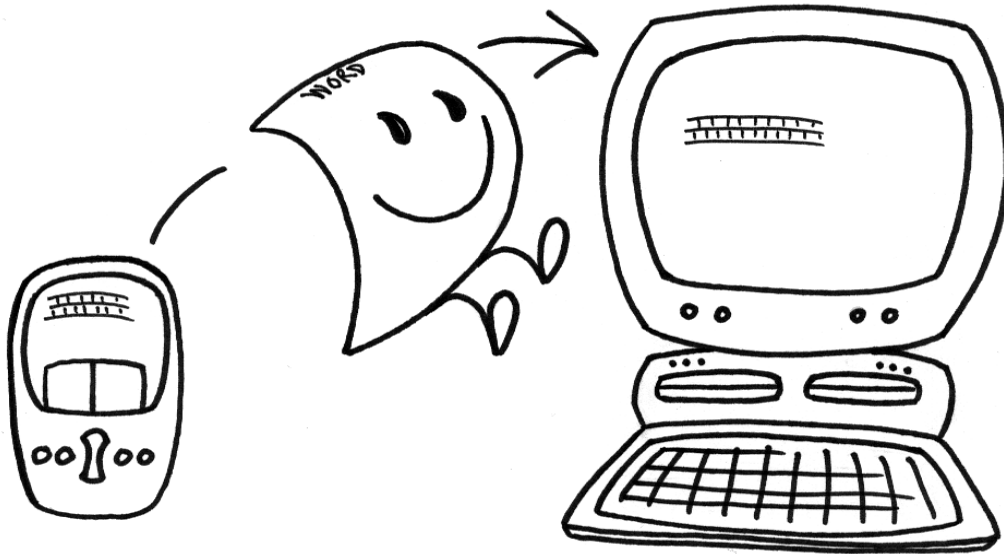


University of Aarhus
Department of Computer Science
Aabogade 34
8200 Aarhus N

December 20, 2000

Application Migration in a Pervasive Computing Environment

M.Sc. Thesis



Michael Tyrsted
Gustav Seth Wibling

Abstract

Ten years ago computers were mainly used at the desk in an office environment. With the advent of new computer devices specifically designed for a certain purpose it has become possible to use computers in a variety of new work situations. Handheld devices have made the user mobile and wall displays such as electronic whiteboards have made it possible for several users to interact using a computer.

In addition to the arrival of such devices new wireless network technologies have enabled devices to maintain a network connection in all situations. This combined with the existence of the World Wide Web have created a vision of a Pervasive Computing Environment in which the user can seamlessly shift between different devices and independently of the current device access his data and applications.

One requirement for this vision to come through, is that the software architectures support seamless shifts between different devices. Current software architectures do not support such seamless shifts. Instead they require the user to synchronize his data and install needed applications each time he shifts device.

In this thesis we make a thorough examination of the characteristics of Pervasive Computing Environments based on which we argue that migration of applications could be the central mechanism to support seamless shifts. In relation to this, we have investigated how migration is used in various systems covering process, object, and agent migration. Based on these investigation, we discuss the issues and propose an outline for the architecture of Pervasive Computing Environments. Through an experiment we demonstrate a way to seamlessly migrate applications between heterogeneous devices.

Contents

1	Introduction	1
1.1	Vision	1
1.2	Problem Domain	3
1.3	Taxonomy	4
2	The Pervasive Computing Environment	5
2.1	Mobile Computing Personae	6
2.2	Characteristics of a PCE	7
2.3	Communication	8
2.4	Mobility	10
2.4.1	Address Migration	10
2.4.2	Location-Dependent Information	11
2.4.3	Physical vs. Virtual Path Distance	12
2.5	Portability	12
2.6	Seamless Shifts	13
2.6.1	User Interface Adaption	13
2.6.2	Pervasive Applications	15
2.7	Summary	20
3	Communication Models for a PCE	21
3.1	The Client/Server Model	22
3.1.1	Properties of the Client/Server Model	22

3.2	Mobile Agent Systems	24
3.2.1	Properties of the Agent Model	27
3.3	Summary	28
4	Migration in Distributed Systems	30
4.1	LOCUS	32
4.1.1	The LOCUS Architecture	33
4.1.2	The Migration Mechanism	34
4.1.3	Residual Dependencies	36
4.2	MOSIX	36
4.2.1	The MOSIX Architecture	37
4.2.2	The Migration Mechanism	38
4.2.3	Residual Dependencies	39
4.3	Sprite	39
4.3.1	The Sprite Architecture	40
4.3.2	The Migration Mechanism	41
4.3.3	Residual Dependencies	43
4.4	Charlotte	43
4.4.1	The Charlotte Architecture	43
4.4.2	The Migration Mechanism	44
4.4.3	Residual Dependencies	46
4.5	Accent	46
4.6	Native Code Process Oriented Migration	47
4.6.1	System Architecture	47
4.6.2	The Migration Mechanism	47
4.7	Emerald	49
4.7.1	Emerald Objects	49
4.7.2	Object Migration	51
4.8	Migration Summary	53

4.8.1	Migrating Processes	53
4.8.2	Object Migration	57
5	Migration in Agent Systems	59
5.1	Mole	60
5.1.1	Agent Migration in Mole	61
5.1.2	Example	62
5.1.3	Summary	64
5.2	Ara	65
5.2.1	Agent Migration in Ara	65
5.2.2	Example	67
5.2.3	Summary	68
5.3	Migratory Applications	68
5.3.1	Obliq	68
5.3.2	System Architecture	72
5.3.3	The Migration Mechanism	73
5.3.4	Example	75
5.3.5	Summary	79
5.4	Comparison	80
6	System Architecture	82
6.1	Communication	83
6.1.1	Network	83
6.1.2	Location	85
6.1.3	Latency	86
6.2	Platform	87
6.2.1	Code	88
6.2.2	User Interface	88
6.2.3	Resources	89

6.3	Migration Mechanism	90
6.4	Pervasive Applications	92
7	The Experiment	97
7.1	The Migration Platform	97
7.1.1	Hardware	97
7.1.2	Software	99
7.1.3	Virtual Machine	99
7.2	The Inside of the Waba VM	99
7.3	Design of the Migration Mechanism	103
7.3.1	The Migration Server	105
7.3.2	The Migration Protocol	105
7.4	The Migration Mechanism	107
7.4.1	VM Init Data	107
7.4.2	Class Code	108
7.4.3	Migration of the Application State	108
7.4.4	Hooks	110
7.5	Evaluation	112
8	Conclusion	114
8.1	General Lessons Learned	114
8.2	Future Work	115
8.3	Conclusion	117
8.4	Acknowledgements	117

List of Figures

2.1	Client side proxy approach, [Casey, 95]	17
2.2	Server side proxy approach, [Casey, 95]	18
2.3	Proxies on all hosts approach, step 1, [Casey, 95]	19
2.4	Proxies on all hosts approach, step 2a, [Casey, 95]	19
2.5	Proxies on all hosts approach, step 2b, [Casey, 95]	19
3.1	The Client/Server model, [Coulouris et. al., 94]	22
3.2	Overview of an agent system, [Baumann et. al., 98]	25
4.1	Steps involved in remote tasking in LOCUS, [Popek & Walker, 85]	33
4.2	Remote Process Creation in LOCUS, [Popek & Walker, 85]	35
4.3	The three layers in the MOSIX Kernel, [Bharak et. al., 93]	38
4.4	Negotiation phase, [Artsy & Finkel, 89]	45
4.5	Transfer phase, [Artsy & Finkel, 89]	45
4.6	X: Global object, Y: Local object, Z: Direct object, [Jul et. al., 88]	50
4.7	Segmented Object Call Stack, [Jul et. al., 88]	51
4.8	Virtual memory transfer	55
5.1	Degrees of mobility, [Rothermel et. al., 97]	59
5.2	The Mole Agent's lifecycle, [Baumann et. al., 98]	61
5.3	Ara System, [Peine, 97]	66
5.4	An Ara agent, [Peine, 97]	66
5.5	Ara Core & interpreter, [Peine, 97]	67

5.6	Overview of Obliq semantics	70
5.7	Transmission of a closure in Obliq, [Bharat & Cardelli, 97]	71
5.8	Example of remote invocation and code transmission	71
5.9	Copy of both mutable and immutable nodes, [Bharat & Cardelli, 97]	72
5.10	Definition of <code>migrateTo</code> , [Visual Obliq implementation]	74
5.11	Definition of the <code>continuation</code> procedure, [Visual Obliq Implementation]	75
5.12	Definition of the <code>Agent</code> procedure, [Visual Obliq Implementation]	75
5.13	<code>CommentsForm</code> , [Bharat & Cardelli, 97]	76
5.14	<code>QuestionsForm</code> , [Bharat & Cardelli, 97]	76
5.15	<code>SuggestForm</code> , [Bharat & Cardelli, 97]	77
5.16	Distribution of application state	80
6.1	System architecture model	83
6.2	Different types of devices connected to the network	84
6.3	Platform architecture model	87
6.4	Relation between a pervasive application and a mobile agent	95
7.1	The Palm Pilot handheld	98
7.2	Format of an object on the object heap	101
7.3	Format of an array on the object heap	101
7.4	The native method stack frame	102
7.5	The virtual method stack frame	103
7.6	Migration of application from Palm to PC	104
7.7	left: Palm architecture model, right: PC architecture model	104
7.8	Format of a heap object message	106
7.9	The memory layout of an array containing 5 unsigned int16	109
7.10	The memory layout of an <code>mainWindow</code> object on the window platform	111
7.11	The memory layout of an <code>mainWindow</code> object on the Palm platform	111

List of Tables

- 4.1 Summary of the systems 54
- 5.1 Comparison of migration systems 80
- 6.1 Defining and characteristic properties of pervasive applications and mobile agents 95

Chapter 1

Introduction

In recent years, a variety of different computer supported devices have emerged. Computers are becoming more and more integrated into our every day life, with CPUs in everything from cars, coffee machines and radios to workstations and PDAs¹. Especially in computer-aided work situations a growing number of new technologies and devices have emerged. Examples of such devices are Palm Pilots², Windows CE devices as well as paper and whiteboards, augmented with computer support. Besides having devices everywhere, they will all be connected to each other through the Internet, making it possible to exchange data between any two devices.

1.1 Vision

All these connected devices imply new work situations. It is possible for the user to choose between different devices depending on the situation, e.g., a PDA when he³ is mobile and a electronic whiteboard when he is at a meeting. This implies that the user shifts between various devices according to the work situation.

These shifts happen throughout the user's work and may happen because the user moves on to a task which requires another type of device or due to the current work continuing at a new location. For the shifts to be as seamless as possible the user's data and his applications should be accessible independently of his location and the device he is using.

The architecture of current computing environments does not support seamless shifts, thus a new computing architecture is needed for this to be possible. A

¹Portable Digital Assistant

²Henceforth called Palms

³Male/Female

computing environment supporting seamless shifts between devices is henceforth called a *Pervasive Computing Environment* (PCE).

In order to give the user access to his applications and his data, a PCE should make use of a network. If the user is stationary the network may be wired as a LAN. If the user is mobile, the network should be wireless. In a PCE it should be easy for the user to shift between devices and the shifts should be as seamless as possible in order not to interrupt the user in his work.

To illustrate how we imagine a work situation in a PCE, we have set up the following scenario.

A software developer has been out of town for a modelling session with some domain experts. On his way back, he gets an idea of how to model part of the problem domain in a better way. He takes his PDA with his modelling tool, Knight [Damm et. al., 2000a], [Damm et. al., 2000b], and opens the diagram they had been working on. He sketches his ideas with the modelling tool on the PDA. Back at the office he moves Knight to the more comfortable desktop computer and continues his work there. At the next modelling meeting, his work is presented at the electronic whiteboard on which further modelling is done; still using the Knight tool.

In the scenario, the application follows the user independently of the device he is using. For this to be possible in a PCE, we have come up with the following three approaches:

1. The first approach is to have the user interface at the client device, i.e., the PDA or workstation, and the actual application at a central server in a distributed system, - somewhat like XWindows. This implies a need for continuous network communication between the client and the server. For mobile devices using wireless network technology it is unlikely that continuous communication can be maintained due to noise and other interference. Constantly communicating over the network is also costly in terms of battery which is a limited resource on mobile device.
2. The second approach is to migrate the state of the application from device to device and set up the application at the destination with the transferred state. This reduces the need for network connectivity, but requires the application to be installed on all devices the user might access.
3. The third solution is to migrate both the state and the code of the application. This demands more network traffic at the point of the migration but in this way there is no need for the application to exist on all platforms that the user wants to utilise.

We have conducted an experiment, where we have chosen to focus on the last approach since it, in our opinion, is the most flexible solution, and flexibility is a keyword in a PCE.

1.2 Problem Domain

The main focus of this thesis is how to migrate applications in a PCE. To understand how this is done we have looked at the characteristics of a PCE to find out what issues should be addressed in such an environment.

To gain a thorough understanding of how migration works, we have investigated how migration is used in different contexts, namely the Distributed Operation System context, the Mobile Object System context and the Agent System context.

Based on the knowledge gained from these investigations, we have proposed a System Architecture suitable for a PCE. To further understand how to migrate applications between heterogeneous devices we have made an experiment in which we migrate an application from one device to another. This involves extracting the application's state on the source device and adapting it to the destination device.

In the experiment we have focused on the problems involved in migrating an application with one process containing one thread, from a Palm to an Intel PC. The applications we migrate are written in Java and runs on the Waba VM⁴, [WabaSoft Inc., 00]. This has been done by modifying the Waba VM to make it capable of migrating applications. This involved investigating how to retrieve the application's state from the object heap, the native method stack, the virtual method stack, and the class heap.

In addition to this, we have looked at how to decide what data and code to move, since platform dependent data and code such as windows and fonts may not be compatible with the destination platform's. This implies investigating how to set up the application against platform-dependent code once it is migrated, ie., hook it up to a GUI window, fonts etc.

Another issue when migrating an entire application is efficiency. We have not addressed this issue, besides the above mentioned concern of only migrating the necessary information. This will therefore be a pointer to future work and will be discussed in chapter 8.2.

We have assumed that an underlying network for communication is present. As of now we use a serial link between the PDA and the PC. We also assume that

⁴Virtual Machine

individual applications do not communicate with each other and we have not investigated the problems concerning global file access or how to migrate open files.

1.3 Taxonomy

We need to define a few terms and concepts that we are going to use throughout the thesis:

- *Device* - The computer hardware the user interacts with. It consist of some input/output units, a processor, memory, storage, and other physical resources.
- *Virtual Machine* - Simulates a physical device that interprets and runs applications written in bytecode.
- *Platform* - The environment on which applications are running. It consists of both the hardware and all the underlying software such as the virtual machine, the operating system, drivers, interface to the communication layer, migration mechanism.
- *Host* - A platform that hosts a number of running applications.
- *Program* - A sequence of instructions describing how to perform a certain task.
- *Process* - The dynamic representation of a program being executed.
- *Application* - A special type of program that is intended for a human user. It contains a user interface and is a tool that the user can use to perform a certain task.
- *Location* - Can either be virtual or physical. A virtual location is the location of a device or a user in a PCE. A physical location is the location of a device or a user in the real world.
- *External resources* - A resource external to the Virtual Machine in which the application executes. Examples of such resources are files, services, printers, sockets, screens, and windows.
- *Autonomous entity* - An entity which is independent/not part of the surrounding environment.

Chapter 2

The Pervasive Computing Environment

The new buzz-word in computer science is pervasive computing. But what does it mean? According to [Dictionary.com, 00] being pervasive means: “Having the quality or tendency to pervade or permeate: *the pervasive odour of garlic*”. [Merriam-Webster, 00] says: “To pass or spread through the whole extent of; to be diffused throughout”.

From these definitions, a PCE could be conceived of as an environment where computers can be found and used everywhere, just as electricity or engines [Weiser, M. 91]. For a computing environment this means that not only can the computer hardware be found throughout the whole environment, but also the software. Applications and computing services will pervade computing systems and environments available to the user anywhere, any time.

In the last couple of years a lot of research projects concerning a PCE have been started, e.g., IBM Pervasive Computing [IBM PCE, 00], CoolTown at HP [HP CoolTown, 00], IPCRES at Indiana University [IPCRES, 00], and lately also at Daimi and CIT with “Centre for Pervasive Computing” [CIT, 00].

According to IBM the users of a PCE require:

“complete access to time-sensitive data, regardless of physical location. We¹ expect devices – personal digital assistants, mobile phones, office PCs and home entertainment systems – to access that information and work together in one seamless, integrated system. Pervasive computing can help us manage information quickly, efficiently, and effortlessly.”

¹the users

They also say that a PCE should:

“give people convenient access to relevant information stored on powerful networks, allowing them to easily take action anywhere, anytime.”

In the CoolTown project at HP the idea is that a PCE is tightly connected to the internet. In the project, people, places, and “things” all have Web pages and are networked together through the World Wide Web.:

“In this world, Web servers will be embedded in a wide variety of places and devices, which will be able to transmit information to each other.

For example, a person wearing a wireless information appliance in a mall could automatically receive information about products and services of personal interest just by entering the Web “space” of the mall. A jeans store with its own Web server could let a customer know it has his favourite style of jeans, in the right size, in stock and on sale.

Devices could also interact independently with each other. A “smart” alarm clock that knows the user’s schedule and route to work could be linked to traffic and weather reports so that it could wake him sooner if he had an early-morning appointment and there was a problem on the freeway that would prevent him from arriving on time.” [HP News, Nov. 4, 99]

Another idea of the pervasive environment and the vision of a *Mobile Computing Personae* is presented in the article [Banerji et. al, 93], described next.

2.1 Mobile Computing Personae

The vision is based on a scenario in which a user is writing a paper for a conference. This work is done using both his home computer, his UNIX workstation at the office, and his PDA on the bus and in a colleague’s office. Generalising on this vision, the paper, [Banerji et. al, 93], argues that we are moving away from machines with multiple users to users with multiple machines. This new work situation affects the way users think about and use their computers, and thus new approaches are needed to support this change.

When a user works, he runs applications, specifies preferences for them, and is allowed to access resources and files according to local rules. These elements, together with the name-to-resource mappings, can be referred to as a user’s *Com-*

puting Persona (CP). Today when a user is using different machines, a new CP is established on every machine he uses. During execution this CP will change state when applications are started, environment variables are set etc. When the user switches from the current computer to another all the information in the CP is lost. Since the user often switches working computer it would be beneficial to have the environment preserved between these switches. In this way, the switch would be more seamless and less annoying to the user. In [Banerji et. al, 93] it is stated that the user is interested in having a consistent working environment between all his devices, regardless of the connection between them. To achieve this the concept of a *Mobile Computing Personae* (MCP) is invented.

The MCP is an environment, which is established the first time a user logs into a computer. It includes a presentation of the system resources and environment, such as Windows or the Macintosh desktop. As well as the resources available to the user and the mechanisms employed to access them, such as a complete, well-ordered name mapping to files, devices and other resources. This is the same as the CP.

The new idea is that when a user logs into another computer the MCP is capable of migrating from the previous computer to the new one and in this way the user is capable of seamlessly continuing his work from the point where he left off. This requires the MCP to be migrated automatically by the underlying system. How this is done is explained later in section 2.6.2.

2.2 Characteristics of a PCE

Based on the above examples we try to decide what characterises a PCE. First of all computers are everywhere and some of them will typically be portable, e.g., a PDA, laptop, or mobile phone. All of these devices are connected by some kind of network technology. The mobile devices are typically connected by wireless technology, while the stationary devices are continually connected by some wired technology, ie., we have a mixture of different networks. We must also anticipate that users move around with their portable devices. They may also access different types of devices and a device may be used by different users.

Apart from the above characteristics, there are also some common demands placed on a PCE by the users. The users expect to have access to their data and their applications no matter where they are, also during transit. They also expect to be able to seamlessly switch device with their current applications following them, ie., they do not want to worry about synchronising their data, setting up their applications or installing applications at new devices.

In the following we will investigate some of the issues, that must be addressed

in a PCE. We will also look at some systems that address these issues. In this chapter we try to be as objective as possible and only present the issues and some of the solutions addressing them.

2.3 Communication

In order to provide access to a user's data and applications, the devices need to communicate. This is typically done using network technology. Current distributed systems already make extensive use of networks to provide distributed access to resources, such as files and applications. However, most distributed systems mainly consist of stationary devices, such as workstations and mainframes, which can be given a fixed address and have very fast and reliable network access. In a PCE, a lot of devices will be mobile. Because of this, wireless network technology is often used to communicate. When using wireless communication, the surrounding environment will interact with the signal, both blocking it and introducing noise and echo, [Forman & Zahorjan, 94]. This results in low bandwidth, high error rates, and frequent disconnections, which in turn gives higher communication latencies, retransmission timeout delays, error control protocol processing, and short disconnections. This results in a wireless connection being of lower quality than a wired.

Apart from this, a mobile device may also lose a connection if the user moves out of the area of coverage or moves into an area with high interference. The number of devices using a specific connection point², may vary dynamically so a user might experience decreased throughput because he has to share the network capacity with more users than previously.

In the rest of this section we will look at the issues *disconnection*, *lower bandwidth*, *high fluctuation rates*, *network heterogeneity*, and *increased security risks*.

Disconnection. In order to cope with disconnections at least two approaches can be used. The first is to use additional resources to make the network more reliable, and thereby prevent disconnections. This could be done by providing better coverage by setting up more or stronger connection points.

If the first approach is not possible either due to money or size of area, a second approach could be to work around the disconnections instead of avoiding them. This can be done by making the device more autonomous, i.e. less dependent on other resources such as the network. When a lot of disconnections occur, the device would then be capable of acting as a stand-alone computer. One thing that

²The stationary transmitter currently used by the mobile device.

may be done to make a device more autonomous is using a mobile file system for file handling, like Coda [Jing, 99] or Mobile File System [Saldanha & Cohn, 94]. In Coda, caching of files is used to give the user file access when disconnected. Information from the user's profile is used to predict what files to cache. The files are then automatically reconciled when the user connects to the network. If a file has been modified by another while the current user was disconnected, the reconciliation fails, [Kistler et. al, 92]. Another system, capable of working in disconnected mode, is the email protocol IMAP, [IMAP, 00]. IMAP uses a cache to store a user's emails when disconnected. When the user reconnects the cache is reconciled with the user's central IMAP service, that holds all information about mail folders, emails received etc. By using IMAP it is possible to access ones emails from an arbitrary location via the Internet. It should be mentioned that the problem solved by the IMAP protocol is simpler than the problem concerning file reconciliation. This is due to the emails in the IMAP system being read-only.

Lower Bandwidth. The wireless network technology as it is currently provides lower bandwidth than the wired technology. Moreover, because of the dynamic number of users sharing a given connection point (and its capacity), the bandwidth available to the user may change dynamically. A way of addressing this could be by providing additional connection points to the wired network to provide sufficient bandwidth. Another way would be to use compression or pre-fetching. Pre-fetching means retrieving data before it is requested by the user. In this way the information is available when the user needs it, even though the connection is down. Pre-fetching, however, can result in lower throughput because of bad guesses, ie., fetching something not used. Intelligent communication scheduling could also be used, in which the applications currently used by the user are given higher priority than background processes, [Forman & Zahorjan, 94]. If the bandwidth suddenly drops, bandwidth is passed to the user applications, and in this way the user does not experience the drop. The other processes can then be served when the user does not need the bandwidth.

High Bandwidth Fluctuations. The fact that the bandwidth may change radically during an application's execution, either because of the dynamic number of users currently competing for it, or because of the user moving into an area with bad coverage, should be considered when designing applications for a PCE. Three approaches can be used to address this. First an application may choose to assume a high bandwidth and not work when the required bandwidth is not available. Second it may choose to assume a low bandwidth which it is very likely to be granted at all time and finally it may chose to adapt to the bandwidth available. The approach taken depends on the needs of the application.

Network Heterogeneity. A mobile device should feature several different methods for connecting to a network. This is due to the mobile device being likely to encounter different network types when being used in different locations. It may, e.g., encounter a network operating via an infrared interface when being used inside a building. When moving outside, this technology is no longer useful and the mobile device must therefore change to some radio-based network technology such as Bluetooth [Bluetooth, 00] or DECT [DECT, 00]. When being situated at a certain location for a longer period of time connecting the device to the local network would save battery and achieve a better, more reliable connection.

Increased Security Risks. Because of the ease by which any person can connect to a wireless network, the risk of an intruder tapping is higher than for a wired network. When a user is mobile, there is also a question of how to give him the correct access rights when changing from one security domain to another.

A solution to the problem concerning tapping could be encryption of all data sent via the wireless links. This encryption could be done either by hardware, e.g., the CLIPPER chip, or in the software [Forman & Zahorjan, 94]. To authorise a user, MIT's Kerberos and its secure authorisation services, [Neuman, 91], could be used. More issues are addressed in section 2.5.

2.4 Mobility

The fact that a user moves around with his device raises several issues that must be addressed in a PCE. Information that is considered static for stationary computers is suddenly becoming dynamic, e.g., information about the server through which the device accesses the net. This server can be specified once and for all on a stationary computer, but a mobile computer should preferably use the nearest, and therefore this information will change during the life of the mobile device. Some of the issues we must address are *address migration*, *location dependent data*, and *physical path versus virtual path distance*.

2.4.1 Address Migration

When a device connects to a network it is typically assigned an address at which it can be reached. This address is typically based on the current subnets address domain. Also when a system knows the address of a host, it caches the address to ease further requests to this host. When a device is mobile this can be a problem, since the device may be moved from one subnet to another and thereby be

assigned a new address at which it can be reached. This can happen even during the processing of a request. To address this issue, [Forman & Zahorjan, 94] propose four solutions:

Selective Broadcast. Broadcasting the message to the mobile device will typically ensure that the device gets the message. However on large networks this is probably a bad idea because of the traffic overload. Instead the message could be selectively broadcasted to the cells (wireless connection points) next to the one the mobile device was last connected to. There is a good likelihood for the device being connected to one of these now.

Central Service. A central service is another solution, in which the address of the mobile device is maintained by the central service. Here the mobile device contacts the central service each time it changes address. Replication could be used to make this approach more reliable. The approach is similar to the one used to update Domain Name Servers on the internet which all are synchronised twice every 24 hours. However, when a lot of devices become mobile, the synchronisation will have to be done at a much more frequent rate.

Home Bases. This is a special case of the above central service approach, in which a mobile device is assigned a home base. Each device will then report its current address to the home base when moving. This way the home base can be used to locate the whereabouts of the mobile device.

Forwarding. To avoid having a central service or guessing like in selective broadcast, forwarding can be used. When a mobile device is assigned a new address, the old subnet's router can be told and the router can then forward the messages it receives to this new address. To avoid too many forwarding points the router could send an update message to the source telling it the new address of the mobile device. The source could then update its cache. The greatest disadvantage of this approach is that some active entity must be present at the old address to forward the messages. This may not always be the case, but typically the mobile device will know its last local router, which can then be updated with the address of the new local router.

2.4.2 Location-Dependent Information

Some information only makes sense at a certain location. This could be information concerning the nearest printer and name server. The mobile device should

somehow adapt to its local surroundings in order to serve the user in the best possible way. One way of doing this could be having the connection point update the device with local information when the device started using it.

2.4.3 Physical vs. Virtual Path Distance

When a mobile user changes location, the device might choose to use a new connection point due to the new connection point being closer. However, this may not always be beneficial since the new connection point may belong to a different router, thereby providing a much longer communication path for the device. The longer communication path will imply longer response times and higher risks for network failures.

Another related issue that must be considered is that when changing to a new connection point, the device may change to a different *virtual location*, [Cardelli, 99]. Virtual locations exist on all wide area networks, due to the presence of potential attackers. They are created by erecting barriers, e.g., firewalls, between the mutually distrustful domains, [Cardelli, 99]. If a device suddenly moves to a new virtual location it may not be possible to access resources available at the previous virtual location. Also some kind of authorisation of the device must be available in order for the device to access resources in the new virtual location. We imagine that some kind of virtual passport could be used in the same way as the physical passport to identify the device, when moving into a new domain. This passport may also be used when accessing resources in a *virtual location* other than the current.

