current environment.

To validate that KaSAAC works and can be used in existing applications, we have simulated several existing formal models. By implementing other communication models we have not only demonstrated that KaSAAC really works and can be used, we have also shown that the KaSAAC communication model is at least as powerfull as the current models. Because we can simulate the other languages, we can use any construct from any other language if we want to. If we would like to use an alt, command, we can borrow it from CSP, just as remote method invocation can be borrowed from actor systems.

The model uses asynchronous communication, just as we wanted for our mobile agent environment, and it has a dynamic way to create new agents. Therefore it can be easily incorporated into an existing mobile agent system. The model itself isn't very demanding, and it can be easily adapted to form the communication model for a mobile environment like Borg.

From the examples above we can deduce that it might be usefull to have a construct ssend and srecv. The synchronous send and the synchronous receive. But this is just syntactic sugar. Also the KaSAAC model can be combined with remote method invocation. And since we can simulate them, all constructs from the formal models can be used. A par/alt from CSP, process passing from π-caluclus, ...

But there are other advantages. KaSAAC introduces the power to access the meta-level communication model. We can access the meta level message queue. Suppose we replace every send(t,v) by send(t, [t,v]). With this cosntruct we can fake the host that sent a message. Thus giving us the power to take messages out of the queue and putting them back. This allows us to poll for messages and gives us complete power over the meta message queue.

KaSAAC is not just a formal model. During its design we have taken care that the model was easily implementable in a real language. In fact this is exactly what the agent programming world needs. Up till now no high level agent communication model existed. So agent applications have no mathematical prove for validating the communication model they implement. In fact, many applications just implement every communication function they can think of. Just take a look at PVM or CORBA. These applications present a huge amount of different communication standards. Actually many systems hardly get past socket communcation (again PVM is a good example)

So in short, KaSAAC is an important formal model that is high level enough to be implemented in an existing programming language. It is at least as powerfull as any model used today, and it can be easily unified with mobile technology.

Chapter 8

Conclusion

Throughout this document we have shown what a mobile agent system might look like and what its major advantages and drawbacks are. We have discussed some problems that are introduced due to mobility and concluded that most of these problems directly relate to communication. Then a brief introduction to several existing models has been given, and we have seen that one of these models (CSP) can be easily unified with Borg. It appears that this unification does not have to be restricted to CSP but can be generalized to other models as well. The following theorem seems to hold in general.

8.1 Traditional Models Are Still Valid And Provide A Good Base For Communication In Mobile Multi-Agent Systems.

The major problem that arises is getting information over the network. Almost all models have some kind of send and receive construct. These functions take a name, link or channel (at least a reference of some kind) as parameter to specify their communication partner. It is this reference that creates the problem. In a mobile architecture we have no way of determining on what physical machine our communication partner is situated. In a practical implementation, we have to specify the physical machine we want to communicate with[1].

Fortunately none of the formal models specify how to get a message from one machine to another . They just assume that there is some sort of messaging system that gets their message from the sender to the receiver. If we can device a messaging system that can deliver a message in a mobile environment, all models remain valid. A solution to this problem can be found in a centralized node that remembers all physical addresses of all

[1]Broadcasts can solve this problem, but from an implementation point of view this is unacceptable

machines in the system. Since all communication must go through this node, this centralized node thus becomes the messaging system. When moving an agent we update the centralized node, making sure we lose no messages in progress. But the trick with the centralized node isn't favorable in practice. In real life, a better solution would be to implement your own routing system, just like the Borg-team did. However since technologies are not part of the existing models, all existing formalisms still work in the presence of mobility.

Because many of these formal models have demonstrated their usefulness by withstanding the test of time. Almost always it will be a good idea to extend a mobile language with any of these models. They will be almost always a good base to start building a real protocol from.

So in short, every communication model that does not specify its messaging system (all models we know of), can be incorporated in a mobile environment.

8.2 Advantages Of KaSAAC

The previous paragraph proves an interesting and reassuring property. We don't have to trash any knowledge we have picked up in our search for mobile agents.

Any agent system poses special demands on its communication model. Agents, and especially mobile agents, that may roam the internet, cannot always afford the delay imposed by synchronized communication. Therefore an agent communication model must offer asynchronous communication and a way to dynamically create agents. Almost none of the traditional models offer asynchronous communication or mechanisms to construct a dynamical agent-environment. Those who do are often too low-level to be of any practical use.

Thus although existing communication models remain valid in a mobile environment, they are not usefull! Almost all these models where written for static small local networks and become useless in big dynamical and slow environments like the internet. Agents request different communication needs than the existing models offer. Therefore, we have developped a new communication model, written especially to statisfy these needs.

