Vrije Universiteit Brussel
Faculty of Science
Department of computer science
2001 - 2002

Mobile Platforms for mobile agents

Verspecht Dimitri

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

Promotor: Prof. Dr. Theo D'Hondt

Acknowledgment

First of all I would like to thank Johan Fabry. He gave me great support and motivation and has sacrificed a lot of his time to read, correct and discuss my work.

I would also like to thank Theo D'Hondt for allowing me to do this research and Werner Van Belle who gave me great feedback during the meetings.

All the other PROG members and my fellow students deserve some words of gratitude too, as they frequently helped me when I had some problems. They also provided me with the necessary feedback and motivated me to keep improving my work.

Final thanks go to my parents for the motivation and support and to Sandrine David who was so kind to check the entire text for grammar and other language mistakes.

Mobile agents for mobile platforms

Dimitri Verspecht

May 26, 2002

Contents

List of Figures

1 Introduction

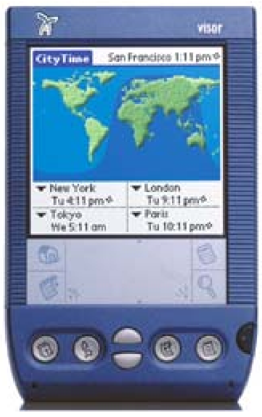


Figure 1: The visor prism, a sophisticated PalmOS device

A few decades ago the PC was a heavy device the size of a large closet. Over the years the PC evolved, it became more powerful and compact. Its functionality increased steadily with, as a result the full grown multimedia machine we know today. Today the PC is a large box with much functionality like sound and movie playback, games, programming, working, ...

A decade ago PC technology became so advanced that manufacturers could shrink their machines even further. Mobile PC's like laptops or even palmtops, little PC's that fit in the palm of your hand (that is why they are called palmtops) were created. In the beginning these little computers lacked much of their big brothers functionality, but as they evolved this quickly changed. Nowadays laptops are not far behind their desktop partners and even the palm sized devices now have advanced functionality like programming, multimedia, network, several work applications, ...

A modern palmtop has the following characteristics:

- A compact size in a shape that makes the device fit comfortable in the palm of your hand.
- A processor which is optimized for low energy consumption
- Limited memory (2 Mb → 64 Mb) due to space constraints
- A large screen covering the most of the palmtops front side
- A stylus to tap the screen which provides the only way of inserting data and navigating programs.
- A simple graphical OS (PalmOS, Windows CE, Pocket linux)
- A battery that provides at least several hours of autonomy

- Other outputs (speaker, led's, IR)
- Expansion slot for expansion or memory modules (Visor, Pocket PC)

These palmtops have become tremendously popular in past years. Because of this success there is a lot of research in the area of mobile computing. This thesis is about the research of "Mobile agents on mobile platforms". Such palm sized palmtops are called mobile platforms. Mobile because you can take them everywhere with you, a platform because they can support agents.
These agents will be supported by a platform called borg. Borg is an agent platform build on a virtual machine called pico. Agents are best seen as autonomous objects. A program in borg is created in the same fashion as in any other OO language. The difference between the two is that agents give you extra functionality. I will go through these extras in the topic about mobile agents. One of the most important properties is the strong mobility of the agents; this strong mobility allows the agents to move from one platform on machine X to another on machine Y while they are running/calculating.
Mobile agent platforms can work on different machines: handhelds, PC's, GSM, embedded computers, ...

This thesis will handle the topic of strong mobility. Chapters 2 and 3 are an introduction to the topic of mobile agents. Chapter 2 handles with the mobile computers. Here we zoom in the exact specifications, advantages and disadvantages of the palmtop computer. Chapter 3 explains in more details the working of our mobile agent platform borg. It also explains what mobile agents exactly are and what their difference is from other agents (known from the AI research). To point out the unique advantages of mobile agents we compare them with 2 other popular solutions to create distributed programs: Java and the Client/Server protocol.
Chapter 4 handles case studies. In total 4 case studies are examined: a mail management system, a meeting planner, a route planner and an advanced route planner. Each case is explained in detail and leads to some interesting conclusions. From all the cases we can conclude that the moving behavior of our agents is often the same and can be programmed in a separate Jump agent. This jump agent can be used in any program to make good use of the weak/strong mobility property of mobile agents.
Chapter 5 will handle the future. The borg mobile platform is far from being complete and extra functionality may greatly improve the use of mobile agents.
Chapter 6 summarizes all conclusions in this thesis.

2 Mobile platforms

Computers give users the benefit of manipulating and organizing data in an easy way. Most users are content with a PC at home or at their work where they can insert their data and work with it. Most commonly used features of a PC are text processing, spreadsheets and multimedia. Some users however do not have the luxury of being able to insert and process their data at home. They have to collect, access and process data while they are on the move. Before the mobile computers this was done with an agenda. Depending on how punctual the user is, such an agenda can be a useful or messy tool. The agenda was used to keep appointments, contact lists and short notes. Wouldn't it be useful to have the data manipulating and organizing skills of a computer with you all the time? This is what the people of the Palm and Psion company must have been thinking when they developed their first organizer. [2] It all started with a very basic organizer by Psion. It had a two line display and 4K data memory. The organizer had the same functions than an agenda: short notes, reminders, calendar, contacts and as an extra also a clock and alarm. Inputting the data on the small keyboard took longer than quickly writing it down. This and the fact the organizer was expensive, still large and limited in functionality made that the device didn't have a real success.

Things changed when Palm introduced their first product: the Palm Pilot [1]. This was the first mobile computer with the characteristics we find in all today's palmtop computers. It has a size that made it perfect to hold in the palm of your hand (thus palm sized or palmtop PC's) and a large screen taking more than 60% of the device surface. It has no keyboard; inserting data and navigating programs is done by a stylus. A stylus is used to tap on the screen, these screen taps have the same effect as mouse clicks on PC's. This method makes it easy to navigate through programs. Data is inserted with graffiti, a hand writing recognition tool. You have to learn an alphabet of simplified characters that can be recognized by the device if you write them on the special parts under the screen. The simple alphabet and writing method is easily learnt and allows users to insert data almost as fast as just writing it down. The device has a full graphical environment, which is easy to use and offers a lot of functionality. It also has the fast and intuitive programs most users require: a simple word processor, an agenda with reminder function, a clock, a to do list (a list of things to do), a short memo manager, ... All these features made and still make the Palm Pilot a record selling machine.

2.1 What makes mobile platforms popular

In paragraph above, we already partly explained what made the palmtop computer so popular. This success wasn't a coincidence. Palm did intensive research before they launched their first product. They wanted to know what users really expect and want from a mobile device. This approach proved to be very fruitful and explains the success of Palm. Palm discovered following requirements for a mobile device [1]:

- The device has to be small and light. Preferably so small that it can be stored in a vest or a jacket pocket. Working with the device has to be comfortable in all situations, this is why Palm made the device fit in the palm of a hand.
- The device has to be easy to use: the device has to be usable for people who do not have computer knowledge. This means that the graphical interface of the operating system and programs has to be simple and self explaining (through use of icons, text and others).
- Programs have to be fast: this is by far the most important requirement for users of a mobile device. Most users use their mobile device for a short period of time many times a day. When they want to save for instance a phone number in the middle of a conversation they won't appreciate that the device takes several seconds to boot and several more to open the requested application. This is why a Palm device is immediately ready to use when powered on and PalmOS programs are extremely fast.
- Easy user input: a mobile device is useless if it doesn't give the user a fast and easy way to input data. A keyboard is the fastest way but would make the device a lot bigger and heavier. The Palm® graffiti writing recognition method and its simple alphabet is easily learned and provides a data input speed almost equivalent to normal writing.
- Long battery life: having to replace batteries too often can lead to a situation where you need the device but forget to replace the batteries in time. That's why Palm made it a requirement to give users a battery life of at least 2 weeks of normal use.

Palm implemented all these requirements and more into their palmtop computers. This enables users to organize and input their data much faster than, for instance, with an agenda. This, together with the low price for the devices and a successful marketing strategy made the Palm Pilot devices tremendously popular.

2.2 Differences between mobile and fixed platforms

What are the main differences between normal PC's and palmtop PC's?
This question is quite important because it can point out certain benefits of mobile agents on mobile devices which do not apply on a normal PC. These possible advantages will be discussed in chapter 3.

Mobile platform (Palm pilot) [3]	Fixed platform (PC)
Limited processing power because of space and energy consumption constraints (a typical modern PalmOS processor runs at 33 Mhz)	Powerful processor (over 1 Ghz) with lots of processing power
Limited memory (2 $\rightarrow$ 8 Mb on most Palm devices) used for both data storage and work memory	Large memories (several Gb for storage and several hundreds of Mb work memory)
Limited network speeds (slow processor, some network hardware is missing due to space constraints)	Fast network speeds (100 Mbit and beyond)
Small screen with small resolution (160 x 160 pixels)	Large screens with huge resolutions (1024 x 768 pixels and beyond)
Compact, light & portable	Fixed (difficult to transport) & heavy
Strong & designed to withstand shocks	Vulnerable to shocks, easily damaged in transport
Immediately ready to use when powered on	Requires a (long) time to boot before the system is ready
Is frequently turned on and off	Unfrequently turned and and off
Input through stylus taps and graffiti (not suitable for long textual inputs)	Input through mouse and keyboard (suitable for lots of text)
Simple user interface due to all above constraints	Very sophisticated user interface with lots of functionality
Bad program compatibility between mobile devices of different brands (due to different hardware and operating system)	Reasonably good program compatibility.

Table 1: The differences between a PalmOS palmtop device and a standard PC.

As you can see, most differences are caused by the compactness and energy consumption restraints of the mobile device. A palmtop PC is highly battery, size and weight optimized which causes a tremendous sacrifice in processing power, memory size, functionality and general speed.

2.3 The future of mobile platforms

Mobile platforms may be very popular, but will it remain that way? By analyzing the past and the current situation, we can conclude that the mobile revolution has just started. First computers became mobile in the form of laptops. In 1996 the first Palm Pilot made it to the market and soon many other companies followed in an attempt to make the PC fully mobile. Then the GSM revolution started. First a GSM offered no more features than an ordinary phone, but soon more functionality was added. SMS (ability to send short messages), WAP (internet on GSM), games, ... extended the abilities of GSM. Although the mobile PC market and GSM is currently separated, experts believe they will fuse together. Already now there are GSM/palmtop devices which offer the best of both worlds: telecommunication (phone, email, chat, sms), multimedia (audio, video, games), work (text and spreadsheet processors), etc. New GSM evolutions like gprs, umts and I-mode will turn the GSM into a full grown multimedia device.[4]
Palm and other palmtop producers also follow this trend and are creating devices with built-in GSM/bluetooth functionality and expansions to upgrade their existing models. The GSM part will greatly expand the internet and communication abilities of the palmtop computer while the palmtop computer will give all its advantages (work, multimedia, ...) to the GSM, thus creating a better device. If this fusion happens, the end product that will evolve from it is anyone's guess. What is certain now is that the revolution of mobile PC's is just starting. This thesis and the research it is based upon (mobile agents) is optimized for this next generation mobile devices. This is because mobile agents can offer lots of advantages but they need a (fast) mobile network (like future GSM's can offer) for that.

Recent articles state however that Palm is getting behind in the hard battle between PalmOS and PocketPC. Where Palm still has a slow 33 Mhz processor that is unable to do multimedia like video and sound (mp3), PocketPC offers processor speeds of 400 Mhz now (by using the new StrongARM intel processor). The memory of PocketPC's is now 64 Mb while the best Palm device is stuck at 16 Mb. Just recently Ati introduced a new graphic chip that will offer PocketPC advanced 2D/3D and video decoding capabilities. Palm still has some advantages. Palm programs still run smoother than most Windows CE programs. The new Windows CE 2002 based on the windows 2000 kernel may change this situation and allow programs on PocketPC to run as smooth as on a Palm device.
Just recently Palm announced that its new PalmOS will support new high speed processors. This will bring Palm back in the fight.

No matter who wins, mobile device will continue to exist and evolve.

3 Mobile agents

In this chapter I will explain what mobile agents are and what kinds of mobility exist. I will also explain the working of our mobile agent platform borg and the 2 types of mobility it supports: weak mobility, that allows the agent to be moved when idle, and strong mobility that allows an agent that is busy doing something to be moved. Once this is explained I will give a short introduction to programming agents in borg.

The second part of this chapter deals with the benefits and disadvantages that mobile agents have on fixed and mobile machines. These advantages become clearer when we compare our mobile agent solution with two other systems to create distributed applications: Java virtual machine and the client/server protocol.

3.1 What are mobile agents?

The concept of agents is fairly new and is best known in AI (Artificial Intelligence) research groups. [5] The agents used in this thesis however have nothing to do with their AI colleagues. Agents are pieces of software written in a high level programming language. These agents can have varying intelligence (from simple text copying to very advanced pattern recognition and intelligence). Different agents can work together on one platform. A platform is an environment on a machine that is capable of supporting and running these agents. A platform also offers communication capabilities between the agents, users and also with other platforms. [5]

The platform here is borg [6], a virtual machine which provides a platform for agents on the machine on which it runs. It also provides a programming environment and thus the ability to program new agents. Agents in borg are little pieces of independent software. These agents run on the borg agent platform that allows them to communicate with other agents or the outside world (user, other system, application, data, etc).
An agent is called mobile if it has the ability to move freely from one platform to another without losing its operability. This is only possible if the platform supports this (borg does), the different PC's are connected to a network and when they all provide a platform compatible with the agents who are mobile. A program in borg is made by making different agents to do the different jobs of your program. It is like OO (Object Oriented) programming with the difference that agents are fully independent and more sophisticated than objects. It is not required for a borg agent program that all agents (if there is more than one agent) run on the same machine [7]. It is perfectly possible that the different agents from one program are all on different machines. It is even possible for agents to move from one machine to another while the program is running. It is this advanced functionality and mobility of agents that can offer many benefits.

Agents in borg offer two different types of mobility:

1. Weak mobility: this type of mobility allows the agents to be moved from one machine to the other machine while they are idle. An example :

   -We have a simple application consisting of a GUI agent to provide the user interface and a calculating agent who performs the calculations in the program. As long as the calculating agent is not calculating something, it can be moved to another platform/machine using weak mobility.

   This mobility moves the agent data from one platform to another using the platform's (borg) router. A special nameserver on the network is informed about the move and will forwards all calls and messages toward the agent to its new location.

2. Strong mobility: this type of mobility is identical to weak mobility with the extra that it can also move agents that are calculating (not idle). This means that if the calculating agent in the above example is currently calculating it can still be moved. The advantages of this type of mobility will become clear in the next chapters.

   Strong mobility moves the agent in exactly the same way as the weak mobility. In this case however the agent is calculating and has a certain data stack (borg works with stacks). In order to allow the calculation to continue on the destination platform this entire stack has to be transferred to the the remote machine as well.

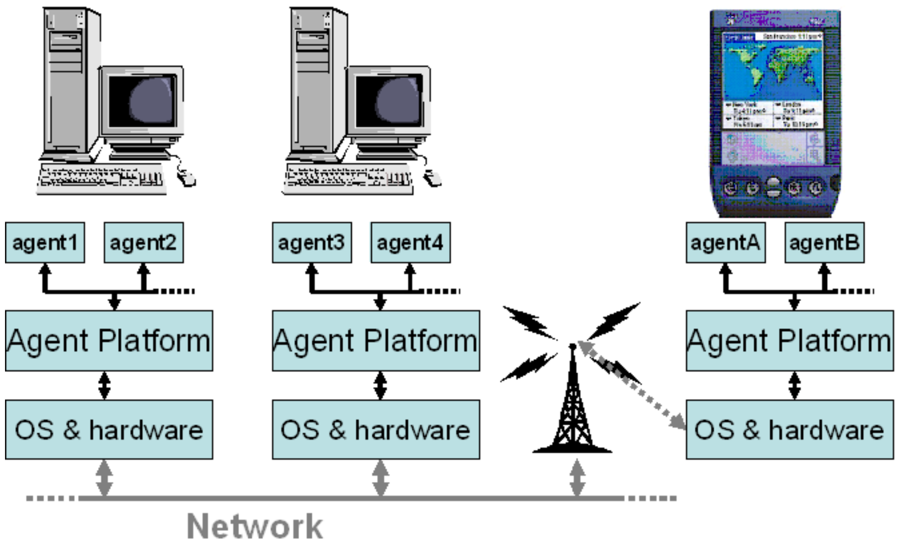


Figure 2: A mobile agent network

3.2 Borg, a mobile agent platform [6]

Borg is a virtual machine. It is built on the pico virtual machine developed by Theo D'Hondt. [8] On top of this virtual machine borg has the needed features to program, support and run agents. It also has the necessary features to allow the agents to communicate with the outside world (through network, users, ...). The borg agent platform supports mobile agents; which means that it has the needed extras to support this mobile behavior of its agents:

- Network capability to allow the agents to move from one platform to another.
- Communication: between agents on the same platform, between agents on different platforms (through the network), with the user operating the machine the platform runs on, possible other platforms/systems, ...
- Programming environment where you can program new agents and run them on the platform running in the background. This can be in a textual environment or in a full GUI (graphical user interface)
- Compatibility with many different machines and operating systems. This to provide maximum mobility of the mobile agents who can then move to different machines with different functionality. Currently only PC is supported with the linux/unix OS. A version for PalmOS will be available soon.

3.2.1 The parts of borg

Borg is mainly developed in the linux operating system. It consists out of 2 main parts [9]:

1. The borg core: this part is made out of portable code. This means it is easier to adapt the code to work on other OS'es or machines. The borg core contains all the code that implements all borg functionalities:
    - The borg virtual machine: translates the borg agent code to machine code and thus allows the execution of agents.
    - The borg router: Provides borg with TCP/IP functionality so one borg platform can communicate with other borg platforms on other machines. This allows agents to communicate with one another even when they are on different machines. The borg router makes use of a nameserver who keeps track of all the agents current location. If a message is passed to an agent the nameserver will tell on which machine that agent currently resides so the router knows where to send it to. The borg router has been extended recently to support QoS (Quality of Service). This is a method to give messages passed from one agent to another a certain priority. This priority allows the message to be processed faster by the borg routers if it has a high priority or to be delayed in favor of other messages when it has a low priority.

- Borg native sockets: these allow borg agents to communicate with other machines not running a borg platform. This communication is done through TCP/IP sockets and allows agents to communicate with other applications and machines like mail servers, web servers, ftp, etc.

2. The borg UI: This is non portable code which means it is very platform specific. The best UI for borg is supported under linux using KDE 1. This UI provides a program environment where new agents can be created and an interface to the agent platform that allows the user to communicate with the borg agents. In linux this UI is a graphical text editor window where you can edit, run and communicate with agents.

3.2.2 Programming in borg

Writing agents in borg is very simple. You just start typing the code in the borg GUI window where it can be executed as well. A borg agent is written in the pico program language. [8] Pico is a high level programming language that is easily learnt by any programmer. The agents are developed like objects in an OO (Object Oriented) program. You separate your program in its different functionalities and program a different agent for each one. This means an agent's structure is much like a class used in OO programming.

```
{
     {
     aFunction(arguments):: {
        code
     };

     anotherFunction(arguments):: {
        morecode
     }
}
```

But agents are not just classes which means they offer some extra functionality. This extra functionality is their ability to pass messages to each other. These messages can be sent to themselves (a procedure call using a message) or to another agent. Agents provide all the extra functionality to enable this message sending. Some of these functions are naming (each agent needs a unique name), getting that name, ...

```
{
\\Agent one sends a message
    sendMessage(msg,agentref):: {
        agentref->acceptMessage(msg,agentname)
    };
    messageOK(agentref,msg):: {
        display(text(agentref) + " received message: " + msg)
    }
}
```

```
{
\\agent two accepts a message
    acceptMessage(msg,agent):: {
        doSomethingWithIt(msg);
        agentref->messageOK(agentname,msg)
        }
}
```

Above code shows two simple agents. Agent one has a procedure to send a message msg to another agent with reference (name) agentref. This agentref is the unique name of the agent we want to send a message to. This name is used as a reference to the agent and used by borg to route the message successfully. A message is sent by calling a procedure on the receiving agent. The procedure here calls acceptMessage and takes a message and an agent reference agentref of the sender as arguments. Each agent keeps its own reference in the variable agentname. Agent two receives the message, processes it and uses the reference of agent one to send a receive confirmation to it. The above code allows 2 agents to communicate with one another in a very easy way. The programmer does not need to know on which machine both agents are.
A message passed from one agent to another is stored in a queue. This is necessary because an agent can only do one thing at a time. Thus it can not process a new message while it is calculating or still processing another message. Messages can also be used to do procedure calls in the agent itself. This and the queuing property of the messages allow the creation of some powerful tools that allow other messages to be processed while the agent is busy:

```
{
continue_process: true; processdata: 0;

processLoop(arguments):: {
    processCode;
    ...;
    if(continue_process,
        agentname->processLoop(arguments))
    };

stopProcess():: {
    continue_process:= false
    };

statusProcess():: {
    displayCurrentProcessData
    ...
    }
}
```

The above program demonstrates a simple agent that can still process messages while it is busy. A process is busy and is looping (like most processes). The next execution of the procedure in the loop is not done by a local call to the procedure but by sending a message to itself. This message is stored in the message queue. If other messages arrive they will be stored in the message queue as well and will be placed before a next loop execution message. Thus the message will be handled first and the loop will continue once all messages before the next loop call message are processed. The above agent demonstrates the easiness of implementing such a system. It demonstrates a looping process in procedure processLoop that is looping by sending a message call to itself. Two other procedures can stop the processloop or ask current information about the process while the agent is working without aborting the current process.