2.5 Portability

In order for a device to be portable, some physical properties cannot be overlooked. Due to the size and weight constraints of the portable device, some resources are scarce. Screens have to be fairly small which affects the input methods; typically a pen is used for input instead of a mouse. It might also be favourable to trade buttons for gesture or voice recognition. Handwriting instead of keyboard typing is also an option. All these factors should be considered when designing software for a portable device.

Currently the battery power is often limited, since the battery should be as small and light as possible. This affects other resources such as processor, memory, storage, and resolution of the screen³, which are typically limited compared to a

³the screen size of a Palm is 160x160 pixels whereas we can expect more than 800x600 pixels on a desktop computer

stationary workstation.

Some approaches usable, in order to save power, are for an application to listen instead of talking over the network, since talking requires as much as ten times as much power, [Forman & Zahorjan, 94]. Spinning down a disk after a few seconds of idleness can also help save power according to [Li et. al. 93].

Portable devices also typically feature a limited storage space, so applications should address this as well. This could be done by using compression or by writing the application in a script language since the size of compiled code often is larger than script code.

However if the development of devices continues as until now, some of the differences between stationary and portable devices will be diminished. It is already possible to get devices with over 200MHz processors and more than 200 Mb of storage. According to [JP, 22/11/2000] portable devices will, within a few years, have processing power above 1 GHz and constant internet connection.

Another issue that should be addressed is the security of portable devices. Because of their size and portability, portable devices can easier be stolen or lost. Some kind of access control or other security precaution would be beneficial in order to secure the data of the portable device.

2.6 Seamless Shifts

The above issues have mainly been concerned with the mobile devices part of a PCE. Seamless shifts, however, apply to all the devices part of a PCE. The users of a PCE should not have to worry about how to change from one device to another. In other words it should be seamless how the shift is accomplished. Several issues must be addressed in order to achieve this. First of all some user interface adaption must be performed when changing from one screen size to another, ie., from a Palm's screen to a workstation monitor. Second the issue of how to make the application follow the user around must be addressed. Other issues, such as how to communicate, are addressed later in chapter 3 and chapter 6.

2.6.1 User Interface Adaption

When an application is displayed on a workstation monitor it will often make use of several windows, buttons, text fields, etc. This all works well because of the monitor screen size and the way with which we interact with the workstation, using mouse and keyboard. In a PCE the user might choose to change to another

device such as a PDA while executing the previously mentioned application. Due to the limited screen size of the PDA some adaption of the user interface must be performed. This adaption could be mapping the active window into the current window on the PDA and hide the other windows. Also buttons and the like may have to be mapped into an pop-up menu to fit into the new screen size. Several methods for mapping user interfaces can be used. In the following we will present two approaches, namely the one presented in 2K, [Román et. al., 99] and the one presented in a project on a Waste Water plant, [Nielsen & Søndergaard, 99].

2K. 2K is a distributed system, in which users are given seamless access to their data and applications either via stationary computers or Palms. This is done using componentised applications, adaptable middle-ware, and standardised protocols. Since the data and applications are accessible from either a PDA or a stationary computer, it is possible for the user to access it while being mobile. To accomplish this they have developed a PalmORB client, which can be used to issue requests to the wired part of the network. This part consists of workstations on which DynamicTAO, [DynamicTAO, 00], is running. DynamicTAO is an adaptable ORB used for interacting with the Palm clients. All exchange of information is performed through these ORBs.

User interface adaption is not yet implemented in the 2K system. However they propose a solution in which XML documents, [XML, 00], are used to specify the user interface components and then use device-specific *Cascading Style Sheets*(CSS), [CSS, 00], for showing these components on a given device. In this way, the application would send the same XML pages to the device not having to worry about the device's type, e.g., whether it is a small handheld or a white-board. The device will then interpret these pages according to its specific Cascading Style Sheets.

One of the application in 2K, a video streaming application, does feature some kind of user interface adaption. The adaption is done by adaptable proxies which relieve the Palm of some of the computation involved in showing videos. The proxy resides on a powerful workstation and does therefore have a substantial amount of resources available compared to the Palm. Due to the Palm only being capable of showing four different scales of gray the extra colour information is removed by the proxy on the workstation. The proxy will also decompress the video frames, since decompression demands a fair amount of computational power. The frame rate and size are also reduced by the proxy to fit the Palm. Finally it should be mentioned that the proxy is also capable of handling disconnections by acting as the Palm client when the client itself is disconnected. The information received during disconnection is cached until the Palm client reconnects.

Waste Water Plant: Interface Adaption. In [Nielsen & Søndergaard, 99] the work process at a waste water plant is analysed. The reason for this analysis is that the inspectors at the plant often find themselves needing information from the central computer, in the office, when they are on their inspection rounds on the plant. To make this possible, using PDAs is proposed as a solution to access information while not in the office.

The analysis also shows that in order for the inspectors to trust the information they receive at the PDA the user interface at the office computer and the PDA should be similar, or “tightly integrated” as it is called. However it is not needed for the PDA user interface to be an exact miniature copy, but using some similar components are useful to couple the user interface on the PDA to the one on the PC thereby making it easier for the users to relate the two systems.

Apart from using similar components to relate different user interfaces we find that the way of interacting on various devices should be alike for a better user understanding. Typically it is more difficult to relate two programs with different user interface interaction methods than if the interactions are of the same kind. It does for instance not help if the interaction on one device is based on toolbars and pointers and on another mainly on gestures. However using the same interaction method on all devices may not always be possible. For instance on a mobile phone the size constraints limit the possibilities for interacting.

2.6.2 Pervasive Applications

Ordinarily, when a user logs into a device, a computing environment is built according to some preferences specified by the user’s startup files. If the user decides to continue his work on another device, the process is repeated, and the user has to restart all his applications, load data etc. In a PCE this should not be necessary. Instead the user should have the feeling of the applications “following” him around. This is what we call a *Pervasive Application*. One way of accomplishing this is presented by [Banerji et. al, 93] in section 2.1 above. Here the key component is the MCP which contains the state of the user interface, the active applications, hidden services, environment variables, and other settings of the user. If, at some point, the user becomes inactive, all this information is stored away in files created for the same reason. Through these files, the exact state can be reestablished when the user becomes active again. This means that if the user decides to change from the current device to another, the MCP is stored and sent to the new device where it can be restored. This involves saving the applications’ states, which are then loaded into the equivalent applications on the new device in order for the user to be able to continue from the point where he left off. This idea is similar to the one used in Sunray appliances, [Sunray, 00], in which the user’s state is stored on a smart card. The smart card is carried

around by the user and when inserted into a Sunray appliance the user's state is restored. This is possible by having all Sunray appliances connected via a LAN and then running applications centrally on a Sunray Server thereby ensuring that the same applications are available at the different Sunray appliances.

A more direct implementation of the MCP idea is proposed in [Casey, 95]. According to the paper the first thing needed to create a PCE using a MCP is global access to files. To achieve this, a mobile file system is suggested, such as Coda or Mobile File System mentioned earlier. By using a mobile file system, access to files can be achieved even during disconnections. [Casey, 95] does not implement a file system but assumes that an adequate one is to his availability.

Apart from file access, some way of storing the state of the running applications is needed⁴. Here [Casey, 95] uses checkpointing. When a checkpoint is issued each application is responsible for saving the part of its own state necessary for it to be restarted with the same look and feel. By making it the application's responsibility, it is argued that restarting the application on another, possibly different, device is reasonable easy since the application explicitly knows what information is needed and what is not. However the applications must of course agree on the format in which the information is stored.

In order for the MCP to follow the user around each device features a **Persona Manager**. The **Persona Manager** takes care of mapping the MCP into the specific device's environment. It can also request a MCP from another device by contacting the device's **Persona Manager**. The requested **Persona Manager** will then issue a checkpoint after which it will send the information to the requesting device.

Some information cannot be stored by the application, viz. site-specific information about files and sockets. This information must be specifically addressed by the system. For files, [Casey, 95] assumes that their state is handled, ie. migrated, by the mobile file system. For sockets, the site specific information is the IP address denoting the current host. This information is not useful at a remote host and should therefore be transformed when migrated. [Casey, 95] suggest extending the current TCP/IP suite with *mobile sockets*. Mobile sockets should be capable of moving along with an application, when it migrates and be able to change their address according to the host they reside on. As a temporary solution until mobile sockets become a part of the TCP/IP suite, [Casey, 95] suggests using communication surrogates. Three suggestions are presented. In the first setup, a surrogate exists on the client host and the client application communicates with the server through the surrogate, see figure 2.1. The server

⁴It is important to stress the fact that [Casey, 95] does not migrate applications, but only their state. This means that applications must be available on the receiving platforms before the state is migrated.

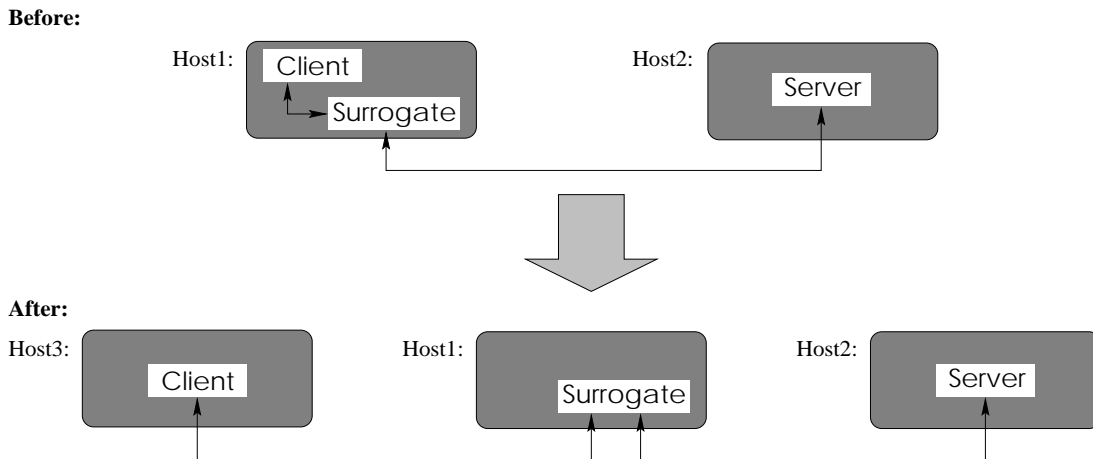


Figure 2.1: Client side proxy approach, [Casey, 95]

communicates with the client through the surrogate. However the server is not aware of that it is not speaking directly to the client application. When the client migrates to another host (host 3), the server will not notice due to the surrogate buffering and acknowledging the requests sent by it during the migration. When the migration is finished, the buffered data is requested by the client migration on the new host. The disadvantage of this approach is that all traffic must go through the previous client host and if this is a PDA it is probably not a good solution.

The second approach is to have the surrogate live on the server, figure 2.2. In this way, the server application will communicate through the surrogate and the surrogate will be able to respond to the server when the client application is migrating. The client application will then reconnect to the server every time it has migrated and the surrogate can then forward all buffered data directly to the new site. In this approach, the server application may also migrate and let the surrogate forward requests to its new destination. This is a very centralised approach in which everything revolves around the surrogate.

If both the server application and the client application are meant to be mobile, another more flexible approach is possible. Here a surrogate resides on all platforms in the network, figure 2.3. Each time an application migrates it contacts its local surrogate and asks it to buffer arriving data during the migration. When the application has migrated it tells the remote application of its new whereabouts, after which it connects to the old surrogate to get the buffered data, figure 2.4. It may also ask the remote application's surrogate to buffer data for it, instead of its local surrogate. When it has migrated it now only has to reconnect to the remote application and ask it to send the buffered data, figure 2.5. From then on, the remote application knows how to send requests to the migrated application's

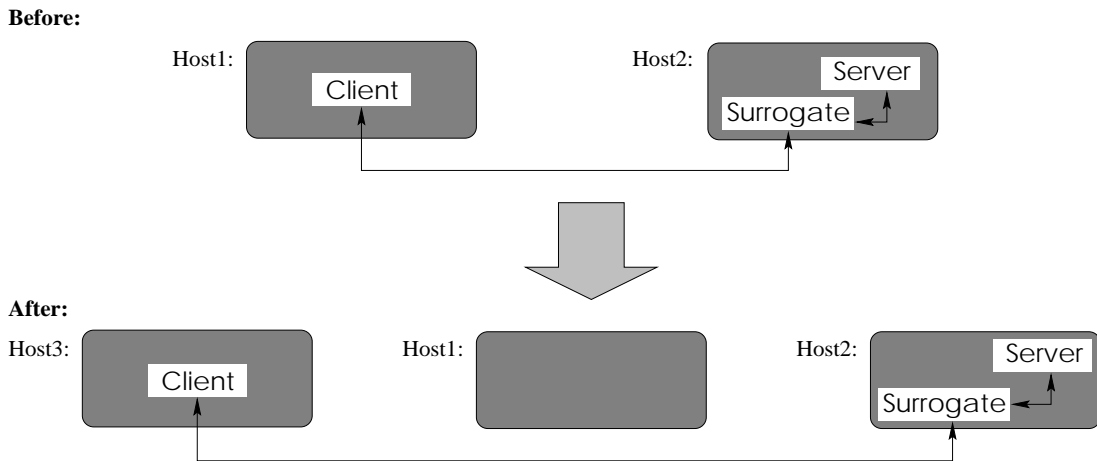


Figure 2.2: Server side proxy approach, [Casey, 95]

new address.

We find the idea of a MCP good. However the idea has three drawbacks. First we do not believe it to be possible to move the entire environment each time a user changes device. When a user is active, the environment can become very large and very complex containing several applications and resources. Small handheld devices use wireless network technology and is therefore still limited in bandwidth, therefore moving large amounts of data will stress the battery and network resources. Second [Casey, 95] implies that the user should not have to log out, when leaving his device and that the MCP can be requested by another device when the user starts using it. But not logging out from a device poses a serious security risk, which then should be addressed by some other means. Third [Casey, 95] also mentions using a PDA as cache for the MCP when a user moves between two non-linked devices. This requires the device the user is departing from to figure out that the user is leaving and then send the MCP to the PDA. How this should be done is not addressed in either [Banerji et. al, 93] or [Casey, 95]. One could imagine that a user's MCP is placed on the PDA each time the user leaves a room and then extracted from the PDA when he enters another. This will require the PDA to be turned on at all times in order for the MCP to follow the user. Since a lot of data has to be transferred, the user may have to wait before leaving the room until all data have been migrated, which is not very seamless.

Finally, it is not necessarily all applications the user wants to follow him. For instance, if he has started a computationally heavy process on a powerful workstation, he does probably not want it to follow him home on his PDA and home computer.

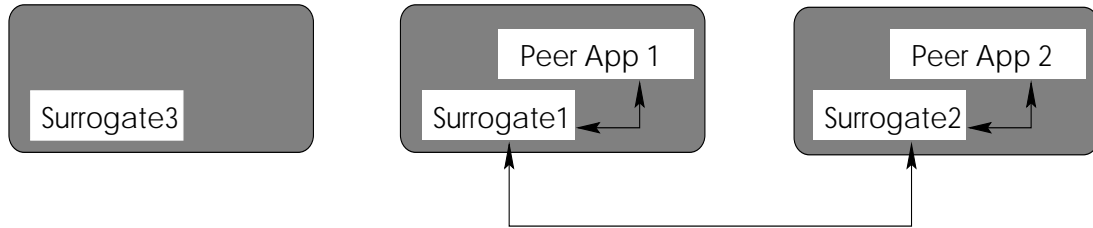


Figure 2.3: Proxies on all hosts approach, step 1, [Casey, 95]

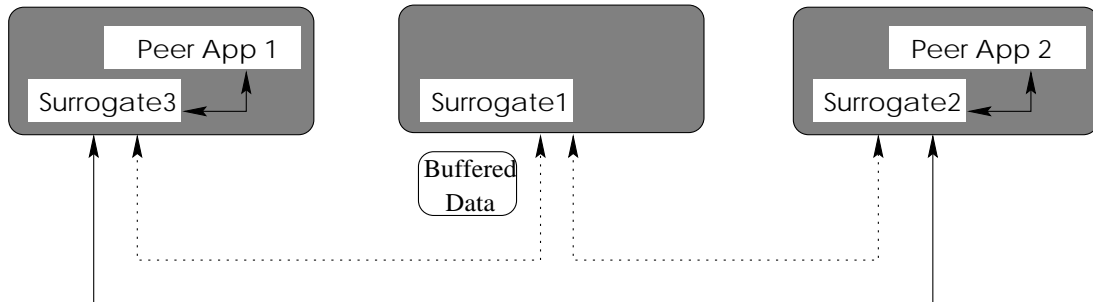


Figure 2.4: Proxies on all hosts approach, step 2a, [Casey, 95]

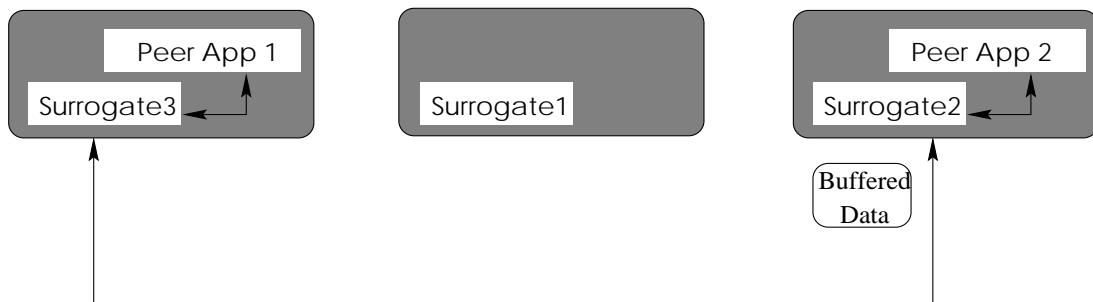


Figure 2.5: Proxies on all hosts approach, step 2b, [Casey, 95]

2.7 Summary

In this chapter, we have presented a vision of a PCE in which applications are able to follow the user, and some of the issues we find should be addressed when developing such a PCE. Some of these issues are multiple network technologies, disconnections, limited bandwidth, high bandwidth fluctuations, mobility of devices and users, ie., address migration, location dependent information and physical path vs. virtual path, portability constraints, seamless shifts, ie., user interface adaption and pervasive applications.

To conclude this chapter, the pervasive environment is characterised by users that shift among different types of devices while working. For the shift to be as seamless as possible it is necessary that network connections can be established from most places, also while moving. At the same time, it should be possible for a user to access his resources no matter from where he connects to the network. On top of this the user's applications should be able to follow him when he shifts to a new platform.

Chapter 3

Communication Models for a PCE

In this chapter, we will investigate what properties we want a communication model and the applications in it to have, in order for the applications to follow the user. In chapter 1 we mentioned three suitable ways of doing this:

1. Have the user interface on the current device of the user and the application on a central server in a distributed system.
2. Migrate the state of the application from device to device and set up the application on the destination with the transferred state. This requires the application to be installed on all the devices the user makes use of.
3. Migrate both the state and the code of the application. In this way there is no need for the application to exist on all the platforms the user wants to utilise.

We do not find the idea of having all applications installed on every device feasible, due to the changing needs of a user. Every time a new application is needed it would have to be installed on all devices the user might access or by the user each time he accesses a new device on which it is not installed. This is not very seamless and the work involved with this, as well as the problem of how to know which device the user will access in the future, makes this approach unsuitable for a PCE. We have therefore not investigated it any further. Instead we have looked at what requirements the other two approaches have and how we fulfil these requirements.

The first approach have properties similar to those of the Client/Server model, henceforth called the *Client/Server approach*, in which resources are centrally managed and can be accessed via the network. The last approach have properties like mobile agent systems, such as Ara [Peine, 97], in which agents move between places and perform different tasks, in the following called the *full migration approach*.

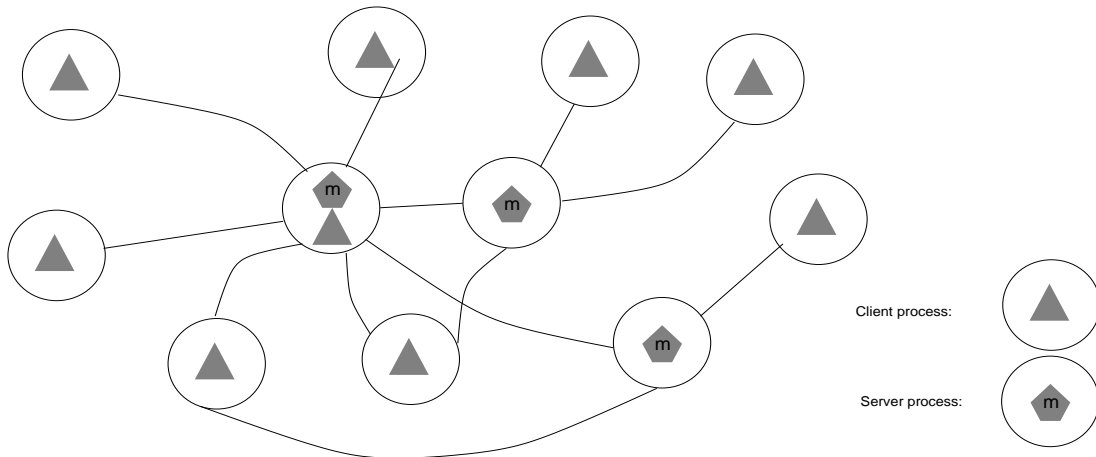


Figure 3.1: The Client/Server model, [Coulouris et. al., 94]

In this chapter we will first look at the Client/Server approach and the properties of the Client/Server communication model. Then we will address the full migration approach by investigating the properties of mobile agent systems and how they communicate.

3.1 The Client/Server Model

The Client/Server model has been used in many distributed systems to provide distributed access to shared resources, [Popek & Walker, 85], [Bharak et. al., 93], [Cheriton, 88], [Douglis & Ousterhout, 91], [Artsy & Finkel, 89]. These shared resources can be both hardware resources, such as printers and disks, and software resources such as files, applications and databases. Systems based on the Client/Server model can be depicted as in figure 3.1, in which clients access the resource through an interface presented by the server's resource manager. To achieve better availability and failure tolerance, replication can be used. The Client/Server model normally uses a request/reply protocol to communicate between the client and the server.

The Client/Server model is generally well known and we will not describe it any further. More information can be found in [Coulouris et. al., 94]. Instead we will look at how the properties of the Client/Server model fit into a PCE.

3.1.1 Properties of the Client/Server Model

In the Client/Server model, resources are centrally managed which simplifies maintenance of resources. This implies a single point of failure, which means

that if a central server breaks down, no user relying on this server can continue his work. However, replication can be used to create more failure tolerant systems and thereby avoid a single point of failure.

In relation to a PCE, we imagine applications to be resources, which the user may access across the net. This requires only the user interface to be displayed on and adapted to the client. This is simpler than migrating and adapting the entire application. It also relieves the client of almost all the computation related to the application, which can be a benefit when dealing with resource-limited devices such as PDAs.

Because only the user interface is moved, the client must be continuously connected to the server in order to update the user interface. This makes the client closely coupled to the network, which is no problem when dealing with local area networks or even WANs connecting stationary computers. However, when using PDAs with wireless connectivity, connections are likely to be lost, at least for short periods of time due to the nature of wireless networks, see section 2.3. Another problem is that a constant connection will cost a lot of battery power, which is a limited resource as of today.

If a PCE is expected to span larger distances, the latency of the network comes into play as well. Due to the speed of light, the latency from one side of the earth to the other is about 1/10th of a second. If the user updates his window this may require several remote messages to be exchanged between the client and the server, each with a possible latency of 1/10th of a second. Replication can also be used to solve the latency problem. This will, however, decentralise server management to the number of replicated servers.

To summarize, the resources in the Client/Server model are centrally managed and accessed via the network using a request/reply protocol. The server typically features a resource manager that presents the resource interface which the clients may use to access the resource. The Client/Server model is very dependent on a reliable network in order for the clients to have access to the resources they need. This can be a problem in relation to a PCE where many devices use wireless network technologies which are less reliable and less powerful than the wired technologies, section 2.3.

The benefits of having applications running on the server is that the clients are relieved of computation and that only the user interface of the application needs to be adapted.

3.2 Mobile Agent Systems

We will now take a look at the properties of the Mobile Agent System model and how they relate to a PCE.

To understand what a Mobile Agent System is, we must first understand what a mobile agent is. No commonly agreed upon definition of the term *agent* exists. We will therefore try to describe what an agent is by mentioning some of the characteristics often associated with an agent and then sum up how we will use the term agent.

Several definitions of the term agent is compared in [Franklin & Graesser, 96]. The result of the comparison is that an agent may have some or all of the following characteristics:

1. **Autonomous** - exercises control over its own actions. Being independent of the environment in which it executes. To interact with the surroundings a standardised API must be provided.
2. **Reactive** - reacts to changes in the surrounding environment in a timely fashion, e.g., not by crashing.
3. **Goal-Oriented** - does not simply act in response to the environment, but have a goal which it tries to achieve.
4. **Temporally continuous** - is a continually running process, ie., it does not start over and over again due to external circumstances such as being migrated. Preserves the state of the process.
5. **Mobile** - able to transport itself from one device to another, typically by calling a system function which performs the migration or at least parts of it.
6. **Communicative** - communicates with other agent and perhaps also people, typically through a communication interface provided by the environment in which it executes.
7. **Learning** - changes its behaviour based on previous experiences, such as knowing a site is down and therefore not try to move to it before checking if the site is back up.
8. **Flexible** - actions are not scripted, ie., the agent is not dependent on executing a specific part of its code to continue execution.
9. **Character** - believable “personality” and emotional state. The agent is capable of giving the impression of being a person or other emotional creature.

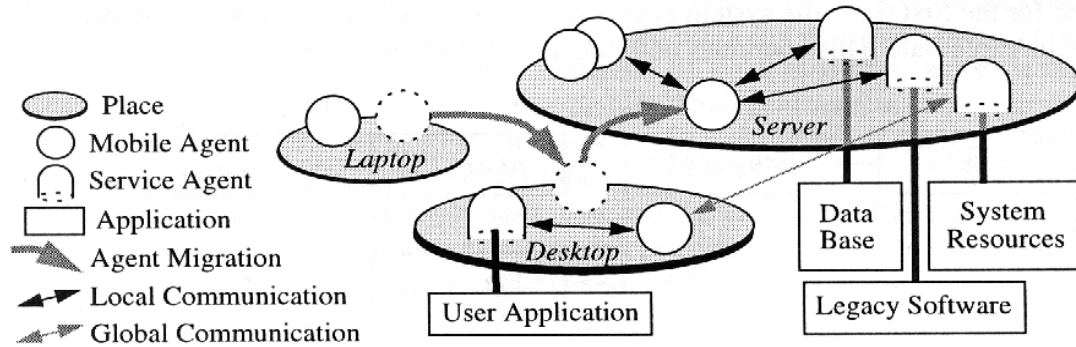


Figure 3.2: Overview of an agent system, [Baumann et. al., 98]

It is important to stress that the agent does not have to possess all of these characteristics. However an agent is always characterised as being autonomous, [Peine, 97], whereas the rest of the characteristics can be mixed. The actual set of characteristics depends on the view of the developers and the area in which they work, i.e., in agent systems most agents are mobile, in Artificial Intelligence environments agents are typically learning, but most likely not mobile.

In [Franklin & Graesser, 96] an agent is defined to have at least the first four characteristics. [Peine, 97] defines an agent to be autonomous and mobile. For now we will not define the term agent but only state that it is autonomous as well as something more, in which the *more* depends on the context in which the agent is used. In a PCE context, agents are assumed to be mobile as well as autonomous and the system described below and the ones in chapter 5 are all mobile agent systems, which also assume that agents are mobile. Later in this chapter, we will look at which of the other characteristics we would like pervasive applications in a PCE to possess, and we will discuss what the differences are between pervasive applications and mobile agents. Before we do this, we will describe the type of system in which mobile agents typically roam. In agent terms such a system is called a *mobile agent system*. The mobile agent system consists of one or more *places* which the agents can move between, figure 3.2. Each place is completely contained in a given host and can be thought of as the platform that provides the agent with an execution environment and access to system resources such as GUI components, sockets, and files.