Our model consists of an imperative base language and 4 simple primitives. We have chosen Pico as the imperative language to base our model on. One of Pico's main goals is to be simple. And since few imperative languages come in any simpler form it is a superb candidate for a formal model.

Next we have extended Pico with four communication primitives being send, recv, sync and new. Each one does exactly what is says, but bear in mind that the send and receive work asynchronous. An asynchronous receive returns a message when it is available or a special identifier "nomesg" when no messages are available. The main advantage of an asynchronous recv is that we have full access to the communication protocol's meta level. (the programmer can fiddle with the internal message queue as much as he likes).

We have also shown that KaSAAC can be used to model CSP, π-calculus and actor systems. This proves that KaSAAC has at least the same expressiveness and communication power as any of these models. Until today we haven't been able to show an equivalence between any of these languages. Maybe this is because the other formalisms are too low-level, less powerfull, or just too difficult to reason with.

The simulation of the other formalisms also backs up the previous theorem. If we can simulate these models with KaSAAC, and we can implement KaSAAC in Borg, then we must also be able to implement these formalisms in Borg.

Also this unified model might have an impact on existing agent languages. Up till now not much effort has been spent on a solid communication model. Most languages just support every communication function they can think of. When you take a look at a popular system like PVM or CORBA, one finds that its communication model is just a big mix of every thing they could implement. Some languages even hardly get passed the TCP/IP layer. A formalization of communication models is really needed, but up till now, all models were too low level or not practical.

In short, KaSAAC is a high level communication model, very powerfull, easy to understand and implementable in a real application.

8.3 Future Work

If we get positive feedback on this system, we would like to do further research in this direction. After all, KaSAAC still needs to be implemented, and there still are a few alternatives and extra communication natives that might be worth taking a look at. Reifying the message queue into the language might be a good idea, just like event handlers that are activated when a message with a specified pattern arrives.

Apart from KaSAAC, we have also been playing with the idea to take a look at agent security. The main reason why mobile agents aren't roaming the internet today is because of lack of security. We hope to find a way to protect hosts from malicious agents (viruses) and to protect the agent from bad hosts (hosts that look at the agents internal variables or even brainwash them).

Bibliography

[1] INTERNET: http://pico.vub.ac.be/

[2] C. A. R. Hoare, Communicating Sequential Processes, 1985, Prentice Hall

[3] G. Agha, I. A. Mason, S. Smith, C. Talcott, A foundation for Actor Computation, January 1993, Cambridge University Press.

[4] L. Cardelli, Obliq A language with distributed scope, May 30, 1995 11:18PM.

[5] A.D. Gordon, Nominal Calculi for Security and Mobility, januari 1999, Cambridge University Press.

[6] G. Agha, I. A. Mason, S. Smith, C. Talcott, Towards a Theory of Actor Computation, January 1993, Cambridge University Press.

[7] R. Milner, The Polyadic π-calculus: a Tutorial, october 1991.

[8] R.Milner, J. Parrow, D. Walker, A Calculus of Mobile Processes Part I, june 1989.

[9] R.Milner, J. Parrow, D. Walker, A Calculus of Mobile Processes Part II, june 1989.

[10] C. Fournet, Security within a Calculus of Mobile Agents?, february 1997.

[11] R. Focardi, R. Gorrieri, Non Interference: Past, Present and Future.

[12] Antigua Facile, reference guide to the language, october 1993, ecrc-93-20.

[13] C. Meadows, Detecting Attacks on Mobile Agents.

[14] V. Swarup, Trust Appraisal and Secure Routing of Mobile Agents, february 1997.

[15] C. A. Gunter, P. Homeier, S. Nettles, Infrastructure for Proof-Referencing Code, February 26, 1997.

[16] P. Lee, G. Necula, Research on Proof-Carrying Code for Mobile-Code Security, March 1997.

[17] W. V. Belle, Reinforcement Learning as a routing technique for mobile multi-agent systems, januari 1998.

[18] W. V. Belle, T. D'Hondt, Agent Mobility and Reification of the computational state, november 1998.

[19] Java Remote Method Invocation, Sun Microsystems, http://chatsubo.javasoft.com/current.

[20] W. D. Meuter, Agora98 Language Manual, 1997

[21] W. V. Belle, Ontwikkeling van een mobiele multi agent architectuur, mei 1997.

[22] R. D. Smet, Toepasbaarheid van prototype gebaseerde object georienteerde programmeertalen in de context van distributie, mei 1998.