3.3 Benefits & disadvantages of mobile agents

3.3.1 General benefits of mobile agents

Mobile agents can move between platforms while the program is running. This is a very interesting feature of which we will examine possible benefits. The moving itself can give programmers the ability to implement special strategies which can solve certain problems. You can program the agents moving behavior to implement smart systems. These smart systems are programs with intelligence that can help themselves to run smoother in certain conditions.
A few examples:

- Sometimes a machine that is running an agent platform can be heavily loaded. This can be because too many agents are currently running on that machine or the machine is doing some other very processor intensive tasks. Eventually the machine can become loaded to the extent that the working of the agents is slowed down. This will not only slow down the agents currently running on that machine but also all other agents that depend on them. In general the entire program's performance may suffer greatly because of one agent running on an "overloaded" machine. This is a logical consequence of distributed programming. When one part of such a system is slow and other parts depend on it, the entire program can be slowed down. This effect is explained in the cases test environment chapter. In such a case where the current machine is too loaded to allow agents to run at full speed; the agents can move to another less occupied machine. By doing this the program will keep running smoothly.

- Another advantage is in the area of distributed/parallel programming. When a programmer wants to do very processor intensive tasks with his agents, he creates several of the agents who will do the calculations and make them spread out over different machines. The agents can be programmed in such a way that they will pick the powerful machines on the network which are currently not heavily loaded. This will speed up the calculation work and will not hinder other users on the network. This way mobile agents can become a powerful parallel processing tool. [10]

- Mobile agents can also be used to create programs which can recover themselves in case of a partial failure in the network. When the agents are running on different machines they are vulnerable. This is proved in the next page of this thesis.

  One or more machines running your agents can crash or their connection with the other machines can be severed by a network failure. In this case your program will most likely stop functioning and data can be lost. You can program agents which will duplicate the agents of your program and recreate killed (crashed) agents in case of a failure. This way there is always at least one of each agent type operational so the program will not suffer a failure. [13]

3.3.2 General disadvantages of mobile agents

While mobile agents obviously have some general advantages they also have several disadvantages as well:

- As we said in the previous section, mobile agents are more vulnerable. Agent based programs are distributed programs so they inherit some of the disadvantages of distributed systems. If agents run on several machines over a network the chance that the program fails due to either a network problem or a machine crash increases. The possibility of a failure becomes bigger when more agents of the program are running on different machines:
  - Let be θ the total chance for a crash and α the chance a certain machine crashes.
  - On a single machine $\theta = \alpha$.
  - When the program runs on n machines $\theta = \sum_{i=1}^{n} \alpha$ + chance of network crash.
  - This means that the more machines are running our agents the more likely our program will fail due to a failure in the network or due to a machine crash. This can be solved by a partial failure recovery system. Such systems are being researched all over the world and also by a fellow student. [13]
- Another disadvantage is that program performance becomes dependant of the speed of the other platforms the agents are working on. This causes program response speeds to differ. If one or more agents of a program run on a slow or heavily loaded machine they will slow down other agents depending on them and your program itself. A small example:

  -We run a distributed system and the program and its data are divided over several agents and machines. When an agent needs information from another agent it will send a message and wait for a response. This other agent may be running on a slow system and is still calculating/processing a previous message. If the agent is on a too slow platform the messages sent to it will be accumulated in its message queue. Other agents will stop working because they expect messages with crucial data from the agent on the slow machine. In this case the entire program will wait for the slow agent to finish working and is slowed down considerably.

  This problem is also explained and illustrated in the cases test environment chapter further on in the thesis.

  This situation can be solved by making agents intelligent enough to move to another machine when their current one is under heavy demand or is just too slow to do the requested tasks within a given time. This intelligence can be something that is measuring the length of the message queue. If it becomes too long this indicates the agent cannot process incoming messages fast enough and needs to be moved to a faster machine.

- A last major disadvantage is that program response time and speed become more dependant of network speed than available processing power. This is best seen in the route planner case when the map and routeplanner-agent are on different machines. The network can not handle the intense network traffic between the two agents and the program slows down considerably. This problem is also explained in the cases test environment chapter further on in this thesis.
  This means the network will become the bottleneck of your program. This will certainly be true when the network or part of it is heavily loaded or when one or more agents run on platforms with a slow network connection (modem, GSM). Only one or more agents have to be on a machine with a slow connection in order to slow down other agents dependant on them and thus the entire program.

  When we again take a program made of several agents running on different machines in a network, the program and data are again divided over different machines and the agents continuously send messages to each other with data to be processed. If the network is very slow in response time the problem is obvious. A message between agents can then take several seconds to arrive. If the sender is waiting for an immediate response this will slow down both agents. If the network bandwidth is too low this might give complications when messages with a lot of data are sent. If an agent who processes such large messages is on a slow connection and gets large messages from all the other agents, it will not be able to process them in time. This is because it can take several seconds or even minutes before a message is transferred to the agent.

  This can of course again be solved by giving the agents sufficient intelligence to move to machines with a fast connection. They can also move to another network area when the network becomes loaded. However there is a more suitable solution to this problem here: QoS (Quality Of Service) which allows you to set a priority to packets sent over the network. Agents could then be given a higher priority than other network operations so the communication between them runs smoothly even on a loaded network. [12] [16]

3.3.3 Benefits of mobile agents on mobile platforms

This thesis is about mobile agents on mobile platforms. We have seen that mobile systems are somewhat like their bigger brother the PC. There are, however, certain differences between the two (see table 1.)
By studying these differences more closely we can find two extra advantages :

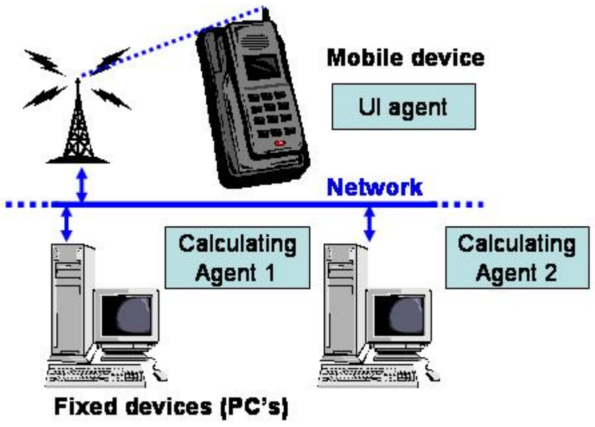


Figure 3: Advantages of mobile agents on mobile platforms: offloading processor & memory intensive tasks

1. Processing power & memory constraints: mobile platforms lack the processing power and memory sizes of a PC. This greatly hinders the complexity and functionality of mobile platform programs. However, we can use mobile agents to provide the mobile platform with more processing power and memory:

   -The agents mobility can be used to offload some of the processor and memory intensive tasks of a program to other (more powerful) platforms (figure 3). Suppose our mobile device has a (wireless) mobile connection and supports mobile agents. Our program consists of a UI (user interface) agent and other processing agents. It is then possible that only the UI agent and some other required agents run on the mobile platform and all other agents (which do the calculations and thus need processing power and memory) migrate to other more powerful platforms on the network. This would take a significant load from the CPU and memory of the mobile device and speed up programs. This migration can be performed by using the weak or strong mobility property of the agents.

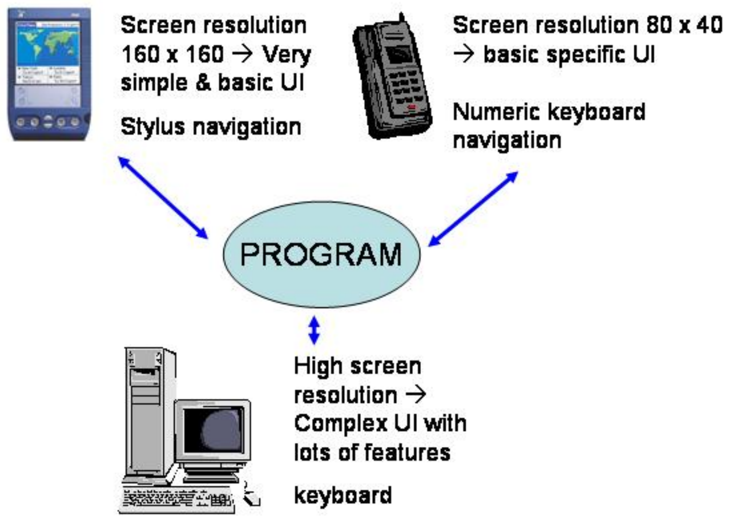

Screen resolution 160 x 160 → Very simple & basic UI
Stylus navigation
Screen resolution 80 x 40 → basic specific UI
Numeric keyboard navigation
PROGRAM
High screen resolution → Complex UI with lots of features
keyboard

Figure 4: Advantages of mobile agents on mobile platforms: mobile platform diversity.

2. Mobile platform diversity: another property of mobile devices compared to PC's is that they all have radically different specifications (Figure 5). Mobile platforms are very diverse: they can go from the Palm Pilots and Visor like devices discussed in this thesis to GSM, Windows CE PocketPC's, ... Take GSM's for instance; those are all mobile devices with different specifications (type/power of processor, screensize&type,data input,memory, OS, programming environment,...) yet they offer all the same functionality (calling, SMS, WAP, games, ...). With current techniques, a program has to be especially designed for a specific type of GSM although it always has the same functionality. It would be much easier to make a program for such devices if GSM's supported mobile agents. We could then again split the program in several agents. The processing agents which offer the program's functionality could then remain constant while only the UI agent has to be GSM specific (which uses the GSM's screen and data input to its best). That way a programmer only has to create different UI agents for each type of GSM while the rest of the agents do not need any changes. Figure 4 demonstrates this diversity. We have a visor with a 160 x 160 sized color screen. It has a stylus for data input which is good to navigate through programs but not to enter a lot of data. The GSM has an even smaller screen and has only a numeric keypad to insert data. The PC has a large screen and a mouse to navigate through programs while allowing the user to enter large quantities of data through a keyboard. Making a program that has to work on all these devices would be much easier with agents; the central program remains the same while only the agents needed for the UI are device specific.

3.3.4 Disadvantages of mobile agents on mobile platforms

Agents on mobile platforms suffer from the same disadvantages as on other machines. The programs on a mobile platform, with its agents spread over different machines, are more vulnerable for a system/network failure. The program response speed will also be dependant on network and platform loads.
The specification of the mobile device and its general network speed create a new disadvantage:

Mobile platforms are best connected through the network with a wireless solution so that their mobility is preserved. For GSM's this is straightforward, for other mobile platforms diverse network solutions exist. The main disadvantage lies in the fact that those wireless networks are slow. Most advanced GSM's have a network speed of 56000 kbit/sec (4Kb/sec) using the new gprs protocol. Only few GSM's already use this protocol and without it the network speed is limited to only 9600 bit/sec (1 Kb/sec). Several companies now offer wireless network solutions for Palm, Visor and pocket PC. These networks allow theoretical transfer speed up to 10 Mbit/sec which is very fast. While this speed is reached on fast mobile devices like notebooks, this is not the case with palmtops. The devices themselves are too slow to make any use of that speed. This is because the network hardware is very simplistic (due to space & energy constraints) and most of the network processing is done by the central processor. Like mentioned in the introduction to mobile devices and palmtops this processor is not very fast. The fact that the processor spends most of its time running your program, makes that it spends little time processing network operations. So, in general, the network connection is slow (1-50 Kb/sec depending on the current load of the processor) and thus inevitably becomes the bottleneck. This means that moved agents from a mobile device to another machine require some time, certainly if we use strong mobility and the agents stack has to be send as well. When too many agents remain local the mobile platform becomes the bottleneck because they ask more processing power than the processor in the mobile device can offer. When too many agents are migrating the network will become the bottleneck because it cannot transfer the agents and the messages between them fast enough.

3.4 Mobile agents compared to other solutions

Now it is time to compare our mobile agents with some other solutions. We will examine if the weak/strong mobility property of mobile agents is that unique. Other systems like Java offer weak mobility and client/server which is a well-known, widely used, architecture to make programs communicate over a network. Java and client/server will be closely examined in order to find out if they can offer the same advantages than our mobile agents.

3.4.1 Java applets [14]

Java is an object oriented programming language. The big strength of Java is that users can download a Java virtual machine and are then able to run Java applications and applets. Applets are small programs you can download from the internet. They are automatically executed on your computer using the Java virtual machine. Java applets are smaller than a compiled program; this is because only Java bytecode (Java classes) are downloaded which are then executed by the Java virtual machine. This Java virtual machine contains all the libraries necessary to run any application/applet. This way the usually large libraries do not have to be sent along which results in a smaller and faster download. This way even very complex programs (like games, bank managing systems, ...) can be downloaded very fast.
Java is an object oriented programming language. Java allows the creation of two types of programs:

1. Applets: these are small programs created to be integrated into a webpage. That way they can offer extra functionality to webpages like:

    - Enhanced multimedia: since Java can interface with OpenGL and DirectX it is able to enrich webpages with powerful 3D graphics and sounds like 3D games, 3D interfaces, ... It also allows the integration of other features like special buttons (with nice animation effects and sound), simple games, etc.
    - Enhanced communication features: like chat boxes and web cam interfaces on web pages.
    - Database support: Java has also SQL and other popular database interface possibilities which allow applets to interface with an online database. This allows users to access data in a database via a website.
    - Search engines: a Java based search engine to search web pages, databases, ...
    - Secure login: Java applets allow secure login and data transfers on websites. Java applets are used to manage bank accounts and other private data that needs the high security a web browser can't provide.

    Java applets are compact in size because they need to be downloaded in an acceptable time even on slow connection. This compactness means that programmers usually limit Java applets in functionality.

2. Applications: Applications are executed locally and do not have to be downloaded like applets. This means they can be larger and are thus able to offer greater functionality.

Java programs which I will refer to as applets in the rest of this text can vary from 3D games (interfacing with DirectX or OpenGL) to online database management systems. In order to have web based features, Java programs also have communication capabilities.

Couldn't these Java applications, which are already much more common than agents have the same benefits? Both systems use a virtual machine which means those benefits are the same (platform independence). Java is an object oriented programming language which most programmers know well so they can easily start programming. Mobile agents however are a new approach and are not as widely known as OO languages by programmers. Although agents may have some advantages over object oriented languages they are not so well-known where OO languages offer tons of programming strategies, design patterns and documentation. [15]

The main difference between Java applications and mobile agents is that Java applications have no strong mobility. But is this strong mobility really necessary to obtain the same advantages? Let's examine this by looking if the Java applications can implement the advantages of our mobile agents:

- Java applications can be used for the same distributed processing as agents can be. It is very common on the internet nowadays for computer users to download small applets who then start some calculations when the machine is not busy (screensaver or so). Java applets however can not move to another machine when a current one is too heavily loaded. They can however be moved to another machine when they are not executing. In such applications, who allow users to share processing power, it doesn't really matter whether the machine is loaded or not, it is more a question of having as much machines calculating as possible. In this field Java applets can be as useful as agents.

- Since Java applications do not have strong mobility they cannot move to another machine when it is heavily loaded. This is an advantage of mobile agents that is very difficult to implement using Java applications/applets.

The general advantages of mobile agents seem to be unique. Java applets fail to offer the very useful advantage of distributing processing power by moving parts of the program to other machines when current ones are too loaded. How do Java applets and mobile agents compete when we examine the advantages on mobile platforms?:

- You can make distributed programs with Java which means you can offload processing power from one machine by distributing the calculations over several machines. You can follow the strategy of our agent based systems and split a program in its functionalities. You can then implement these functionalities in different applets and provide them with the necessary means to communicate properly. The applets can then be distributed to special servers where they can run. One applet remains on the host machine and provides a UI for the user. In this manner processing power and memory are offloaded to other machines. However, the fact that applets aren't mobile gives this approach many disadvantages:
    - It is difficult to distribute the applets. Agents can move from machine to machine as long as the machines have a virtual machine that supports them. The Java virtual machine also provides the communication between the agents wherever they are. Java applets can be provided with instructions so they will move to a Java applet server with the best specs, but implementing these instructions is very difficult. [15]
    - Java applets are unable to move to other platforms once they are running, thus they are unable to respond to heavy server or network loads. This will seriously jeopardize program performance as only one applet of the distributed system has to be on a slow machine and/or with a slow network connection to slow down the entire program.
- Java applets are perfectly able to handle platform diversity. As with mobile agents we can put the main functionality of a program on powerful servers while the mobile machines (GSM, Palm, ...) are only provided with a unique UI especially designed to work better on the specific mobile machine.
- Java applets can also provide a more unified programming environment. All mobile machines only need to have the Java virtual machine installed and whatever program created in whatever Java programming environment will run on it.

As we can see, Java applications have difficulties in implementing the advantages of mobile agents; this is because of two reasons:

1. The platform of mobile agents automatically provides the communication between agents wherever the mobile agents are currently running. Although Java uses a very simple interface (Java RMI) as well, interactions between two Java programs have to be programmed with more care. [9] [6]
2. Agents in a mobile agent based system can freely move from one machine to another while the program is running. Agents are also capable of moving when they are busy performing a certain task (strong mobility). It is this lack in strong mobility that disables Java applets/applications to offer the same advantages than our mobile agents.

We can conclude that Java programs may have certain advantages on mobile platforms. However, the advantages they offer are in a different area than our mobile agents. The lack of strong mobility can not be compensated by the weak mobility provided by Java. Java programs are thus unable to reproduce the same advantages than mobile agents.

3.4.2 Client server systems

Compared to the other solutions, client/server is not a programming environment nor a programming language but a type of protocol. The client/server protocol is used by many programs which have to communicate certain data over a network. Web browsers, internet games, chat programs are all client/server based. In the client/server protocol you have two different types of communicating programs:

- The server: the server is a program running on a powerful machine and offers certain services to other users. It contains most data and has client management systems. An example of such a server is for instance a 3D gaming server. The server allows multiple clients to log on. The clients all run a 3D game that is constantly sending data (position & direction, player, type of weapon, current action, ...) to the server. The server then forwards all data from one client to all other clients so that they are informed of the other player's location and actions. That way, the clients can play against foes controlled by other people in the 3D game.
- The client: runs the program of the user and is connected to a server for certain services. In our 3D gaming example the client connects to the 3D server in order to get the data from the other players so the user can play with/against them.

Almost every known programming language gives programmers the ability to create client/server based applications. Thus most programmers know how to implement a client/server based application. However, these systems have the same drawback as Java programs. They are not mobile and are incapable to reproduce the same advantages as our mobile agents:

- With client server systems you can easily do distributed calculations. In this case the server is sending packets to the connected clients containing calculations. When the client is done, it sends the result back to the server which then processes the result and sends a new set of calculations to the client. Java based programs for instance use this client/server system for communication between different programs.
- Offloading calculations from workstations with client server based systems is also commonly used.

  Example: a database management program is running on the server. The client sends database operations to the server which will perform these operations on the database and return the result to the client. Old database systems made the server do all the work. If the client needs two tables to be joined, the server would do this and send the complete end result to the user. Nowadays the server only returns database data and checks its consistency. The client then does the user requested operations on the received data.