Most mobile agent systems such as [Peine, 97], [Johansen et. al., 95], [Baumann et. al., 98], [Puliafito et. al., 98], and [Gray et. al., 96], do not let the agents run directly on the real machine processor, memory, and operating system. Instead they make use of virtual execution environments, i.e., virtual machines. The reasons for doing this are mainly security and portability.

With respect to security, the virtual machine makes it possible for the hosting

place to ensure that a given agent cannot cause serious harm. This is important since agents are capable of deciding for themselves when and where to move. By letting each agent execute in a virtual machine, the place gains a fine-grained control of the agent allowing it to decide what resources the agent can access, for instance by using a security model such as the Java sandbox model, [Java, 00]. It also ensures that if an agent crashes it does not crash the entire device.

The other benefit of using a virtual machine is portability. Due to the agent being interpreted by the same virtual machine's interpreter, agents can execute independently of the underlying hardware architecture, just as Java Applets in the JVM [Java, 00]. This is necessary considering the amount of different devices available today.

In order for an agent to present information to the user, it needs access to external resources such as windows, printers, and files. This access is gained through uniform interfaces which abstract away the device-specific interfaces on the different devices and enable the agents to access resources in a hardware-independent manner. This approach makes it possible to check whether agents are allowed to access a resource and what rights they should be given to that specific resource. The use of a resource can also be monitored to see whether it is used correctly.

Since agents are aware of when and where they move, they are responsible for setting and resetting handles they might have to external resources. This is normally not done by the agent platform, ie., place. However, Migratory Applications by [Bharat & Cardelli, 97] use agents to migrate both the application code and the state of the user interface. This state is then set up on the destination by the agent platform. Here a migratory application can be thought of as a mobile interactive agent, running on top of a system which makes use of subagents to migrate the application. We will go into more details about the Migratory Application system in chapter 5.

To sum up, mobile agent systems provide one or more places on which agents can execute and communicate. Each place provides a virtual machine by which fine-grained control of the agent and agent portability is achieved. A place also provides a uniform resource interface through which agents can access external resources. This gives agents a uniform way to access external resources independently of the hardware on which they execute. It is normally the agents that maintain the handles they have to external resources.

We will now look at the properties of mobile agents and mobile agent systems to see how these may be applicable in a PCE.

3.2.1 Properties of the Agent Model

As mentioned above, agents can be mobile, which allows them to move between places almost seamlessly, often by one system call such as `ara_go` in Ara, [Peine, 97]. For this to be possible, it is beneficial for the mobile agents to be autonomous. This gives mobile agents the advantage of being independent of and loosely coupled to the platform on which they run. In particular this means that the mobile agents are independent of the network once they have arrived at a place. Autonomy would be an advantage for pervasive applications in a PCE with a lot of mobile devices, since we no longer have a single point of failure, apart from the actual client device, which cannot be avoided. The use of a virtual machine and environment to make the involved devices homogeneous to the mobile agent is also useful in a PCE, since we have several very different devices present.

Apart from autonomy and mobility, agents may also be reactive and goal-oriented. Neither of these properties are necessary in a PCE, since we do not imagine pervasive applications as being either. However it should pose no problem if an application actually had either of the properties, we just do not want to demand them and cannot see what benefits a PCE would have of its applications having either of the properties. This does also apply for the character and the flexible properties, since we do not imagine an application as being a person and there is no need for it not to be scripted.

With regards to the learning property, it would be beneficial for an application to learn from the input it receives. This would help it in predicting the needs of the user. This is already present in applications such as Microsoft Office in term of helper agent in Word where the user is monitored by the helper and in case it knows of a better solution to a specific problem it presents it to the user. It is however not a property that makes a difference with regards to a PCE or the demands of a PCE's users as they are specified in chapter 2.

We do not view an application as being temporally continuous but more as being event-driven since it is based on a graphical user interface through which events are generated. An application does not need to preserve state from one session to another. However, a pervasive application should be temporally continuous because it will experience interruptions during which the state should be preserved, ie., when it is migrated.

The pervasive application should be communicative in order to present information to the user and to exchange data with the system and other (pervasive) applications.

3.3 Summary

With respect to the two models presented, their properties and the advantages/-disadvantages mentioned, we believe that a pervasive application should be able to follow the user using migration and not simply through moving the user interface. This is due to the close coupling to the network required if the Client/Server model is used. This demand cannot be met by the mobile devices part of a PCE. The problems concerning enough processing power are likely to be solved in the near future due to hardware advances.

For the pervasive application to follow the user by migration, we want them to be mobile and autonomous. To make the migration seamlessly to the user the state of the pervasive application must be preserved, i.e., pervasive application must be temporally continuous. To interact with the user we want pervasive applications to be communicative.

According to our discussion of agents, this is enough to characterise a pervasive application as a mobile agent. However, we still believe there is a difference, since we find that the mobile agent is aware of its own mobility whereas the pervasive application is migrated by request from the user and the migration itself is handled by the system. So the pervasive application should not notice that it has been moved to a new device. This requires the external resources and their state to be handled by the platform on which the pervasive applications run and not by the pervasive application. On top of this, we do not imagine a pervasive application as being reactive, adaptive, goal-oriented, flexible or as having a character.

The concept of pervasive applications versus mobile agents is of course based on our perception of what constitutes an pervasive application, namely that it is an application, i.e., a tool that can be used to produce a result. However the tool must be wielded by some user in order for it to produce anything. This is discussed further in section 6.4.

In the future it may be that one chooses not to distinguish between pervasive applications and mobile agents, and future pervasive applications may start featuring more of the agent characteristics such as reactive and adaptive. In Migratory Applications, [Bharat & Cardelli, 97] imagine their applications as being mobile, interactive agents, i.e., agents with a user interface capable of moving.

Whether one chooses to call the pervasive application a mobile agent or a pervasive application is mainly up to the person, since especially the term agent is very widely and vaguely defined and will be discussed further in section 6.4. The important thing is that the pervasive application must be able to migrate in order to follow the user around in a PCE.

We have therefore decided to investigate the area of migration further. By migration we mean both migration of processes, objects and agents. We will start this investigation by looking at how process migration is done in some Distributed Operating Systems.

Chapter 4

Migration in Distributed Systems

Migration: Moving from one country, place, or locality to another
[Merriam-Webster, 00]

Investigation into migration of active entities began in the early 1970's where focus was on process migration in distributed operating systems. *Process* is the key concept in operating systems, both distributed and non-distributed, and basically it is a program in execution. According to Tanenbaum, [Tanenbaum, 92] a process consists of the executable program, the program's data and stack, its program counter, stack pointer, registers, and other information needed for the program to execute.

The development of distributed operating systems has for many years been focused on realizing a transparent system¹. Here transparent system means a collection of computers which transparently act as one. In such a system the user does not need to know on which server a specific resource resides, e.g., on which server a specific file is located or as we shall see on which remote host his process is executing.

For process-migrating distributed operating systems to be fully transparent it is necessary that the migration task is transparent both to the user and to the process. This means that the user is not aware of one of his processes migrating during execution. To the process it means that the process is not aware of being migrated.

Before we turn to the distributed operating systems we will explain some of the terms used in the chapter. We will start by explaining the term *Load balancing*,

¹The trend in today's distributed computing is moving away from transparency, [Waldo et. al., 94], or at least discussing some of the difficulties it gives.

which is the motivation for implementing migration in most of the distributed operating systems.

Load Balancing. The philosophy behind Load balancing is that in a network there will always be idle hosts or at least hosts with a light workload. The developers wish to harness this processing power and the best way to do this is process migration.

An important part of a load balancing system is the *load balancing policy*, which decides when and where to a process should migrate. Load balancing policies can be split into three different categories.

- **Static:** These policies are based on an *a priori* allocation of processes with no regard to the current load of the involved host(s). Once allocated, a process executes on the given host until terminated or finished.
- **Dynamic:** Is like static load balancing policies, except that allocation of hosts are based on information about the current load of network hosts. This information is based on statistics gathered from all hosts on the network.
- **Preemptive:** Even though dynamic load balancing is based on the load of network hosts, once the process has been allocated it may not move throughout the rest of its life. The policy is therefore dependent on the unceasing creation of new tasks if it is to balance the load. Under a preemptive load balancing policy, a process may begin its execution at one site and due to fluctuations in the system load be reassigned to another site later.

Residual Dependencies. Another issue when migrating process are *residual dependencies*. They are dependencies left behind by the system when it migrates a process. Residual dependencies create an on-going need to maintain data structures or provide functionality for a process even though the process is now executing on another host.

Residual dependencies are undesirable for three reasons: reliability, performance and complexity. They decrease reliability by allowing the failure of one host to affect processes on another host. They decrease performance, since they require remote operations, where local ones would have sufficed. They increase complexity by distributing the state of a process over several hosts.

Some residual dependencies cannot be removed. For example when a process accesses a hardware resource on a specific host and is then migrated. To maintain

transparency and functionality the process has to forward its calls to the host on which the resource resides.

Residual dependencies can also be used to improve transparency, such as the *Process Control Block* (PCB) kept at the home site in the Sprite system, described later. The PCB allows system calls such as `kill`, `wait` and `exit` to be performed at the home site, even though the process has migrated.

In some cases distributed operating systems also create residual dependencies to achieve better performance, such as the home structure used for fast process location in LOCUS and MOSIX or the lazy fetch mechanism used in Accent to fetch virtual memory pages.

In the rest of this chapter we will take a look at how process migration and object migration have been done in existing systems.

4.1 LOCUS

The first system we will look at is LOCUS, [Popek & Walker, 85], which started as a distributed operating system project at the University of California, Los Angeles in 1979. It was developed to be a distributed, fault tolerant version of UNIX, featuring a distributed file system.

The main idea behind LOCUS is to unite a collection of heterogeneous computers, both workstations and mainframes, into one big system. By doing this they hope for the system to be as easy to use as a single computer. Unifying the computers is done by providing transparent network support for both the user and the applications.

It enables users and applications to access resources and services in the entire system without knowing their actual location. It is also possible to perform *remote tasking* meaning that processes are easily started remotely or migrated to a remote host. Even if a user, sitting at one type of architecture, wants to execute a program only available on another architecture LOCUS will automatically execute that program on the appropriate one, completely transparent to the user.

The motivation for migration in LOCUS was to support transparent remote tasking between homogeneous architectures. They wanted the user to be able to easily execute his process on one host and if needed migrate the process to another host during execution.

4.1.1 The LOCUS Architecture

The architecture of LOCUS is based on the monolithic UNIX system, extending it with network support². It permits the user to execute programs at any site in the network, just as easy as executing the program locally. To do this the system calls `fork` and `exec` have been extended to work in a distributed environment.

Besides this, LOCUS also makes it possible for the user to transparently shift execution site at run time, thus providing the ability to dynamically choose a new site. This is supported through the functions `run` and `migrate`.

Logically `run` can be seen as a combination of `fork`, that starts a new process followed by `exec`, that creates a process in which the program is executed. `Migrate` is used to change a process' site of execution at run time.

`Fork` and `migrate` is only possible between CPUs of the same type since the process retains most of the internal state information and mapping state between different architectures is not supported.

When `exec` and `run` are called they allocate a new memory image on the destination host making the calls independent of the source host which originated the call. This makes it possible for them to be initiated on a host of different architecture than the destination host, see figure 4.1.

	old image	new image
move old process	<i>migrate</i>	<i>exec</i>
create new child	<i>fork</i>	<i>run</i>

Figure 4.1: Steps involved in remote tasking in LOCUS, [Popek & Walker, 85]

Transparency. To achieve system transparency in a system with remote tasking the following transparencies should be supported:

- **Name Transparency** - Names has to be globally unique. Files on all hosts are covered by a single tree structure and the name of a file consist of both a logical file group id and an inode, i.e. a file descriptor. *Process names* (PID) are made up of a host identifier and a local process identifier. Since processes can migrate, the content of the PID does not indicate the current host of execution.

²UNIX did not have network support at the time LOCUS was developed.

- **Location transparency** - Location transparency is necessary in order for the network to seem as one computer. It cannot be handled by having the location of the resource encoded in the name, since the resource can migrate. Instead the name, the PID or file inode, contains the name of a site which knows the location of the resource, ie., both files and processes have a specific site which keeps track of the current file or process' location. For files this site is called a *Current Synchronisation Site*. It can be found from the logical file group id. For processes, the id consist of the process' creation site's ID and a local process ID. The creation site always knows where to find the process.
- **Semantic consistency** - System services, support libraries, commonly used application programs and the like must have the same effect independent of the site on which they execute. This is also called *environment consistency*.

4.1.2 The Migration Mechanism

Migration of a process in LOCUS is started either internally by the process calling the `migrate` function, or externally by signalling the process with a `SIGMIGRATE` signal. Both ways cause the local system to assemble some process state information before sending a migrate request to the destination site.

The migration request is handled the same way as the three other remote tasking calls (`fork`, `exec` & `run`). When one of the four kernel calls are called a *Fork Process Request* (FPR) request is sent to the destination site, first step in figure 4.2. The FPR request includes parts of the source process' *proc* and *user* structures³ which will be needed by the destination to create a new process, along with open file information.

When receiving a positive acknowledge (RFPR) the source process is suspended and the destination site will take control and be responsible for pulling the memory image of the process across the network. This is called the *pulling strategy* and was used due to a lack of buffer space, probably because of memory cost.

When the destination host receives the FPR request, it forks a destination process that handles the retrieval of the source process' image. First the destination process checks what kind of remote process request was sent, ie., `fork`, `exec`, `run` or `migrate`. This is needed in order to determine what other data must be obtained from the source process. If it is a `fork` or `migrate` request the source process' *user area*, which consist of the user data and the user stack, has to be

³We assume they contain information about the process and the current user environment. However it is not explained in the book, [Popek & Walker, 85]

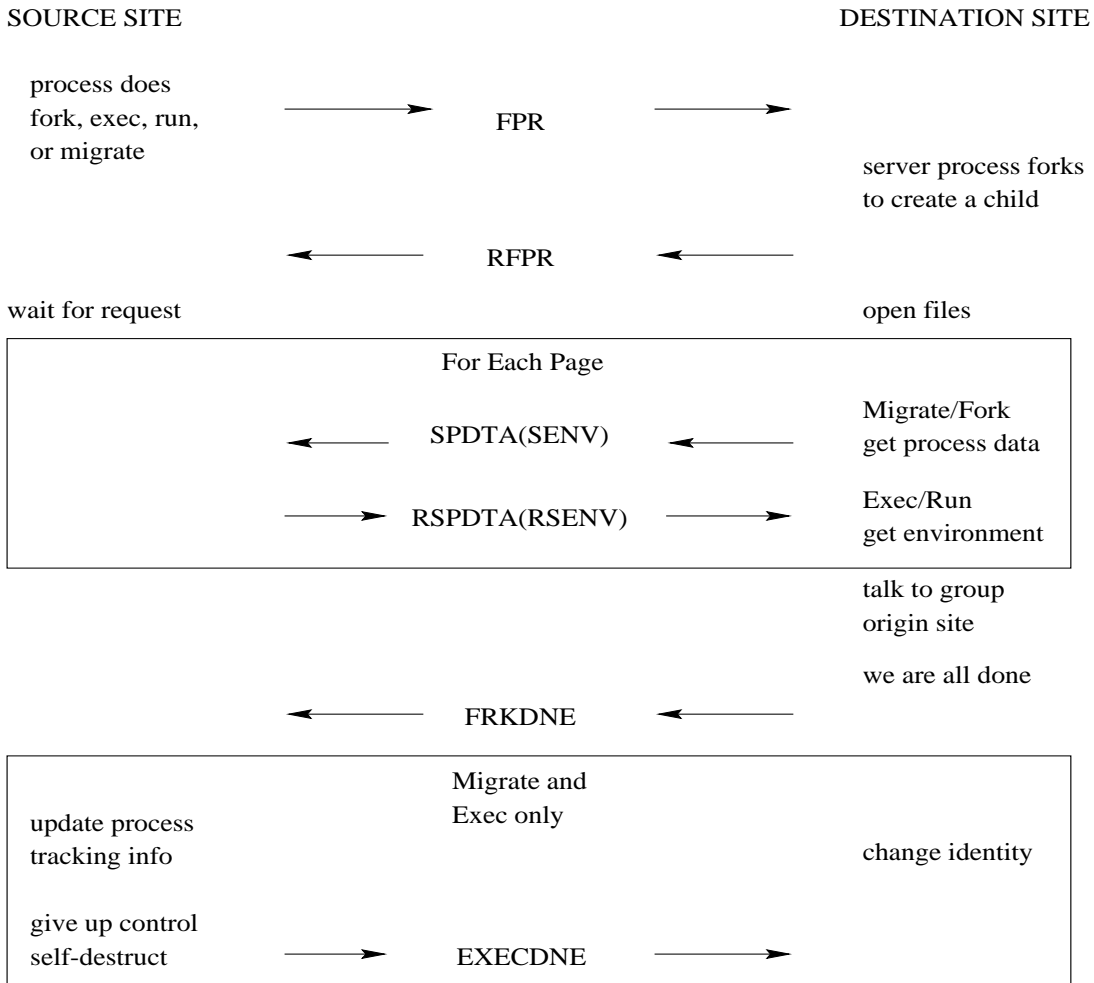


Figure 4.2: Remote Process Creation in LOCUS, [Popek & Walker, 85]

copied. If the request is `exec` or `run` no part of the *user area* is needed. Instead command arguments and environment data are needed.

To retrieve the source process' image or environment paged over the network the destination process sends either a SPDTA request or a SENV request, depending on the type of call, ie., if it is `migrate` or `fork` a SPDTA requests is sent otherwise a SENV request is sent. Paging of the process image is thus strictly by demand and in standard sized blocks to ensure the buffer is not overflowed. The destination process is also responsible error checking and getting the data in the right order.

Having transferred the data from the source process, the destination notifies the source process, by sending a FRKDNE response. In case of the request being a `exec` or `migrate` request the destination process will change its PID to that of the source process, taking over the source's identity.

Finally all signals received while the migration took place are sent to the destination process in a EXECDNE response, the process' site information is updated at the process' original creation site and control is returned to the process' program code.

Since Locus features a distributed file system it is not necessary to migrate files.

4.1.3 Residual Dependencies

The only dependencies that are left behind when a process migrates to a remote site is the information on the process' creation site used for locating the process.

When a process wants to signal another process it first searches the local process table to see if the process is found locally. If not, the creation site is asked of the process' whereabouts. If the process is at the creation site, the signal is forwarded, otherwise a message is sent back with information of the process' location. In case the creation site is down, a replicate takes over. The replicate is decided in advance, probably at the creation time of the process. However exactly how this works is not specified in the book.

4.2 MOSIX

The next system we have chosen to look at is MOSIX, [Bharak et. al., 93], which was developed at the Hebrew University of Jerusalem starting in 1981. MOSIX, is a Multicomputer Operating System designed to integrate a cluster of loosely connected, independent computers into a single UNIX-like environment.

Migration in MOSIX is used for load balancing. The system features a preemptive load balancing policy. As in LOCUS transparency issues are important. Location and access transparencies are supported to make the network invisible to the user and to applications. An important characteristic in MOSIX is autonomy of the hosts. In MOSIX each host is capable of operating independently of the other hosts. This is due to the fact that no central servers are used to provide essential services, and each host holds a complete kernel and its own file system.

MOSIX is capable of spanning several heterogeneous architectures, but preemptive load balancing, ie., migration, is supported only among homogeneous subsets of hosts.

Even though the hosts are autonomous they are meant to work closely together. The loose connection between hosts was maintained to allow better scalability and dynamic configuration of the system.

To provide a distributed file system MOSIX unites all the disjoint file trees found on each individual host, into one distributed file tree. Each individual tree is a complete UNIX file system, with regular files and devices on the leaves and directory files on the internal vertices. To combine these trees a super-root is used (`/...`). The super-root is root of all the hosts currently connected to the network. The path `/.../m5` will for example refer to the root of host 5's local file system.

4.2.1 The MOSIX Architecture

The MOSIX kernel is built by restructuring code from the UNIX kernel into three layers. The lowest layer, which is hardware dependent, contains routines that access local resources (device drivers, files, process structures). It is tightly coupled with the local processor and has complete knowledge of all local objects. It can only access local objects.

The uppermost layer executes in a hardware independent manner. It provides the standard UNIX system interface to applications. Since the layer is hardware independent it does not know the identity of the processor on which it executes. It is capable of accessing all objects to which it holds a reference.

Communication between the two layers proceeds via the `Linker` that provides RPC services between the two layers, making it possible for an upper layer to communicate with a lower layer on either a local host or a remote host, transparently, see figure 4.3.

To access a file the upper layer of the kernel uses `universal inode/file structures`. This structure consists of a host id, a local pointer and a version number. The link layer uses this information to find and call the lower level of the kernel (possibly on a remote host). The lower layer accesses the file on the disk using the local inode which is similar to that of standard UNIX.

Generally system calls are split up into an upper and a lower system call. Thereby when an application makes a system call it happens in the upper layer. The upper layer will then make a call to the lower layer through the link layer, which will forward the call to a, possibly remote, lower kernel layer, figure 4.3.

MOSIX kernels use remote procedure calls, called `Scalls`, for internal kernel-to-kernel communication across the network. When a kernel on one host wants to call procedure `<name>` on a different host it calls `S<name>`. This call is forwarded by the link layer to the remote host. At the remote host an `ambassador` process handles the call by calling the `<name>` procedure for which it received the `S<name>` call.

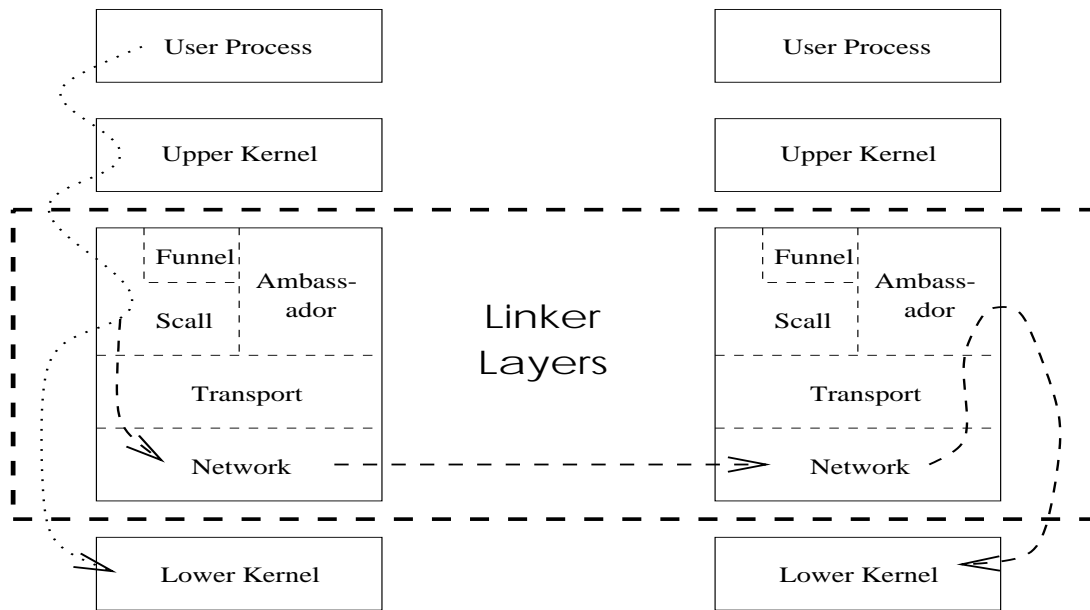


Figure 4.3: The three layers in the MOSIX Kernel, [Bharak et. al., 93]

4.2.2 The Migration Mechanism

Migration in MOSIX can be initiated either by the systems load balancing algorithm or by the user.

The user can initiated a migration by making a:

```
migrate(destination: address; lock: int)
```

system call, where destination is the address of the host and lock is used to check whether the process is locked on the current processor and therefore not allowed to move.

If the process is allowed to migrate a call of

```
passto(destination: address; userRequest: bool; loadBalancing:
        bool; maxLoad: int)
```

is made. `passto` can also be called directly by the load balancing algorithm. These system calls are handled by the upper layer of the MOSIX kernel.

The first thing `passto` does is to check whether the process is allowed to migrate and whether the specified destination accepts it. If this is the case a `Smakeproc` call is made to allocate a process frame on the destination and initialise the process' memory regions.

As described above the `Smakeproc` call on the source host causes the `ambassador`

process at the destination host to call `makeproc`. `Makeproc` makes sure that it is allowed to accept processes from the source processor and if the source process was moved due to load balancing, it checks if the load of its own processor is still under an acceptable level. It then makes sure that the process maintains its ability to communicate with the same remote processes as before. Next the process' data is passed. First the `u area` is transferred. The `u area` contains the runtime information of the process. When transferring the process memory image only the modified memory pages are passed. The rest are either being demand-paged from the original executable file or marked as zero filled. Finally `makeproc` checks the *process identification number* (PID) before it creates the new process frame. If all went well the `ambassador` returns a positive acknowledge and creates a process in which `makeproc` can fill in the memory regions according to its arguments and incoming `S<name>` calls.

When the acknowledge is received on the source host. It makes a `Spassproc` call. This is the commit point of the migration. If it succeeds the home site of the process is informed of the process' new location. The source finally calls `Sactivate` and pass any pending signals to the destination host. On the destination host the `activate` call activates the process or if something went wrong during the migration cleans up the mess.

Finally, the memory and process are cleared on the source host.

4.2.3 Residual Dependencies

To locate a process a home structure is updated each time the process migrates. The host, on which the home structure is located is found by mapping the process id to a processor number using a simple function. This is actually the only dependency that is left, when migrating a process. Files are not necessary to migrate along with the process, since MOSIX uses a distributed file system. As mentioned above files are always accessed through the universal inode. Since the universal inode consist of a host id migrating a file would require all the universal inodes present in the system to be updated. Therefore migration of files are not possible in MOSIX.

4.3 Sprite

Another process migrating system is Sprite, [Douglass & Ousterhout, 91]. Sprite was developed at the University of California, Berkeley beginning in the fall of 1984. Sprite was built to harness the power of idle workstations connected to the network, and they found process migration a good way to, transparently,

do this. Just like the above systems Sprite can only migrate processes between homogeneous architectures.

When a process starts it is automatically assigned an idle workstation on which to run. When the user of that workstation returns, the process will be evicted from the workstation in order not to steal processing power from the “rightful” owner. This implies a different migration mechanism than the ones we have seen in the previous examples. This migration mechanism will be described in the following subsections.

As in the previous systems, transparency were given high priority during the development, but also performance had high priority. This inevitably lead to some residual dependencies and more complexity.

4.3.1 The Sprite Architecture

Like the other systems mentioned Sprite is a distributed operating system which supports a distributed file system in order for all processes to transparently access files no matter their location.

To further support transparency a process’ behaviour should not be affected by a migration. The process should appear as if it had never left its home site and any operation that is possible on an unmigrated process should be possible on a migrated process.

By making kernel calls location-independent, process location transparency would be fulfilled. Four techniques were used to achieve location-independent kernel calls.

1. Changing the file system, so every host accesses the same name space. Thereby making `open`, `close`, `read`, `write`, etc. location-independent.
2. Transfer state from source to destination host, so normal kernel calls can be executed locally. State transfer was used for transferring virtual memory, open files, process and user identifiers, resource usage statistics, and more.
3. Forward some kernel calls to the home site, e.g., the call `gettimeofday`, because hardware clocks are not synchronized. Another is `getpgrp` where the information is kept at the process’ home site, due to that processes in a group may be distributed across the network. Forwarding can also occur from the home site to the remote host, e.g., in case of the `kill` kernel call.
4. The last method used is patching of some kernel calls that need to update information, both on the home site and on the remote host. An example of such a call is the `fork` kernel call, that needs to create a PCB on both

the home and the remote host in order for the forked process to seem as if it is running on the home site. The kernel calls `wait` and `exit` must also be patched.

In the Sprite system the selection of idle hosts are fully automated. To monitor the usage of a host, Sprite has a `load-average` daemon, that notifies a central migration server if a host is idle and ready to accept migrated processes. Processes that invoke migration can then call `Mig_RequestIdleHosts` to obtain an identifier for an idle host. This host id is then given to the kernel when asking it to migrate a process. When a user of a previously idle host returns an *eviction* happens. This is detected by the `load-average` daemon, which notifies the process that initiated the migration. It is then up to that process to migrate the remote processes back to their home site.

Two tools can be used by the user to migrate a process. They are `mig` and `pmake`. `Mig` can be used to execute a command on a remote host. It will select an idle host and invoke the appropriate kernel calls to migrate the process to the remote host. `Pmake` is the equivalent to the UNIX `make` tool, except that it run as many processes as possible in parallel.

`pmake` remigrates processes in case of an eviction, `mig` does not.

4.3.2 The Migration Mechanism

A major part of migrating a process concerns migrating its state. Sprite looks at the following parts of a process' state:

Virtual Memory Transfer. This is the limiting speed factor in migration. To reduce the cost of transferring the whole memory image to the new environment, Sprite flushes the dirty pages to disk and starts the process on the remote host with no pages loaded. The memory pages will then be loaded when needed by the application. In this way only the dirty pages are stored (transferred to the file server) and only the pages needed at the remote end are transferred from the file-server to the remote host. This can have the disadvantage of accessing the disk twice, first writing them to the file server and second reading them back to the remote host. But since Sprite's file system is cached, most of the time the pages will be placed in the file server's main cache, so no disk access is necessary.

Another approach could be to transfer the dirty pages directly to the remote host and avoid the two accesses to the file server. This is not done because Sprite processes are mainly migrated when started or when evicted. When processes are started they have no virtual memory, and when processes are evicted it is

important they leave quickly and whether the pages are put on the file server or at the remote site makes no difference to the speed of the eviction,.

The advantage of the Sprite approach is that it does not leave any residual dependencies on the host from which the process is evicted.

Sharing of memory is possible in Sprite, but if this is the case, then the involved processes are not allowed to migrate. Instead of denying these processes to migrate another solution could be to migrate all the processes sharing the memory.

Open Files. The state of a Sprite file consists of a file reference, caching information and an access pointer. All of this should be moved when migrating a process.

When they first tried to move an open file reference, they closed the file and reopened it on the remote host. This approach did not work because the file might be deleted in the meantime, due to another process holding the only reference and therefore being allowed to delete the file. Therefore they had to move the reference without closing the file.

Moving the file cache was done by flushing it to the file server, the file pages could then be retrieved when needed on the remote host. A note about cache is that to handle consistency the cache is disabled when two or more processes access a file from different locations.

The access pointer is moved with the process if the process is the only one using this particular access pointer. If the process has forked a child process, that also uses this access pointer, the pointer is stored at the server and the value read from there.

The Process Control Block. Apart from virtual memory and open files Sprite keeps a PCB that consists of all the remaining information about the process, such as the state of message-channels, the execution state and other kernel state. This block is migrated along with the process and an updated copy is also kept at the home site. The home copy is used to assist in some aspects concerning transparency.

Generally the PCB is easy to migrate since it is not as bulky as virtual memory and does not involve distributed state like open files.

4.3.3 Residual Dependencies

The same residual dependency concerning the home structure is found in Sprite. Here it is also used to make the location of the process transparent to the user, as mentioned in section 4.3.1.

4.4 Charlotte

The Charlotte system is another distributed operating system, that offers process migration, [Artsy & Finkel, 89]. It was developed at the University of Wisconsin-Madison, in the mid-80's as a host for experimentation with distributed algorithms and load distribution strategies.

In contrast to the above systems, Charlotte is a message based operating system and provides higher abstraction mechanisms for communication. Communication with other resources, such as other processes or files, happens via logical links that are handled by the kernel and different utility processes. It is the only system we have looked at, where process migration leaves no residual dependencies at all. Process location is handled by the abstract link mechanism and not by maintaining a home structure or a PCB.

For clarity the migration mechanism is separated into three phases: *Negotiation*, *Transfer* and *Establishment*. As in the above systems the main motivation for migration was to provide load balancing among the hosts in the network.

4.4.1 The Charlotte Architecture

The Charlotte system runs in a homogeneous environment with a multi-threaded kernel on each host. The kernel is responsible for simple short-term process scheduling and provides a message-based interprocess communication protocol for communicating over a token ring network.

Besides the kernel there are different utility processes in the system. One of their duties is to handle the pre-emptive migration policy. The migration mechanism is handled by the kernel. By letting the migration policy be handled by utility processes and the migration mechanism by the kernel a clean separation is achieved between the two. This is one of the characteristics of the Charlotte system and it makes it simpler to renew and maintain the migration policy.

Processes are completely unaware of the location of their communicating partners. This is due to the link mechanism which hides the location of the involved processes (or files) in the kernel. Obtaining a link to another process or file is

done either through another process, e.g., a parent process, or by using a name server. All information concerning a link is stored in and maintained by the two involved kernels. This includes the location of the communicating processes. A rich set of IPC primitives including non-blocking, asynchronous communication is provided for link communication.

The migration policy in Charlotte is effectuated by the **Starter** utility process. **Starter** bases its decisions on statistical information gathered by all the kernels in the network. Statistics include information on platform load, individual processes and selected active links for measuring the communication rate. Gathering of the information happens in the individual kernels running on the different hosts.

A **Starter** process executes a policy procedure when it receives messages carrying statistics, advice, or notice of process creation or termination.

4.4.2 The Migration Mechanism

The migration event is, as mentioned above, split into three phases: Negotiation, Transfer and Establishment. It is initiated by the **Starter** utility.

Negotiation. The *source* (S) and the *destination* (D) must agree to the transfer and reserve required resources to migrate the *process* (P).

In the case where one **Starter** process decides to migrate a process from one host to another, it asks the **Starter** on the other host if it will accept the process, step 1 in figure 4.4. If the answer is yes (step 2), the source kernel is informed (step 3) and the process is offered to the destination kernel (step 4). This offer is passed on to the **Starter** (step 5) which responds whether to accept the source process (step 6 & 7).

Included in the offer is the size of the memory space needed by the process. This information is used to allocate enough resources for the process to exist and is also a factor in the decision of accepting the process. Pre-allocation guarantees successful completion of multiple migrations at the expense of reducing the number of concurrent incoming migrations.

Transfer. The transfer of P involves moving its virtual address space, step 8 in figure 4.5, and its contexts (step 10) to the destination. In between the two steps all kernels maintaining links to P is sent an update message, which informs them of the new address of P, ie., the other end of the link. If either of the processes communicating with P tries to contact P, the kernels on which they run will withhold the message. It is then up to the kernel on D, for which P is

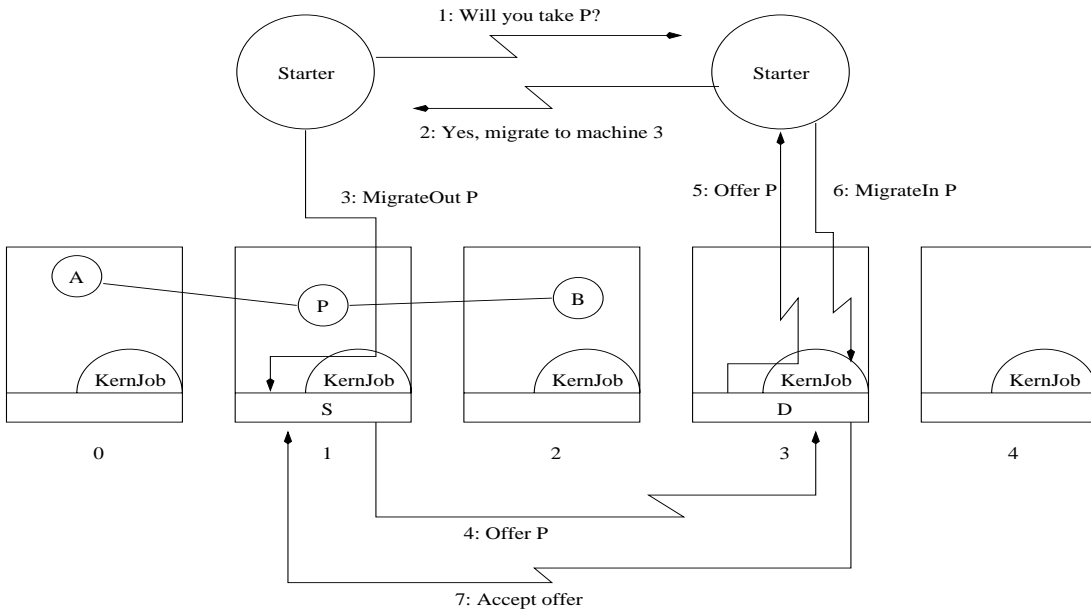


Figure 4.4: Negotiation phase, [Artsy & Finkel, 89]

designated, to request the messages once the migration has finished. This implies that after S has notified all P’s communicating partners it need not concern itself with future messages for P. In case it receives one it assumes it is requested later by the kernel on D and discards it.

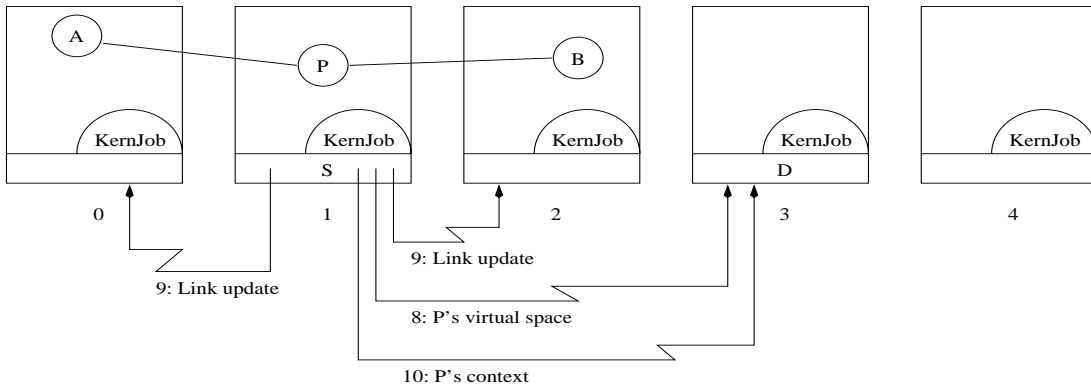


Figure 4.5: Transfer phase, [Artsy & Finkel, 89]

The contents of the context message in step 10 includes control information, the state of all of P’s links, and details of communication requests that have arrived for P since the transfer began. Pointers in S’s data structure are tracked down, and all relevant data are marshalled together. The requests are not to be confused with the messages exchanged between processes, they are solely used by the kernels to inform each other that a message is pending.

If there is a failure on either host during the transfer phase. It is detected by the other host, which then aborts the migration, terminates the migrant, and cleans up its state.

Establishment. The establishment phase is interleaved with the transfer phase. Data structures on the source are read as part of marshaling, and the corresponding destination data structures are written during demarshaling. After the transfer, P's links and pending events are adjusted by D. D then inserts P in the appropriate scheduling queue. Communication requests that were postponed while P was moving have been cached by S and D; they are now directed to the IPC kernel thread in D in the respective order of arrival (for each link, all requests cached at S precede those cached at D).

The kernel data structures pertaining to the migrant process must also be transferred and established. The data structures are marshalled by copying them to a byte-stream buffer and converting data types, e.g., pointer types. They are then transferred to D and finally demarshalled. In this way no information related to the migrant is retained at the source, i.e., no residual dependencies for the process.

4.4.3 Residual Dependencies

Because of the link mechanism a migrating process does not need to update any home-structures as in the three previous systems. Instead the link is updated by the underlying layers, i.e., the kernel and the utility processes.

4.5 Accent

The largest performance problem when migrating a process is migrating the virtual memory. In [Zayas, 87] an alternative way of doing this is presented where memory pages are fetched when referenced on the destination site. [Zayas, 87] has implemented this way of migrating virtual memory in the Accent System, [Rashid, 88], a distributed operating system already featuring *copy-on-write*, [Tanenbaum, 92].

The modified version of Accent provides process migration between homogeneous hosts like the previous mentioned systems. But instead of performing a physical migration of the process' memory, Modified Accent only performs a logical migration. A logical migration means that only the code and memory needed to restart the process on the destination is moved when migrating. The rest will be transferred when needed. Based on tests [Zayas, 87] shows that only a relatively small portion of the memory is actually ever referenced. So by not moving all

of the memory at migration time [Zayas, 87] is able to achieve better migration performance.

When a process accesses a memory page it is copied from a previous host memory image. So we have a kind of *copy-on-reference* mechanism similar to copy-on-write mechanism presented in [Tanenbaum, 92]. The downside of this approach is that residual dependencies are created each time the process migrates. In this way a process can end up depending on all the hosts the process has visited during its execution.

4.6 Native Code Process Oriented Migration

The last system providing process migration is a modified version of V, [Cheriton, 88], created by [Shub, 90]. We chose to include it, because it is capable of migrating processes between heterogeneous architectures and in a pervasive environment it is highly unlikely that the architectures, between which the applications migrate, are homogeneous, see section 2.6.2. The motivation in [Shub, 90] for removing the requirements for identical hosts is that more networks would be able to utilise process migration for balancing their workload.

4.6.1 System Architecture

The current system is a prototype, built upon the V distributed OS. V already supports migration between homogeneous hosts and is implemented on both the SUN and the VAX architecture. Due to the same kernels running on both architectures mapping external state (process information) is easier, since the same data structures are used for storing it.

The current prototype specifically looks at how to migrate a process from VAX to SUN. The process is coded in a strongly typed language and may only contain one thread. The migration is initiated by the process itself.

4.6.2 The Migration Mechanism

[Shub, 90] defines something called the *migration unit*. The migration unit contains all the code, static data and mapping tables needed by the process. Below we will take a look at the contents of the migration unit.

Code. Instead of mapping code when moving from one architecture to another, [Shub, 90] chose to put the code for all the involved architectures in the

executable. In this way it was only a matter of loading the right code, when executing on a different architecture. The downside of this approach is the size of the executable, which is much larger than needed for any specific architecture. However only the code needed on a given architecture will be loaded from virtual memory into physical memory.

Data. Instead of also carrying different data around for all architectures with the same values but different layouts, [Shub, 90] decided to map the data when moving from one site to another. In order to make this mapping as easy as possible [Shub, 90] uses the Greatest Common Denominator when allocating space for primitive types. This means that an integer will always have enough space allocated, no matter the architecture. This of course wastes some space on the architectures, that uses fewer bytes for their primitive type. With this layout the pointers, that points to other data, can all point to the same address and do not have to be mapped when the process migrates. However all pointers that points to code must be mapped to point at the code valid for the current architecture.

Two kinds of data exist in a process: *static* and *dynamic*. Memory for static data is allocated at compile time, and the mappings for this can also be generated at compile time. However memory for dynamic data is typically allocated by a `new` statement at runtime, why information about how to map the data should be computed by the `new` statement. In this way the migration mechanism can map it appropriately when the process migrates.

State Mapping. Apart from data, mapping of function entries and return addresses must also be done. This information can be generated at compile time, since all the code is available. When migrating a process the entry point of a function is mapped to the place in the new architectures code, where the function is located. The same is done for return addresses. Mapping of static and dynamic data is done on a component by component basis. The mapping includes accounting for change in byte ordering.

The last thing that must be mapped is the activation history. Here the dynamic allocated storage is mapped similarly to dynamic data. The registers used must also be mapped taking into account differences between the architectures.

To sum up [Shub, 90] is capable of migrating processes between two heterogeneous architectures, SUN and VAX. A process carries around code for all the hosts it may migrate to. This blows up the size of the executable. Apart from this the application also carries around the mapping information used for mapping between any of the architectures, which also increases the size of the executable. Finally the memory allocated by the application will always be the size of the largest

memory image of all of the involved architectures, no matter which architecture the process is executing on.

4.7 Emerald

The last system we will look at in this chapter is Emerald, [Jul et. al., 88]. Emerald is not a distributed operating system, but a distributed object-oriented language and kernel which enables development of distributed applications by the use of distributed objects. Objects in Emerald are capable of migrating from one host to another. Migrating objects encompass both migrating processes and migrating data, since an object with an active thread can be thought of as a process and an object with no methods or threads can be thought of as pure data. Thus the system is an example of a more fine-grained computation migration mechanism, than the ones we have been looking at until now.

The Emerald system consists of a local area network of homogeneous devices. It is assumed that the network is secure and of approximately 100 devices. On each device an Emerald kernel is running. The kernel takes care of all traffic in and out of the host. It also provides a runtime system in which the objects can live.

Below we will take a look, first at Emerald objects and their structure, and then at how the migration mechanisms works.

4.7.1 Emerald Objects

Conceptually only one type of object abstraction is presented to the programmer. Objects can be thought of as carrying around their own code, which is important in a distributed environment since having the code represented at a single place will involve too much network traffic. All objects have a unique network-wide name, a set of operations, an optional process and a local representation of data. Objects containing a process are called active objects and objects with no methods are called data objects.

At implementation level three different representations are used to improve performance. They are *global objects*, *local objects* and *direct objects*.

Global Objects. Global objects are capable of moving between hosts and they can be referenced and invoked from other hosts. For each global object referenced from a host, the host keeps an object descriptor. The object descriptor contains information on whether the object is located at the current host or located at a remote host. It also contains a *forwarding address* telling which host it was last

known to be on. If the object is present at the current host the object descriptor points to the objects data area. The data area contains all the values of the object's fields, such as pointers and primitive values. It also has a pointer to the objects concrete type. The concrete type contains the object's code and a template describing the layout of the data area as well as a template for each method describing the activation record. The templates are used when migrating and when garbage collecting the object. See figure 4.6 for the structure of the global object.

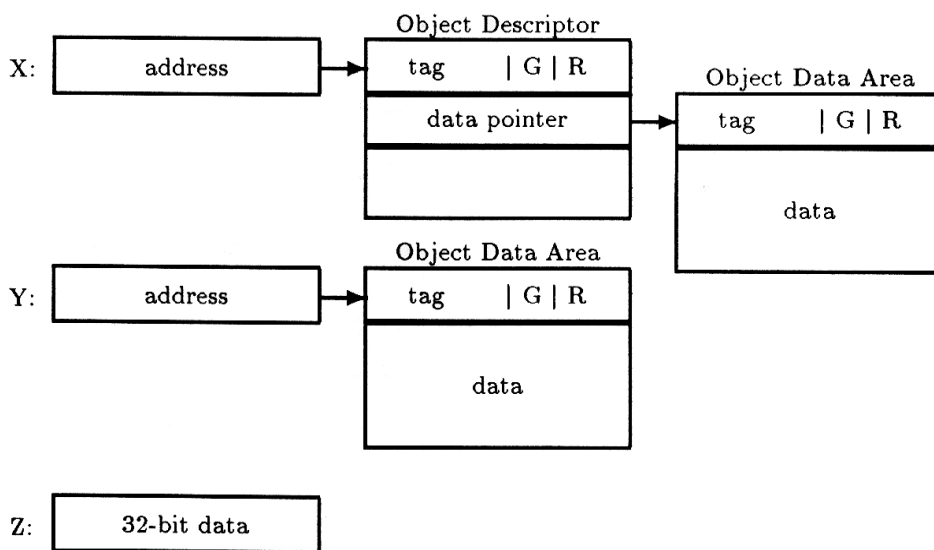


Figure 4.6: X: Global object, Y: Local object, Z: Direct object, [Jul et. al., 88]

Local Objects. Local objects are completely enclosed by another object, typically a global. They cannot migrate unless the surrounding object migrates and they cannot be referenced from outside of the surrounding object.

Because local objects cannot be reference globally, they do not need an object descriptor. The data area of the object act as a simple object descriptor, see figure 4.6. The data area is similar to the data area of the global object and also contains a pointer to the concrete object type.

Direct Objects. Direct objects are used for primitive types, such as integers and strings. They are allocated directly in the representation of the enclosing object.

4.7.2 Object Migration

A global object can be migrated by calling either `move`, `fix` or `refix`. The syntax of the calls are:

```

move <object> to <host>
fix <object> at <host>
refix <object> at <host>

```

`Move` is a suggestion to move an object to a host, the runtime may choose to ignore it. `Fix` and `refix` explicitly move the object. `Refix` is used if the object is already fixed at the current host.

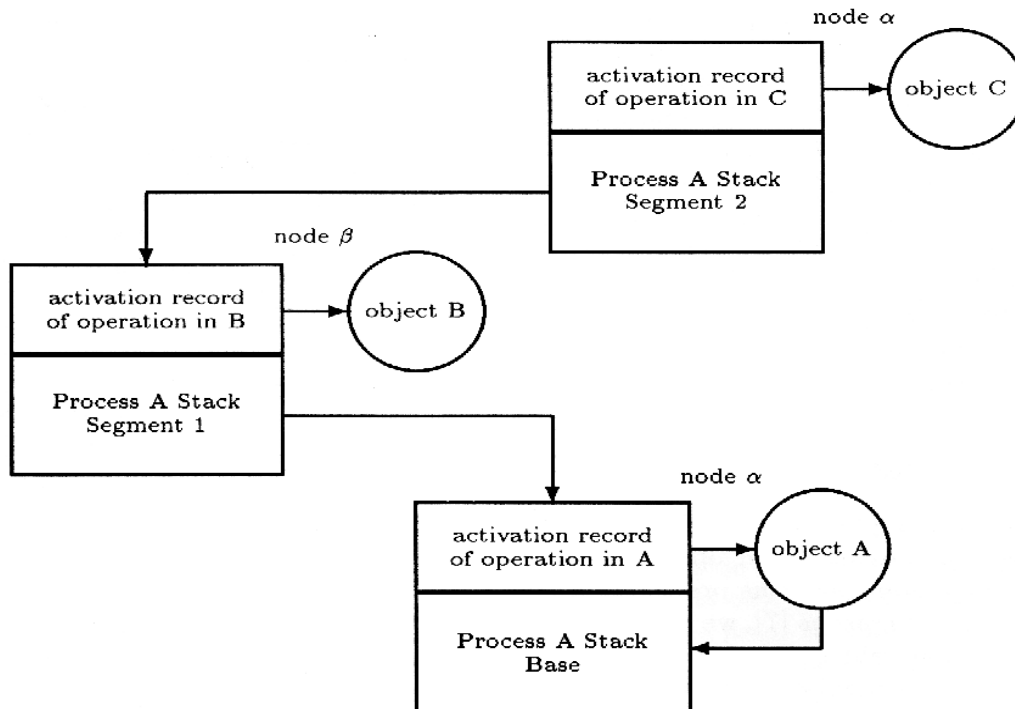


Figure 4.7: Segmented Object Call Stack, [Jul et. al., 88]

If an object contains both methods and data the activation records of the object are moved along with it. This may segment a process' call stack, as in figure 4.7. An Emerald process is thought of as a stack of activation records. If the object contains no methods, no activation records have to be moved, when it migrates. We will start by looking at how to migrate this type of object.

Data Objects. For performance reasons Emerald uses direct memory addressing, as opposed to indirect. Therefore memory pointers must be mapped when moving the object.

When moving a data object, the kernel creates a message containing the data area of the object and a table describing how the pointers should be mapped. If the object is a global object the kernel send the OID(the unique Object IDentifier), the forwarding address, and the address of the object descriptor. The OID is used to find the object descriptor on the destination if it is already there. If it is not there the information sent can be used to create a new object descriptor.

If data object is a local object only its address is sent along, which can then be mapped into the enclosing (global) object at the destination.

If the concrete object type is already at the destination, the data area can immediately set its pointer to it. If it is not present, the destination kernel requests it from the source host. In this way the code is only moved if necessary.

The destination kernel will use the template of the data area found in the object's concrete type to traverse the data area and the table received from the source host to remap the data area's pointers to their new addresses.

Active Objects. When an object containing methods is moved, the activation records belonging to these methods must be moved along with the object. To find out what activation records belong to a given object a list of activation records is maintained for each object. For performance reasons the activation records in this list are not linked to the objects unless the actual object is moved. When a move is initiated the list is traversed and all activation records belonging to the object are linked. This approach requires a method to check the list when it returns to see if the activation record is linked and if so unlink it.

When an activation record is moved it is taken out of the stack in which it was previously located. This splits the stack into (at most) three parts. A middle part, the one moved, a bottom part and possibly a top part, see figure 4.7. The top part stays on the source host but is copied onto a new stack segment. Each of the stack breaks are modified from local pointers to remote pointers, since the methods now must return over the network. In order to find the stack-breaks they use activation record templates, found in the object's concrete type. These templates describe the parameters, local variables and the registers used by this method.

The registers are used to optimise access to local variables. When a method is invoked it stores the contents of the registers it wants to use to be able to restore them to their previous values when it returns. When an activation record is moved the activation records above it must be traversed to see if any of them

uses the same registers. If this is the case the above activation record will have a copy of the register contents as they were before it started using the registers. These copies must be sent along with the moving activation record, so it can restore the registers on the destination host to the expected values.

Summing up, the Emerald system was developed for creating object oriented applications, that are distributed across a homogeneous network of hosts. Emerald provides support for migrating objects between hosts. Since objects can be both pure data and a process, object migration subsumes both data exchange and process migration. The hosts are highly reliant upon each other since the application is distributed across all the network hosts.

4.8 Migration Summary

In this chapter we have looked at six different process migrating system and one object migrating system. The purpose of this was to gain a better understanding of how migration was done in tightly coupled systems. In table 4.1 we have briefly summarised the characteristics of the systems in order to created a quick overview.

This summary is split into two. The first part looks at what and how the process' components are migrated in the reviewed systems. The second part looks at what is added by migrating objects.

4.8.1 Migrating Processes

Migrating a process can be split into migrating the state of the process and migrating the code of the process. The process' code is not a problem because it can always be requested via the distributed file system. The part of the code that is currently loaded will be automatically transferred when transferring the virtual memory. The main task of process migration concerns migrating the process' state. This can be split up into the following four things.

- **Virtual memory** - The address space of the process containing all the information computed or loaded at this point in the execution.
- **Open files** - The state which the process associates with files.
- **Process Control Block** - The runtime information kept by the Distributed Operating System at run-time. This could be Program Counter, working directory etc.

	What is migrated?	Migration environment	Load balancing	Residual Dependencies	IPC
LOCUS	Process	Homogeneous	None, Remote tasking	Home structure for process location	RPC via the PID
MOSIX	Process	Homogeneous	Pre-emptive	Home structure for process location	RPC via the PID
Sprite	Process	Homogeneous	Dynamic	PCB on creation site for transparency	?
Charlotte	Process	Homogeneous	Pre-emptive	None, process location and transparency is handled by abstraction mechanism in the kernel or utility process	Via logical links
Accent	Process	Homogeneous	?	Everything is left and transferred on demand	?
Modified V	Process	Heterogeneous	?	None, as it is described in [Shub, 90]	?
Emerald	Object	Homogeneous	None	Forwarding list, other objects which this object calls	RMI via the global OID

Table 4.1: Summary of the systems

- **Channels & links** - The state associated with the Inter Process channels or links currently used by the process.

Virtual Memory. Three different approaches are presented for migrating virtual memory. Each are depicted in figure 4.8. In LOCUS, MOSIX and Charlotte all of the virtual memory is moved to the destination prior to the process continuing execution. This has the advantage of leaving no residual dependencies at the source. One optimisation here is to only moved the modified pages, since the rest can be loaded from the executable which contains the code and the static data.