While in this example calculation power is divided over clients and server it is very difficult to offload processing power and memory usage of a simple program using this approach. This would require a server doing the processing part of the program and a client on the users machine communicating with it. The server thus has to offer all services for all programs a user can run. This is very complex and since client/server systems do not have any form of mobility it will quickly result in overloaded servers and networks.

This problem can be partly solved by allowing the client to upload the tasks it wants to do remote to the server. This however will result in higher network traffic since the entire server program has to be uploaded to the server. You could implement a platform on the server which can execute certain program code uploaded to it, but this would be complex. Mobile agents and Java offer such a system by default.

- Client/server systems can handle the diversity in the mobile market. Since the programs functionality can reside on the server and the client is the only part that has to be device specific. However, since client/server based systems do not use a virtual machine, the client has to be especially programmed in a language environment for a specific (mobile) device. This causes the development of the client to be much more complex compared to agent or Java based approaches.
- Client server based systems also do not provide a general programming environment since they are not a programming language or virtual machine, but a protocol that can be used in many different programming languages.

Client/server based systems are easy to implement and almost all programming environments (also mobile programming environments) support them. They can be used to offload simple distributed calculations. However splitting an ordinary program and distributing it over the net is far more complex than in the case of Java programs or mobile agents. The lack of any true form of mobility makes that distributed systems based on the client/server system cannot handle situations where a server overloads.

We can conclude that client/server based applications offer almost none of the advantages we find with our mobile agents or even Java programs.

3.5 Conclusion: is mobility so important?

It looks like none of the above solutions can match the advantages provided by mobile agents. Java applets and the client/server protocol are very powerful tools too, but their lack of strong or any form of mobility and the bad communication support between client and server on different machines impeache them to match with mobile agents. Mobile agents are truly programmed to be mobile and communication between agents is as straightforward as communication between different objects in an ordinary program.
Some readers may have doubted the usefulness of mobile agents. But, as we can see, this property of being able to move from one machine to another ,whether the agent is busy or not, has some unique advantages:

- The ability to offload processor intensive parts of your program to another platform.
- Ensure fast execution of a program by moving agents to a machine with a faster processor/network connection.
- The ability of agents to use their mobility to divide processing power.

If we look at our ultimate goal, making programs for mobile devices run faster, some readers might think all above discussed approaches will work as well without mobile agents:

- For client/server based applications and Java applets there will have to be powerful servers which will execute the processor intensive tasks. This might work since mobile programs are very simple and basic compared to PC programs and they will never consume much memory and processing power. While this might work now, we claim this will not be so straightforward in the future:

  -Five years ago mobile devices were rare. GSM's were just invented and mobile devices were still slow, heavy and expensive. Within just 5 years, we had a major growth of mobile devices. Within a few years from now, half the computers in the world can be mobile. Early GSM's had very limited functionality, but recent models have a decent graphical screen with a simple processor. This means the GSM is becoming a mobile computer as well. If we take all GSM's as mobile devices, everybody will agree that there are already now more mobile than fixed PC's in the world. Whether or not current GSM's are true mobile computers can be a topic of discussion right now. The next generation of GSM's which will support gprs, umts and I-mode [17] will not need this discussion anymore. Those devices which are already in use in Japan right now, have internet, video and audio playback and many more multimedia features. These devices are undoubtedly true mobile computers.

Placing the servers and networks that can handle the distributed programs of that many mobile devices will be a costly operation. This is because the entire network infrastructure with many powerful computers is already available: the internet. Each processor of a computer connected to the internet is, on average, less than 10% used by the average user. This means that all the computers on the web offer a lot of available processing power. In the future users may allow their machine to become a platform (for a small fee) for mobile agents. This will allow their machines to support agents and process certain tasks while it is idle. The development of such systems is done already today.

- For simple distributed calculation tasks Java programs or client/server systems may still be adequate to implement a system like described above. For more complex programs however we require mobile agents: users will not appreciate it if they can't run a processor intensive program because in the background, a process hungry Java or client/server application is running and can't be stopped until it has finished with whatever it is doing. So in this case an agent based distributed system would be better. An intelligent agent that performs calculations on a users machine will see the increased load on the CPU and decide to move to another platform. If it has migrated, the user's PC is free for whatever he/she wants to do. Once the agent is on his new machine, it continues its calculations like it had never been interrupted.

- The mobile property of agents can also be useful in ways that cannot possibly be implemented with a Java or client/server program. Let's look at the following example:

  Example: a land surveyor wants to measure certain data of a piece of land. He has a GPS and measures his position at each corner of the land. These positions are entered in his mobile device. This mobile device triangulates all given positions in order to calculate the area of the land. The program itself consists of two or more agents. One or more that accept user input and display the information on the screen and another that does the triangulation and surface area calculations.
  When the land surveyor starts, little data is entered and the data is quickly processed. After a while, the process starts to slow down. At a certain point the calculations start to take too much time and the agents which do the calculations decide to move to a faster platform (in between or during calculations). This way the speed of the program is maintained.

> This is a situation that Java programs or client/server systems can not implement. Either the remote calculation applet/server is immediately started (even though it might never be necessary to use it) or the program is run only locally. There is no way a part of the program is going to copy itself during calculations from one place to another when using Java programs or client/server systems.
>
> It is this property that makes mobile agents so fit for mobile applications as we will discuss further on in this thesis.

We can already conclude that the mobility of agents definitely has unique benefits. The cases will further examine these possible advantages and disadvantages.

4 Case studies

The case studies will be used to see if the weak/strong mobility property of agents in agent based programs offer real benefits on mobile platforms.
In order to find this out we will examine four different cases, each with a different objective:

1. Mail agent case: this is an agent based mail management system. A server agent gets mails from a mailserver and stores each mail in a mailagent. So each mail is an agent. Downloading a mail is done by moving the specified mailagent from the remote mailserveragent to the local machine. The mailserveragent provides functions to search and manage the mails in the mailagents. This case demonstrates the easiness of making a distributed system with agents. It also shows the easiness of implementing weak mobility using agents.

2. Meeting planner: this case is developed by Pieter Verheyden.[10] It is a case that allows users to plan a meeting. Each user has his own meeting planner agent and all these agents communicate with one another to come to a successful meeting. A meeting is successful if the place where the user wants the meeting is available at the given date and time and when there are enough participants.
This case is a true peer-to-peer system. This case is used to find out if a distributed system runs better/worse when a slow machine like a palmtop PC joins the distributed community.

3. Route planner: the route planner demonstrates the use of strong mobility. In this case not only processor power but also memory use is important. A route planner needs a large map thus there will not be much free memory. On top of that the route planning calculations take a lot of work memory. To prevent the palmtop from running out of memory the calculating agent is moved when free memory is low. This case demonstrates the ease of implementing and using strong mobility in a program and will test if using this property has impact on program performance.

4. Advanced route planner: this case is basically the same as the above route planner but has one big difference. While the route planner above demonstrates the use and possible advantages of strong mobility, this route planner will go even further. It will try to find out what type of mobility is best in a certain situation. When is strong mobility the best choice or when is weak mobility the best choice. Weak mobility may be preferable on slow network connection where the moving of the agent and its stack would take too much time. In the case of weak mobility calculations will be aborted and data already calculated will be lost. Strong mobility will move the agent slower, but once it is moved the calculations continue from where they where interrupted.

Eventually we will try to make an agent that will choose how to move a calculating agent in a certain situation, the jump agent. This jump agent has a high intelligence and tries to get a maximum performance gain by choosing between different types of mobility to move the calculating agent.
All the case code can be found in the borg repository. [9]

4.1 The testing platform

There are several important factors when testing a distributed agent program in terms of performance. Next chapter will explain what those factors are and why they are important.
The second part of this chapter describes our "test environment" for our cases. This test environment simulates different conditions so we can test our cases in different situations and thus come to a better conclusion.
The last part explains the types of mobility that will be used in the different cases.

4.1.1 The test factors

There are two factors that have a big influence on the performance of a distributed program: network and processor speed.
Like any program a distributed program is largely dependant on the processing power available. Unfortunately processing power distribution is very important too in terms of performance of distributed programs. A simple example will illustrate this:

We have three agents, all three agents are on a different machine and are working on a distributed problem. In order not to complicate things we assume that the three machines are connected to a fast network with no noticeable delays. They all start the same calculations and when they have finished they will send the results to each other. After they have compared the results they will start a new calculation. We have two different situations now:

1. All three machines have the same processor speed. Lets assume we have three machines with processing power 100 which means they can perform 100 calculations per second. If the agents have to perform a calculation that requires 300 steps, all will calculate about 3 seconds:

Time	Agent 1 (100)	Agent 2 (100)	Agent 3 (100)
1	start calculation	start calculation	start calculation
2			
3	Done	Done	Done
4	Comparing	Comparing	Comparing
5	start calculation	start calculation	start calculation
...	...	...	...

Tab 2: Distributed system with all agents on a system with equal processing power.

We can see that each set of calculations takes 3 seconds for all agents. All agents finish at the same time and no agent has to wait for the results of an other agent.

2. The situation is entirely different when the agents are on machines with different speeds. Lets assume that agents 1 and 2 are on a machine with speed 150 and agent 3 on a slow machine with speed 50. Logical reasoning would tell us that the average speed of this system is $(150 + 150 + 50)/3 = 117$ and thus faster than the average speed of 100 from previous example. So we expect a performance gain of our distributed program since more processing power is available:

Time	Agent 1 (150)	Agent 2 (150)	Agent 3 (50)
1	start calculation	start calculation	start calculation
2	Done	Done	
3	Waiting	Waiting	
4	Waiting	Waiting	
5	Waiting	Waiting	
6	Waiting	Waiting	Done
7	Comparing	Comparing	Comparing
8	start calculation	start calculation	start calculation
...	...	...	...

Tab 3: Distributed system where the machines the agents reside on have different processing speeds.

Unexpected, our distributed system with 50 more total speed performs only half as fast as the previous example system. The table immediately shows what is causing this unexpected bad performance. Agents 1 and 2 with their high processing power finish their calculations in a fast 2 seconds. However, since they have to wait for the result of agent 3 before they can continue they have to wait until agent 3 finishes. Agent 3 with only speed 50 takes 6 seconds to complete the calculations and only then all agents can compare their results and start a new calculation.

This example may be extremely simplified and is really the worst case scenario, but it clearly shows that average processing speed tells nothing about the possible performance of a distributed program. The performance depends greatly on the slowest component in the system. Many will say that this problem can be solved by making your program the way that agents 1 and 2 can do other things while waiting. This will indeed solve the problem, but in a real distributed agent program each agent has a specific functionality and thus sooner or later one or more agents will have to wait for the result of a calculation that can only be performed by an agent on a slow machine. If one agent is on a much slower machine compared to the rest this will cause a negative performance hit, no matter how fast the other machines may be.

A second very important factor when measuring the performance of distributed systems is the speed the different machines can communicate with one another. A slow network connection between two or more machines in a distributed system may cause a bigger performance drop than a slow machine in the system. To illustrate this we can take above example. When the agents stop their calculations, they will send the results to one another. They will compare these results which will take one second and then send a confirmation to start the next calculations. The agents do not compare or start a new calculation before all messages have been received.

We can again distinguish two extremes:

1. All machines in the distributed system can communicate through a fast network with one another. On a fast network it takes only 1 second to send a message from one agent to another.

Time	Agent 1 (100,fast)	Agent 2 (100,fast)	Agent 3 (100,fast)
1	start calculation	start calculation	start calculation
2			
3	Done	Done	Done
4	send result 1	send result 2	send result 3
5	receive result 2,3	receive result 1,3	receive result1,2
6	comparing	comparing	comparing
7	send confirm 1	send confirm 2	send confirm 3
8	receive confirm 2,3	receive confirm 1,3	receive confirm 1,2
9	start calculation	start calculation	start calculation
...	...	...	...

   Tab 4: A distributed system with all agents connected with a fast network.

   Through a fast network no agent has to wait for another and our distributed system can work at full speed taking optimal advantage of the available processing power.

2. Two or more machines in the distributed system have a slow network connection. All machines still have the same processing power. Agent 1 and 2 have a fast network connection that can send any message in 1 second. Agent 3 is unlucky again and his machine has a very slow network connection with agent 1 and 2. It takes 3 seconds to send a message to agent 1 and 2 and it takes 3 seconds to receive a message from agents 1 and 2.

Time	Agent 1 (100,fast)	Agent 2 (100,fast)	Agent 3 (100,slow)
1	start calculation	start calculation	start calculation
2			
3	Done	Done	Done
4	send result 1	send result 2	send result 3
5	receive result 2	receive result 1	
6			
7	receive result 3	receive result 3	receive result 1,2
8	comparing	comparing	comparing
9	send confirm 1	send confirm 2	send confirm 3
10	receive confirm 2	receive confirm 1	
11			
12	receive confirm 3	receive confirm 3	receive confirm 1,2
13	start calculation	start calculation	start calculation
...	...	...	...

Tab 5: A distributed system with a slow network connection between two or more machines.

The slow network connection between agent 3 and the other agents is disastrous for the performance of our mobile system. Agents 1 and 2 quickly receive a message from one another. After that, both agents have to wait until the message from agent 3 is received. In this case the available processing power is partly lost due to the slow network connection.

Above examples illustrates clearly that the performance of a distributed system greatly depends on processing power and network speed. In the case of processing power, distributed systems with a PalmOS device will have problems as seen in table 3 because the PalmOS device is much slower than the other machines in the network.

Palm devices also suffer from a max network speed limitation. The network operations that are performed by the network hardware on a PC are partially done by the processor in mobile devices. This is because this extra hardware cannot be fitted into the small space of a network upgrade module and to limit power requirements. This causes the maximum network speeds on mobile devices to be much lower since the processor of a mobile device is already much slower than its PC counterpart and it has to perform extra operations to perform network operations. On a Palm device who's processor only runs at some 30 Mhz this maximum network speed limitation is very obvious (some 10 - 20 Kbit/sec maximum). On other mobile devices that now come with new 400 Mhz processors this limit is less apparent (maximum transfer speeds of some 100 Kb/sec) but it becomes questionable if we can still call such devices slow. Since we examine our cases on slow mobile devices like a Palm, we will assume that the maximum network bandwidth for such device is 20 Kb/sec.

4.1.2 The test platform

The cases will be tested in several different situations to form a conclusion:

- All machines are fast with a fast network connection
- All machines are fast with a slow network connection
- All machines but one (our Palm device) are fast with a fast network connection between the Palm device and other machines. The network between the other machines is fast.
- All machines but one (our Palm device) are fast with a slow network connection between the Palm device and other machines. The network between the other machines is fast.

The test platform consists of only PC's, connected with a 10 or 100 Mbit network connection. This means our testing environment is the fast-fast case. In order to test the other situations it is necessary to simulate a slow machine and/or network.

Simulating a slow machine can be done by putting wait() procedures in between the normal calculations procedures. The wait() procedure just waits X time before allowing a further piece of code to be executed. This wait() procedure is simply to implement:

```
wait(time) :: {
    i : time;
    display("waiting..." + eoln);
    while(i>0, i:=i-1)
}
```

To simulate a fast machine this wait time is 0 or small. To simulate a slow machine this wait time is set to a higher value.

Simulating different network speeds is a little more complex. A network has two properties that are important to evaluate its speed:

1. Bandwidth: This property is known by most PC users. It indicates the amount of data that can be transferred over the network in a given time. Usually this speed is expressed in Kbit/sec or Mbit/sec where a 8 Kbit = 1 Kbyte and 8 Mbit = 1 Mbyte. Simulating this is very difficult and involves putting a wait() in front of a message send with the wait time proportional to the size of the message to send. Since borg has no memory monitoring this is very difficult to accomplish and thus the wait() uses a guess of the estimated message size. Since the only place where messages of larger than some Kb are sent is in the case of agent moves, this is the only place such delay waits are simulated.
2. Response time: Less known by PC users and well-known by online gamers is the response time of a network. This is the time it takes for a signal to travel from one point in the network to another. This delay depends on three factors :

(a) The number of routers between the two points in the network: a router needs some time to process and forward an incoming packet. When the router is loaded and has to process many packets, our packet can be placed in a queue where it has to wait some time or can even be erased in which case the packet has to be send again.

(b) The network load: when a part of the network is loaded there will be a lot of collisions. In case of a collision the packet is lost and has to be send again.

(c) Less important is the total cable length between the two points in the network. Since signals can only move at 2/3 time the light of speed in cable networks and at the speed of light itself in optical networks it takes a small time for a signal to travel a certain distance over a network. This time can be estimated by the formula *time = length / speed.*

This response time is measured in ms and is usually small when machines are close to one another (on UTP, coax, optical fibre, ... networks). In our testing environment where the machines are connected through only some meters of cable, one router and an average loaded network, these delays will remain small and will not exceed 20 ms. A mobile device that uses a mobile network solution like GSM has a much larger delay. This because the signal has to travel through different types of networks and is processed many times. In this case delays become larger and easily exceed 200 ms. Such a delay is easily simulated by placing a wait() before the message send that will hold the message X time from being send. This has the same result as the delay on a network.

A general wait() to put in front of a message sent to simulate different network response time and bandwidth is:

```
wait(delay,size,bandwidth):: {
    i: (delay + size / bandwidth) * factor;
    display("waiting..." + eoln);
    while(i>0, i:=i-1)
}
```

Delay contains the network delay to simulate, size and bandwidth give the size of the message and the network bandwidth. All this is to simulate the time it takes to send a message over the network. Factor contains a machine dependant constant that multiplies the result from the formula (delay + size / bandwidth). This is necessary because the time it takes to execute 1 loop in wait is not equal with 1 ms.
Testing the program performance itself is done by starting a timer just before the calculation, we like to monitor, starts. When the calculation finishes the time is returned in seconds. Seconds are good for large delays, but they are inadequate for more precise monitoring. Unfortunately borg does not support anything else but seconds.

4.1.3 Types of mobility

The cases will use three different types of mobility. To prevent confusion in further chapters I will explain the different types here:

- Weak mobility: this type of mobility moves an agent who is currently idle to another machine. Only the agent (without its stack) is transferred.
- Strong mobility: moves an agent who is currently calculating to another machine. When the agent is transferred, it will continue the calculations on the remote machine like it was never interrupted. Strong mobility requires the agent and its stack to be transferred.
- Forced weak mobility: this type of mobility is new and will move an agent who is currently calculating using weak mobility. The agent's calculations are aborted, and its stack is emptied. The only data that remains in the agent's stack is the procedure call. By leaving this procedure call in the stack, the agent will automatically restart its calculations once moved. Moving an agent using forced weak mobility only requires to transfer the agent and its small stack to the other machine.

4.2 The mail server system

This case demonstrates the ease of moving agents between machines. It shows that the use of weak mobility using mobile agents is very simple. The case is a simple mail managing program that can download, view, manage and search mails. The big difference with a standard mail manager is that the mails are not stored in some database. In this case every mail is a mail agent. Only the reference to this mail agent is kept in a database.
The mailserveragent runs on a remote machine. This agent downloads mails from a mail server. It creates a mailagent and stores all mail data in it. The mailserveragent only keeps a reference to this mailagent for future calls and operations on it. This mailserveragent can manage the mailagents and can search their content. The usermailagent runs on the local platform (our mobile device), providing a user interface through a GUIagent. This usermailagent is responsible for managing and searching mailagents whether they are locally or not. When the user decides to view an email the mailuseragent will get the list of reference to the mailagents from the mailagentserver and give the mailagent the order to move to the local machine. At that point the mail can be read without being connected to a network. A user also has the ability to view the mail page by page. In this case only parts of the content are transferred. This allows quicker viewing of the mail but will cause loading times between each page.

In the following sections we work out this idea to demonstrate the ease of creating a distributed system with mobile agents. It also demonstrates the simple moving behavior of agents.

4.2.1 General program structure

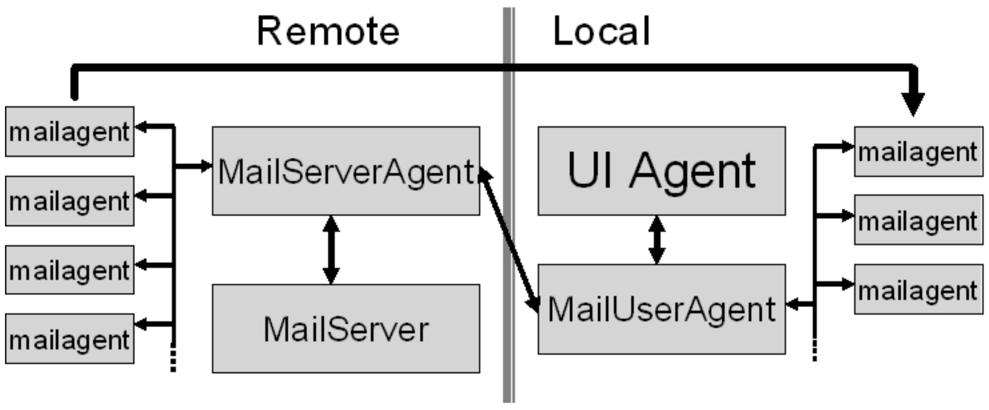


Figure 5: General structure of the mail agent program.

This program consists out of 5 types of agents:

1. GUI agent: this agent is the GUI version of the usermailagent. It is responsible for the communication between the user and the usermailagent. It provides a simple GUI that allows the user to use all the functions provided by the other agents in the program. This agent's job is to translate user actions into messages that the usermailagent can understand.

2. Mailagent: this type of agent is used to represent one email. Each received email is stored in a unique agent. This agent keeps all necessary information of the email (sender, date, subject, content). This agent also has the necessary functionality to search for certain keywords in his mail content and subject. It is capable of moving itself from one machine to another when ordered to do so by the mailserveragent or mailuseragent.

3. Mailserveragent: this agent can download mails from a mailserver and keeps a list of references to al the mailagents. The mailserveragent has the following functions:

    - Retrieving mails from the mail server
    - Keeps a list of references to al the mailagents
    - Searching mails when usermailagent is uncapable to do this operation. (Machine where mailuseragent resides on is too slow).
    - Send the list of references to the mailuseragent

4. Usermailagent: this is an agent that runs on the local machine and provides the user with the ability to communicate with the mailserveragent and offers extra functionality like viewing and managing mails.
   The user to view a mail in two ways:

    (a) The mailuseragent orders the mailagent to move itself to the mobile platform. For this it first gets the list of references to all mailagents from the mailserveragent.
    When the agent is moved, it will display the information to the user. Downloading the mail like this may take a while since the entire mailagent has to be transferred, but when the agent has moved the entire mail can be viewed quickly.

    (b) Only parts of the mail content are transferred. This allows the user to read his mail page by page. The mail is displayed more quickly (only part of content and not entire mailagent is transferred) but there will be a downloading delay between each page.

Since communicating with a real mailserver is quite complex and the main idea of this case is to demonstrate mobility, we will simulate the downloading of mails using an extra agent:

- Mailserver: this is a simple agent which simulates a mail server. Its only functionality is that it can communicate with the mailserveragent and that it can upload emails to it.

4.2.2 The program

Implementing the mailcase is quite straight forward. I will explain the program agent per agent by telling how it performs its tasks. I will illustrate some of the key functions of an agent with code. The code is very readable and is easy to understand:

- Mailagent: the mailagent is one of the most easy agents of the program. The mailagent is a mail, so it holds everything a mail is supposed to hold. It has the necessary variables to hold the subject, sender, receiver, time of receiving and contents of a mail. Most of the procedures in the agent are quite simple and serve mostly for communication between the mailagent and other agents. These procedures allow other agents to ask all or specific data from the mail and to change certain data. The mailagent also contains the code to search in its own data. This search algorithm is quite complex and checks if a string of keywords is present in subject and/or content. Date, sender and receiver comparing functions are also implemented to offer even greater searching power.

```
searchInText(keywordstring,tobesearched):: {
    pos:1;
    result:1;
    busy:true;
    searchstring: keywordstring;
    while((busy & (result>0)), {
        ‘ extract keyword out of string
        pos:=strstr(searchstring," ")-1;
        ‘ Last keyword has no space
        if(pos = -1, {  pos:=length(searchstring);
           busy:=false }
        );
        keyword:substr(searchstring,1,pos);
        searchstring:=substr(searchstring,pos+2,
                length(searchstring));
        ‘ search the keyword in the subjectstring
        result:strstr(tobesearched,keyword);
        pos:=0
        }
    );
    if(result>0,true,false)
}
```

  The above piece of code shows how the text search algorithm is implemented. It checks the tobesearchedstring to see if it contains all the keywords from keywordstring. If all keywords are present it returns true, if not, false.

The procedure that instructs the mailagent to move to another machine is very simple:

```
jumpTo(machine):: {
    agentmove(machine)
}
```

Note that it does not check whether or not the machine exists. If the machine does not exist the mailagent will move anyway and will be lost.

- Mailserver: this agent is the most simplistic one of the program. It only contains a set of mails stored in tables and the code that allows the mailserveragent to download them.

```
sendMail(ref):: {
    tmpmail:mailtable_.getElement(1);
    mailtable_.deleteElementAtIndex(1);
    ref->receiveMail(tmpmail)
}
```

The above procedure sends a mail to the agent ref when called. Tmpmail is a table containing the mail data:
[subject,sender,receiver date&time, content]

- Mailserveragent: this agent has the code to download mails from the mailserver. It downloads a mail and creates a mailagent for each mail. To allow further communications with the mailagents a reference to each mailagent is kept in a table.

```
‘ Creating the agent and apply its mail data agentref:
new(mailagentname,load("thesis/mailcase/MailAgent.borg"));
agentref->setMailData(receiver,subject,ownemail,content,timrec);
 ‘ Store the mailagents reference in the table.
mailagenttable_.addMailAgent(agentref, false,1);
```

This code creates a mailagent with mailagentname as name. The reference to this agent is stored in the mailagenttable table. Each mailagents name has to be unique.

```
mailagentname: makeMailAgentName(timereceived)+eraseBlanks(subject);
```

This line makes a unique name for each mailagent. The name consists out of the received time (returned by makeMailAgentName(timreceived)) and the subject. In the subject the blanks need to be removed since they would make the agentname corrupt.

This erasing is done by eraseBlanks(string):

```
eraseBlanks(inputtext):: {
    oldstr: inputtext;
    newstr: "";
    pos: 0;
    while(pos < length(oldstr), {
        size: length(oldstr);
        pos:= strstr(oldstr," ")-1;
        if(pos= -1, pos:=size,
            {
            newstr:= substr(oldstr,1,pos);
            newstr:= newstr + substr(oldstr,pos+2,size)
            });
        oldstr:=newstr
    });
    oldstr
}
```

The mailserver has many other procedures to allow it to communicate with the mailagents and usermailagent. It is also capable of searching mails. This code is duplicated in the usermailagent (and explained there) to allow searching of mails when not connected to the mailserveragent.

- Usermailagent: has procedures that allow it to communicate with the mailserveragent, mailagents and GUIagent. It keeps a table of references to the mailagent identical to that of the mailserveragent. This allows the mailuseragent to access mails that have been downloaded when the connection to the mailagentserver is severed. It also contains the code that searches mailagents. The search procedure allows the user to search all mailagents with certain parameters. Only agents which comply with all the given search parameters are included in the search result:
  - The subject contains all the given subject keywords
  - The content contains all the given content keywords
  - The sender address is the same as the given address
  - The received date is the same as the supplied date

```
startSearchMails(subjectkeywords,contentkeywords,
    datedata,sender):: {
    foundmailagenttable_:= makeVoidTable(1);
    if((localmailagenttable_.getNumberElements()) > 0,
        {
        inertable: localmailagenttable_.getElement(1);
        agentref: inertable[2];
        agentref->searchMail(sububjectkeywords,contentkeywords,
            datedata,sender,agentname,1)
        }
    )
}
```

The above procedure starts searching the mailagents. It gets the agent at index 1, and asks it to see if it contains all the given search parameters. The agent returns the answer to the usermailagent by calling searchresult.

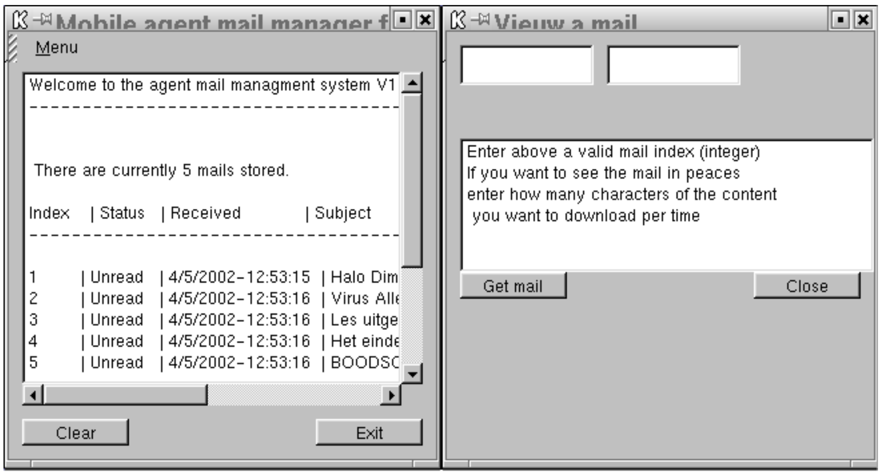
```
searchResult(agentref,index,result,subject,from,
subjectkeywords,contentkeywords,datedata,sender):: {
    if(result,
        tmptable:[from,subject,index];
        foundmailagenttable_.addElement(tmptable)
    });
    index:=index+1;
    if(index <= (localmailagenttable_.getNumberElements()),
        {
        inertable: localmailagenttable_.getElement(index);
        agentref: inertable[2];
        agentref->searchMail(subjectkeywords,contentkeywords,
            datedata,sender,agentname,index)
        },
        printResults()
    )
}
```

searchResult will store the agent in a list of found agents when it complied to all search results. The mailagent returns in "result" wether or not it complied to the search criteria. Then searchResult gets the next mailagent reference out of the localmailagenttable (if there is one) and asks the new mailagent to check itself to see if it matches with all the search criteria. This mailagent will again call searchResult,... This continues until all mailagents have been visited.

Other procedures print the search results to a string for display in the GUI.

This agent also contains the code to display a table of all currently available mails together with their unique index number. It then allows the user to get and display a mail using this index number. When the user chooses to download and view the entire mail, the usermailagent will order the mailagent at the given index in the reference table to move to the local machine. When it has moved itself the usermailagent will pass all the mail data to the GUIagent which then gives a nice display of the mail. If the user chooses to view the mail in parts the mailagent remains where it is. The usermailagent will get mail subject, sender and data and display this. It will then get the content page by page.

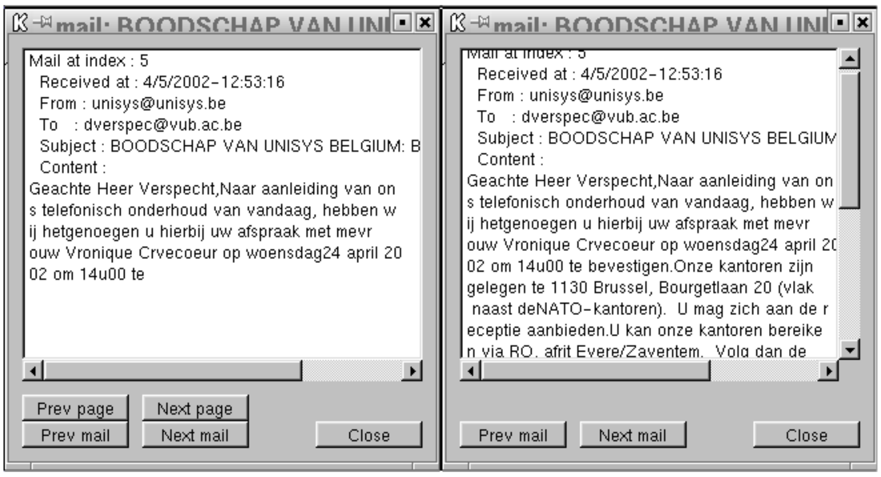
- GUIagent: contains the code that displays a basic gui and which allows user actions to be translated into messages that the usermailagent can understand. The GUI code is very straight forward and does not need any explanation since it does not implement any program functionality. The GUI is constructed using the palmwidgets borg library that allows the construction of PalmOS compatible user interfaces. This user interface is very simple since only windows, checkboxes, push buttons and text fields are supported in this user interface library.



```
Welcome to the agent mail managment system V1
-----------------------------------



 There are currently 5 mails stored.

Index    | Status   | Received             | Subject
-----------------------------------


1        | Unread   | 4/5/2002-12:53:15  | Halo Dim
2        | Unread   | 4/5/2002-12:53:16  | Virus Alle
3        | Unread   | 4/5/2002-12:53:16  | Les uitge
4        | Unread   | 4/5/2002-12:53:16  | Het einde
5        | Unread   | 4/5/2002-12:53:16  | BOODSC
```

```
Enter above a valid mail index (integer)
If you want to see the mail in peaces
enter how many characters of the content
 you want to download per time
```

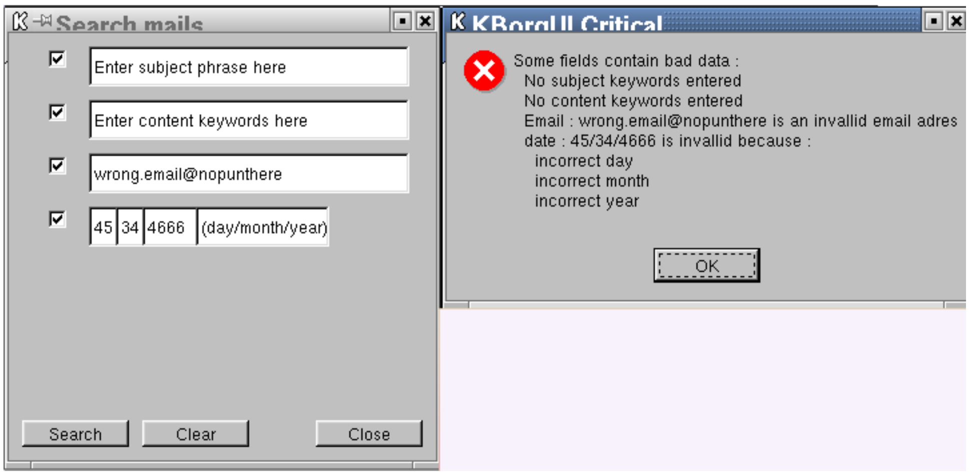
Figure 6: The mailcase GUI: managing mails.



```
Mail at index : 5
  Received at : 4/5/2002-12:53:16
  From : unisys@unisys.be
  To   : dverspec@vub.ac.be
  Subject : BOODSCHAP VAN UNISYS BELGIUM: B
  Content :
Geachte Heer Verspecht,Naar aanleiding van on
s telefonisch onderhoud van vandaag, hebben w
ij hetgenoegen u hierbij uw afspraak met mevr
ouw Vronique Crvecoeur op woensdag24 april 20
02 om 14u00 te
```

```
Mail at index : 5
  Received at : 4/5/2002-12:53:16
  From : unisys@unisys.be
  To   : dverspec@vub.ac.be
  Subject : BOODSCHAP VAN UNISYS BELGIUM
  Content :
Geachte Heer Verspecht,Naar aanleiding van on
s telefonisch onderhoud van vandaag, hebben w
ij hetgenoegen u hierbij uw afspraak met mevr
ouw Vronique Crvecoeur op woensdag24 april 20
02 om 14u00 te bevestigen.Onze kantoren zijn
gelegen te 1130 Brussel, Bourgetlaan 20 (vlak
 naast deNATO-kantoren).  U mag zich aan de r
eceptie aanbieden.U kan onze kantoren bereike
n via RO. afrit Evere/Zaventem.  Volg dan de
```

Figure 7: The mailcase GUI: viewing mails page by page or entirely



```
☑ Enter subject phrase here
☑ Enter content keywords here
☑ wrong.email@nopunthere
☑ 45 34 4666 (day/month/year)
```

```
Some fields contain bad data :
  No subject keywords entered
  No content keywords entered
  Email : wrong.email@nopunthere is an invallid email adres
  date : 45/34/4666 is invallid because :
    incorrect day
    incorrect month
    incorrect year
```

Figure 8: The mailcase GUI: searching emails

4.2.3 Program conclusions

We will now evaluate the programs performance, using the four different situations described in the previous section.

On fast machines with a fast network connection:
This is the situation where we run our case on PC's connected with a fast network. On a fast machine each action a user can perform in the mail case is simple enough to be done without any noticeable delay.

On a fast network (take 2400 Kbit/sec or 300 Kbyte/sec), agent migration using weak mobility is very fast. Only for very large mails a small delay may be seen. On fast networks with a small response time delay the time it takes to migrate an agent can be estimated by the formula:
migration time = agent size / network bandwidth
On fast networks this time is always below 1 second. This is because the largest mail in our test has a content of approximately 20 K. Even if the mailagent would take up 100 Kb this would mean that it would be able to migrate in
100 Kb / 300 Kb/sec = 0,3 seconds to another machine. On this fast or even faster networks it would take mails with contents larger than 1 Mb before delays can be monitored.

Reading a local mail is even faster. Getting the mailagent from internal memory takes only a few milliseconds. The user however will not notice a difference between the 300 ms it takes to get the mailagent over the network or the 3 ms it might take to get the mail from local memory.

Getting a mail in pieces does not offer any noticeable benefit at all in terms of downloading delays. It is however a much more comfortable way to view large mails on the small Palm Pilot screen.

On fast machines with a slow network connection:
This is the situation where we run our case on fast machines connected with a slow network. Examples of such a situation can be a laptop or PocketPC connected with a GSM to offer access to a network. In this case processing power is more than sufficient to prevent noticeable delays whatever mailcase operation is performed.

On a very slow network (only some Kb/sec like GSM) the situation is a lot more different from the previous case. Migrating the mailagents takes more time and here it becomes useful to view large mails in parts in terms of speed. We can estimate agent migration time by using the formula:
time = (agent size / network bandwidth) + total response time delay
This formula takes network delays into account. Simply adding the network delay would cause bad results since an agent migration is done by sending several packets over the network.

Each packet has a network response time delay, depending on whether or not some packets get lost, the total delay can differ greatly. The only thing we can say about the delay is that it is proportional with the size of the mailagent.

When we try to migrate an average sized mailagent, say 20 Kb, this will take significantly more time than in the previous case.
Even without incorporating the total response time delay, our formula gives us that it will take 20 Kb/sec / 2 kb/sec = 10 seconds to move our agent. When migrating a mailagent with a large content (> 100 Kb) this time is not longer measured in seconds but in minutes.

This leads to the conclusion that viewing mails using a machine with a slow network connection is best done by viewing the mails page by page. If we choose to download mails in parts, for instance 10000 characters (= 10 Kb) per time, the mail content will be downloaded in about 10 Kb/ 2Kb/sec = 5 seconds. The user will have to wait 5 seconds between each mail page for the next piece of the content to download.

This situation clearly shows us that a slow network connection like a GSM connection is actually too slow to allow weak mobility migration of agents.

On a slow machine with a fast network connection
In this situation we can have Palm devices with a fast wireless Ethernet connection or the brand new umts GSM/organizers. Like we have seen in the test environment section, Palm devices have a max network bandwidth of 20 Kb/sec. On slow machines like Palm devices, the borg GUI becomes an enormous bottleneck. The mailcase itself runs fast enough on both fast and slow machines. The GUI however is not suited for slow machines. The GUI update delays on fast machines are only 2-3 seconds which is reasonably fast. On slow machines this delay exceeds 10 seconds.

Transferring a mail using weak mobility does not take too long. A standard mail of 50 Kb is moved in less than 3 seconds. It takes however at least 10 seconds to built the new GUI windows which takes 13 seconds total time to view the mail.
When we view the mail page by page we gain only 1-2 seconds in transferring the mail content. Making the initial window to view the mail takes again 10 seconds, but once this window is created getting the next content requires only a minor GUI update (2 sec). Getting the next page of your mail then takes only some seconds which is an acceptable delay.

Here the user is perfectly free to use whatever mail viewing method he wants. They both have acceptable delays. Viewing the mail page by page is again better for viewing large mails on the small palmtop screen.

On a slow machine with a slow network connection
This is the worst case situation: here not only the GUI causes delays but also the downloading of the mail data takes a lot of time. Devices that are in this situation are the new GSM/organisers, offering a Palm like processor with a GSM network connection. Currently, these devices support network speeds between 9600 to 28800 depending on the GSM network they are operating on. Current modern devices with gprs will get 56000 which is still considered slow.