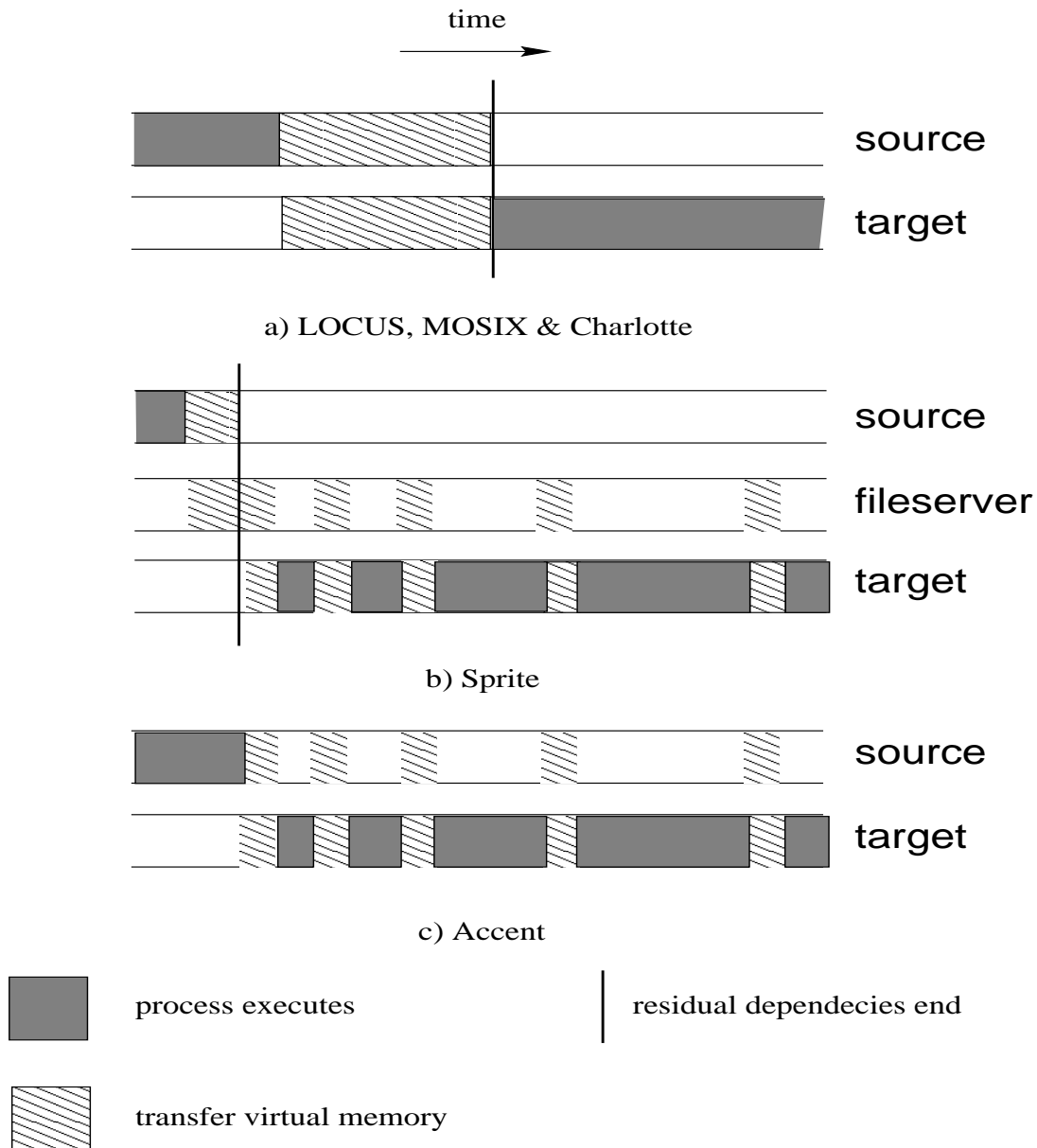


Figure 4.8: Virtual memory transfer

In Sprite all of the virtual memory, ie., the dirty pages, is also moved when migrating. However Sprite decides to place the dirty pages on the distributed file server. The pages can then be requested from the file server when needed. This approach creates no residual dependencies with respect to the source however it does have a residual dependency with respect to the file server. We have not counted this residual dependency in table 4.1 due to all of the presented system being dependent on their distributed file system. In a PCE, however, this dependency can be very crucial and some way of migrating the files in order to make

the individual clients more autonomous are probably a good idea. Otherwise one can choose to access data in a different way than through files, e.g., databases and tuple spaces, see section 6.2.3.

The last approach presented concerning migration of virtual memory is the one used in Accent, which only logically moves the address space. All of the virtual memory stays behind on the source and is transferred to the destination upon reference. This approach spreads out the work of moving the virtual memory and a lot of work can be saved due to some memory pages never being referenced. However the residual dependencies created by this approach makes it unfit for a PCE.

Open Files. The problems involved when migrating open files are related to the state of the files being distributed across the network. The simplest way of handling migration of a file is to close it and open it upon arrival on the destination. This can however cause the file to be deleted during the migration, since another process could believe that it was the only one with a reference to the file and therefore delete it. Another simple approach would be to ask the destination to open the file before closing the file at the source. In this way the file could not be deleted however in Sprite this would cause the local file cache to disabled. In Sprite they have chosen a different approach namely to migrate the state of the open file handle, ie., never close the file.

Process Control Block. In systems providing migration purely between homogeneous hosts this information is easy to migrate. This is due to the memory layout and the data structures being completely equivalent on source and destination. The only problem would be if the information were spread across different parts of the system kernel. Then it would be more difficult to gather it all, but this problem will only arise on basis of a bad system kernel design. In the heterogeneous environment, ie., [Shub, 90], no problems arise due to the architectures running the same distributed operating system with the same data structures representing the process information. However byte-ordering might be different and should be accounted for.

Channels & Links. Migrating links in Charlotte is done by updating the link with the new location of the migrating process, after which the link is transferred to the destination. At the same time the kernels of the communicating processes are notified of the migration. This results in all messages being withheld by the kernels from which the messages originates. The kernel of the migrating process can then request the messages, when the migration is done. In this way there is no need for buffering and forwarding messages.

If instead the system makes use of RPC, some kind of buffer must be used to collect the requests received on the source during migration. The requests should then be forwarded to the destination once the process has been migrated. The hosts from which the requests originated should also be notified of the process' new address. This could be done by attaching the new address to the response of a forwarded request.

In a PCE we do not believe that continuous connections are possible when mobile devices are involved, see section 2.3. A solution could instead be to use the agent model for communicating or maybe to use mobile sockets as presented in 2.6.2 and [Casey, 95].

Heterogeneous Issues. In [Shub, 90] two important issues concerning heterogeneity are presented. The first issue concerns the different code needed for the process to be able to execute on different architectures. In [Shub, 90] this is solved by having all the different types of code in the process executable. It is then only a matter of loading the right code on a given architecture. This approach may work well enough on a limited amount of architectures. In a PCE, however, it is a bad idea due to the number of different architectures present causing a serious increase in code size. If a new architecture is suddenly added to a PCE all applications would somehow have to add the new architecture code to their executable, possibly by recompiling the application with the new code.

The second issue raised when migrating between heterogeneous architectures concerns data layout. Different architectures can have different byte-ordering and different sizes of their primitive types. [Shub, 90] address this by the Greatest Common Denominator approach, ie., he always allocates the amount of memory needed by the architecture which uses most memory. Byte-ordering is handled by mapping the data on migration.

4.8.2 Object Migration

Since object migration subsumes process migration as well as data and passive object migration Emerald has a more fine-grained migration entity. It is possible to create a process migrating system by creating the processes as active objects which are then moved. However it is also possible to create one distributed process consisting of several objects that interact. One of the main differences between a purely process migrating system and Emerald is that the relations between the objects must be handled. This includes handling stack segmentation, when an object is migrated, that has already called a method in another object, as in figure 4.7.

Using objects, the way it is done in Emerald to create a PCE, is not a good idea. The Emerald system was created with focus on small secure networks. Due to tight integration between the hosts, the system is not meant to be used on wide area networks. However one could imagine some of the server applications running in a cluster of homogeneous devices in order to provide enough processing power for the many request likely to occur in a PCE.

Chapter 5

Migration in Agent Systems

We will now turn to another type of system using migration, namely the *Agent System*.

Some of the properties of pervasive applications are similar to the properties of agents. We have therefore dedicated this chapter to investigate the migration mechanisms used to migrate agents. We will start by taking a look at Mole, [Baumann et. al., 98], which features *weak migration*. Then we will take a look at Ara, [Peine, 97] which features *strong migration* and finally we will describe how Migratory Applications, [Bharat & Cardelli, 97], uses agents to migrate applications. Before we start looking at the systems, we will briefly explain the terms weak and strong migration.

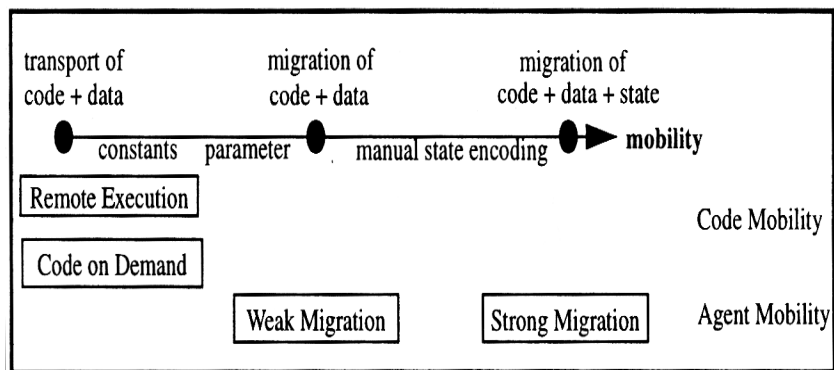


Figure 5.1: Degrees of mobility, [Rothermel et. al., 97]

In figure 5.1 three degrees of mobility are depicted. The simplest form of mobility presented is *remote execution* and *code on demand*, where the code is either pushed onto a client or requested from a server before execution. During execution the code cannot move. Technologies that makes use of this form of mobility

are `rsh` in UNIX and Java `Applets` from Sun Microsystems. This form of code mobility is not enough for mobile agents (or PCEs), since an agent (or PCE Application) should be capable of moving while executing.

The next degree of mobility is weak migration. Here the code and the data state can *migrate* during execution. The data state consists of the contents of the global variables and the instance variables. When using weak migration the application programmer have to take care of threads, local variables and parameters since they are not part of the data state and therefore not migrated. This somewhat complicates the process of writing agents. Some systems featuring weak migration even require the programmer to decide what global variables and instance variables should be migrated. The advantage of weak migration is that it is simpler to implement for the system programmer and when it is not possible to modify the underlying platform it is often the only possible form of migration. This is due to most platforms not allowing the programmer to get hold of the execution state which is needed in order to support strong migration. To give control back to the application after the weak migration, it is required that the application has a `start` method, which sets up the application and continues the execution.

The last degree of mobility is called strong migration. Strong migration differs from weak migration by also migrating the execution state (ie., the contents of the local variables, local parameters and executing threads). Systems providing this kind of migration relieves the application programmer of all worries concerning the actual migration of the agent. To extract the execution state of an agent the system must provide functionality to externalise and internalise the stack and threads. Few systems provide this functionality, so most systems using strong migration are built from scratch. Also since the execution state can be quite large, strong migration may be more time consuming.

With the definitions of weak & strong migration in place, we will take a look at Mole, a system that makes use of weak migration.

5.1 Mole

The Mole Agent System, [Baumann et. al., 98] is built on top of an unmodified Java VM. It is based on the concepts of *places* and *agents*. The overall system contains a number of places on which local resources can be accessed and agents can execute and communicate. Several places can coexist on the same host, but one place cannot be distributed across several hosts in the network.

Agents are active entities capable of moving around between different places in

order to perform their tasks. Each agent is identified by a unique immutable identifier based on their creation site. When an agent moves from one place to another its data state and code are migrated along with it. Places are divided into two categories depending on their connectivity. If a place is continually connected, ie., wired, it is called a *connected place*. If a place is only temporarily connected it is called *associated*. Workstations are typically connected and PDA and laptops are typically associated.

Since Mole is built on top of the Java VM, strong migration, as described above, is not possible. This is due to the fact that it is not possible to capture the state of a running thread in the current JVM or to access the contents of the stack. This is one of the reasons why Mole features weak migration.

5.1.1 Agent Migration in Mole

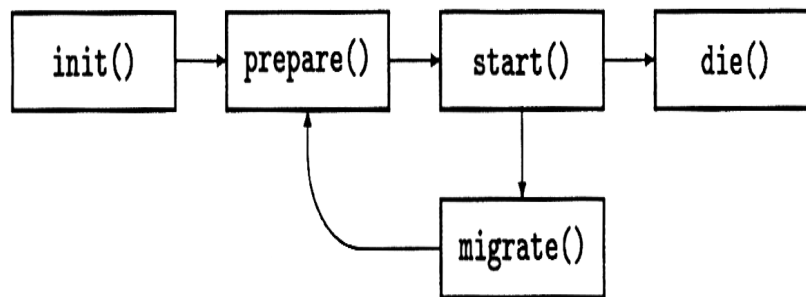


Figure 5.2: The Mole Agent's lifecycle, [Baumann et. al., 98]

Mole makes use of the Java Remote Method Invocation (RMI) package, [Java, 00], to transfer agent code and state. When an agent migrates it calls the `migrateTo` method. This results in no new messages (RPC) being accepted for this agent. After handling pending messages, the agent's threads are stopped and the agent is removed from the list of active agents. The agent is then serialized using the object serialisation feature provided by Java RMI. This serialisation computes the transitive closure of all objects reachable by the agent, apart from transient objects and threads. This platform independent representation of the agent's state is then sent to the new place on which it is reinstated. In case some classes (code) are missing they can be requested either from the source site or from a code server, as described in [Hohl et. al., 97]. Since only the data state can be migrated, the agent has a `prepare` method, see figure 5.2, that is called in order to set up the agent. After this is done the `start` method is called. It is up to the agent programmer to fill out the body of these methods. If all is successful a *success message* is returned to the source, which will then terminate all the suspended threads. If at something went wrong during the migration, the

source threads are resumed and control flow continues from after the `migrateTo` statement. An exception is thrown, which can be caught for error handling.

5.1.2 Example

To get a better understanding of how Mole works we will describe what is required of the agent programmer in order to program a wandering agent which prints its name and its origin on each host it visits. On the following pages we have the code of the Mole Wanderer Agent.

```
public class Wanderer extends UserAgent
    implements MobileAgent
{
    protected LocationName home    = null;
    protected transient LocationName current = null;
    protected String[] listOfHosts;
    protected int nextHostIndex;

/**
 * initialization of the agent
 */

    public boolean init(Hashtable parameters)
    {
        String s;

        s = (String)parameters.get("Home");
        if (s != null)
            home = new LocationName(s);
        else
            return false;

        nextHostIndex = 0;
        listOfHosts = {"Host1", "Host2",
                      "Host3", "Host4"};
        return true;
    }
}
```

Two basic agents are available in Mole: the `UserAgent` and the `SystemAgent`. System agents extend `SystemAgent` and are used to present interfaces on a place to user agents. System agents belong to a specific place. User agents extend `UserAgent` and do not belong to a certain place. They are often mobile but they need not be. If a user agent is meant to be mobile it must implement the

`MobileAgent` interface. This interface is used to tag an agent as mobile and requires no extra methods to be implemented.

As can be seen in the code the `Wanderer` agent is a mobile user agent with four variables. A `home` variable describing the agent's origin, a `current` variable describing the actual location, a list of hosts to visit and an index describing which host to visit next.

The `init` method is called when the agent is created either by the system or by another agent. It takes a hash table as parameter, which can be used to provide the agent with information about its surrounding or other information needed to initialise the agent properly. `Init` is only called when the agent is created not when it has been migrated. For adapting an agent or setting up its execution environment, Mole agents have a `prepare` method that is called by the system just after `init` or after a migration. In our example we use it to retrieve the current location of the agent. In more advanced agents it would be used to set up threads or other execution state that is not migrated in Mole (it featuring only weak migration, see start of chapter).

```
/**
 * final preparations before the agent comes to life
 */

    public boolean prepare()
    {
        current = getCurrentLocation().locationName();
        return true;
    }
```

After the `prepare` method has returned the system creates a new thread in which it calls the agent `start` method. This is the point where control is given back to the agent and it is in this method that the actual agent computations should be placed (or in methods called from it). Below we can see how our example agent prints a message on the current location after which it migrates on to the next host.

```
/**
 * start callback
 */

    public void start()
    {
        // awake after migration or creation

        String currentName = current.toString();
```

```
String homeName = home.toString()
Engine.out("Wanderer Agent: arrived at " +
          currentName + " originating from " + homeName);
}

if (!currentName.equals(homeName))
{
    LocationName nextHost =
        new LocationName(listOfHosts[nextHostIndex]);

    migrateTo(nextHost);
}
else
{
    Engine.out("Home, sweet home");
    die();
}
}
```

During the call of `migrateTo` the system calls the agent's `stop` method which may be used by the agent programmer to perform some last minute computation or clean-up before the agent is serialized by the system and transferred to the new place.

The Wanderer Agent uses this method to increase the `nextHostIndex` to make the agent migrate to the next host in the `listOfHosts`.

```
/**
 * called before the actual migration
 */
public void stop()
{
    nextHostIndex++;
}
}
```

5.1.3 Summary

Mole is based on places and agents. Places are logical platforms on which the agent can meet and communicate. Places provide different kinds of service in form of stationary service agents. Most agents are however mobile and capable of migrating from one place to another. Agents in Mole may be multi-threaded but since Mole features weak migration, the execution state of an agent is not

migrated when the agent moves. Weak migration is also the reason for having the `start` and `prepare` methods, since an entry point in the agent is needed where the execution can be resumed. By placing some of the burden concerning agent migration on the agent programmer, the Mole developers are capable of providing an agent system, that can run on top of an unmodified Java VM. This may be beneficial in terms of getting other people to use Mole.

5.2 Ara

The Ara Agent System, [Peine, 97] is currently implemented on MS Windows and in several UNIX dialects. As in most other agent systems Ara is based on the concepts of agents and places. Each place features an Ara core on top of which Ara processes may run, see figure 5.3. The core is the run-time system for the agents providing them with all the language-independent functionality they need, such as a communication API and a GUI API for accessing external resources. Ara agent can be programmed in both C/C++ and TCL and more languages can be incorporated. The requirements for incorporating a new language is that an interpreter exists for the language such as the MACE virtual machine, [Peine, 97], for C/C++ and the TCL interpreter, [TCL, 94], for TCL.

The language-dependent functionality needed by the agent to access the Ara core is placed in the agent interpreter. e.g., the TCL interpreter is modified to provide stubs by which a TCL can access the Ara core and likewise for the MACE virtual machine. The interpreter together with the agent constitutes an Ara process. This makes it possible to have agents, written in different languages, running side by side on the same Ara core.

The Ara system features strong migration as opposed to Mole. This is possible because Ara has its own process abstraction. Thereby access to all information needed to migrate the execution state of an agent is provided. How the agent is migrated is addressed in the following section.

5.2.1 Agent Migration in Ara

When an agent wants to migrate it calls `ara_go`, as in figure 5.4. `Ara_go` is an Ara core system call, which handles the migration of the agent code and state. Before migrating an agent its state is transformed into a portable, platform independent representation. This representation is then remapped to the destination site's specific requirements, thereby handling byte-ordering and primitive type differences.

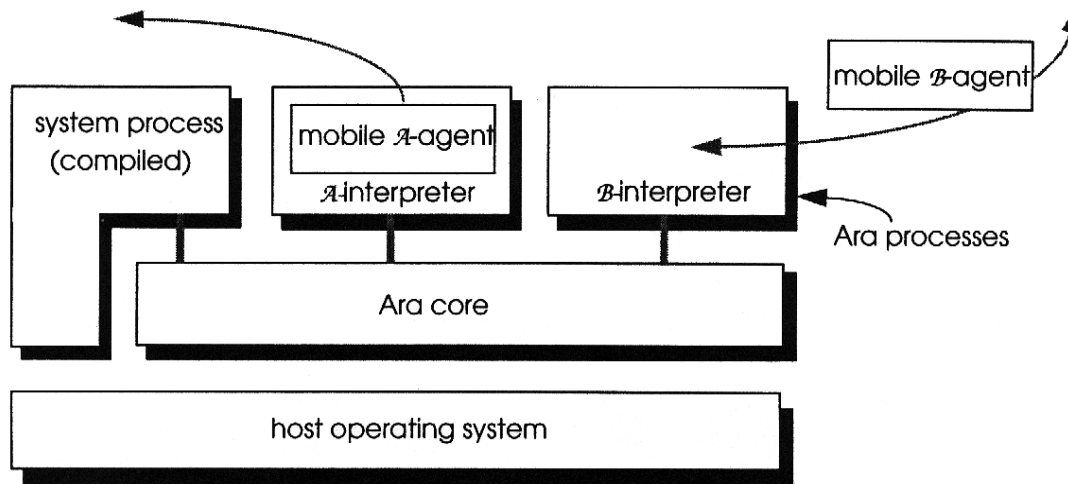


Figure 5.3: Ara System, [Peine, 97]

```

ara_agent {
  set home [ara_here]
  ara_go thor
  puts "Hello at thor, I've come from $home!"
  ara_exit
}

```

Figure 5.4: An Ara agent, [Peine, 97]

The agent's state is composed of two parts. The state of the agent interpreter and the state of the underlying Ara process in the Ara core. The Ara core transforms the state of the Ara process itself. In order to transform the state of the interpreter the core makes use of a dedicated function defined by the interpreter, see figure 5.5. This function (ie., upcall) takes care of transforming all the state of the interpreter. This involves transforming the run-time stack into a portable form, which can then be transformed back upon arrival to the destination. The implementation of this function is the major challenge in adding a new interpreter to the Ara system. The implementation can make use of Ara core functions for transforming primitive types.

When the agent's state has been transformed, the state and code are sent to the destination site via the communication process, a stationary agent which listens for incoming agent-parcels and sends out-going agent parcels. When an agent-parcel is received it is passed on to the Ara core.

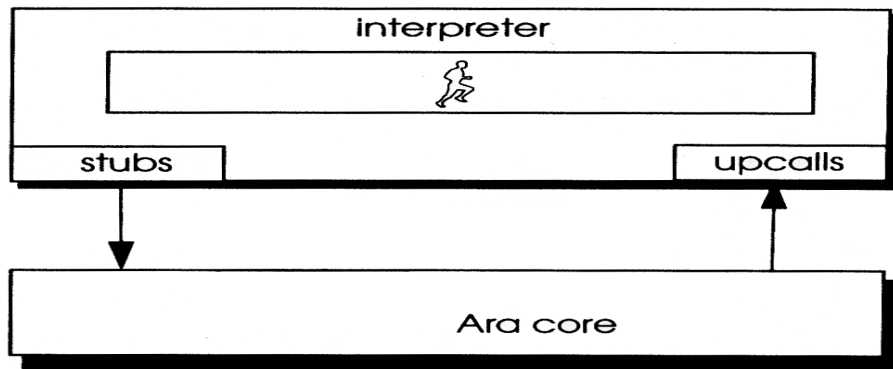


Figure 5.5: Ara Core & interpreter, [Peine, 97]

5.2.2 Example

We have created a similar agent in TCL for the Ara system, as the one presented in Mole.

```
set Wanderer [
  ara_agent {
    set home [ara_here]
    set listOfHosts {Target1 Target2 Target3 Target4}

    foreach target $listOfHost {
      ara_go $target
      puts "Wanderer Agent arrived at $target originating from $home"
    }
  }
]
```

```
    ara_go $home
    puts "Home, sweet home"
    ara_exit
  }
]
```

As can be seen the code is substantially smaller than for the Mole agent. This is because Ara features strong migration which means that execution is continued right after the `ara_go` call. This relieves the programmer of worrying about how to set up the execution state and it makes it possible to produce more readable code. However it also requires more work by the system, since it has to extract the execution state of the agent as well as the data state.

5.2.3 Summary

The Ara agent system provides strong migration for migrating agents. This is achieved by having their own process abstraction giving them complete control over the Ara process, as well as access to its state. Ara is capable of running agents written in several different languages. The interpreters for these languages must be modified to be able to transform the agent's state into an independent representation. This representation can then be transformed back on the destination. The interpreters must also provide an interface to the Ara core functions to give the agents access to the system resources.

5.3 Migratory Applications

The last system we will look at is Visual Obliq, [Bharat & Cardelli, 97], a framework for developing migratory applications. The system is based on the untyped, distributed scripting language Obliq.

We will start by taking a look at the Obliq language and its network semantics. Then we will describe the Visual Obliq system architecture, ie., its run-time system, and its application structure. Finally, we will look at how the migration mechanism works.

5.3.1 Obliq

Obliq is an object-oriented interpreted language supporting distributed lexical scope. It provides support for migrating code between different hosts in the network. Supporting both distributed lexical scoping and code migration can

cause a problem due to a *variable identifier* being bound to a variable on one site after which the code (and the variable identifier) is transferred to another site, ie., another context. To solve this problem, Obliq transforms the local pointer of a variable identifier into a network reference, thereby the variable identifier will denote the same variable independently of the context in which it resides. Obliq is built using the Modula-3 network library and network objects which makes it easy to have network references.

The distributed lexical scope gives Obliq a very powerful distributed semantics, which makes it possible to seamlessly move code from one site to another. We found the description of the Obliq semantics to be very complex which may show in the following explanation.

Obliq Distribution Semantics. The distributed semantics of Obliq is based on the notion of *sites*, *locations*, *values*, and *threads*, [Cardelli, 95].

Sites are address spaces and specific for a given host. It is possible to allocate locations, ie., addresses, at a site, figure 5.6. Locations are place holders for values. Locations always belongs to a specific site and cannot move. The value of the location can however be copied to a new location, possibly on a different site and the value of the original location can be changed to point to the new location. Locations are also called mutables, since their contents may change. They are denoted by variable identifiers.

Values are also called immutables, since they cannot change. Obliq values include *basic values*, *objects*, *arrays* and *closures*. Basic values are values such as integers, string and other primitive types. Objects, arrays and closures are complex values containing locations, ie., an object and an array can be thought of as a collection of *embedded locations*. A closure is some source code (ie., a procedure) and a table containing the variable identifiers denoting the locations needed to provide a closure for the code.

In objects, the embedded locations are the objects fields, in arrays they are the elements, and in closures they are the free variables. Values are denoted by *constant identifiers*.

In contrast to locations, Obliq values can be transmitted over the network. However in the case of objects and arrays, only a reference is transmitted since they consist purely of locations which are specific to a given site and cannot move. An Obliq closure consists of procedure code and a table containing variable identifiers denoting the free variables needed by the procedure. When a closure is moved the procedure code (an immutable value) and the table containing the variable identifiers are moved to the destination site. All the variable identifiers will be

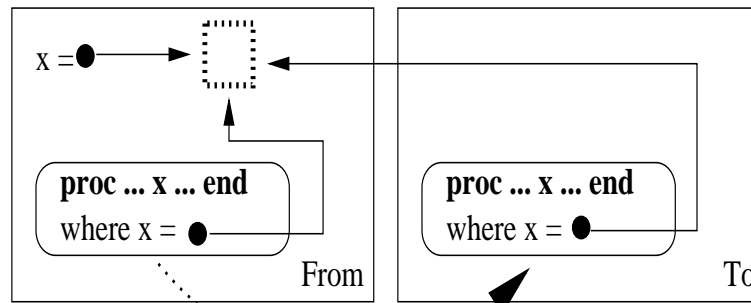


Figure 5.7: Transmission of a closure in Obliq, [Bharat & Cardelli, 97]

```

let remoteInvoke = proc(host)
  var x = 2;
  try
(1)    let networkRef = net_importEngine("RemoteObj", host);

(2)    networkRef(proc(inc) x = x + inc end);
      true
    except else
      false
    end
  end;
end;

```

Figure 5.8: Example of remote invocation and code transmission

procedure that resides on another host. The method or procedure can be reached through a network reference that can be obtained either by using a name server or as the return value from a method or procedure. As an example see figure 5.8 in which a network reference is obtained at (1) which is then invoked with a procedure (ie., some source code) as arguments at (2). Since Obliq always transfers closures and not just the source code, the invocation will result in the procedure being moved to the destination with a table containing the variable identifier x which will be converted to a network reference that points back to the source host as in figure 5.7. On the destination, the argument `inc` should be provided to the procedure which will then add the value of this argument to x that still resides on the source host. It is important to notice that it is the transitive closure of values that are transmitted. If the procedure given as argument in figure 5.8 had nested procedures these would have been moved as well. However locations are not moved, instead network references are created that points back to the site on which the location resides.