Here, transferring an average 50 Kb sized mailagent will take quickly more than 25 seconds if we take an average 2 Kb/sec network speed. Larger mails of some hundred Kb will take minutes to download. In this case it is recommended to get the mail in pieces. Downloading a 10 Kb piece of the content takes approximately 5 seconds, together with the 2-3 seconds to update the GUI this is an acceptable time.

Conclusion
For the mailcase, we can conclude that network speed is very important if we want to use mobility. On fast networks the delay of migrating a mailagent using weak mobility is barely noticeable by the user. On a slow network weak mobility becomes a pain to use since transferring an average sized 50 Kb mailagent already causes delays of more than 20 seconds. Small mailagents can still be transferred using weak mobility, but once migrating a mailagent takes more than 20 seconds, it is better to start downloading the mail in parts. Certainly on fast machines with a slow network connection the page by page mail download is a far better option.

Slow machines do not change the situation much. We have learned that due to processor power and space constraints the maximum network speed is seriously compromised in palmtop devices. Weak mobility moves a mailagent in an acceptable time on a slow machine and only very large mails are best viewed using page by page download. On slow connection it is always the better option to use the page by page download. The difference between slow and fast machines is that everything goes a little slower on the slow machine because of slightly larger network delays and slower GUI updates to display the mail.

The GUI is the real performance bottleneck in this case. Certainly on mobile devices this GUI causes very large delays. It takes over 3 seconds to create a new window and still about 2 seconds to update it on a fast machine. We can only guess how much slower the GUI will be on a slow Palm device. The delays will be certainly more than a factor 5 larger which will cause unacceptable delays.

Generally we can conclude that weak mobility has proven its use. However when using a very slow network connection even migrating a mailagent using weak mobility requires too much time. In such situations a page by page mail download becomes preferable. The biggest bottleneck on slow machines is the GUI that will have to undergo some major performance tweaks before it will be usable on a mobile device.

4.3 Meeting planner:

The meeting planner case is a distributed system that allows users to set up a meeting in a certain room at a certain time with a minimum number of participants, etc. Each user gets a GUIagent and an MP-agent which allow the user to create, view, manage and delete meetings. The agents communicate with each other in a way that is known as peer-to-peer communication. Each user keeps its own data (GUI, MP agent and created meeting and their data in meetingagents). This means that every set of agents have unique data for each user and that all the agents of all users in the distributed system, will have to communicate with each other to come to a conclusion (meeting ok or not).

This testcase will try to find out if the presence of a slow machine in a distributed system can influence the performance of the program. In distributed systems, all data and processing are distributed. The presence of a mobile device in such a system may influence the speed of the meeting planner. If it does slow down the program, we may be able to find a solution to the problem using mobility.

4.3.1 General program idea

The meeting planner consists out of 3 agents:

1. MP-agent: which represents a user or place. This agent contains the code to setup, cancel, view or edit meetings. It contains commando's from the user through the GUI-agent. It keeps a list of references to all other MP-agents of other users and another list with all references to meeting-agents the user has created.

2. GUI-agent: this agent is the graphical user interface which communicates with the MP-agent with asynchronous messages. When the MP-agent sends a message to the GUI-agent it will not wait for the GUI-agent to reply before continuing. Such type of message passing is called asynchronous. The GUI-agent has to be on the users machine, but does not need to be on the same machine as the MP-agent.

3. Meeting-agent: this agent represents a meeting. A meeting agent is created by the MP-agent. The MP-agent will kill this agent if a meeting has passed or is cancelled. This agent is responsible for the actual planning of the meeting, it will check all other users to see if they can participate. If the user given minimum number of participants is not reached, the meeting-agent will kill itself cancelling the meeting. If the minimum number of participants is reached, the meeting-agent will inform all users that the meeting is planned and that they are no longer free on that given date and time.

The user uses the meeting planner by logging in using a name and password. He then can choose to plan a new meeting. To do this, he fills all necessary data: organizer, date, start time, end time, meeting subject, place (room), guests and a minimum percentage of people who have to attend the meeting. This data is entered through the GUI-agent which sends all entered data to the MP-Agent. When all data have been correctly entered, a meeting-agent is created by the MP-agent. The meeting-agent will communicate with all other meeting-agents to check if the place where the meeting will take place is available at a given date and time. Should this not be the case, the meeting-agent will warn the user who then will have two choices: trying another place or aborting the meeting. When the place has been reserved the meeting-agent will try to reserve sufficient participants for the meeting. It does this by contacting all other users logged on to the meeting planner system. If there are not enough participants or when the place is not available at the time of the meeting, the meeting will be cancelled and all users will be warned about the annulation. If their are enough participants the meeting will be confirmed to all users.

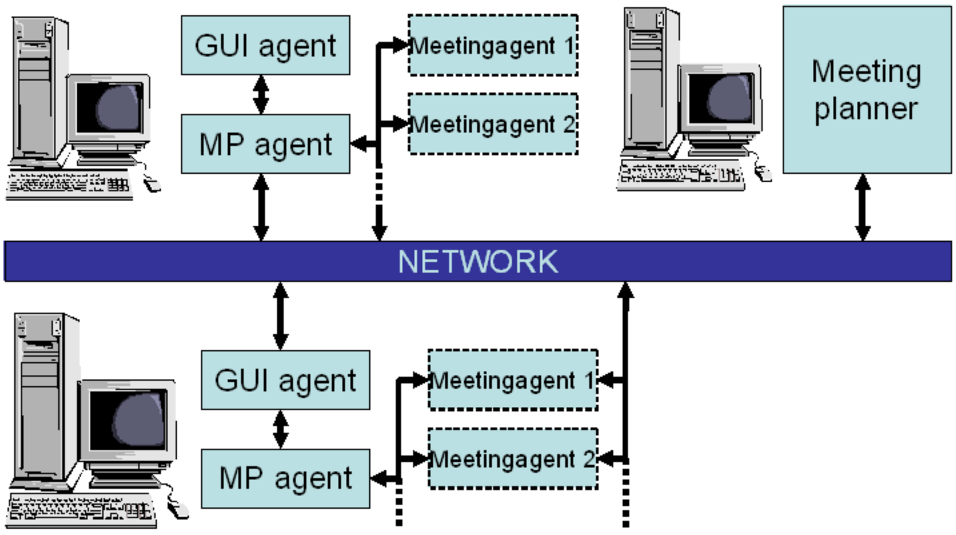


Figure 9: The meeting planner: agent distribution

Figure 9 illustrates how the agents are distributed. Note that the MP-agent and meeting-agents do not need to be on the same machine than the GUI-agent. Only the GUI-agent needs to be local to allow communication between user and program. Standard the MP-agent is loaded on the same machine than the GUI-agent and the MP-agent will create meeting-planner agents on his machine. The agents can easily be modified to use weak mobility. It is for instance possible to move the MP-agent immediately to another machine once created where it will create all meeting-agents as well. This may seem useless now, but will prove to be very useful further on.

4.3.2 The program

The mailcase is programmed by Pieter Verheyden. The mailcase is made of 3 types of agents which communicate in a way that is known as peer to peer:

1. GUI agent: this agent is the most straightforward agent to program. It creates a simple graphical user interface using the PalmWidget library for borg. Since the GUI-agent does not offer any program functionality and its code is very straight forward I will not illustrate it with sample code.

2. MP-agent: the MP-agent is the agent that offers all the program functionality. It is the main agent that enables a GUI-agent for itself. The MP-agent keeps all data necessary to set up a meeting. It has a list with references to all its meeting-agents so the user of the MP-agent or other users can communicate with them. It also has a list of all other currently logged in users so the meeting-agents can contact them to get participants and warn all users about a meeting success or failure. A new meeting is planned using this code:

```
setupMeeting(datetable, guesttable, percentage, subject, place):: {
    if(checkDate(datetable),
        if(((guestmeetingtable_.checkForFreeTime(datetable))
    & (mymeetingtable_.checkForFreeTime(datetable))
    & (waitingmeetingagentstable_.checkForFreeTime(datetable))),
            {
            meetingagentname:makeMeetingAgentName(datetable);
            maref:makeMeetingAgentObject(meetingagentname,
                datetable, agentname, guesttable, percentage,
                subject, place);
            mymeetingtable_.addReservation(maref, datetable);
            if(!is_void(guiagent_),
                guiagent_->setupMeetingFinished()
            )},
            returnError("No free time found")
        ),
        returnError("You entered an invalid date and/or time")
    )}
```

   The above procedure is called from the GUI-agent and contains all data necessary to start a meeting: the date and time, the guests that have to be present, the percentage of participants, a meeting subject and place. This data was entered by the user in the GUI and send to the MP-agent. First of all, the data is checked to see if it is correct (correct date, does the place of meeting exist, ...). If the place is not yet occupied at that time by another meeting (a meeting from another user or one from the current user) a new meeting-agent is created. makeMeetingAgentName(datatable) creates a unique name for the meeting-agent using date, start and end time of the meeting. The agent is then added to our local table of meetings which keeps a reference to all meeting-agents created by the user controlling the MP-agent.

Cancelling a meeting can be done when the place can not be reserved or at any other time:

```
cancelMyMeeting(datetable):: {
    meetingagent: (mymeetingtable_.getMeetingAgent(datetable));
    if(is_false(meetingagent),
        returnError("You did not setup a meeting on " +
            dateToString(datetable)),
        {
        meetingagent->cancelMeeting(agentname);
        mymeetingtable_.deleteMeetingAgent(meetingagent);
        if(!is_void(guiagent_), guiagent_->cancelFinished())
        }
    )
}
```

This code will check if there is a meeting-agent at the given time. If there is a meeting-agent, it will be killed and deleted from our meeting-table. Note that only the user who created the meeting-agent can kill it and thus cancel the meeting.

The MP-agent also has the needed code to reserve a place:

```
tryReservate(meetingagent, datetable):: {
    if(waitingmeetingagentstable_.checkForFreeTime(datetable),
        {
        waitingmeetingagentstable_.addWaitingMeetingAgent
            (meetingagent, datetable);
        if(((mymeetingtable_.checkForFreeTime(datetable)) &
            (guestmeetingtable_.checkForFreeTime(datetable))),
            reservationSuccess(meetingagent, datetable),
            reservationFail(meetingagent)
        )
        },
        meetingagent->reservationFail(agentname)
    )
}
```

This code checks if the current user and the place are available at the time of the meeting. If not, the user will be prompted if he wants to pick another place and/or time.

Besides all the above codes, the MP-agent contains extra code to change the data of a meeting, and to do some extra GUI-interaction, like confirming participation to a meeting of another user, log in, password management, etc...

3. Meeting-agent: a meeting-agent contains all code and data to manage everything in a meeting itself. A meeting-agent keeps its time, data, subject, place, minimum number of participants and a list of participants. It is also responsible for finding participants for the meeting.

```
checkMeetingStatus():: {
    nbrparticipants:participanttable_.getNumberElements();
    if(nbrparticipants>=MINNUMBER, true, false)
}

trySetupMeeting():: {
    if(checkMeetingStatus(),
        {
        caller->setupMeetingSuccess(agentname, datetable);
        place->guestMeetingSuccess(agentname, datetable);
        for(i:1, i<=(participanttable_.getNumberElements()),
            i:=i+1,
            {
            cparticipant:(participanttable_.getElement(i));
            cparticipant->guestMeetingSuccess(agentname,
                datetable)
            }
        )
        },
        {
        nbrparticipants:participanttable_.getNumberElements();
        caller->setupMeetingFail(agentname, datetable);
        place->guestMeetingCancel(agentname, datetable);
        agentdie()
        }
    )
}
```

The above code checks to see if there are enough participants for a meeting. If the meeting planning is a success, all users are informed so they can add the meeting in the datetable and know they are occupied on the date and time of the meeting. If the meeting fails because there are not enough participants all users are informed about this failing so they will delete their reference to this meeting-agent. Also the user of the meeting-agent is informed about the meeting planning failure. When this is done the meeting-agent kills itself

```
meetingFail():: {
    nbrparticipants:participanttable_.getNumberElements();
    for(i:1, i<=nbrparticipants, i:=i+1,
        {
        cparticipant: participanttable_.getElement(i);
        cparticipant->guestMeetingCancel(agentname, datetable)
        }
    );
    place->guestMeetingCancel(agentname, datetable);
    agentdie()
}
```

The above code will inform every user in the meeting planner system that the meeting in this meeting-agent has failed. This allows the MP-agent of those users to erase the reference to these agents.

The meeting agent contains some other procedures that allow the user to cancel the meeting manually or that enable getting and setting data in the meeting-agent like the place and date.

I extended the meeting planner with some minor changes.

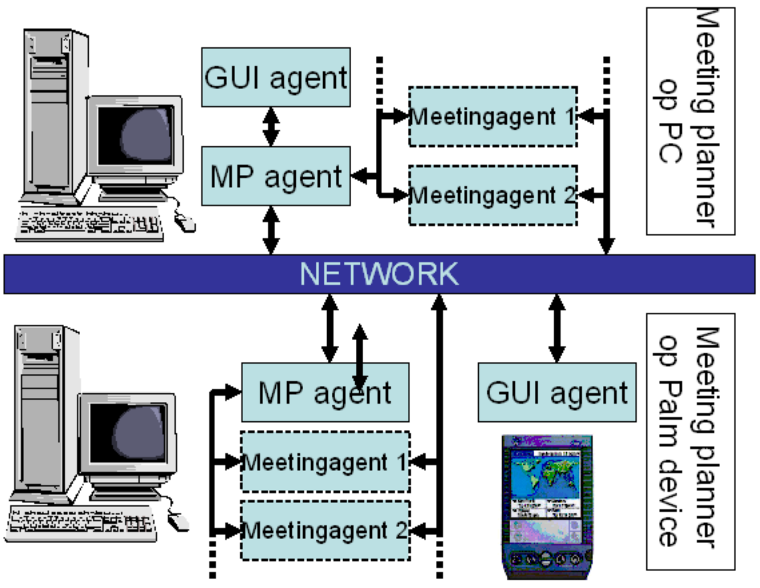


Figure 10: The meeting planner: agent distribution when a mobile device is used

Figure 10 clearly demonstrates what I changed about the meeting planner. When a user logs in using a PC everything remains as it is explained above. When the user logs in using a mobile device all agents except the GUI-agent can be moved to another platform. The reason for this change will become clear in the next section.

4.3.3 Conclusion: Use weak mobility

Since the meeting planner does not use any form of mobility when running we will use different test situations here. The program is distributed which meas that the network speed will have a general impact on the programs performance. The messages the agents send to one another are usually very small so the network bandwidth is not important. As the agents need to communicate a lot, the network response time delay will have an impact on the performance. The larger this delay, the slower messages are send and the slower the program will perform. Since this network delay is difficult to simulate and is only of minor importance we will not test different network speeds.

In the chapter about our test environment, we saw that the performance of a distributed program is more dependant on the slowest machine in the network than on average processing speed. This is why in this case we will test two situations:

All meeting planner agents run on a fast machine

When the meeting planner runs on a PC platform all the agents of the system have plenty of processor power, memory and network speed. In this case, the program runs smooth and no noticeable delays are monitored. Our meeting planner extension that can move agents to another platform is useless because almost every PC is fast enough to finish all the different meeting planner operations in time.

All meeting planner agents but one run on a fast machine

This is the case where a user logs on using a slow machine like a mobile device. The slow machine in this situation in a Palm device using a mobile network connection. In this situation there are 2 possibilities:

1. <u>All meeting planner agents remain on the slow machine</u>: in this situation the GUI-agent, MP-agent and meeting-agents the user might create remain local. In this situation the meeting planners performance suffers slightly because of two reasons:

   (a) Slow machines, like mobile devices use a mobile networking solution. This mobile networking solution has less bandwidth and a larger response time delay compared to networks between PC's. Like mentioned above, the bandwidth is not really an important factor. A larger delay however will cause messages between agents to travel longer. This will cause a performance drop like table 5 illustrated in the test platform section of this chapter. This is especially the case when using a GSM network connection that has large delays.

   (b) The slow machine cannot perform the requested calculations as fast as the other machines in the distributed system. This causes the other machines to wait for the slow machine to finish its calculations. This way precious processing power is wasted which causes a performance drop. This is illustrated by table 4 in the test environment section of this chapter.

The meeting planner case has a small performance drop due to the combination of both a slightly slower network connection and lesser processing power. The slow machine is still capable to finish all calculations in an acceptable time, this is why the user will notice only a small performance loss. The biggest processing power eating function is again the GUI. When a GUI update is done (new window) the slow machine needs almost all its processing power for this update. This causes that other operations on the slow machine temporary stall whenever a GUI update is performed. This causes longer calculation times and thus longer waiting times for the other agents on other faster machines which need the results.

While the performance hit may be barely noticeable by the user, there is definitely a waste of processing power because the fast machines need to wait for the slow machine to finish its calculations. On more complex distributed systems that require more intense calculations, this will cause very big performance hits. This is why I added the extension to the meeting planner that allows it to move all agents, except the GUI-agent, to another (faster) machine.

2. Only the GUI-agent remains on the slow machine: this is illustrated by figure 10. In this case only the GUI-agent remains on the slow machine (mobile device). The MP-agent will move itself to another fast PC platform once it is created. All meeting-agents will be created on the same machine as the MP-agent and thus will be on a remote machine as well. This causes all agents that offer program functionality (and do the calculations) to be on a fast remote machine.

   In this situation the distributed system seems to function without any delay at all. This is easily explained since we created a distributed system where all agents reside on a fast machine like illustrated in table 3 in the test environment section of this chapter. The GUI agent is still on a slow machine, but since it does not perform any calculations, there are no agents on other machines that are waiting for results from the GUI. The MP-agent just sends GUI-update messages which are handled by the GUI independently of all other agents. Since the messages between the MP and GUI-agent are asynchronous the MP-agent is not influenced by delays in the GUI-agent.

   The user of the slow machine will notice a slightly better performance since the machine can now focus entirely on GUI actions.

In this case an enormous advantage of weak mobility becomes clear. A slow device can have a great performance hit in a distributed system. By simply moving all agents but the one responsible for the user interface to another faster machine, a slow machine like a mobile device can participate in a distributed system without causing any performance loss.

This system requires only minor modifications to existing distributed agent systems. All agents are loaded locally; once all agents are loaded, the agents that can be moved (agent which are not responsible for user or other interactions on the mobile device) are moved to another faster machine using weak mobility. The only code that has to be added is the code that will move the agents to another machine. The only disadvantage of this solution is that the time to start the program will increase because moving the agent costs some time. How much longer the start-up will take depends on how fast the network is.

The solution presented in this case can be used in all agent based distributed programs thus enabling a mobile device to join a distributed or peer-to-peer "community" without any penalty in general program performance.

4.4 The GPS route planning system

Everybody has heard of GPS navigation, the miracle devices which ensure their users will never get lost again. You simply insert where you want to be, add some via points (if wanted) and off you go. The navigator calculates the fastest route towards your destination and displays a nice map of your current location where the road to follow has a nice bright color. While you are driving information appears on crossroads and obstacles ahead and a gentle voice informs you of actions to take.
No wonder GPS navigation systems have become so tremendously popular in past years. Many people have GPS navigation now; some have a GPS navigator in their cars, others have a small handheld GPS device with very limited navigating skills. Some have a PC or laptop with an optional GPS module.
Just recently some companies started manufacturing GPS and route planning expansions for known palmtop brands like Visor, Palm and Pocket PC. While these can offer at least the functionality of a mobile GPS navigator, they have two major drawbacks:

1. Finding an optimal route in a map containing a lot of data is a very complex and difficult task. Thus route planning systems require lots of processing power.

2. The map containing the route data of your current location contains a lot of data and therefore takes a lot of memory. If that isn't bad enough, the route planning algorithm itself requires a lot of memory too.

We have seen that palmtops lack these two items. This case will demonstrate that mobile agents can provide enhanced route planning for palmtop devices.

4.4.1 General program idea

In all cases a GPS navigating system is made of the same parts:

- GPS (to known current location, speed, heading, ...)
- Processing unit (to do the necessary calculations)
- Software (to do the calculations and for user interface)
- A digital map (a map containing the information on all the routes at present position, this is a large collection of vectors which represents the different roads)
- A display to inform the user with either very basic data (raw position in longitude and latitude) or more complex data (display of all routes on current locations, actions to take, ...)
- Some optional parts like, for instance, audio to inform the user by voice about his position and/or actions to take.

When looking at the list above, we can see that all the parts required by a GPS navigation system are present in PC and palmtop devices. Only the GPS module is missing. Some companies noticed this as well and in order to offer the user cheap GPS navigation systems they created a PC compatible GPS module that can be used with any popular PC navigating software. Such a module attached to a notebook gives people an excellent alternative for GPS navigation in cars compared to embedded car navigation systems.
However, notebooks are too big and heavy for more mobile GPS navigation (for use on foot). Better candidates are our palmtops like Palm, Visor andPocketPC.

4.4.2 Overcoming some problems

Like mentioned in the introduction there are some complications with GPS navigating on a mobile device. The implementation of such a system is not straight forward because the processing power of a mobile device simply isn't good enough to do the complex calculations in an acceptable time. Also the memory of mobile devices is much too small to store detailed maps of an area. Last but not least the remaining work memory is too small to allow calculations of a large route since such calculations require lots of memory.
While these are all very good arguments not to attempt to create a mobile route planner many have tried to anyway. [18]
If the device is intended to be fully mobile, the best use for it will be on short excursions on foot. Such routes can be complex, but are often not very long. The user will most likely plan his route before he starts walking so only the map data relevant on this route has to be uploaded. In this case the local memory and processing power are sufficient. This allows the user to navigate without the need for a network (which is most likely absent in mountains, forests,...)

For other uses, like on longer excursions or navigation in a car the local processing power and memory may become insufficient. The maps, even if only the needed parts are uploaded, become too large to be stored locally at full detail. Also the calculations on these complex maps will take a lot of workmemory. This is where our strong mobility comes in. When the calculations get too complex and take too long to complete they can be offloaded to a more powerful platform. Long complex routes are only used in a car where we most likely do have a network (GSM). In that case the calculations can be offloaded when they take too long and the user can consult more detailed maps via the network.

4.4.3 General program construction

Our program will consist of the usual GPS navigator parts like described in the introduction of this case. There will be a map, a user interface and route calculating part. Our program will consist of 4 agents:

- GUIagent: this agent implements a simple GUI like in our mail case. This agent is responsible for forwarding user actions (pressing a button, tapping the screen, ...) to the routeplanneragent which will then process these commands. It is also responsible for displaying the information it gets from the routeplanneragent to the user.
- Routeplanneragent: this agent's main job is to calculate the shortest route between two points. It does this by using the map data in the mapagent. This agent also accepts user requests from the GUI (start planning a route, print route, ...) and sends information to the user via the GUI (optimal route results, route planning status, ...)
- Monitoragent: this agent monitors the routeplanneragent. It closely monitors the time the routeplanneragent is calculating and the memory it is using. If it calculates too long or takes too much memory this agent will order the route agent to move to another machine.
- Mapagent: this agent simulates the data map (local & remote). Its only function is to hold the map data and to give it to the routeplanneragent when asked for.

The routeplanner can be in 2 main different conditions where we can split one condition in three subconditions:

- The route planner is running completely local and all agents remain local during the planning of a route.
- The routeplanneragent has moved to other machines and the agents have to communicate over the network.
  - The mapagent remained local and the routeplanningagent still uses it to plan its route. This will cause heavy network traffic since there is intense communication between the two agents.
  - The mapagent moved to the same machine as the routeplanneragent or the routeplanneragent gets its data now from another mapagent on the same machine. This is an ideal situation where the route planner process can run at full speed without heavy network load.
  - The mapagent is moved as well but is on another machine than the routeplanneragent or the routeplanneragent gets its data from another mapagent on another machine. In this case, the network load between the machines containing the active mapagent and the routeplanningagent will be high. Program performance will depend greatly on the network speed between those two machines.

We will now discuss the exact working of the program in a specific state to explain betterits mobility behavior.

<u>The program runs completely locally</u>

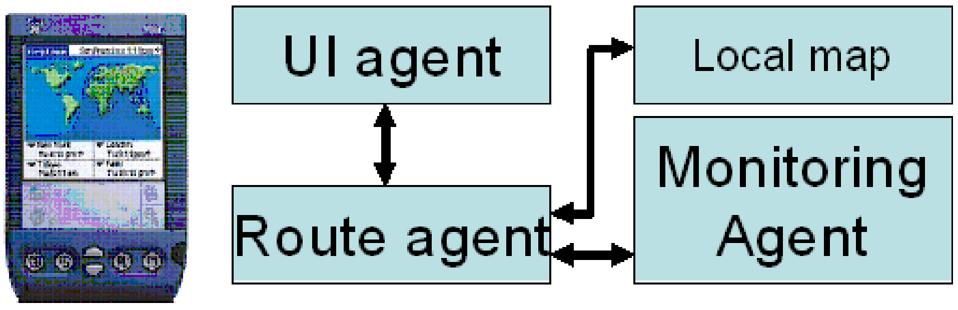


Figure 11: GPS navigating program when running completely local.

In this case all map data is stored in the local map. If the data requested by the user is not in it the program can optionally consult a remote map. All agents run locally so the operations can not be very complex.

The first case of the program will most likely occur when planning short trips. To explain the working of the program better we look at a simple example run of our program:

- The user uploads the map he needs to the mobile device
- The user plans a route by giving a start and an end point
- The GUIagent forwards user actions to the routeplanner agent
- The routeplanneragent calculates the route using the data from the mapagent
- The calculations are finished in time and the monitor agent does nothing.
- The routeplanneragent prints the route and forwards it to the GUIagent which displays the route to the user.
- The GUIagent displays the info sent by the user agent.

The above example run shows how the agents behave when they all remain locally. The routeplanneragent uses data from the local map in the mapagent and the monitoring agent does not do anything since calculations are simple enough to finish in time.

The program runs on different machines

This will happen when the monitor agent sees that the routeplanneragent takes too long to calculate the route or starts eating too much memory. It will then order the routeplanneragent to move itself to another machine. On this remote machine a new monitor agent can be created. This will allow the monitor agent to move the route agent again if:

- The remote platform is too slow to finish the calculations in time
- The remote platform is too busy with other agents/applications
- The network connection between the remote and local platform is too slow

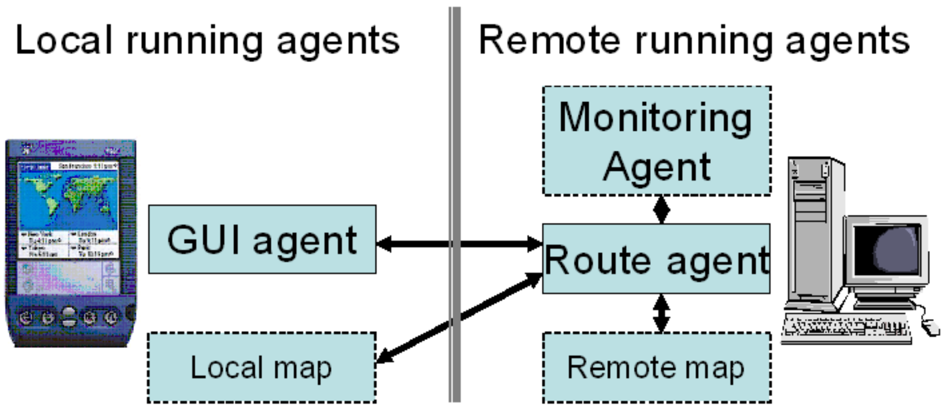


Figure 12: GPS navigating program when routeplanneragent exceeds either the max memory use or max calculation time threshold. The agents reside on different machines.

As figure 12 indicates, the communication between the agents is now more complex. The fact that the program runs partially local and partially remote will not be noted by the user. To understand the working of the program we look at a simple example run of our program:

- The user uploads the map he needs to the mobile device or uses the basic map. If local memory is too low the route agent will use a remote, more detailed, map. These maps are represented as a mapagent.
- The user plans a route by giving a start and an end point in the GUIagent. The GUIagent start the routeplanneragent with this start and end point.
- The routeplanneragent calculates the route using the local and/or remote mapagent.
- The calculation takes too long or the memory use is too high and the monitor agent decides to order the routeplanneragent to move itself to another machine.

- The routeplanneragent moves itself and its current data stack to another machine (preferably one with a fast network connection with the machine containing the active mapagent).
- The routeplanneragent continues its calculations on the remote machine and returns a route to the GUIagent when it finishes.
- The GUIagent prints this route to the user

The only step that differs from the above situation is the step where the monitoragent checks the routeplanneragent. In the first case, the routeplanneragent's memory use and calculation time remains under the move threshold of the monitoragent and nothing happens. In the latter case described above, the routeplanneragent passes one of the thresholds and is ordered to move itself.

This illustrates that the moving of an agent is transparent to the other agents which are unaware of the new location of the routeplanneragent and keep communicating with it like it never moved at all. This demonstrates the power of the mobile agents platform independency.

4.4.4 The program

Implementing the route planner case was a lot more difficult than the mailagent case. I will explain all different agents and illustrate some of their more important functions with borg code.

- Mapagent: this agent contains a small test database. This database consists of two parts:
  1. Nodes database: nodes represent crossings, villages or other special locations on the map. Each node in the database has unique ID and one or more names.
  2. Edges database: edges represent pieces of roads in between crossings and where road conditions (orientation, speed, ...) remain the same. An edge always goes from one node to another and thus contains a start and an end node. It also has information about its maximum speed, length and cost. Cost is used to represent roads that are not free like in France where you have to pay a sum according to the distance you travelled on the highway. Note that a road can consist out of multiple edges. An edge also has the name of the road it is a part of.

  These databases are stored in borg/pico tables:

```
nodetable_: [
    '[node id, node name]
    ["node01","Gent"],
    ["node02","Erpe mere"],
    ["node03","Aalst"]
    ["node04","Dendermonde"]
]
```

```
 edgetable_: [
    ‘[startnode, endnode, length, max speed, cost, route name]
    ["node01","node02",10,120,0, "E40"],
    ["node02","node03",10,120,0, "E40"],
    ["node03","node04",9,70,0, "Dendermondse steenweg"]
]
```

Several procedures in the mapagent allow other agents to get data from this database. The functions used by the routeplanneragent to plan a route are:

getNode gets the node data from the database where the nodename is ”city” and returns this data to agent ”agentref”

```
getNode(city,agentref):: {
    found:false;
    tmptable:0;
    index:1;
    size: size(nodetable_);
    while(((!found) & (index < size)),
        {
        tmptable:= nodetable_[index];
        if(strstr(tmptable[2],city) > 0, found:= true);
        index:= index + 1
        }
    );
    agentref->acceptCity(tmptable)
}
```

getEdge returns all data of the edge between ”node1” and ”node2”. This procedure is used to print a route and to check if two nodes are neighbours (they are if this procedure returns an edge)

```
getEdge(node1,node2,agentref):: {
    found:false;
    tmptable:0;
    index:1;
    size: size(edgetable_);
    while(((!found) & (index < size)),
        {
        tmptable:= edgetable_[index];
        if(strstr(tmptable[1],node1) > 0,
            if(strstr(tmptable[2],node2) > 0,
                found:= true
            )
        );
        if(strstr(tmptable[1],node2) > 0,
            if(strstr(tmptable[2],node1) > 0,
                found:= true
            )
        );
        index:= index + 1
        }
    );
    agentref->acceptEdge(tmptable,node1,node2)
}
```

getEdges is the most important function to plan a route. It returns all edges containing a node "nodeid". The routeplanner will pick one of these edges to continue its route construction and store the others for backtracking.

```
getEdges(nodeid,node2,agentref):: {
    foundtable: makeVoidTable(1);
    tmptable:0;
    index:1;
    size: size(edgetable_);
    while((index < size),
        {
        tmptable:= edgetable_[index];
        if(strstr(tmptable[1],nodeid) > 0,
            foundtable.addElement(tmptable)
        );
        if(strstr(tmptable[2],nodeid) > 0,
            foundtable.addElement(tmptable)
        );
        index:= index + 1
        }
    );
    agentref->noNeighbour2(foundtable,nodeid,node2)
}
```

- Routeplanneragent: this agent does the actual route planning and communicates with the user through the GUIagent. the routing planning code is quite complex and too large for proper illustration. So I will just explain the principle of the route planning. The route planner has to keep several data in order to successfully plan a route:
  - A best route: contains the list of edges forming the route with the lowest cost until now. The route cost is the total length of the route and is used to see if another found route is better (it is when it has a lower cost).
  - A route: this is a table that contains the edges of a route in construction.
  - Visited nodes: this is a table that contains nodes that have been visited. This table is necessary to avoid picking edges that have already been used.
  - Backtrack stack this stack stores all information of the route planning each step in process. This information can be used to backtrack and to continue the route planning using an alternative choice at that point.

The working of the route planner is best illustrated by explaining every step in the process:

1. A user starts the route planner by giving 2 cities
2. The GUI forwards these city names to the routeplanneragent
3. The routeplanneragent will ask the mapagent to send it the nodeid of these cities.
4. If the city names exist, the mapagent will return the nodeid's to the routeplanneragent
5. The routeplanner checks if the two nodes are neighbors, it does this by asking the mapagent if an edge exists between the two nodes.
6. The mapagent returns either an edge in which case a route has been found and the routeplanneragent will add it to his route table that contains all visited edges (and thus our route). The route is stored and backtracking is done to see if a more optimal route exists. (step 8)
   No edge is returned in which case the route planning continues to construct the route further. The routeplanneragent will ask a list of edges from the mapagent containing the current node.
7. The routeplanneragent gets a list of edges. It will filter all edges in this list and erase those which contain nodes that are already visited (these are stored in a table) and will pick the first edge to continue its route planning. The other edges are stored in a table and are pushed on the backtrack stack together with all other information necessary to resume route planning in case of backtracking.
   The first edge we picked is saved in our route table and the current node is stored in the visited table. A new route cost is calculated. This cost is the total route length so the new cost is simply
   cost:= cost + length(newedge).
   -If this new cost is smaller than the cost of the previous found optimal route (or the initial cost if the first route still has to be found) the route planner continues by again checking if the new current node and destination node are neighbors, ... (step 5)
   -If this new cost is larger than the cost of the previous found route, this route is no longer optimal. The route planning will stop and backtrack.
8. The backtrack procedure is called which will pop the latest route planner configuration from the stack. It checks the remaining list alternative edges.
   -If there are still alternatives (that haven't been visited yet) the route planner will pick the alternative edge, store this one in route, store the current node in visited and check if the new current node and destination node are neighbors. (step 5)
   -If the list of edges is empty, this means all edges on this level have been tried and the backtrack procedure will restart, popping the previous state from the stack, checking the edges, ...
9. Eventually every possibility is tried. When no possibility remains the route planner stops and the most optimal route found is printed.

- <u>GUIagent:</u> provides a simple GUI where the user can start the route planning process by giving start and end city. The user also can give an initial cost. This cost is to prevent that the route planner will search too far. This cost value has to be larger than the length of the shortest route between the two cities or the route planning will fail.
- <u>Monitoragent:</u> The Monitoragent has a simple timer function that is activated when the route planner starts calculating and stopped when it finishes. The monitor agent checks continuously the calculation time of the routeplannignagent. When this time exceeds a given threshold, it will move the agent to another platform. This seems simple, but there are some problems using this kind of monitoring system. The routeplanningagent is very busy calculating and while it calculates, it does not check for new messages (like a move request). This problem is solved in two ways:
  - The routeplanneragent continuously needs to contact the mapagent and wait for its reply messages when planning a route. If in the meanwhile a jump request message is sent to the routeplanneragent, it will be stored in its message queue and will be before the mapagent reply. Thus the routeplanneragent will comply with the jump request before continuing with the calculations.
  - The procedure calls in the routeplanneragent are not like in a regular OO program. Instead of calling a local procedure by simply calling its name and passing argument *procedure(arglist))* the procedure is called by passing a message to it. This can be done by sending a message (with the procedure call) from the procedure that needs to call another procedure to its own agent. This can be done with:

    agentname→procedure(arglist)

    If during the execution of a procedure a jump request is received, this request will be stored in the message queue of the agent. When the procedure stops and calls another procedure with a message, this message will be stored after the jump request message. Thus the agent will move itself to another machine first before continuing with its calculations. This may seem a lot of trouble to implement the use of strong mobility in an agent but is actually necessary for the next case. There weak mobility and forced weak mobility will be used. Since borg has not all the features to stop an agent and empty its stack till the procedure call, we need to simulate those in the routeplanneragent itself. This is why the agent needs to be capable of receiving and processing messages at any time. In the jumpagent section we shall see that all this work is not necessary once borg supports all needed functions.

  Not only the routeplanneragent but also the monitoragent itself has this problem. The monitoragent is in a loop which continuously checks if the time or memory threshold has not been passed yet. Thus it is also incapable of receiving a stop-timer commando.

This is solved by not making the check loop with a standard loop statement (for, while) but with a procedure that constantly calls itself with a message containing a procedure call to itself. Once again if a stop-timer request is received, it will be stored in the message queue. When the procedure calls itself this call will be after the stop-timer request and thus the monitoragent will stop the timer. This stop-timer request simply sets a variable from true to false which makes that the loop-procedure no longer calls itself and stops:

```
monitorLoop():: {
    if(timePassed()>=max_time,orderJump(jumptoref));
    if(stacksize > max_memory,orderJump(jumptoref));
    if(stillmonitor,
        agentname->monitorloop(),
        display("Monitoring stopped."+eoln)
    )
}

stopMonitor():: {
    stillmonitor:= false
}

getStackSize(size)::{
    stacksize:= size;
}

startMonitor(agentref):: {
    display("starting monitoring..." + eoln);
    calcagentref:= agentref;
    stillmonitor:=true;
    starttime:=combineDateTime(date(),time());
    monitorLoop()
}

orderJump(machine):: {
    calcagentref->jumpTo(machine)
}
```

The monitoragent receives an updated size of the stack each time routeplanneragent adds an element to it to backtrack. It will order its calculating agent (in this case the routeplanningagent) to move to another machine when either the passed time is greater than the max_time or when the stack memory size(memory the calculating agent uses) becomes bigger than the max _memory treshold.

4.4.5 Program conclusions

This route planner case eats a lot of memory. The memory use is far from optimized and the procedures can be made much more performant. This may seem a bad thing but is actually done on purpose. This route planner has only a small database an thus memory and processor use will be limited anyway. With this case we try to simulate a full grown route planner that eats memory and processing power.

Before we can continue testing this case in different situations, we need to estimate the size of the route planner agent during its calculations. Since borg does not allow memory managing, the stack size is estimated. The elements stored on the stack are pico tables with 10 elements. This table contains start and endnode (6 characters each). It contains the route already constructed at that time which can contain 1 to 10 edges (maximum of 15 edges in databases but not all can be visited in the same path). So lets assume an average of 5 and a max of 10 edges are in this route table. An edge contains two nodes, 3 numeric values and a name: [”node01”,”node02”,12,120,0,”E40”]. If we say the name has an average length of 7 characters the average size of an edge is 6 + 6 + 2 + 2 + 2 + 7 = 25 bytes. This means 5 edges take 125 bytes The stack also contains a list of visited nodes that can go from 0 to 12 in our case. With an average of 6 and 6 characters per node such a visited table contains 36 bytes. The other fields are filled with current speed, length, cost and value of the current route in construction. These are all 32 bit integers and take up 8 bytes. The ninth element contains a list of alternative edges to use when backtracking. This list has between 0 and 4 edges maximum. With an average of 2 edges, this field takes 50 bytes. The last field contains the current node processed and the endnode taking 12 bytes. So in total each stack element takes an average of 12 + 125 + 36 + 8 + 50 + 12 = 243 bytes. The stack gets on average 11 deep which means its total size is 2673 bytes. When we monitor max stack memory use, we get around 210 average use per cell which is pretty close to our theoretical guess. These memory sizes may seem very small, but what if we have 150 in stead of 15 edges in the database?
Each element now takes on average 12 + 25*100 + 100*6 + 8 + 2*25 + 12 = 3184 bytes per element. The stack gets almost 7 to 10 times deeper with sizes easily passing 80. In this case the stack uses not a measly 3 Kb but a respectable 254 Kb. This indicates that memory use grows exponential in proportion with the total number of edges in the database. If we assume the agent itself with all other data takes at least another 20 Kb, we have an average routeplanneragent size of 23 Kb (mapagent contains 15 edges) and 274 Kb (mapagent contains 150 edges).