Even though locations are not automatically moved, Obliq has a `copy` primitive

that, given a network reference, will copy all the values and the contents of the locations to the site on which it is executed, figure 5.9. It makes use of the `alias` primitive to change the value of all the copied locations to make them redirect future accesses to the new site. In this way it is possible to simulate that an object has moved. The `copy` primitive is useful when migrating applications as we shall see in section 5.3.3. In the next section we will describe the system architecture of the Visual Obliq framework.

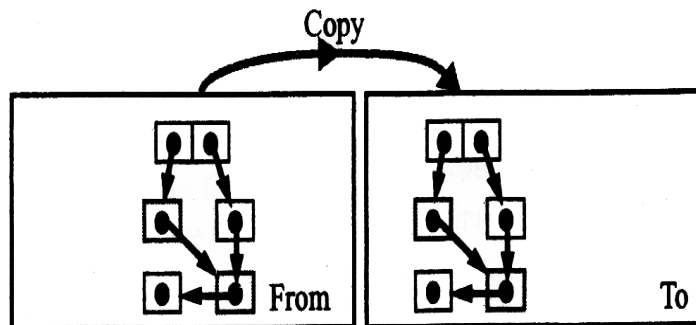


Figure 5.9: Copy of both mutable and immutable nodes, [Bharat & Cardelli, 97]

5.3.2 System Architecture

The Visual Obliq framework is composed of *Agent Servers* and *migratory applications*. The migratory applications can be thought of as mobile interactive agents (we will use the term migratory applications). These migratory applications can migrate from one Agent Server to another by calling `migrateTo`, a system call that via a sub agent migrates the state and code of the migratory application to the destination Agent Server. Exactly how this is done is described in more detail in section 5.3.3. We will here describe the structure of the Agent Server and the migratory application.

The Agent Server is an extended Obliq interpreter and a runtime system with support libraries for migration and for accessing local resources such as local UI components, network communication, files, and processor. Migratory applications can only access local resources through the interface provided by the Agent Server.

The Agent Server can be thought of as a simple compute engine accepting code and executing it. Each time it receives a new piece of code, it provides it with a *Briefing* (this is similar to the procedure in figure 5.8 where the `inc` argument is

provided by the Obliq interpreter on the destination host). The Briefing describes the local resources which may be accessed by the new code.

Migratory Applications are applications built of forms and widgets. Forms are equivalent to windows and widgets are the buttons, text fields, frames etc, that can be placed in a form. When an application is started, an instance of each of the forms composing the application is created. Each form may be extended with new functionality or data fields. This code is local to each instance of a form and is called *Form support code*. When a widget is added to a form, the programmer specifies what actions should happen when the widget is activated, e.g., a button is pressed. This code is called *callback code* since it is called when a local UI component representing a widget is activated, ie., it is a callback from the local UI component library to the actual application. The programmer may also attach *global code* to the application. This code is accessible from all parts of the application and is typically global variables used for maintaining some global state. Finally a programmer may specify *session-constructor code* which is additional code that should be executed when the initial form instances are created.

When an application wishes to migrate, it uses the `migrateTo` system call, described in the next section.

5.3.3 The Migration Mechanism

Migration in Visual Obliq is handled by the `migrateTo` system call. It is stated that this call must be made from some of the callback code in the application. Why it cannot be called from a global procedure or some form support is not clear to us. The signature of the call is:

```
migrateTo(host)
migrateTo(agentServer, host)
```

It is possible to specify what Agent Server on the destination host the application wishes to migrate to. If this is not specified the default Agent Server is assumed. `MigrateTo` performs the following steps when called:

1. It contacts the Agent Server at the destination to make sure that the migration is allowed. If not it returns `false`.
2. If migration was allowed, each form and widget is traversed and updated with the information in the local UI component equivalents, ie., the user interface state is collected.

```

let migrateTo = proc(dest)
  try
(1)    let AgentServer = net_importEngine("VORceiver", dest);
        foreach formList in allForms do
(2)      packFormList(formList());
        end;

(3)    AgentServer(proc(arg) agent(copy(continuation), arg) end);
        true
    except else
        false
    end
end;

```

Figure 5.10: Definition of `migrateTo`, [Visual Obliq implementation]

3. The user interface is destroyed, breaking links between the application's state and the running UI thread.
4. Links to the runtime system are removed.
5. A suitcase is created which contains links to all relevant application code and state. A sub-agent is sent to the destination where it will copy the suitcase, ie., the application code and state to the destination and set it up.

Step 2 is necessary because much of the information about the size and content of a widget is in the actual UI component displayed by the current system. This information should be collected before the application is allowed to migrate. Once it is done, the links to the actual UI may be removed. The UI is automatically reclaimed by the Obliq garbage collector in step 3. In step 4 other links to the run-time system such as files and sockets are removed. It is up to the application to ensure that necessary information concerning these resources is collected before the call of `migrateTo`. In step 5, the actual migration of the application code and data is performed. To understand how this is done lets take a look at the `migrateTo` code, defined in figure 5.10.

At (1), `migrateTo` checks to see if it can access the remote Agent Server and if so obtains a network reference to it. At (2), all forms and widgets are traversed and their state is checkpointed. This results in a continuation (the suitcase mentioned in step 5), figure 5.11, that holds references to all code and data reachable in the application. During the traversal all links to the runtime and the UI have been broken. At (3), the network reference is given a procedure, `proc`, as argument. As described above this results in the procedure and all immutable nodes, ie., the agent code, being sent to `AgentServer` for execution. At the `AgentServer`

```

    let continuation = proc (local)
(1)  foreach formlist in allForms do
(2)    unpackFormList(formlist());
      end;
    end;

```

Figure 5.11: Definition of the `continuation` procedure, [Visual Obliq Implementation]

```

    let agent = proc(contin, local)
      contin(local);
    end;

```

Figure 5.12: Definition of the `Agent` procedure, [Visual Obliq Implementation]

the procedure is provided with a Briefing as described in section 5.3.2, ie., `arg` now contains a briefing. Next a local copy of the `continuation` is created using the `copy` primitive and both arguments are passed to the `agent` procedure, see figure 5.12. The `agent` is the sub-agent used for transporting the application.

The `continuation` argument is actually a procedure that unpacks the checkpointed user interface, see figure 5.11 (2). It is important to notice the `allForms` variable identifier that denotes an array holding references to all the forms part of the application. It is through this variable identifier the application code and data is reached.

The invocation of the `continuation` argument is done in the `agent` procedure, figure 5.12, where it is invoked with the local briefing supplied by the Agent Server.

5.3.4 Example

The focus of the Migratory Applications framework is not to produce agents, but to produce applications that can be migrated. One of the differences is that the application is always communicative, ie., has a user interface, which is not always the case with the agents mentioned in section 3.3.

The example we will use to illustrate how migratory applications are created is borrowed from [Bharat & Cardelli, 97]. The example is an application, that visits a number of hosts. On each host the application asks the user some questions. It is possible for the user to add another host and user to the list of destinations, which the application is going to visit.

The process of building a migratory application starts in the Visual Obliq GUI builder. Using this builder, forms and widgets are created and each form and widget can have some code attached.

The example contains three forms, two toplevel forms, figure 5.13 and figure 5.14, and one pop-up form, figure 5.15.

Figure 5.13: CommentsForm, [Bharat & Cardelli, 97]

Figure 5.14: QuestionsForm, [Bharat & Cardelli, 97]

When the application arrives at a host, it presents the two toplevel forms to the user. When the user has filled in the questionnaire and any comments he may have, he clicks **Done**, which will migrate the application to the next host in the list. He may also choose to click **Suggest Someone** which will pop-up the Suggest Form in which he can suggest a new host and user which the application should visit on its tour.

When creating a new application the default is to create one instance of each form. The forms may then be reached through `CommentsForm[0]`, `Questionnaire[0]`

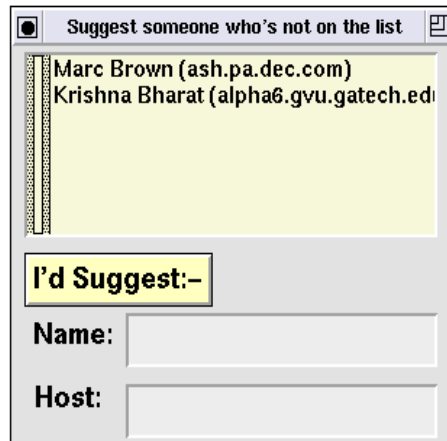


Figure 5.15: SuggestForm, [Bharat & Cardelli, 97]

and `SuggestForm[0]`. This default initialization is enough for this application and no extra session-constructor code is needed.

To keep track of the people and hosts the application must visit, the following global code is added.

```
var NumVisited = 0, people = [], hosts = [];
let OriginalHost = volibLocal.getHostName();
```

It is then up to the first user to use the suggest form to add people and hosts for the application to visit.

To make the suggest form pop-up the `Suggest Someone` button has the below callback code attached.

```
SELF.SuggestForm.show();
```

Here `SELF` refers to the `CommentsForm` in which the `Suggest Someone` button is placed. The `SuggestForm` is anchored to the `CommentsForm` which is why it can be reached by `SELF.SuggestForm`.

When the user has typed in the user he wants to suggest, he can add the data to the global `people` and `hosts` arrays by clicking `I'd suggest`. The callback for this button is:

```
let name = SELF.Name.getText();
let host = SELF.Host.getText();
```

```
people := people @ [name];
```

```
hosts := hosts @ [host];
SELF.Agenda.append(name & " (" & host & ")");
```

The callback for the slider in `QuestionsForm` copies the current slider value into a field named `Qn2`.

```
let n = SELF.Success.getValue();
SELF.Qn2.putText(fmt_int(n));
```

We now only need to attach some callback code to the `Done` button. The `Done` button should gather all the information given by the user and place it in the `Transcript` window in the `CommentsForm`. After this is done the code should check whether all hosts in the `Agenda` have been visited and if not migrate the application to the next host. If all hosts have been visited the application should migrate to `OriginalHost`.

The code looks as follows:

```
let comments = SELF.Comments.getText();
SELF.Comments.putText("Please Type Your Comments Here");
SELF.Transcript.appendText(people[NumVisited]
    & " said\n" & comments & "\n"
    & " Qn 1: " & QuestionsForm[0].Qn1.getChoice()
    & " Qn 2: " & QuestionsForm[0].Qn2.getText() );
loop
    NumVisited := NumVisited + 1;
    if SELF.Agenda.numElements() is NumVisited
    then
        dest := OriginalHost;
    else
        dest := hosts[NumVisited];
    end;
    if MigrateTo(dest) then exit end
end;
```

Once the above code has been typed in, the Visual Obliq builder is capable of generating a Migratory Applications with a user interface. The application will perform a survey, where it visits the hosts added to its `Agenda` and asks the user on each host two questions. All the answers are gathered in the transcript editor which can then be read by the user initiating the survey.

5.3.5 Summary

The Visual Obliq framework provides the programmer with primitives for migrating an application between network hosts. The migration mechanism is capable of moving the application including its user interface from one Agent Server to another. This is done by checkpointing the state of the user interface into a portable format which can then be restored at the destination. All of the application's state and code is moved so no residual dependencies are created. However files and sockets are not taken into account by the framework, so if an application uses such it must account for them itself before migrating. Visual Obliq applications can migrate to any host that presents an Agent Server because the Agent Server provides a hardware-independent platform with an extended Obliq interpreter and a runtime capable of interpreting the Obliq scripting language.

The distributed scope in Obliq gives problems when migrating applications. In case two applications (A and B) share some of their state, the following situation will occur when one of the applications (B) migrates to another host. The state of the B will be copied by the Visual Obliq framework to the new host. At the same time the `copy` primitive, see section 5.3.1, redirects the locations on the source host to the new locations on the destination host, see figure 5.16. This results in A's state being distributed across several hosts thereby making B dependent on the destination host. This situation may get arbitrary complex in case more applications are involved.

The problem arise due to the way boundaries are defined for applications, viz. that all state reachable from the application's instances should be moved. We have recently found out that the creators of the Visual Obliq framework have come up with a new framework in which boundaries are defined in terms of *Ambients*, thereby avoiding state overlap between two applications.

An Ambient is defined as a confined place in which computations happen. It has a unique unforgeable name, a collection of processes, and a collection of subambients. Ambients can move in and out of each other subject to *capabilities* associated with the ambient names.

It seems to us that Ambients will solve many of the issues present in a PCE. However at the time of writing, Ambients only exist as a definition in a calculus [Cardelli, 99]. This calculus is still under development and we have therefore not addressed Ambients further in the thesis. However the idea seems very interesting and should be investigated in future work.

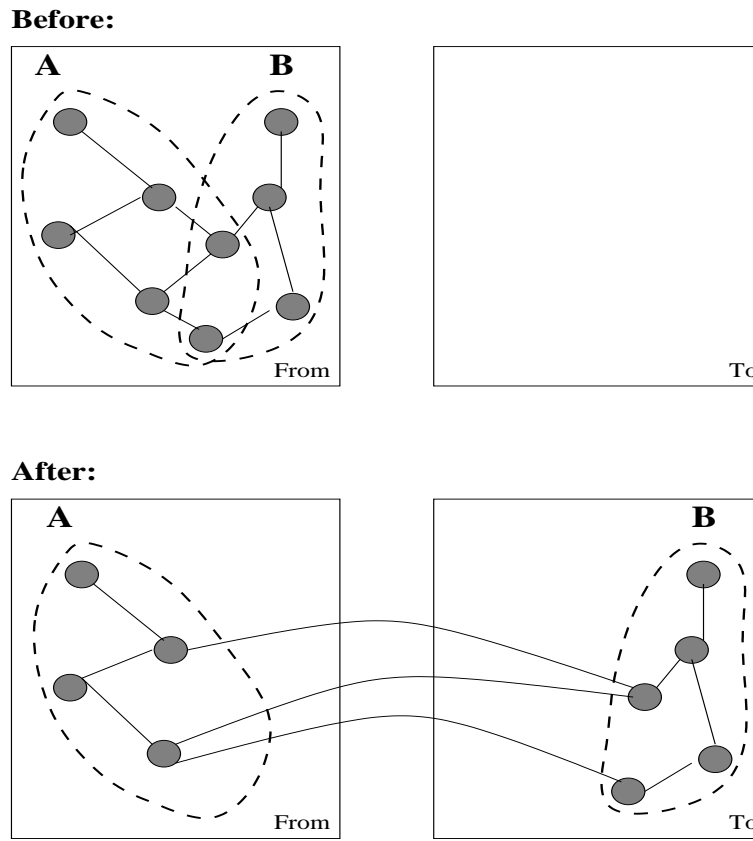


Figure 5.16: Distribution of application state

5.4 Comparison

In table 5.1 we sum up the differences between the above systems.

System	Migration entity	Migrates code	Migrates data state	Migrates execution state	Migrates ext. resources
Mole	Agent	X	X	-	-
Ara	Agent	X	X	X	-
Migratory Applications	Application	X	X	X	Yes, user interface

Table 5.1: Comparison of migration systems

The main difference between Ara and Mole is that Ara migrates the execution state. This involves migrating local variables, parameters and contents of threads. For this to be possible Ara has its own process abstraction, thereby getting access to the needed information.

Both Mole and Ara requires the application programmer to explicitly handle any external resources before migrating an agent. This implies that an agent accessing a file should itself close the file before migrating, ie., the agent should be aware of that it no longer has access to the file once it has migrated.

Regarding expressive power it is possible to create equally expressive agents using Ara or Mole. However more work has to be done in Mole in order to capture and re-establish the agent's execution state.

The Visual Obliq framework differs by migrating applications instead of agents. This implies migrating the user interface as well. The user interface represents several external resources in terms of windows, and buttons. To capture this state the Visual Obliq framework checkpoints the user interface before migrating, thereby copying the state information.

We would like to note that we are not sure whether Mole is capable of migrating the user interface of an agent. To our knowledge it is possible to serialize and send Java AWT components, but whether they may be used in Mole agents, we do not know.

Chapter 6

System Architecture

In the previous chapters we have looked at the basic characteristics of a PCE and how migration could be used to provide pervasive applications. In this chapter we will describe how we think the architecture of the underlying system should look in order to best support the requirements of a PCE. Lets start by summarizing the main characteristics of a PCE.

Computer devices are everywhere. All devices can interact via a wide area network such as the Internet.

Users have access to devices in various work situations. Due to changes in their work situation or location they may shift device either when working at the same project or while changing between different projects. In a PCE, shifts between devices are seamless since the user should not be concerned with how to gain access to his applications or how to keep his data synchronized.

As we have argued in chapter 2, a PCE supporting seamless shifts should at least make it look as though the user's applications follows him from one device to another. We have also argued in chapter 3 that the best way to do this is by letting the application actually follow the user, ie., migrate the entire application from one device to the next.

These characteristics imply new requirements to the architecture of the computing system and the applications in it. Devices, applications as well as users should be able to move freely in a PCE and still be able to communicate or access remote resources. The system should be able to adapt dynamically to the environment as it changes. To meet these requirements, we have recognised four parts of the system architecture that must be addressed; the communication layer, the platform, the migration mechanism, and the applications running on top of the platform, see figure 6.1.

In the rest of this chapter we will discuss the issues that arise with-in each of

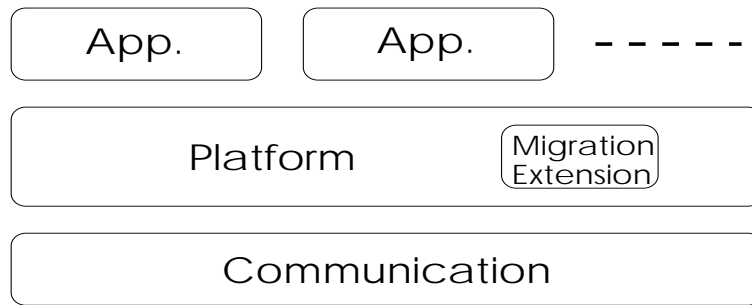


Figure 6.1: System architecture model

these areas.

6.1 Communication

Three important issues, viz. *network*, *locations*, and *latency*, related to a PCE should be handled in the communication layer. The network enables the user to access remote resources in a PCE. Locations arise due to the user and his device being mobile and therefore dynamically change location during the work process. Latency will have to be addressed by the communication architecture because of the size of the area the network spans and because of the use of wireless connections, see section 2.3. We will start by looking at the requirements of the network.

6.1.1 Network

The network is the hardware technology and the low-level protocol used to communicate between the devices. It should enable the users to access any remote resource which the user has permission to use. Two approaches may be chosen to accomplish this. Either we demand network access to all resources and devices on the net or we ease the network access demand and use a cache when the network is not present. The latter idea is used in [Banerji et. al, 93], in which the user's Mobile Computing Personae (MCP) follows the user around using the net when possible and when not the MCP is cached either in the user's PDA or in a badge he is wearing.

By having the entire mobile computing personae, containing all of a user's resources, cached in a mobile device, the different devices do not necessarily have to be connected to a network at all since the MCP could just be transferred by an internal connection such as an infrared transmitter. This way some of the problems we will be investigating in this chapter would vanish.

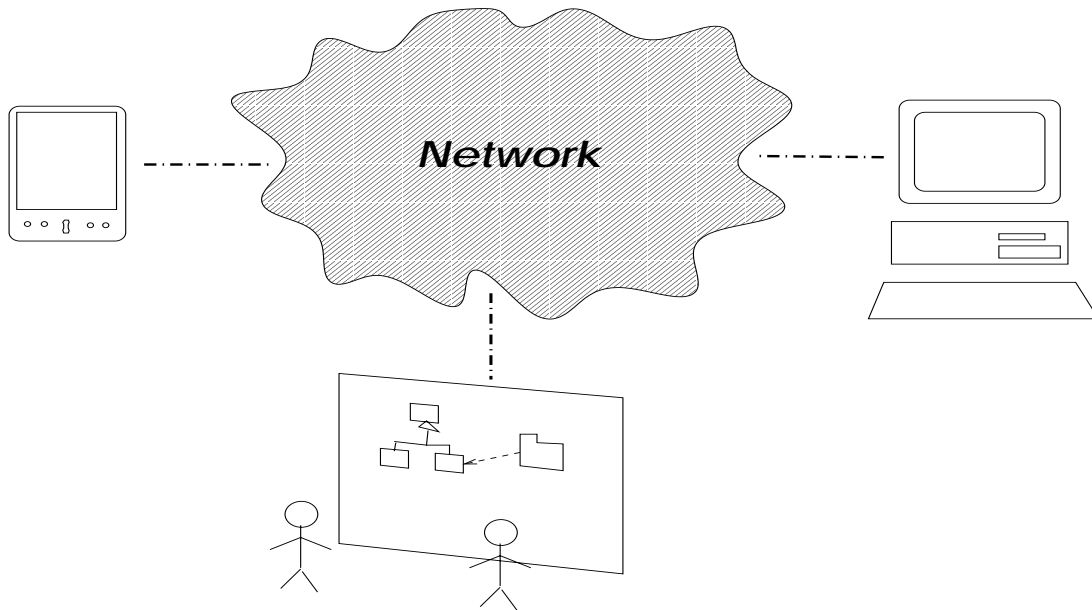


Figure 6.2: Different types of devices connected to the network

However, as described in chapter 2, we do not believe that it is possible to migrate the whole MCP every time a user move out of range of the network. One of the problems is how the computer and the network figures out that the user is leaving the network. Also sharing resources using the cached approach is very difficult, e.g., if a shared file is located in somebody's cache it is impossible for other users to access it.

Thus access to the network should be available at all times in a PCE. This gives the user distributed access to shared resources from every location of the network. The network should cover the entire area of a PCE. Outside of the network, resource access will no longer be seamless and the idea of pervasiveness will fail.

Domains. Since the movements of a user are not confined to his normal working environment but may span large areas such as whole countries or even the entire planet, we need a network that covers as large an area as possible.

By using the Internet, we will have worldwide coverage and the user would be able to move to any location from which he can access the Internet.

Due to the hierarchical structure of the Internet, a notion of domains emerges. Each domain is often subcomposed into many smaller domains that each are locally administrated. Each of these smaller domains will typically span a certain area belonging to a specific company or organisation. Due to this company's fear

of attacks from the outside, the domain will often be protected by a firewall. This creates small islands that each require the user to be authorized before he gains access to their resources. The resources available will also depend on the domain in which the user is currently located, as described in chapter 2.

Mobility. The mobile devices present in a PCE requires the network to feature wireless technology for accessing resources while moving. However, in the case that the user is not mobile it is beneficial to use wired technology due to its reliability and speed. We therefore propose a mixture of wired and wireless networks to ensure the best form of connectivity for the user. Examples of wireless technologies that could be used are Bluetooth, DECT, and the mobile phone net. For wired networks, the existing ethernet technology and the ordinary phone net may be used. For communication in the same room, devices may use infrared transmitters or cables.

A side effect of wireless connections is that if the user moves out of range of the current connection point, the connection should automatically change to a new, if possible.

Since the user is going to use the network from a variety of devices another issue must be addressed by the network technology. The network should handle the diversity of the different devices present in a PCE, figure 6.2. It is important that communication does not have any hardware specific requirements. This implies that some standard protocol, such as the IP protocol should be used for communication.

6.1.2 Location

A phenomenon that will arise in network communication is *location*. A location can either be physical or virtual. A change in physical location happens, e.g., when the user shifts from one device to another, or when the user with his mobile device moves through time and space. In doing so the user may be assigned a new local printer or other local resource, i.e., changing physical location may cause the local resources available to the user to change as well.

The notion of virtual locations arise because of the presence of distrustful domains as described in 6.1.1, in which we describe why resource access is dependent on the domain in which the user is located. A user may change virtual location by connecting to a different wireless connection point belonging to a different domain. This could cause the user to be denied access to the files he was previously accessing, since he does not have the same privileges in the new domain. It may also cause the user to have to pay for using a certain resource such as the printer, because in the new domain he does not have status of being an employee.

Due to the user and the device being able to dynamically change domain, some way of specifying locations that is not tied to a certain domain is needed. On the Internet the IP protocol is used for communicating between devices, identifying each device with an IP number. These numbers are specific for the domain and it will therefore require the device to be assigned a new IP number each time it changes domain. If this happens while the device is talking to some other device the communication will be lost. It is therefore necessary with a protocol or communication mechanism that is able to follow the user and his applications.

A solution could be the mobile sockets used in [Casey, 95] enhanced with some identification, ie., a passport that applications and users could authorise themselves by as described in 2.6.2.

6.1.3 Latency

The last issue related to communication is *latency*. If the network spans an area as big as the Internet there will inevitable be latencies in the communication purely due to physical distances. A procedure call to the antipodes on earth requires at least 1/10th of a second to get from the source to the destination, independently of future improvements in networking technology, simply due to the speed of light.

Another reason for latencies to arise is that some of the communication in a PCE will take place over wireless connections. As describe in section 2.3 wireless connections often disconnect for short periods of time because of noise, shifts in connection points, and other interruptions. To the user of the wireless connection this will look as latency in the communication.

One way of addressing this issue is by using agents for communication. We believe that in a few years, the bandwidth of both wired and wireless connections will reach a level that will make the amount of data transferred unimportant and only the latency will be left to affect the response time. Thus sending more data and code in the form of an agent will not be a problem. The benefit of using agents is that all intermediate messages (messages other than the first request and the final result) are no longer needed. In this way, the communication is less affected by latencies.

Associating agents with an application and a user, through a application id and a global user id, can also be used to address the problem of changing domains. When an agent is about to return to the client device it passes the user id and application id to the underlying system which will then locate the current location of the client. In this way the agent need not know the location of client while performing its task but only at the time when it wishes to return. If the user is

no longer present in a PCE the agent could return to a predetermined docking station, as in [Gray et. al., 96], where it could reside until the user reconnects.

6.2 Platform

In this section we will investigate the properties of the platform on which applications are running. First of all the platform should be able to execute the application. On top of this it should provide access to both local and remote resources. Another important feature of the platform is the migration mechanism which we will investigate in more detail in section 6.3. The platform layer is the middle-ware connecting the applications with the underlying communication layer and hardware.

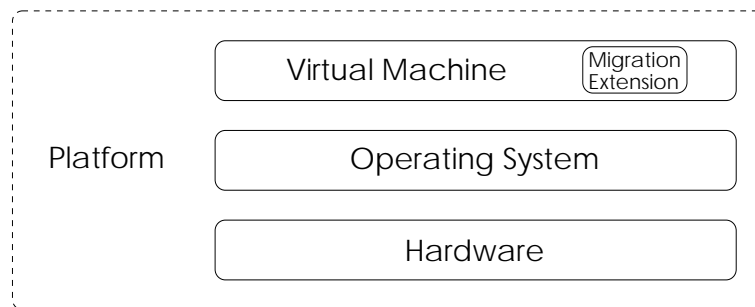


Figure 6.3: Platform architecture model

Since the user is going to use a variety of heterogeneous devices and because applications are going to migrate between these devices, the architecture of the platform should provide a unified environment for execution and for accessing resources.

Besides providing a unified environment, the platform must be able to adapt itself to the surrounding environment as well as adapt incoming applications to itself. The platform adaption is needed when the device enters new domains or when the present domain changes. The application adaption is needed for adapting arriving applications to the platform environment.

For the platform to meet these requirements it has to handle code, user interface and resources. Lets start by looking at how we provide a uniform execution environment for the code.

6.2.1 Code

For migrating applications to be able to execute on different hardware devices running different *Operative Systems* (OS) two approaches can be used. One is to adapt the code to the platform, ie., code mapping, as in [Shub, 90]. Another is to adapt the platform to the code, ie., use a virtual machine to interpret the same code on all platforms.

Due to the reasons mentioned in section 4.6.2, code mapping is not a feasible solution in a PCE. Code mapping would require every device to be able to map code from any other existing device requiring a huge programming effort. If the approach used in [Shub, 90] is chosen, in which every applications carries code for each hardware architecture it may meet, the size of the executable would explode.

We therefore find the virtual machine a much better solution since this only has to be made once for each platform. Of course this implies that only programs translated into byte code are capable of migrating. Virtual Machines has previously been perceived as slow compared to natively compiled languages, however with the newest advances in dynamic compilation, e.g., HotSpot [Java, 00], this gap seems to be less significant.