A fully grown routeplanner like route 66 eats uses 23 Mb of memory. If we assume it uses 3 Mb for GUI and other purposes this means that it requires 20 Mb to do its calculations. this may still be an acceptable amount knowing that a route planner like route 66 contains several ten thousand edges. The program code is also quite small taking some 5 Mb, but the maps are huge taking an enormous 500 Mb.

Luckily route planners for mobile device like Palm use smaller more local maps and thus much lesser memory. Let's assume that a modern route planner for Palm devices uses 1 Mb to store its map, 100 Kb program code (which is very large for a palm device) and 2 Mb work memory to do its calculations. With these numbers we can estimate how a real agent based route planner will react in the different situations below.

On a fast machine with a fast network:
On a speedy machine with lots of memory like a PC the route planner case runs smoothly. It has only a very small database and plans an optimal route in a reasonable time of about 20 seconds. If we make the calculating agent move to another machine (by setting a very low memory or time threshold in the jump agent) the agent will move quickly. In our case, the calculation agent uses max 274 Kb. A fast network easily allows transfer speeds of 300 Kb or beyond. This means that our routeplanneragent can be moved in less than one second using strong mobility. Moving the routeplanneragent offers no benefit in this kind of situation. When we set the monitoragent thresholds sharp enough, the routeplanneragent will be moved, which will cause a huge amount of network traffic because the routeplanningagent continuously needs data from the mapagent. Due to network delays, this will cause a tremendous performance loss since the routeplanneragent will have to wait continuously for data from the mapagent. When the mapagent is also moved, the performance of the program will depend on the speed of the machine the agents moved to. If this remote machine is faster than our local machine, there will be a performance gain, if not, a performance loss.

When the machine, our route planner case is running on, is fast enough to finish calculations in a reasonable time (lets say below 2 minutes), it is unnecessary to use strong mobility. Moving the routeplanneragent and mapagent will only result in a minor performance gain in the best case (mapagent moves along with routeplanneragent)and tremendous performance drops in the worst case (mapagent remains local).

On a fast machine with a slow network:
Here using strong mobility has even worse results than in the situation above. Lets assume that our slow network connection is a modern but slow 4 Kb/sec gprs GSM connection. Our routeplanneragent will take a full 274 Kb/4Kb/sec 70 seconds to transfer. This and the even larger network delays when mapagent and routeplanneragent need to communicate cause an enormous performance drop. This is best illustrated by an example:
Lets assume that our fast computer finishes calculations in just 2 minutes when all agents remain local. If we decide that we want to move the agent after 30 seconds, calculations will take 30 seconds + at least 70 seconds moving the agent. When the routeplanneragent has a mapagent on its new machine and starts calculating again at full power, it will still calculate at least one minute (if the new machine is faster). This means that total calculation time is no longer 120 seconds but 30 + 70 + 60 = 160 seconds in the best case!!

This and previous situation leads to the conclusion that strong mobility is useless on a fast machine. Its only purpose is to offload a processor intensive task (the route planning) to another machine. If this is done, the mapagent has to be moved along also or there has to be a mapagent on the same machine as the routeplanningagent. If this is not the case program performance will suffer greatly from network delays because of the intense network traffic between these two agents.

On a slow machine with a fast network:
Slow machines like a Palm device use a processor that is much slower than a desktop PC processor. Lets assume that when a route planning calculation takes 20 seconds on a PC, it will take at least 300 seconds or 5 minutes on a Palm device. In this case, strong mobility may definitely help.

Even when the network connection is fast (20 Kb/sec for a Palm device) the strong mobility of the agent will take some time. Moving the routeplanneragent will take 274 Kb /20 Kb/sec = 14 seconds to transfer. This is already much longer than our less than one second result we got when a PC was connected to a fast network, but it is still acceptable fast. If we set our monitor agent with a "max calculation time" threshold of 30 seconds the routeplanneragent will migrate to another platform once this threshold is passed. Will this moving have a performance gain in this situation? Lets take a look at another run of our program. We calculate 30 seconds, the agent is transferred in 17 seconds. If the mapagent moved along or another one is available on the machine, the calculations can finish at max speed within 18 seconds (10% or 2 seconds of the calculation work is already done on our Palm device) . Thus instead of 5 minutes we calculated a mere 30 + 17 + 20 = 67 seconds. This is an enormous performance gain of about 450%. If the mapagent does not move along, the calculations will take longer because of the intense network traffic between the two agents, but will still finish a lot sooner than when the routeplanneragent would have remained local.

We can conclude that in the situation of a slow machine with a fast network, strong mobility is very useful. The agent is transferred reasonably fast and calculations finish much sooner.

On a slow machine with a slow network:
In this situation everything changes. Our Palm device is now connected to the network using a slow GSM connection (4 Kb/sec max). We take the same example as in previous paragraph:
A route planning calculation that takes 20 seconds on a PC is performed on a Palm device where the monitor agent is set to move it after 30 seconds. In this case, moving the agent will take 274 Kb / 4 = 70 seconds. When there is a mapagent on the remote machine of the routeplanneragent, calculations will finish in 30 + 70 + 18 seconds = 118 seconds. This is still an acceptable result. When we take our larger Palm routeplanner with a 2 Mb routeplanningagent, it will take more than 8 minutes to transfer the agent.

If on top of that the mapagent remains local and no other mapagent is present on the machine the routeplanneragent currently resides on, finishing the calculations will take even longer leading to unacceptable waiting times.

We can conclude that it is possible to use strong mobility as long as the routeplanneragent stays below 400 Kb. If it becomes any larger than this, the time to migrate the agent will become too large. When the routeplanneragent is too large when we decide it to move, it might be better to abort the calculations, move the routeplanneragent using weak mobility and then restart the calculations. This is the so called forced weak mobility. Moving the routeplanner with weak mobility can be done in a few seconds. Once the agent is moved, calculations can continue much faster, certainly when the mapagent moved along or a copy is present on another machine. This idea will be worked out in the next case.

Route planner conclusions
Strong mobility has a strong advantage, but only when moving from a slow to a much faster machine and when network speeds allow the routeplanneragent to be transferred in an acceptable time. When the network is very slow, the moving takes too much time and the best solution would be to stop the calculations, move the agent using weak mobility and then restart the calculations on the remote machine. This solution allows the calculations to finish much sooner than when we use strong mobility.

It looks like strong and weak mobility alone have only an advantage in specific situations. Next case will check if it is possible to combine these advantages so the mobile property of agent offers a performance gain in more situations.

4.5 The advanced route planner

This case is the same as the previous route planner with one big difference; the intelligence of the monitoragent is much higher here. Instead of always using strong mobility once the max calculation time or max memory use threshold is passed, this agent will use a different approach.

Moving an agent using strong mobility to a faster system will indeed ensure that the calculations will finish sooner IF the network allows quick transfer of the agent and its stack. If the network is slow, this transfer alone may take several seconds or minutes, undoing any time gained by finishing the calculations faster on a fast machine. In such situations it can be better to finish the calculations locally and then move the agent using weak mobility. Then the agent can finish future calculations faster. An even better approach is to cancel the calculations, move the routeplanneragent using weak mobility and then restart the calculations on the faster remote machine. This may prove to be faster than using strong mobility on a slow network or waiting for the calculations to finish locally. This case will have an enhanced monitoragent which is capable of monitoring the network speed. This agent will allow us to find out which mobile behavior is best in a certain situation.

4.5.1 When to use strong or weak mobility?

The above case learned us that the network is a very important factor for this decision. If the network speed is fast, the routeplanneragent can be moved quickly and the calculations will continue much faster even if the mapagent remained local. If the network speed is slow, the moving will take much longer. At that point, it might be better to wait and finish the calculations locally if the memory is sufficient. If the memory is insufficient and a move is necessary on a slow connection, it may be better to abort the calculations, move the agent using weak mobility and then restart the calculations on the remote platform.

The choice between strong and weak mobility will depend on the following factors :

- Network speed
- Calculation time threshold
- Free memory on device
- Memory use of calculating agent
- The time it takes to move the agent (agent size divided by the network speed)

4.5.2 The advanced monitoragent

Our monitoragent has to be extended to be able to do this extra monitoring. Since borg has no methods for measuring the memory use of an agent or current network speed, those will have to be simulated. The monitoragent will get a higher intelligence checking several parameters before deciding how to move an agent.

A movetime function will be implemented which calculates the time it will take to move an agent. This is simply done by dividing the memory use of the agent with the network speed. The result is the time it takes to move the agent to another machine.
A emptystack procedure clears all agent work memory. This way we can move the agent optimally using weak mobility.

More parameters will have to be applied to the monitoragent in order to configure it correctly:

- maximum memory use of calculating agent
- minimum free memory on current platform
- maximum move agent time
- maximum calculating time

With the above functions and extra parameters, the intelligence of the monitoragent can be extended:

- if "max calculation time" is passed:
  - is "movetime" smaller than "maximum move agent time"
    - yes: Move calculating agent using strong mobility
    - no : stop calculations and emptystack (clear all calculation data) Move calculating agent using weak mobility
- if "current used memory" is larger than "maximum agent memory use" :
  - is "movetime" smaller than "maximum move agent time"
    - yes: Move calculating agent using strong mobility
    - no : stop calculations and emptystack (clear all calculation data) move calculating agent using weak mobility
- if "current free memory" is smaller than "minimum free memory:" :
  - is "movetime" smaller than "maximum move agent time"
    - yes: Move calculating agent using strong mobility
    - no : stop calculations and emptystack (clear all calculation data) move calculating agent using weak mobility

This intelligence for the monitoragent is, although not perfect, already much better than the intelligence used in the previous case.

4.5.3 Program conclusions

On fast machines:
We will not test fast machines in this case since they behave exactly the same as in the previous route planner case. There we saw that any form of mobility is useless on a fast machine unless we want to transfer the processor intensive tasks to offload the local machine.
We have to prove in this case that mobility can give a performance gain for agent programs on mobile devices.

On slow machines with fast network connection:
For this and following situation I configured the monitoragent as follows:

- Max moving time for our routeplanneragent is 30 seconds. This means that when our routeplanneragent is 274 kb, network speed has to be at least 274 Kb / 30 seconds = ∼9 Kb/sec. For our full grown route planner with an agent of 2 Mb, we state that the moving time may be at most 120 seconds. This means network speed has to be at least 17 Kb/sec.
- Max calculation time is 60 seconds.
- The memory settings have the same result than the "max calculation time" setting. Exceeding max memory use will have the same results than when the max calculation time is exceeded.
  The only difference is that the max memory use threshold can make the agent move sooner than the max calculation time threshold.

When our Palm device has a fast 20 Kb/sec network connection, we can see that both route planners can be moved in time. Lets assume that a calculation of a route takes 20 seconds on a PC and 5 minutes on a Palm device. Moving the routeplanneragent takes 274 Kb / 20 Kb/sec = 14 seconds using strong mobility. This is below our "max agent move" time, so strong mobility will be used when a threshold is exceeded. After 60 seconds only 20% of the calculations is complete when the "maximum calculation time" threshold is exceeded. Moving the agent is done in 14 seconds and the remaining 80% of the calculations is done in 16 seconds (if a mapagent is on the same machine as the routeplanneragent). Thus the route planning will take a total of 90 seconds which is far better than our 5 minute calculation time if the routeplanneragent had remained local. In this situation we have an 330% performance increase. This is 120 % less than in our previous case but still very acceptable. The lesser performance is because the monitoragent will wait longer before it moves the agent; 60 seconds compared to 30 seconds in the previous case. If the mapagent remains local or the routeplanneragent consults another map agent on another remote machine, performance gains will be smaller, but still well beyond 100%. In such situations, the performance gain greatly depends on the network speed between the machines holding the routeplanneragent and mapagent.

On slow machines with slow network connection:
In this situation our Palm is connected to the network using a slow 4 Kb/sec (GSM like) connection. We again take a route planning calculation that finishes in 20 seconds on a PC and takes 300 seconds on our Palm device. In this situation the "agent move time" is 274 Kb/ 4 kb/sec = 70 seconds and 2048 Kb / 4kb/sec = 500 seconds. In both cases the max move time of 30 seconds is well exceeded which means that weak mobility will be used to move the routeplanneragent whenever a threshold is exceeded. Our routeplanneragent calculates for 60 seconds, then "max calculation time" threshold is exceeded and the monitoragent orders the routeplanneragent to move itself using weak mobility. If we assume our routeplanneragent takes 50 Kb when idle and with empty stacks, it will take 13 seconds to move the agent. Once the agent has been moved, the calculations restart and are finished in about 20 seconds. This means the entire route planning procedure takes 60 + 13 + 20 = 83 seconds. If we compare this with a strong mobility approach in this situation, which would have taken 60 + 70 + 16 = 146 seconds to complete the calculations, this is a tremendous performance gain of more than 75%. This proves that it is better in some situations to stop the calculations and to move the agent using weak mobility

Here we can see that our advanced monitoragent definitely has advantages. An advanced intelligence in the monitoragent that carefully decides what kind of mobility to use in a specific situation can lead more frequent to performance gains than when we stick to one kind of mobility.

Some readers may argue that moving the routeplanneragent sooner or immediately after creation will have even better results. We want however the agent to be local as long as calculation delays become not too long.
Take a simple route planning calculation that needs only 20 seconds to finish on a Palm device and 2 seconds on a PC. When we set our "max calculation time" threshold to 10 seconds, the total calculation time will be :

- When using strong mobility: When the network connection is fast it will take 10 + 14 + X $>$ 24 seconds to finish the calculations, no matter how fast the calculations finish on the remote machine.
  When the network connection is slow, it will take even longer: 10 + 70 + X $>$ 80 no matter how fast the calculations finish on the remote machine.
- When using weak mobility: When the network connection is fast it will take 10 + 3 + 2 = 15 seconds to finish the calculations. This is indeed faster than our 20 seconds IF the routeplanneragent has a fast connection to a mapagent. When the network connection is slow it will take 10 + 13 + X $>$ 23 seconds to finish the calculations no matter how fast the remote machine is.

We want the agent to start calculating locally and to remain local as long as calculations can be finished in a reasonable time. Setting low thresholds will cause the routeplanneragent to be moved quickly. If a fast connection between the routeplanneragent and a mapagent exists, future calculation will indeed finish faster. If this fast connection between the agents does not exist, program performance for small route planning calculations may have been seriously compromised when using a small "maximum calculation time" threshold.

General conclusions:
Here it becomes obvious that the mobility property of mobile agents can give more often bigger performance gains. To accomplish this, the monitoragent needs a high intelligence that carefully chooses which kind of mobility to use depending on the situation. Even this monitoragent has some situations where it causes performance drops. These performance drops only occur when simple route planning calculations are done that require little more time than the "max calculation time" threshold to finish. In that case the time lost by moving the routeplanneragent causes performance loss. This can be resolved by giving the monitor agent an even higher intelligence. If it would be possible for the monitoragent to duplicate the calculation agent, this duplicate can be moved using weak/strong mobility depending on the movetime threshold. The original agent remains calculating locally until the duplicate has moved to another machine and finished its calculations there. In this case, the agent which finishes first wins. This would solve the last possible performance drops when using mobility.

4.6 The Jump agent

The first 2 cases (mailcase and meeting planner) only implement weak mobility. This to prove that weak mobility can be useful on mobile devices and is easy to implement. The third case (the route planner) is used to show the ease of using strong mobility, when it is provided by the platform, with mobile agents and that it can have advantages on mobile platforms. It contains a simple monitoragent that moves the routeplanneragent when either the max calculation time or max memory use threshold is passed. This very basic intelligence only proves useful in the case of a slow local machine with a reasonable fast network connection. This leads to the conclusion that using only strong mobility is not enough. Much better results can be obtained when using a combination of strong and forced weak mobility. The last case (the advanced route planner) uses this idea and has a far better monitoragent with higher intelligence than its predecessor. It uses strong and forced weak mobility depending on the threshold (hard or soft) and the network speed. With this monitoragent we obtained far better results in terms of program performance.

Finding out the optimal moving behavior for a calculating agent is not easy and depends on many factors. In order to make mobility easy to use on mobile platforms a more general form of this monitoragent has to be created, the **jumpagent**. This jumpagent is a more general form of the monitoragent of the advanced route planner with higher intelligence and configurability. This way, programmers can use it in every borg agent program that uses a processor, memory or network intensive calculating agent.

4.6.1 What to monitor:

This jump agent will have a more improved intelligence compared to the last monitoragent. It will move its calculating agent in following situations:

- maximum memory use of the calculating agent threshold is passed
- minimum free memory on current platform threshold is passed
- maximum calculating time threshold is passed
- maximum CPU time use threshold is passed
- maximum network use threshold is passed
- is weak mobility available for our calculation agent?

The first 3 situations are the same as in the monitoragent of our advanced route planner. The last two offer extra functionality that other cases may need:

- Maximum CPU time: this can help when more than one calculating agent is active. When one of the agents starts using too much processing power, it can slow down the entire program. In this case an extra function in the jumpagent will monitor its calculating agent and will move it if it starts taking too much CPU time. This threshold is user specific hard or soft which allows better use of this rule.

This feature is not necessary in programs like our route planner, since there is only one calculating agent, but may prove to be very useful in other programs with more than one simultaneous running process. Such programs can be games which can have a rendering, AI (artificial intelligence), network and user interface part.

- Maximum network use: network bandwidth is limited, certainly on mobile devices using mobile networks. The speed of such networks can vary from 1-2 Kb/sec (GSM) to 10-20 Kb/sec (wireless ethernet, bluetooth). Much higher speeds can be reached, but can not be handled by the internal processor of the mobile device. Since all network functions put a load on the processor it is also imperative to limit network use to offload the processor. The program speed can also be compromised when one agent claims most of the network and so takes the bandwidth necessary for other agents to communicate.
  This can be avoided by adding an extra function in the jumpagent that will move the calculating agent if it takes more than X % of the network bandwidth. This threshold is user specific hard or soft.
  This function of the jumpagent can be used in many programs. Even the route planner would benefit from it. This jumpagent will automatically solve the network speed problem when only the routeplanneragent is moved and the mapagent remains locally. The network cannot take the intense network traffic between the two agents and thus the program slows down considerably. In this case a jumpagent set to monitor the mapagent will move it to another machine when the calculating agent is moved and network traffic rises above the given threshold.

- Weak mobility available: not every calculation agent can just be stopped in the middle of a calculation. Certainly not if the calculation involves several agents which are strongly dependant on one another. This setting allows the user to specify if it is possible for the agent to just abort its calculations, move it using weak mobility and restart its calculations once moved.

The user will be able to give all above parameters plus some extras:

- Max agent move time: the maximum time it may take to move the calculating agent

- A critical minimum free memory barrier that has to protect the program in case the minimum free memory rule is still transferring an agent and memory runs dangerously low. In this case 2 things can happen:

  1. The agent will be moved using forced weak mobility if available.
  2. If forced weak mobility is unavailable, nothing will happen. Either the agent is moved in time and memory is freed or the program will stop.

- Weak or strong threshold: a strong threshold means the agent will have to be moved, no matter what the conditions are. A weak threshold will move the agent if conditions are optimal. Memory use thresholds are always strong since they can cause a program crash when exceeded.

4.6.2 The jumpagent intelligence

The intelligence of our jump agent is even more complex than that of the latest monitor agent:

The jumpagent has the ability to create a duplicate of its calculating agent. This duplicate is either moved to another machine or killed (depending on the situation). The duplicate and moveDuplicate(type) procedure have following intelligence:

createStrongDuplicate(agent)

- Is there enough free memory to create a duplicate of agent "agent"
  memory_use(agent) + current_memory_use <
  total_memory - minimum_free_memory
  - yes: Create duplicate
  - no: Do not create duplicate (fail)

createWeakDuplicate(agent)

- Is weak mobility available for agent "agent"
  - yes: Is there enough free memory to create a duplicate of agent "agent"
    memory_use(agent) + current_memory_use < total_memory
    * yes: Create duplicate, stop its calculations and empty its stack.
    * no: Do not create duplicate (fail)
  - no: Do not create duplicate (fail)

moveDuplicate(type,machine)

- is type of mobility = strong: Does the agent already exist on remote machine
  - yes: We cannot transfer the data of the calculation to start the remote agent. Kill remote agent and move duplicate agent.
  - no: move duplicate agent
- s type of mobility = weak: Does the agent already exist on remote machine
  - yes: Start the calculations on the remote agent. No move is necessary.
  - no: Move the duplicate agent and start calculations.

The thresholds for max calculation time, max CPU, network and memory use have all the same intelligence:

if threshold is passed and is a weak threshold:

- is "movetime" smaller than "maximum move agent time"
  - yes: is createStrongDuplicate a success
    * yes: move duplicate using moveDuplicate(strong,machine)
    * no: move original agent using strong mobility

  - no: is createWeakDuplicate a succes
    * yes: is "movetime" of the duplicate now smaller than "maximum move agent time"

      –Yes: Move the duplicate using moveDuplicate(weak, machine) mobility.
      –no: Kill the duplicate
    * no: do nothing.

If a duplicate agent is created, the agent which finishes first wins.

if threshold is passed and is a strong threshold:

- is "movetime" smaller than "maximum move agent time" or is weak mobility unavailable
  - yes: move agent using strong mobility
  - no: move original agent using weak mobility.

If a duplicate agent is created, the agent which finishes first wins.

The minimum free memory threshold is always hard. If this threshold is passed, it means that the calculating agent is large (eating a lot of memory) and free memory is most likely low. This low free memory may compromise possible other calculating agents so it is imperative that the agent is moved as fast as possible to free up some memory. If the agent will take too long to move and free memory is critical the program can crash. Since free memory is low, creating a duplicate agent is impossible and not wanted in this case. If the memory runs low the agent is moved in order to free up memory:

- if "current free memory" is smaller than "minimum free memory:" :
  Exceeding the minimum free memory threshold can compromise other programs and is thus a hard threshold in which case the agent has to be moved
- is "movetime" smaller than "maximum move agent time"
  - yes: Move calculating agent using strong mobility
  - no: Move the calculating agent using weak mobility. If weak mobility is not available use strong mobility anyway.
- if " critical minimum free memory" threshold is passed:
  If forced weak mobility is available, abort current move if not weak and transfer agent using forced weak mobility.

4.6.3 How to use the jumpagent

A jumpagent like the one just described can make the job of implementing mobile behavior for an agent easier but not easy. Agent based programs are distributed and thus need some proper thinking and design before implementation. A good strategy is an OO based approach with some extras:

- What are the needed parts for your program. Here the programmer identifies different functionalities in his program and decides how to separate them in different agents.
- Once the programmer knows what agents he/she will need, he/she has to work out the communication between agents.
- The programmer now has a good idea of what the processing and network requirements of each agent will be. At this point, he can think of what settings for the jumpagent would be best for each agent in his program. Agents that require a lot of memory need a carefully chosen max memory use threshold. Those which require a lot of processing power and are loaded on a slow machine need a carefully chosen "max calculation time" and "max CPU use" threshold. Agents that need a lot of network bandwidth are best of with a strong "max network use" threshold.
- After completing all previous steps, the programmer can start the design of the agents, programming their functionality, ...
- As a last step, the programmer can tweak the thresholds for maximum program performance.

The third step allows the programmer to configure the jumpagent that way it can ensure that its calculating agent is almost always in an optimal situation. As mentioned, an agent that requires a lot of processing power is best with a strong "max CPU usage" threshold so it will be moved to another machine in an early stage of its calculations when the current machine is either slow and/or loaded. Agents which need to have a fast response time no matter what the other conditions are, are best equipped with a strong "max calculation time" threshold. Agents which need a lot of network bandwidth preferably have a strong "max network use" threshold so the agent will be moved once the network becomes too slow to ensure optimal operation of the calculating agent. The most difficult agents to configure are the ones which require a lot of memory. Since a program can stop when it runs out of memory, the memory thresholds have to be carefully chosen.

This way, strong thresholds will ensure that the agents will always be in the situation the programmer thinks is best for them. The weak threshold can be used to further optimize your program. They are threshold like: "breaking this threshold may compromise the program's performance so try to move the agent but do not force anything". This means that weak thresholds will only move agents when the other situations are good and when the moving itself will not cause more trouble than not moving.

Designing the agents, determine their needs and configure their jumpagent's thresholds accordingly is one thing, integrating the jumpagent into your program is another. There are some rules the programmer has to keep in mind if he/she wants the jumpagent to do its job properly:

- Each calculating agent must use its own properly configured jumpagent. When the calculations start it will tell the jumpagent to start monitoring and when the calculations stop it will tell to stop the monitoring.

  It seems that a procedure to make the agent move is missing; such a procedure is in fact not necessary since a strong mobility move can be ordered from the jumpagent itself with a simple command:

  agentref->agentmove(machine)

  This commando will move the agent whether it is calculating or not. If the agent is not calculating the move will one using weak mobility; if the agent is calculating a strong mobility move will automatically be used. The commando's "stackTrash(agent)" and "clone()" in borg, can be used to empty the stack of an agent or to clone an agent (making a duplicate).

- In order to prevent program crashes due to memory exhaustion it is best that the calculating agents support forced weak mobility. Forced weak mobility will abort the calculations the calculating agent may be performing, empty its stack and move it using weak mobility. This way memory is freed and the agent is quickly transferred so it can resume its calculations. To do this all other agents involved in the calculations should check the status of the other agents on a regular time. When they see one agent's calculations have been aborted, they will all abort and restart so that no calculation anomalies will occur.

4.7 Thesis case conclusions

Out of all these cases we can draw a general conclusion. But first we go shortly over the case conclusions again.

4.7.1 Case conclusions overview

The Mailcase: here we show that weak mobility is really easy to use. On a fast network the agent is transferred quickly and can be viewed much faster once locally. On slower networks however even the weak mobility may become too slow for decent download times in which case it is better to download the mail content data in pieces of user defined length.

The meeting planner: the meeting planner shows a small performance drop once a slow mobile device is participating in the distributed system. It becomes obvious that more processor intensive distributed systems will suffer even greater performance losses once a slow machine is participating. Here we showed that this performance drop can easily be solved by moving all calculating agents to a faster platform. The only agents which remain on the slow machine are agents which communicate with the device or user itself. This case shows us that weak mobility enables slow machines like our Palm device to participate in a distributed system without causing a system wide performance drop.

The route planner: the route planner gives us a glimpse of the performance advantages we can get when we move the routeplanneragent using strong mobility. The case clearly shows that strong mobility can give tremendous performance advantages by moving the calculating agent from a slow machine to a faster one. Unfortunately it also becomes obvious that strong mobility can only be used when the network connection is fast. If the network speed is slow, moving a calculating agent using strong mobility can cause a performance drop instead of gain because the moving of the agent itself takes just too long. It also shows that the network becomes a bottleneck in a distributed program where two or more agents send a lot of messages: if the routeplanneragent and mapagent are on different machines, a lot of network traffic is generated between those machines because the routeplanneragent continuously needs data from the mapagent. The performance drop caused in this situation is more dependant on network delays than on network bandwidth. This is especially the case when small messages are used; only when large messages are used the bandwidth comes into play as well.

The advanced route planner: here the attempt to get rid of the disadvantages of the route planner is done by implementing a system that uses both strong and weak mobility. The case confirms that such an approach helps. The approach chosen depends on the time it will take to move an agent over the network. This means that the kind of mobility we use will depend on the size of the calculating agent and more importantly on network speed. On slow networks it is better to stop the calculations and move the agent using weak mobility. This causes less transfer delays that may cause a performance drop. On fast networks strong mobility is preferable since the agent can be moved fast, no matter how large it is and local calculation results are not lost.

Although moving the agents from a slow to a fast machine often results in a performance gain, there are still situations where we can note a performance drop. This indicates that there is still some improvement left.

The Jump agent: this improvement is attempted in the jump agent. This agent is an attempt to make mobility easier to use. The jump agent can be used in many programs to ensure optimal performance in different situations. The jump agent has a high level of intelligence and has many configuration options so a programmer can optimize it for a specific calculating agent. The intelligence of the jump agent is much higher and it has some tricks to further prevent performance loss in some situations (like duplicating agents). Most of the functions the jump agent uses are not implemented in borg. (network, CPU and memory monitoring, agent duplication, stack emptying, ...) The jump agent still leaves much room for improvements, but it already demonstrates its power. It also indicates that borg is still in a too early stage of development to take full advantage of the mobile property of its agents.

4.7.2 Conclusion: weak and strong mobility help

The first case leads to the conclusion that mobility is easy to implement. The second case showed that weak mobility can be used to allow slow machines like Palm devices to participate in a distributed program without a performance gain.

From the last two cases we can conclude that mobility is very useful to move calculating agents from slow to fast machines in terms of gaining program performance. It has however become clear that weak or strong mobility used separately can give great performance gains but only in very specific situations. The performance advantage can be improved and extended to more situations once we start monitoring certain key aspects like network speed, agent memory size, ... This monitoring combined with some intelligence allows the monitor agent to pick the right mobile behavior (none, weak, forced weak, strong) so that we experience a performance gain in more situations.

This intelligence is further improved in the jumpagent which allows programmers to use the mobile property of agents more easily. The jumpagent can be configured to give optimal performance for its calculating agent. The intelligence can be improved even further to allow for instance interactions between different jumpagents and their calculating agents.

The monitoragents and jumpagent show that borg is lacking some features that are necessary to take full advantage of the mobile property of agents. This is why I will present some needed future extensions to borg in the next chapter.

5 Extensions and future work around this thesis

This thesis has examined the use of strong and weak mobility of mobile agents on mobile platforms. As agent systems and mobile platforms are still developing, there is no doubt that there will be future work about this subject. This section of the thesis gives some pointers about some extensions that would put weak and strong mobility better to use in borg. It also gives some possibilities of future work around this thesis subject.

5.1 Extensions for borg

Borg has developed from a very experimental programming language in the powerful mobile agent platform it is now. [6] [9] Borg has made significant progress, but the work is far from being complete. Even now several extensions are made to borg and several others are planned.
In this thesis, we encountered several situations where we thought: "If we only had that function in borg we could...". So we decided to dedicate a small section in my thesis that contains pointers for extension to the borg virtual machine. Pointers that may enable future developers to make much greater use of the strong and weak mobility property of borg agents.

When using weak and strong mobility it became clear in this thesis that neither of them used separately offers only limited advantages. Weak and strong mobility can give much greater performance gains in programs when the moving behavior of agents is intelligent. This means that depending on several conditions the optimal agent move strategy will be chosen. Borg however has not the means to monitor those conditions. Since future users will most likely need them, it may not be bad if they were implemented in the future. These are the extensions I recommend:

- Agent memory monitoring: this is a function that returns how much memory an agent is currently using. If borg is to be used more frequently on mobile devices in the future this extension will become a necessity. On mobile platforms memory will always be more limited than on PC's due to space constraints. Programs that require lots of memory may get into trouble if memory runs out. Also the size an agent is occupying in memory is a good indicator on how long it will take to move it using either weak or strong mobility. This is why a memory monitoring tool for monitoring the memory use of an agent would be most helpful in borg.

- Free memory monitoring: this will be more useful on mobile devices than on PC's. This monitoring can prevent program crashes due to memory exhaustion. It can be used to force agents to stop calculating or to force them to move. This way much needed memory is freed and a program crash prevented.

- Network Monitoring: this might be more tricky to implement since this will involve contacting several "test machines" to determine network bandwidth and response time. Network speed is always very important when using mobility. Combined with the agent memory monitoring it can be used to give a good estimate on how long it will take to move an agent. This move time can then be used to make a decision on what type of mobility to use to move the agent in terms of program performance.

- CPU usage monitoring: this type of monitoring will be more useful on platforms which contain several calculating agents or on machines which are performing other tasks besides serving as an agent platform. This will allow agents to distribute processing power automatically. If one agent requires too much CPU time on a machine that is under heavy demand it can be moved to a machine that is currently idle. There it can have all the CPU time it needs. This feature will be more of use on PC's and servers which run multiple agents and programs than on mobile devices.

- Agent compression: this may solve the biggest disadvantage of mobility, and more specifically of strong mobility. Moving an agent requires the program to halt in most cases until the agent has been moved and can continue its calculations. For performance reasons it may not be bad to compress an agent (and its execution stack if strong mobility is used) before sending it over the network. This way the move time of the agent can be shortened considerably and program performance will increase. This is certainly true for slow networks where the time lost by compressing the agent is easily regained by sending the agent faster over the network. The agent compression should be optional and configurable (how heavy compression to use?). On fast machines with slow networks a high compression can be used. On mobile devices a medium setting may be advisable.
  If this is implemented the "agent memory use" monitoring could be extended to give a good estimate of how big the agent will be once compressed.

- Agent stack emptying: this is needed in the case where the jumpagent decides that it is better to stop the calculations of the calculating agent and to move it using forced weak mobility. In this case, all calculations have to be stopped and the calculating agent's stack has to be emptied. This can be done using the borg stackTrash function, but this function has a disadvantage. When we use stackTrash we reset the entire calculating agent his stack. This means we have to manually restart the calculations once the agent is moved. This causes the extra trouble of storing all the data needed to restart a calculating agent. It would be better if we had a stackTrash function that would empty the stack till just before the function call. This means the stack contains nothing but the function call. If we move the agent now it will automatically restart its calculations.

- Agent duplicating: this feature that enables agents to be duplicated while they are running can help to counter possible performance losses when using mobility. It can be used to duplicate the calculating agent. The duplicate is sent to a remote machine using strong/weak mobility. The original calculation agents remain calculating locally when possible and the agent which finishes first wins. This prevents performance loss in situations where the calculating agent takes only slightly longer to finish its calculations locally than the "max calculation time" threshold. In that case the original agent will finish before the moved duplicate agent.

5.2 Future work

This thesis has examined the use of weak and strong mobility for benefits on mobile platforms and has suggested some procedures and ideas to make optimal use of this property through the use of a jumpagent. This jumpagent is far from being perfect and far from being complete. Although programmers can configure all its thresholds it is still very basic and lacks real intelligence. This jump agent can easily be extended to have an even higher intelligence that will make the benefit of mobility even bigger. Possible extensions to the jumpagents can involve:

- Communication between the different jumpagents to allow an even higher intelligence. Such communications would allow a "clean" abortion of a calculation that involves multiple agents. In this case a jumpagent sends a message to all involved jumpagents that memory is running low and that the calculation has to be aborted.

- Intelligence to decide where to move the calculating agents. In this thesis we always moved agents from a machine to another predefined machine. This would be difficult to use since all the machines have to be manually entered. It would also undo the advantage of mobile agents to distribute processing power. An intelligence that seeks other remote machines and checks certain parameters on those machines (speed, load, memory, network speed,...) could allow better processing power distribution and thus better performance.

6 Thesis conclusions

As a final chapter we will sum up all conclusions made in this thesis.

In chapter 2, we examined what mobile computers are and what benefits/disadvantages they have. We concluded that these mobile devices like Palm Pilots are complete size and battery optimized. This is why they lack large memories, fast processors and advanced network hardware. We also learned that mobile devices are just at the beginning of their evolution. Next generation umts GSM's, GSM/PDA's and other will mark a new step in this evolution. These devices will have splendid multimedia capabilities and will have a permanent internet connection via the existing GSM networks.

Chapter 3 explained what mobile agents exactly are, what the borg platform is and how you can program agents in borg. We studied the advantages and disadvantages of mobile agents on both fast (PC) and slow (Palm) machines. It became clear that moving agents while they are calculating is a powerful property that has the advantage to offload processing power on a machine while the program is running. There are two major disadvantages which are inherent to a distributed system. A distributed agent program becomes largely dependant upon network speed and is more vulnerable to system/network failures.
Do we really need mobility to have the advantage of offloading processing power? To answer this question we compared mobile agents with two other widely known tools to make distributed systems: Java and client/server. Although Java can support weak mobility it became clear that offloading processing power while a program is running is impossible. Client/server lead to the same conclusion. So it seems that strong mobility has indeed a unique advantage.

In chapter 4 we created several cases in a test environment to prove that mobility can really be used to offload processing power:

- The first case, the mailcase, clearly shows that weak mobility is easy to implement and use.
- The second case, the meeting planner, showed that a mobile device can not join a distributed system without causing performance loss. By using weak mobility to transfer all calculating agents from the mobile device to another machine this problem is solved.
- The route planner case introduced strong mobility and gave a first glimpse of the possible performance advantages we can get when using mobility to move processor intensive agents to another faster system. It became clear that using one type of mobility has only performance gains in very specific situations.
- To solve this, the last case, the advanced route planner, used a higher intelligence to move the routeplanneragent. This intelligence checked the time it would require to send an agent over the network and decided what kind of mobility to use based on that.

Eventually, we try to make mobility easier to use by making a jumpagent. This jumpagent monitors a calculating agent and has a high intelligence that will cause even more frequent and better performance gains.

The last chapter concludes that borg is still too immature to take full advantage of the mobile property of its agents. In this chapter we suggested some needed extensions to make borg an even more powerful tool.

6.1 General conclusion

The use of mobility in agent based programs on mobile devices has definitely advantages. Moving agents using different types of mobility can give big performance gains. In order to have frequent performance gains we need to choose our type of mobility carefully. To make this decision we need an intelligence that monitors CPU, memory and network usage and that decides what mobility to use based on these factors. This intelligence is implemented in a jumpagent. This agent allows the intelligence to be used for a wide variety of calculating agents. This intelligence, although already quite complex, is far from complete and there is still a lot of work to be done on this subject.

References

[1] Links relating to general Palm info like the first Palm device, Palm device requirements, ...
http://www.palm.com

[2] Links to the general Psion site with all the info about Psion products
http://www.psion.com

[3] Links to articles who explain the PalmOS
http://www.palm.com/dev
http://www.palm.com/about/pr/background.html#Q2

[4] Links to PDA/GSM combi devices
http://www.nokia.com/phones/9210/index.html
This siemens PDA can have a mobile network or GSM module expansion
http://www.fujitsu-siemens.com/rl/products/handhelds/pocketloox.html
A pocket PC with build in GSM
http://www.my-siemens.com/sx45
Compaqs ipac, a very powerfull pocketPC that can be extended with network or GSM capabilitys through two compact card expansion slots
http://athome.compaq.com/showroom/static/iPAQ/handheld_jumppage.asp

[5] The AI arti lab at the VUB does some extensive research with agents
http://arti.vub.ac.be

[6] All information about borg can be found here
http://borg.rave.org/

[7] Links to the borg routing principle and working
http://borg.rave.org/research.html

[8] Link to the official VUB pico site
http://pico.vub.ac.be

[9] All code of borg is found here
http://borg.rave.org/cgi-bin/borgcvs/borg/
The borg core files are in cborcore, the user interface specific files are in the Xborg directories.

[10] Link to the borg source code of Pieter Verheydens meeting planner
http://borg.rave.org/cgi-bin/borgcvs/borg/examples/MeetingPlanner/

[11] Other thesisses about borg can be found here
http://borg.rave.org/cgi-bin/borgcvs/borg/documentation/thesis/
http://borg.rave.org/research.html

[12] Thesis Cedric Vanrykel : Quality of service voor mobiele multi agent systemen
Licentiaatsthesis, Programming Technology Lab, Vrije Universiteit Brussel, 2001-2002.

[13] Thesis Pieter Verheyden : Transparante foutentolerantie voor mobiele multi agent systemen
Licentiaatsthesis, Programming Technology Lab, Vrije Universiteit Brussel, 2001-2002.

[14] Here you can find links to articles explaining the working of Java and how you program in a Java environment
The Java architecture
http://www.artima.com/insidejvm/ed2/ch01IntroToJavasArchitecture3.html
General Java information
http://Java.sun.com

[15] Links to mobile Java programs and Java agent systems
http://www.cs.mu.oz.au/agentlab/javaresources.html
http://www.manning.com/Mahmoud/
http://www.it.iitb.ernet.in/ rahul/Links/Links.html

[16] Links to sites relating to Quality of Service
http://www.nortelnetworks.com
http://www.qosmagazine.net/What_is_Quality_of_Service_1.asp

[17] Links to sites explaining the different GSM technologies like gprs, umts, I-mode, ...
http://www.mformobile.com/

[18] Route planner software for the Palm device
http://www.123-gps.com/magellan-gpsvisor.shtml

[19] Here is a reference to other work being done on the subject of mobile agents on mobile platforms.
http://agent.cs.dartmouth.edu/research/Chen.html