6.2.2 User Interface

As described in section 2.6.1, it is necessary with an adaption of the user interface. This is due to different hardware layout and interacting mechanisms used on different devices. This adaption can be handled in various ways. [Casey, 95] suggests using the XWindows protocol. This gives a unified way of accessing UI components such as windows, buttons and other widgets. It however does not provide any adaption of these components to the platform. A window looks the same on every platform.

Another solution is presented in 2K by [Román et. al., 99], where the contents of the user interface is represented by XML documents. On each platform the contents of the XML documents are interpreted according to the Cascading Style Sheets present on this device. This makes it possible to adapt the user interface according to the device while at the same time relieving the application programmer of the problems involved in user interface adaption.

When using a VM capable of loading code dynamically, a third approach would be to have the system class libraries (representing the GUI components) implemented according to the platform layout. Due to dynamic loading, different representations of buttons, windows etc. would then be loaded into the application at run-time, thereby adapting the user interface to the actual device.

The use of a virtual machine and byte code also gives the possibility of interpreting byte code differently on each platform and in this way provide adaption. This gives the option of a very low-level adaption, if the byte code known from, e.g., Java is used. If instead the code interpreted was at a higher level, e.g., source level as in TCL, this adaption may be more useful.

We suggest using either the XML or the System Classes approach.

6.2.3 Resources

As mentioned above, the platform should provide uniform access to local resources. With mobile devices some issues not present with stationary devices arise.

When a mobile device is physically moved to a new domain it should adapt to the new environment. This involves hooking up the device and the applications to local printers and other local domain resources to enable the user to perform his task using local resources instead of the ones of his previous domain. Again this requires some identification of the user and device in order for the platform to gain access. The passport identification idea presented in section 6.1.2 could be used here. In some areas the user may have to pay in order to access the local resources. This requires some kind of billing system.

Not only can devices be moved into a new domain with other resources where they have other access rights to the resources. The resources in a domain may also change dynamically. One way of addressing this is to provide a lookup service with which devices and services register. When a device enters a new domain it would either be given the address of the lookup service or it would broadcast a request asking for this address. The device will then be able to use the lookup service to locate the services in the new domain, as in [JINI, 00].

Not all necessary resources are available locally; files are one example of this. Some sort of distributed access should therefore be provided in a PCE, however using a distributed file system will require a constant connection while accessing a file. A better approach would be to either use a mobile file system, such as Coda, that migrates whole files to the client device, thereby making it possible for the client to disconnect while still accessing the file [Jing, 99].

When accessing files in a domain protected by a firewall the user should be authorized. For validating users, Coda currently uses a token with the user UID. However with small modifications, we believe it could be extended to present the user over a wider area, ie., by using some globally unique user id. Another solution

would be to use the WebDAV¹ protocol [WebDAV, 00], which is an extension to the HTTP protocol by which users can be authorized before accessing a resource. WebDAV also provides versioning and sharing of the resources. Finally, an agent could be used to migrate a file from one domain to another. The agent would be able to identify itself using the user id, as presented in chapter 2.

One may question whether files are the right way to store and access information in a PCE since they are only a way of storing information externally to an application. As mentioned earlier it may be more useful to use databases, tuple spaces, [Cabri et. al, 99], or some other technique to store information. However it should be possible to access and share information from all devices including mobile devices using wireless network technologies. This requires some way to ensure that information is accessible even when devices are disconnected. A solution to this problem may be to migrate the information.

6.3 Migration Mechanism

A central part of the platform is the migration mechanism. A major part of migrating an application is extracting the application specific information from the platform on which it executes. This can be done in two different ways. Either the information is extracted by the application programmer using reflection, [Maes, 87], which gives him access to the representation of the application or it is done as part of a `migrate` system call that have direct access to the internal representation of the application.

The benefit of the reflection approach is that it is more flexible and that the programmer decides what needs to be extracted. However it also requires more of the programmer, ie., he must understand what is needed and what is not. Another issue is that very few language provide the reflection capabilities needed to get hold of the execution state of an application, limiting the migration mechanism to weak migration. Reflection also has the side-effect of introducing a performance overhead.

The system call approach relieves the application programmer of any worries he may have concerning migration. However with this approach, the application programmer have no influence on the migration process and cannot alter it in any way. Future modifications of the migration process are also difficult to implement, since it requires the system to be altered.

In this section, we will not distinguish between the two approaches, but look at

¹Web Distributed Authoring and Versioning

some of the aspects that should be considered when implementing a migration mechanism.

The migration mechanism should be able to unhook any application, collect all of its state, transfer it to a new platform on which it establishes the application again, including hooking it up to the local resources.

This gives four different aspects involved with migration of an application which we will discuss below. These four aspects are Negotiation, Retrieval, Transfer, and Establishment.

Negotiation. Negotiation is necessary for two reasons: to ensure the application access to necessary resources and to ensure the security of the destination platform.

An application have some demands that must be fulfilled in order for it to run satisfactorily, e.g., the application may demand a certain amount of memory for its class/object heap space and native method/virtual method stack space. It may also demand a certain amount of processing power or network bandwidth, or maybe it has demands concerning the size of the display or other I/O devices.

If one or more of these demands are not met by the destination platform, the application may not be able to run there. In order to assure that this does not happen, the migration mechanism on the source platform initiates a negotiation. If the negotiation succeeds, the source can start the actual migration of the application. The destination can use the information received during the negotiation to reserve resources for the application.

The negotiation may also resolve security issues, e.g., what privileges the application may get if it is migrated to another domain. It can also be used to verify the identity of the application and the platform.

Finally, the negotiation may resolve other security issues, such as checking whether the code is coming from a reliable source and if not decide whether the code should be allowed inside the domain.

Retrieval of Application Code and State. If the negotiation succeeds, the migration layer can start to extract the data and code that make up the application. Under the assumption that the application is coded in an object-oriented language, the class and object heap must be traversed to extract the non-platform specific information. For the class heap this means finding all the classes that are not system classes, including contents of the static variables. For the object heap this includes all live objects except the system hooks. All pointers between

the heaps and between object must be mapped to be able to reconstruct them at the destination.

The native and the virtual method stack must also be traversed, and all information from these stored. This also include mapping pointers between the stacks and the heaps. One way of mapping the pointers could be to use relative addressing with the start of the heap or stack as the base. The relative address should take care of the fact that the size of the objects, stack frames or classes may not be the same on all architectures.

Migration Protocol. In order to exchange the application code and state, a migration protocol is needed. This protocol represents the application at the lower levels of the platform. It can be seen as an abstract representation of the application, that enables different communication layers to exchange application information.

Establishment of Application Code and Data. Once the information has been extracted and sent from the source to the destination, it must be re-established in the virtual environment. This implies setting up the class heap, object heap, native method stack, virtual method stack, system hooks and pointers.

To set up the class heap, all the system code present in the heap at the source platform should be loaded as well as the code migrated. Once this is done, the object heap can be re-established, including system hooks and remapping of pointers to code and other objects. If object handles are used these may be re-established now. Finally the stacks can be set up, mapping pointers to their previous state.

It is important to note that the number of system hooks may change for a given object from platform to platform. This must be accounted for. Byte ordering and the size of primitive types may change, so this should also be taken into account.

6.4 Pervasive Applications

In the previous sections we have looked at the communication, the platform and the migration mechanism used in a PCE.

We will now take a closer look at the concept *pervasive applications*, both their properties and how they relate to *mobile agents*. As mentioned in chapter 3 we have no clear way of distinguishing between a pervasive application and a mobile agent. In this section we will make a preliminary attempt to define and relate

the concepts of pervasive applications and mobile agents.

The pervasive application is an entity with which the user interacts. It runs in a virtual environment provided by the user's platform. We think of a pervasive application as being a resource, that the user may access via the net. This access is achieved by migrating the pervasive application along with the user from platform to platform, ie., pervasive applications are applications that are capable of following the user around in a PCE.

By being pervasive, applications gain some of the properties normally held by mobile agents. To recapitulate, a mobile agent may have the following properties: mobility, autonomy, communicativity, reactivity, goal-orientedness, learning, character, temporally continuity and flexibility.

The properties shared with pervasive applications are mobility, autonomy, temporally continuity and communicativity. In section 3.2 we said that to be an agent, the entity should be autonomous as well as some of the other eight properties mentioned, depending on the context in which the entities are used. We could therefore perceive pervasive applications as being mobile, communicative agents. The question is now whether one should distinguish between mobile agents and pervasive applications and if so how this distinction should be made.

Let us start by looking at how Merriam-Webster, [Merriam-Webster, 00], defines the terms agent and application.

An Agent is defined as:

- “one that acts or exerts power”
- “something that produces or is capable of producing an effect : an active or efficient cause”
- “a means or instrument by which a guiding intelligence achieves a result”
- “one who is authorized to act for or in the place of another : as a representative, emissary, or official of a government <crow agent> <federal agent>”
- “one engaged in undercover activities (as espionage) : SPY <secret agent>”
- “a business representative (as of an athlete or entertainer) <a theatrical agent>”

From the above definition we perceive an agent as being active and specialised. *Active* because it acts and is capable of producing an active effect or efficient cause. *Specialised* because it is designated for a certain task, ie., it is a spy, secret agent, or business representative.

An application is defined as:

- “an act of applying”
- “an act of putting to use <application of new techniques>”
- “a use to which something is put <new applications for old remedies>”
- “a program (as a word processor or a spreadsheet) that performs one of the important tasks for which a computer is used”

Contrary to an agent, the application is passive and less specialised. *Passive* since it must be applied in order to produce a result, ie., it is not itself capable of acting. We also perceive applications as being *less specialised* than agents because the tasks they are designed to solve are more general, requiring user interaction.

We perceive the difference between an application and an agent as the difference between a spreadsheet and an accountant. The spreadsheet must be applied directly by the user for the user to balance his accounts. The accountant is capable of balancing the user’s accounts by himself without the user interacting.

To gain a better understanding of the concepts: pervasive application and mobile agent, we will define them according to the definition of Conceptual Clustering as defined in [Madsen et. al., 93], chapter 18. We would like to stress that this is a preliminary attempt to do so and by no means a final answer.

In Conceptual Clustering concepts are organized according their defining and characteristic properties. The defining properties of a concept determine a sharp border member and non-members. The characteristic properties are properties that a member may or may not possess. The Conceptual Clustering view is closely related to the better known Aristotelian view, [Madsen et. al., 93], the difference being that in the Conceptual Clustering view the properties in the intension need not all be objectively decidable.

As can be seen in table 6.1 we propose the defining properties of a mobile agent to be: mobility, autonomy, goal-orientedness and temporally continuity. Since we are looking at a mobile agent it must implicitly be mobile and autonomous, according to section 3.2. To be an agent an entity must be goal-oriented since it assigned an overall task which it should solve on its own, e.g., the accountant is assigned the task of balancing the user’s accounts. To be able to solve a task it is also necessary for the mobile agent to retain its state across migrations, e.g., the accountant must remember where he left (the day before) in order not to perform the same task over and over again.

As mentioned in chapter 3 an agent may also be reactive, learning, flexible, character and communicative. However, we do not perceive these properties as

Properties	Pervasive Application	Mobile Agent
Defining	Mobile Temporally continuous Autonomous Communicative	Mobile Temporally continuous Autonomous Goal-Oriented
Characteristic	Reactive Learning Flexible	Reactive Learning Flexible Character Communicative

Table 6.1: Defining and characteristic properties of pervasive applications and mobile agents

defining.

We find the defining properties of a pervasive application to be: mobility, autonomy, temporally continuity, and communicativity. The first three properties are needed to make the pervasive application able to follow the user in a PCE, as argued in section 3.3. On top of this all pervasive applications need to be communicative in order to interact with the user, e.g., the spreadsheet needs a user interface for the user to enter information.

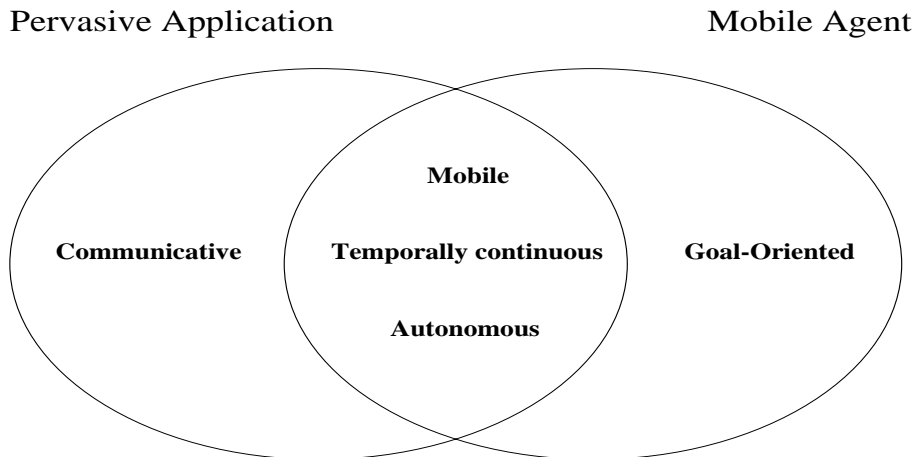


Figure 6.4: Relation between a pervasive application and a mobile agent

The characteristic properties of the pervasive application are reactive, learning, and flexible. We do not perceive a pervasive application as being goal-oriented in the same way as the spreadsheet is not able to balance the user's accounts

without the user interacting.

In figure 6.4 the pervasive application and the mobile agent is related by their defining properties. As mentioned above the mobile agent may be communicative and it can therefore perform the same role as a pervasive application. However, a pervasive application cannot be goal-oriented since it is neither a defining nor a characteristic property. This is because we perceive a pervasive application as a tool which is not capable of acting on its own.

As said above this discussion should be seen as a first attempt to define and relate the concepts of pervasive application and agent. Another concept that could be included in this discussion is mobile objects since they share the autonomy and the mobility properties of pervasive applications. More work should definitely be put into relating these concepts, maybe by classifying the concepts in terms of a classification hierarchy.

Chapter 7

The Experiment

Based on the experience gained from the previous chapters we made an experiment, in which we migrated applications from a Palm to an Intel PC.

As described in the last chapter, this involves Negotiation, Extraction of the application on the source, Transferring, and Establishment of the application on the destination. We decided to concentrate on how to actually migrate an application, and not on the negotiation topic.

We have not focused on the underlying network but decided to use a serial link for communication, even though the communication between two such devices should preferably be wireless.

Other issues that we have not addressed are security, ownership (in case the application is moved to another user's device), and migration of files used by the application.

In the following we will start by describing the migration platform including the hardware, operating systems and virtual machine used. Then we will describe the structure of the virtual machine from which the internal application data is extracted, and finally we describe our implementation of the migration mechanism.

7.1 The Migration Platform

7.1.1 Hardware

In order to best explore what problems exist when migrating applications in a PCE, we chose to migrate between two very different devices. In this way

we hoped to discover as many problems concerning migration as possible. To describe the devices we will briefly list their features.

Palm Pilot. The Palm is a small handheld device using a 20MHz Motorola Processor and 8Mb flash RAM for storage produced by Palm Inc., [Palm, 00]. Information is stored in databases instead of ordinary files and the Palm features no hierarchical name space.



Figure 7.1: The Palm Pilot handheld

The Palm has a 160x160 pixels display which can be manipulated by a pen, called a Stylus, see figure 7.1. Choosing an application is done by pointing at the application's icon on the screen. To write information the Palm has a writing area in which the user can type *Graffiti* letters and numbers. Graffiti is a text recognition input system specifically designed for handheld devices.

To communicate, the Palm has a serial port for linked communication and an infrared port for wireless communication.

Intel PC. The Intel PC used in our experiment features a 233MHz Pentium Pro processor, 128 Mb of RAM and 6Gb of harddisk storage. Unlike the Palm the PC has a keyboard and a mouse for inputting information. The screen is 17 inches and the solution used is 1024x768 pixels.

The PC features a LAN network card, two serial ports and a parallel port for communication.

7.1.2 Software

The Palm runs the PalmOS 3.5 operating system and the Intel PC runs MS Windows. In order to remove as many of the differences between the two hardware and software platforms as possible we needed a virtual machine to run our pervasive applications. For this we have used the Waba VM [WabaSoft Inc., 00], which was designed specifically for small handheld devices, but also ported to Windows.

To communicate over the serial link, we used the standard Serial Communication libraries on the Palm. On the PC we have written a server in Java, using the Java Communication API.

7.1.3 Virtual Machine

We found Waba the ideal virtual machine for our experiment. The source code is freely available which made it possible for us to alter it as needed to support migration. Waba is developed by WabaSoft, [WabaSoft Inc., 00]. It features a language, a virtual machine, a class file format, and a set of foundation classes.

Waba is specifically designed for small devices, such as PDAs and other handhelds. The language, the class file format, and the bytecode is a strict subset of Java and application programmers can use Java development tools for creating applications for Waba VM as long as they only use the supported subset.

Waba is optimized specifically with small devices and their characteristics in mind. This means that features which demand substantial amounts of memory and features that are deemed unnecessary are omitted from the design. Things omitted are for instance exceptions, the primitive type `doubles` and threads. Waba VM is only capable of running one thread due to PalmOS being a single-threaded operating system. The foundation classes provided for Waba were designed to be as small as possible but still provide the functionality needed to write proper programs.

One of the characteristics of Waba VM is that, including foundation classes, it uses less than 64 kb of memory when running and is capable of running programs using less than 10 kb of memory.

7.2 The Inside of the Waba VM

When an application is running in a virtual machine, all of its run-time state is represented in the VM's internal structures. To be able to extract an application

from the VM, it is necessary to know the VM structure. We will briefly describe what parts constitute the Waba VM and how these parts are related.

The Waba VM consists of two heaps, a class heap and an object heap, and two stacks, the native method stack and the virtual method stack. Objects in Waba are always referenced through object handles. Let us start by looking at the class heap.

Class Heap. The class heap contains information about all the classes currently loaded by the virtual machine. This information is contained in the following structure.

```
typedef struct WClassStruct
{
    struct WClassStruct **superClasses; // array of superclasses
    uint16 numSuperClasses;
    uint16 classNameIndex;
    uchar *byteRep; // pointer to class representation in memory
    uchar *attrib2; // pointer to accessFlags
    uint16 numConstants;
    ConsOffset *constantOffsets;
    uint16 numFields;
    WClassField *fields;
    uint16 numMethods;
    WClassMethod *methods;
    uint16 numStatics; // used by sendObjectHeapItem when serializing
    uint16 numVars; // computed number of object variables
    ObjDestroyFunc objDestroyFunc;
    struct WClassStruct *nextClass; // next class in hash table
} WClass;
```

Here the `superClasses` pointer is a pointer to other class structs on the class heap, that are loaded recursively when the class is loaded. The pointer `constantOffsets` points to an array of offsets that can be used to find any constant in the class's constant pool. This array is also allocated on the class heap.

The `fields` and `methods` pointers are also allocated on the class heap. The fields may be either a static field in which case it has a value or an ordinary field in which case the value is the offset of the field in the object's representation. The method structs are used to describe whether a method returns a value and whether it is a constructor. It also contains a pointer to the method code in the class file, ie., the byte-code.

To summarize the following is allocated on the class heap: the `WClass` struct, the `WClassMethod` structs, the `WClassField` struct, and the `ConsOffsets`. Each contain pointers into the class file.

Object Heap. Objects and object handles are allocated on the object heap. The objects are allocated from the bottom and the handles from the top. Objects are only referenced through their handles. Object references are represented as integers which are then mapped into an object handle index used to access the actual object on the heap through a pointer into the object heap. In this way the garbage collector only has to update the pointer in the object handle and not traverse the stack's other objects and static class fields to update the object reference each time the object is moved. Using object handles also simplifies migration where we also only have to update the pointers in the handles on the destination.

Two types of entities are allocated on the object heap apart from object handles. Real objects and arrays. For a real object the format is shown in figure 7.2. For an array it is showed in figure 7.3.

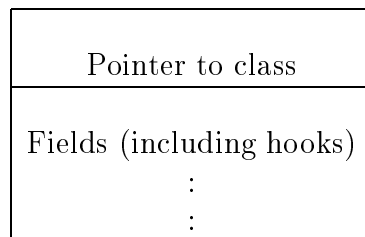


Figure 7.2: Format of an object on the object heap

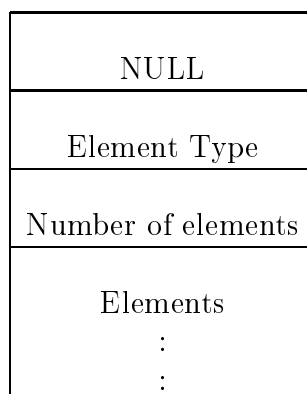


Figure 7.3: Format of an array on the object heap

All arrays start with a NULL pointer and all objects have a pointer to their class on the class heap. This can be used to distinguish between the two.

Object Handles. The object handles are represented by the `HOS` struct. It has three records, the pointer into the object heap, an `order` and a `temp` integer. The `order` integer holds the index of the last known free handle. If no free handles exist up till this handle the value is the index of the handle. The `temp` integer is used when garbage collecting.

```
typedef struct
{
    Var *ptr;
    uint32 order;
    uint32 temp;
} Hos;
```

The handles can be accessed by their index through the `heap.hos` array.

Native Method Stack. The native method stack is used internally in the VM to ensure that objects are not garbage collected during calls to native methods or while another object is allocated. It is also specifically used to place a reference to the application main object, ie., the object created from the class given as argument to the VM upon startup.

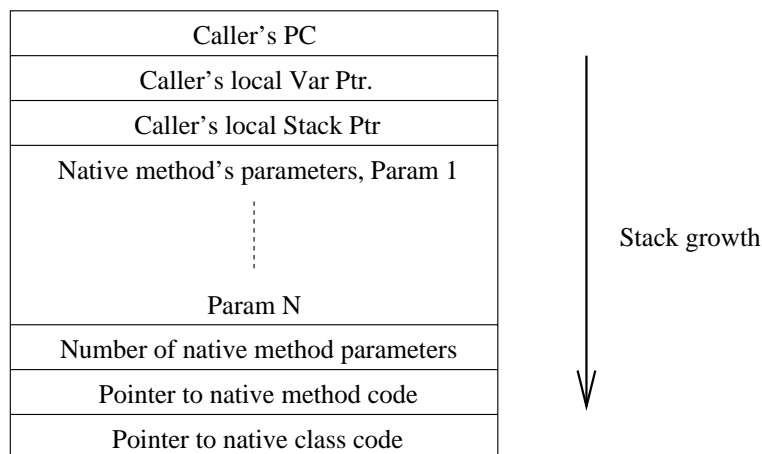


Figure 7.4: The native method stack frame

Virtual Method Stack. The virtual method stack is the “normal” stack, ie., the one used to push all method invocation frames and return frames. Two types of invocation frames exist. The *virtual method frame* and the *native method frame*. The native method frame will always be the last on the stack since any methods called from the native method are placed on the C native stack. The stack for native methods are shown in figure 7.4 and the virtual method stack

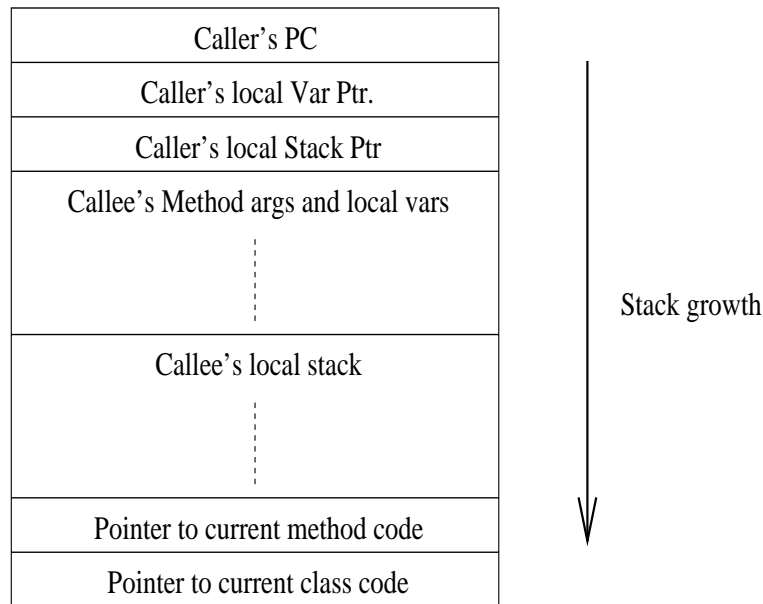


Figure 7.5: The virtual method stack frame

frame can be seen in figure 7.5. The three first entries in both figures are the *return frame* which contains the caller's PC, local variable pointer and local stack pointer.

7.3 Design of the Migration Mechanism

When designing the migration mechanism, we came up with the following steps for migrating an application from the Palm to the PC, illustrated in figure 7.6.

1. **Extract application** - The application's code and data is extracted from the VM. The data consists of the size of the heaps, the virtual method stack, the native method stack and the number of objects. The data is also the object heap, the object handles, the class heap, the native method stack and the virtual method stack. The code is all the non-system class files found in the application classpath.
2. **Transfer application** - The code and data is serialized and sent, via the serial link, to the *Migration Server*. This is initiated in the Migration Extension on the Palm which was the simplest approach since we cannot have a server process running along side the Waba VM on the Palm (it is single-threaded). The Migration Server receives the code and data according to the Migration protocol, described in 7.3.2.
3. **Write application** - The Migration Server writes the data into a file and

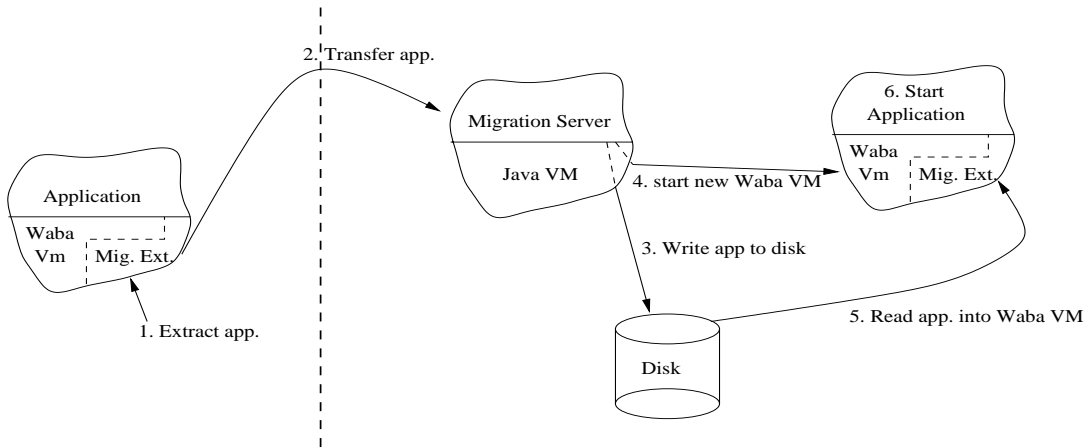


Figure 7.6: Migration of application from Palm to PC

the code into their respective class files. The data is written according to the migration protocol.

4. **Start Waba VM** - The Migration Server starts a new instance of the Waba VM with arguments describing the location of the application.
5. **Setup code and data** - The Migration Extension reads the data file, loads the code and sets up the heaps and stacks accordingly.
6. Control is passed to the application

From the above steps, we created the following system architecture for the Palm and the PC, figure 7.7.

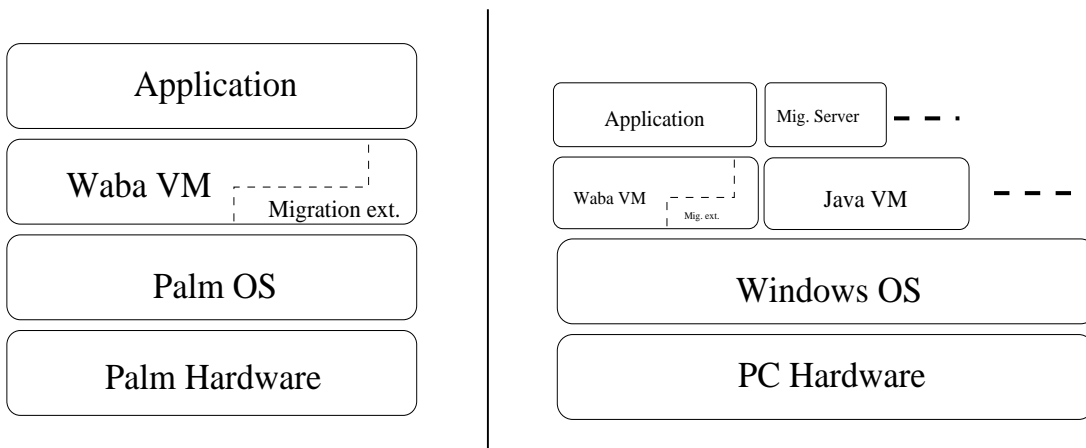


Figure 7.7: left: Palm architecture model, right: PC architecture model

On the Palm, we have the Waba VM running, interpreting the application. Since PalmOS is a single-threaded environment, only one application can be executing at a time. This means that we cannot have a server running in the background receiving applications, while the user of the Palm is using another application. Instead the Palm user could request applications from an application server.

On the PC, we have Windows, a multi-threaded OS. Here we have a Migration Server listening on the serial port for incoming applications. The Migration Server is written in Java and runs completely independent of the Waba VM. Alongside the Migration Server, we have the Waba VM executing the actual application.

Next we will look at the Migration Server.

7.3.1 The Migration Server

The Migration Server has three tasks. To receive code and deserialize it, to receive data and write it untouched to a file, and to start the Waba VM when all code and data have been received. Deserializing the code includes creating a file of the right class name and in the right class path. All the initialization data and the application data should be written, as is, to a file.

When the Migration Server starts the Waba VM it gives the path to the application's directory as argument. In this directory and possible subdirectories the server has stored all the code and the data file. The data file will then be read by the Waba VM. The first data read by the VM is initialization data with the size of the heaps and stacks. Once the initialization is done it will start loading the code that was loaded before migration, and read in the data that makes up the heaps and stack. The data is read according to the Migration Protocol briefly described in the next section.

7.3.2 The Migration Protocol

This protocol is necessary for communicating between the two platforms. The messages represent the application on the lower levels of the pervasive architecture where a homogeneous environment is not present. The protocol can be thought of as the way to represent the application on the net. It is used both when sending data and code over the serial link and when writing and reading data to the disk on the PC. We have defined the following types of messages.

```
H_VmStackSize      // the size of the Virtual Method stack
H_NmStackSize      // the size of the Native Method stack
H_ClassHeapSize    // the size of the class heap
H_ObjectHeapSize   // the size of the object heap
```

```

H_VmStackPtr    // the actual value of the virtual method stack ptr
H_NmStackPtr    // the actual value of the native method stack ptr
H_NumHandles    // the number of object references allocated
H_NumFreeHandles // the number of object references used
H_Class         // the name and code of a class
H_ClassName     // the name of a loaded class
H_Hos           // an object reference incl. gc info and used info
H_ObjectClass   // a heap class object
H_ObjectArray   // a heap array object
H_ObjectRef     // a object reference from the native method stack
H_VmFrame       // the Virtual Method stack frame

```

An example of a message is displayed in figure 7.8.

Type
Object handle index
Name length
Name
No. of superclasses
No. of vars in topmost superclass
Vars of topmost superclass
No. of vars in next superclass
Vars of next superclass
:
:
No. of vars in class
Vars of class

Figure 7.8: Format of a heap object message

7.4 The Migration Mechanism

Until now we have described our platform and the internal structure of the virtual machine. We have also presented the overall steps involved in the migration. We will now look at how the actual migration mechanism is implemented.

When migrating an application, we have to extract all the relevant data, serialize it, send it, and reestablish it on another virtual machine on the destination host. The relevant data is class code, object heap, class heap, native method stack, virtual method stack, and initialization data (ie., size of heaps, stack etc.). In the following we will describe how each part of the required data is extracted. We will start by describing how to extract the initialization information used to set up the heaps, stacks etc.

7.4.1 VM Init Data

The first data we migrate is the data used to set up the virtual machine on the destination. We send this first to be able to set up the virtual machine on the destination before migrating the actual application. The data needed is:

```
Application name
Virtual Method Stack size
Native Method Stack size
Class Heap size
Object Heap size
Current Native Method Stack pointer
Current Virtual Method Stack pointer
Number of Object Handles
Number of free Object Handles
```

Apart from the application name all of the above data is either integers or doubles and is represented in global structures. This makes it easy to extract, serialize, send, and reestablish. The application name is the class name given as parameter to the virtual machine when started. An object reference to an instance of this class is put onto the native method stack to make sure the object is not garbage collected. In order to get the name we just have to look at the name of the object's class. On the destination we create a directory of the same name as the application. This directory is used to store the application code and the state of the application.

7.4.2 Class Code

The next part we migrate is the application code. The code of an application is packaged into one or more databases on the Palm. To launch an application on the Palm, a launcher is generated. This launcher gives as parameters the names of the needed databases to the Waba VM when the application is started. On startup each database is stored in an array for fast access. The first database in this array is the Waba System Classes, the rest are all the application-specific code. This simplifies finding the code since all we have to do is to run through all the application-specific databases and for each record, ie., class, serialize it and send it to the destination. On the destination, a file is created in which the code is put. If the name of the class contains directories, these are created first.

7.4.3 Migration of the Application State

When the code has been migrated it is time for sending the run-time state of the application. The run-time state consists of the two heaps, the two stacks, and the object handles.

Class Heap. We start by traversing the class heap. For each non-system class, we migrate the class name and the value of the class's static fields (stored along with the class). In this way, we know which classes were loaded when the application was migrated and we can load the same classes on the destination before setting up the objects on the object heap. By loading the classes before we set up the object heap, we can map an object's class pointer when putting the object on the heap.

Object Handles. Next we migrate all the object handles. The object handles are allocated from the top and down of the object heap whereas the objects themselves are allocated from the bottom and up. When we migrate a handle, we do not migrate the object pointer since the object might be placed differently on the destination object heap than it was on the source host's heap. Instead we migrate the index of the handle. Since the handles are of fixed size they are easily accessible through the `heap.hos` array. Their index in this array is migrated along with the `order` and `temp` integers. The handles are then placed in the same index on the destination. When we migrate the object, we migrate the index of the object's handle with it. Using the index, we can get hold of the object handle on the destination and set its object pointer when we put the object onto the object heap. Since the objects are only referenced through their object

NULL	
Unsigned Int16	
5	
Element1	Element2
Element3	Element4
Element5	Empty

Figure 7.9: The memory layout of an array containing 5 unsigned int16

handles and not directly, we do not have to worry about other object references in the code.

Object Heap. The object heap contains two different kinds of entities, arrays and objects. Their structures were described in figure 7.2 and figure 7.3. For both entities, we find the index of the object handle denoting them and migrate this index along with the entity. In this way we know what handle was used to reference this entity. On the destination the handle's pointer can now be set when the entity is put on the object heap.

When migrating an array, we also migrate the array's type, size of the type (ie., 2 bytes for a uint16), length, and its elements. The object heap is 4 byte aligned, so each entity must be aligned to this 4 byte boundary. Array elements are stored using only the necessary amount of bytes, so for instance an array with 5 unsigned int16, will have the format depicted in figure 7.9. Here each row represent 4 bytes, but since the elements only need 2 bytes each, they can be stored side by side to minimize storage needs. However it is important to notice that the next entity begins at the row after the last element and not immediately after.

When migrating objects we have to map the class pointer since the class will be placed at a different address on the destination. We map this pointer to the name of the class which is then serialized and sent to the destination. When migrating the fields of the object, we look at the object's class to find out how many non-static fields it has. The number of fields following the class pointer, see figure 7.2, is a sum of this particular class's fields and all of its superclasses. For each class we check to see how many hooks the class has. The hooks are not migrated along since they are system-specific and must be reestablished by system calls on the destination. All of the other fields are migrated. When reestablishing the object on the destination, we have to be aware of the difference in byte order between the two architectures. We do not have to worry about the 4 byte boundry when migrating objects, since all fields are represented by 4 bytes on both the Palm and the PC.

Stacks. As mentioned before we have two stacks in the Waba VM. The native method stack and the virtual method stack. We have to migrate the native method stack to know what objects should not be garbage collected. The virtual method stack must be migrated in order to continue execution right after the `migrate` statement.

The native method stack contains only objects references, ie., integers and is very simple to migrate. The virtual method stack contains class references, method references and program counters. Each of these are specific to the machine on which they are allocated and must therefore be converted into relative pointers that can then be converted back to absolute pointers at the destination. Apart from this, we only have to make sure that the elements are sent in the correct order. Since a migration is always initiated by a native method call of `migrate` this will be the last frame on the virtual method stack. There is no need for sending this frame to the destination, since we know that the call is finished when we are deserialising the application.

7.4.4 Hooks

The last subject taken care of by the migration mechanism is to rebind the application to the new host environment which means a re-establishment of the *system hooks*. The system hooks are pointers to system resources, such as windows, fonts, and sockets. We have not looked at how to setup socket or serial port hooks, since the applications we are looking at do not communicate with others, see chapter 1.

When migrating an object, we do not migrate the hooks of the object. So when setting up an object on the object heap, we check for each class to see whether it is a system class and if it calls a function that creates the system resource and places the values, ie., system hooks, right after the ordinary fields of the current class. For instance, if we migrate a `mainWindow` object it has four superclasses, `waba.ui.window`, `waba.ui.container`, `waba.ui.control` and `waba.lang.object`. Both the `mainWindow` and the `window` class has a system hook on the Windows platform. So the memory layout would be as in figure 7.10.

On the Palm no hooks are necessary for either `window` or `mainWindow` since only one window is displayed at a time. Thus the memory layout on the Palm would be as in figure 7.11.

So when migrating an object from PalmOS to Windows, we have to create the system hooks necessary on the Windows platform. This is done by checking each superclass and the class itself, when putting the fields back on the object heap. When a superclass has some hooks we put the fields used by this superclass on the

Class pointer
fields of object class : :
fields of control class : :
fields of container class : :
fields of window class : : system hook
fields of mainWindow class : : system hook

Figure 7.10: The memory layout of an mainWindow object on the window platform

Class pointer
fields of object class : :
fields of control class : :
fields of container class : :
fields of window class : : :
fields of mainWindow class : : :

Figure 7.11: The memory layout of an mainWindow object on the Palm platform

heap and afterwards make the system calls, that will create the needed system resource. The hooks are then placed after the fields of the superclass. In this way we dynamically create the system hooks needed on this particular platform. The fields of the next class are then placed after the hooks.

7.5 Evaluation

We have extended the Waba VM with functionality that makes it possible to migrate an application from a Palm to an Intel PC. This has been tested by migrating a simple “hello world!” program as well as a more complex modelling application.

We have implemented strong migration because we found it important to relieve the application programmer of how to migrate the application and because we in this way would gain the best knowledge of the problems involved in migration. To provide strong migration we have migrated the class heap, object heap, including object handles, class code, virtual method stack, native method stack as well as some initialization data.

The most difficult part to migrate was the object heap since we had to map class pointers, object handle pointers and take care of system hooks to platform-specific resources. One other issue that caused some confusion was the different byte-ordering used on the Palm’s Motorola chip and the PC’s Intel chip. However it was simple to solve once found.

At first it appeared to us that there was no need for migrating the virtual method stack. We figured that this was due to the Palm OS being single-threaded and event-driven, meaning that only one event was handled on the stack at a time. We later found out that this was due to the way we had coded our test programs in which `migrate` was always called directly from a method, i.e., event handler that was on the C stack (the C stack meaning the stack which is used by the Waba VM process). If we instead had called another Java method than `migrate` and called `migrate` from that method, we would have a method frame on the stack, which should be migrated. However migrating the virtual method stack was no problem, since we already knew how to map pointers and we also knew the different formats of the stack frames.

Some of things we have not looked at are migrating threads, code versions and how to migrate from the Intel PC to the Palm. Depending on whether the virtual machine provides its own thread abstraction or whether it uses the OS thread abstraction, migrating threads will be more or less difficult. With respect to code versioning and availability, checking if the code already exists on the destination would be an obvious performance improvement. Code versioning could be used

to ensure that the correct version of the class code is used, or if applicable the newest.

Finally when migrating from the PC to the Palm some issues must be addressed. None of these issues are related to the actual extraction and establishment in the Waba VM. We will however briefly describe them.

Finding the Class Code. We have to handle files instead of databases. On the Palm we have one or more databases and all we have to do is to migrate them one by one. The system classes are also easily identified and restricted to a specific database. On the PC we have to look in the directories of the class path and their subdirectories. This involves some kind of assessment towards what needs to be moved and what can stay behind.

Creating a Waba Launcher. On the PC the Waba VM is started from the command prompt or by making a system call. On the Palm the Waba VM is started by a specific launcher program, that tells it what databases to use and the name of the first class to load. Some modification should be done to either create a Launcher program on the run or modify the Palm Waba VM to be able to start without.

Receiving the Application. Since the Palm is single threaded we cannot have a server running in the background listening for incoming applications. The user of the Palm would have to request the application in order to receive it. In relation to a PCE we do not find this a problem since the user would have to indicate what application he want to work with on the Palm anyway, due to the single-threaded nature of the Palm.

Chapter 8

Conclusion

8.1 General Lessons Learned

In the future more and more people will make use of mobile devices and each person will have access to several different devices. These users will want to seamlessly access the applications and data they need to perform their tasks. For this to be possible, the applications should be independent of the individual devices and instead be part of a large network. This requires all devices to be able to communicate either by wired or wireless networks. Due to the properties of wireless networks, mobile users cannot expect to have a continuous connection and the mobile devices and their applications should be as independent of the network as possible. This requires the applications to be placed on the device of the user. In this way the device and the application will continue working even during a disconnection.

For an application to always be present on a device, it should be able to migrate. In this way it is possible for the application to be available to the user and at the same time be independent of the net even though the user moves to a device where the application is not already installed.

The best way to make sure that applications can run on any device independent of the hardware is through virtual machines. The virtual machine provides a homogeneous environment to the application by abstracting away any hardware differences that may exist between the involved devices. For accessing local resources a uniform resource interface may be used that will abstract the actual hardware dependent interface away.

Due to the inherent overhead when interpreting code and due to the current processing power in mobile devices, one may question whether mobile devices should make use of virtual machines. We do believe that with the advances in

virtual machines in the areas of adaptive compiling and dynamic optimisation as well as the speed with which new hardware is developed, the problems concerning performance will tail. The main difference will then be the mobile devices using wireless networks and the stationary devices use wired networks. We therefore recommend the use of migration and virtual machines, since it makes the device more independent (autonomous) of the surroundings which is a requirement when being mobile using wireless networks, see section 2.3.

We are convinced that the gap in processing power between mobile and stationary devices will narrow significantly within a few years making mobile devices nearly as powerful as stationary devices [StrongArm, 00]. This will increase the number of applications suitable for mobile devices. However, there will still be applications with extraordinary demands for processing power and interaction methods that will not be suitable for mobile devices. Examples of such could be 3D environments and simulations.

An important issue that must be addressed when migrating applications is adaption of the applications user interface. This is required due to the different screen sizes and interaction mechanisms used on the different devices. On a Palm we have a very small screen capable of showing one window at a time and the user interaction is mainly performed using a pen. On the workstation we have a monitor available for presenting information and both keyboard and mouse for user interaction. These differences should be accounted for. One way of doing this is by interpreting the application's user interface according to the device on which it executes as done in our experiment using GUI system classes specifically designed for the architecture, see section 2.6.1 and section 6.2.2.

8.2 Future Work

Our thesis has only addressed some of the issues present in a PCE. An important issue that should also be addressed is security. This is both the security of the platform and the security of the pervasive application.

The security of the platform is partly solved by using a virtual execution environment. This makes it possible for the platform to decide which resources an incoming pervasive applications should have access to and what rights should be associated with each resource.

To ensure that a pervasive application does not block a resource on a host one could give the pervasive application a specific amount of credits that it could use on resources. Once the credits are used up the pervasive application is denied further access. The credit system could also be associated with the migration

task and thereby prevent a pervasive application from becoming a renegade on the net continuously migrating from host to host.

Concerning pervasive application security, further investigation into the area of how to secure applications from being altered by the host system should be made. One approach could be computing a checksum based on the application code which could then be checked on trusted hosts. This however does not prevent the system from spying on the application. How this is prevented we do not know.

One could imagine using pervasive applications for exchanging information. This could be useful in cases where one wants another to comment on ones work, e.g., showing a document to another for comments or approval without the receiver having to set up the application and finding the right spot in the document. In such a scenario ownership of the application must be addressed, ie., who owns the applications and the resources it accesses? The one who initially started the application or the one on which device it is currently running?

In relation to this, applications could also be rented to a user from an application service. This would be an easy way for a user to gain access to a tool without having to buy it for himself. This requires some way of paying and also some way of making sure the application can be withdrawn from the client's device once the lease expires.

Performance is another issue we have not been concerned with. However for a PCE to become usable it should perform well. Several optimisations could be made in the form of communication protocols, code compression, reuse of code already present on the host, code optimisation, file access etc.

With respect to file access, we are not sure whether distributed file access should be provided or whether information should be accessed through central databases. We do believe that some kind of storage on the client should be provided for code and temporary data. However we are not sure whether a distributed file system is the best approach to access shared data due to the amount of network traffic needed.

In our experiment we have used the Waba VM on both the PC and the Palm. Since the Waba VM is quite limited compared to the Java VM it may be better to use the Java VM (or some other VM) on the PC and the Waba VM on the Palm. In this way the application would be able to take better advantage of the platform on which it executed. We imagine the VMs involved to be able to interpret the same byte code. However, some code adaption may be needed.

Finally, further investigation of the concepts mobile agent and pervasive application should be done. Also investigation into whether reflection or system calls is the best way to extract application specific information should be performed, ie., whether the language should be extended with primitives for reifying the

application or whether it should be accessed purely through system calls.

8.3 Conclusion

In this thesis we have investigated migration of applications between heterogeneous platforms in a Pervasive Computing Environment. This is one step towards a Pervasive Computing Environment in which pervasive applications seamlessly follow the user.

Specifically we have looked at what the requirements are for migrating applications and how the migration mechanism should be implemented. This resulted in a discussion and a suggestion of how to address some of the issues present in a Pervasive Computing Environment. It also resulted in an experiment where we have implemented strong migration by modifying the Waba virtual machine. In this way making it possible to migrate applications from a Palm running PalmOS to an Intel-based PC running MS Windows.

Our research into Pervasive Computing Environments and our experiment showed that providing a uniform execution environment is necessary due to the heterogeneous devices present in a Pervasive Computing Environment. This unification can be done by using a virtual machine which makes applications capable of executing on devices independently of the hardware.

Our experiment also showed that it is possible, using strong migration and a virtual machine, to seamlessly migrate an application without the application programmer having to worry about the execution state.

8.4 Acknowledgements

We would like to thank our supervisor Professor Ole Lehrmann Madsen for his help, guidance and invaluable discussions throughout this work. For corrections and suggestions we would like to thank Ib Bentzen-Bilkvist, Mads Torgersen, Anders Morten Mikkelsen, and Klaus Marius Hansen.

Bibliography

- [Artsy & Finkel, 89] Y. Artsy and R. Finkel *Designing a process migration facility: The Charlotte experience*. IEEE Computer, September 1989, 47–56.
- [Banerji et. al, 93] A. Banerji, D. L. Cohn and D. C. Kulkarni *Mobile Computing Personae*, in Proceeding of the 4th IEEE Workshop on Workstation Operating Systems, 1993, p. 14-20
- [Baumann et. al., 98] J. Baumann, F. Hohl, K. Rothermel, M. Schwehm and M. StraSSer *Mole 3.0: A Middleware for Java-Based Mobile Software Agents* in proc. Middleware'98, Springer Verlag
- [Bharak et. al., 93] A. Bharak, S. Guday and R. G. Wheeler *The MOSIX Distributed Operating System*. Springer-Verlag Berlin Heidelberg 1993.
- [Bharat & Cardelli, 97] K. Bharat and L. Cardelli *Migratory Applications*. In *Mobile Object Systems: Towards the programmable Internet*, in Lecture notes in Computer Science No.1222, 1997, pages 131-149
- [Bluetooth, 00] *Bluetooth wireless network technology*,
<http://www.bluetooth.com>
- [Cabri et. al, 99] G. Cabri, L. Leonardi, G. Reggiani and F. Zambonelli *Design and Implementation of a Programmable Coordination Architecture for Mobile Agents*, in Proceedings of TOOLS Europe 99. p 10-19
- [Cardelli, 95] L. Cardelli *A language with Distributed Scope*, in Computing Systems, 8(1):27-59, January 1995.
- [Cardelli, 99] L. Cardelli *Abstractions for Mobile Computation*. in Lecture Notes in Computer Science, Vol. 1603, Springer, 1999. pp. 51-94
- [CSS, 00] *Cascading Style Sheets*,
<http://www.w3.org/style/CSS>
- [Casey, 95] M. Casey *Realizing Mobile Computing Personae*. Ph.D. dissertation, University of Notre Dame, April 1995.

- [Cheriton, 88] D. R. Cheriton *The V Distributed System*. In Communications of the ACM, 31(3):314–333, March 1988
- [CIT, 00] *The Danish National Centre for IT Research*, <http://www.cit.dk>
- [Coulouris et. al., 94] G. Coulouris, J. Dollimore and T. Kindberg *Distributed systems - Concepts and Design*, printed 1994 by Addison-Wesley.
- [Damm et. al., 2000a] C. H. Damm, K. M. Hansen, M. Thomsen, M. Tyrsted *Creative Object-Oriented Modelling: Support for Creativity, Flexibility, and Collaboration in CASE Tools*. In Bertino, E. (Ed.) Proceedings of ECOOP'2000. Sophia Antipolis and Cannes, France, June 12-16.
- [Damm et. al., 2000b] C. H. Damm, K. M. Hansen, M. Thomsen *Tool Support for Object-Oriented Cooperative Design: Gesture Based Modeling on an Electronic Whiteboard* In Turner, T., Szwillus, G., Czerwinski, M., Paterno, F. (Eds.) Proceedings of CHI'2000, The Hague, The Netherlands, April 1-6, ACM Press.
- [DECT, 00] *Digital Enhanced Cordless Telecommunications*, <http://www.dectweb.com>
- [Dictionary.com, 00] *Online english dictionary*, <http://www.dictionary.com>
- [Douglass & Ousterhout, 91] F. Douglass and J. Ousterhout. *Transparent process migration: Design alternatives and the Sprite implementation*. Software: Practice and Experience, 21(8):757–785, August 1991
- [DynamicTAO, 00] *Dynamic ORB using reflection*, <http://choices.cs.uiuc.edu/2k/dynamicTAO/>
- [Forman & Zahorjan, 94] G. H. Forman and J. Zahorjan *The Challenges of Mobile Computing* in IEEE Computer, 27(6), April 1994
- [Franklin & Graesser, 96] S. Franklin and A. Graesser *Is it an Agent, or just a Program ? - A Taxonomy for Autonomous Agents* in Proceedings of Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996.
- [Gray et. al., 96] R. Gray, D. Kotz, S. Nog, D. Rus and G. Cybenko *Mobile agents for mobile computing*. Technical report PCS-TR96-285, Department of computer Science, Dartmouth College, Hanover, 1996.
- [Hohl et. al., 97] F. Hohl, P. Klar and J. Baumann *Efficient Code Migration for Modular Mobile Agents*. Accepted Submission for the Third ECOOP Workshop on Mobile Object Systems: Operating System support for Mobile Object Systems.

- [HP CoolTown, 00] *HP pervasive computing research project*,
<http://www.cooltown.hp.com/>
- [HP News, Nov. 4, 99] *HP Advanced Research Efforts Help Make Pervasive Computing a Reality*, Palo Alto, Calif., Nov. 4, 1999,
<http://www.hp.com/pressrel/nov99/04nov99a.htm>
- [IBM PCE, 00] *IBM pervasive computing homepage*,
<http://www-3.ibm.com/pvc/>
- [IMAP, 00] *Internet Message Access Protocol*,
<http://www.imap.org>
- [IPCRES, 00] *Indiana Pervasive Computing Research (IPCRES) Initiative*,
<http://www.indiana.edu/ovpit/ipcres/>
- [Johansen et. al., 95] D. Johansen, R. van Renesse, and F. B. Schneider *An introduction to the TACOMA distributed system*. Technical report, University of Tromso, June 1995.
- [Jing, 99] J. Jing, A. Helal, A. Elmagarmid *Client-Server Computing in Mobile Environments* ACM Computing Surveys, Vol. 31, No. 2, June 1999.
- [Jul et. al., 88] E. Jul, H. Levy, N. Hutchinson and A. Black *Fine-Grained Mobility in the Emerald System*. In ACM Trans. on Computer Systems, Vol. 6, No. 1, February 1988, 109–133.
- [JP, 22/11/2000] *Jyllands posten, digital-sektionen*, 22nd of November 2000.
- [Java, 00] *The Java Programming Language and Virtual Machine*,
<http://www.javasoft.com>
- [JINI, 00] *The Jini Community*,
<http://www.jini.org/>
- [Kistler et. al, 92] J. J. Kistler and M. Satyanarayanan. *Disconnected Operation in the Coda File System*. In ACM Transactions on Computer Systems, 10(1), p. 3-25, Feb. 1992.
- [Li et. al. 93] K. Li, R. Kumpf, P. Horton and T. Anderson. *A Quantitative Analysis of Disk Drive Power Management in Portable Computers*. Technical Report, Computer Science Division, University of California at Berkeley, 1993.
- [Madsen et. al., 93] O. L. Madsen, B. Møller-Pedersen and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language* Addison-Wesley, 1993.
- [Maes, 87] P. Maes *Concepts and Experiments in Computational Reflection*, in OOPSLA 87, Conference Proceedings, pp. 147-155, ACM (1987).

- [Merriam-Webster, 00] *Online English Dictionary and Thesaurus*,
<http://www.m-w.com>
- [Neuman, 91] B. C. Neuman. *Protection and Security Issues for Future Systems*. In Workshop on Operating Systems of the 90s and beyond, Springer-Verlag Lecture Notes in Computer Science #563, p. 184-201, July 1991.
- [Nielsen & Søndergaard, 99] C. Nielsen and A. Søndergaard *Designing for mobility - an integration approach support multiple technologies*, in Proceedings of NordiCHI 2000 (CD-ROM) 23-25 Oct. 2000, Royal Institute of Technology, Stockholm, Sweden.
- [Palm, 00] *Palm Incorporated*,
<http://www.palm.com>
- [Peine, 97] H. Peine *An Introduction to Mobile Agent Programming and the Ara System*, ZRI Technical Report 1/97, Dept. of Computer Science, University of Kaiserslautern, January 1997. <http://www.uni-kl.de/AG-Nehmer/Ara/ara.html>.
- [Popek & Walker, 85] G. Popek and R. P. Gerald *The LOCUS Distributed System Architecture*. The MIT Press Cambridge, Massachusetts, London, England 1985.
- [Puliafito et. al., 98] A. Puliafito, O. Tomarchio, and L. Vita. *MAP: Design and Implementation of a Mobile Agents Platform*. Tech. Rep., University of Catania, 1998.
- [Rashid, 88] R. F. Rashid *From RIG to Accent to Mach: The Evolution of a Network Operating System*. The Ecology of Computation (B.A. Huberman, ed.), North Holland, 1988, pp. 207-230.
- [Román et. al., 99] M. Roman, A. Singhai, D. Carvalho, C. Hess and R. H. Campbell *Integrating PDAs into Distributed Systems: 2K and PalmORB* in International Symposium on Handheld and Ubiquitous Computing (HUC'99), Karlsruhe, GE, 1999. http://choices.cs.uiuc.edu/2k/papers/huc_99_ps.gz
- [Rothermel et. al., 97] K. Rothermel, F. Hohl and N. Radouniklis *Mobile Agent Systems: What is Missing?* in proc. of IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems; Cottbus, Germany; 1997.
- [Saldanha & Cohn, 94] J. Saldanha and D. Cohn. *A hybrid model for mobile file systems* in Proc. Workshop on Mobile Computing Systems and Applications, Santa Cruz, CA, Dec. 1994.

-
- [Shub, 90] C. M. Shub *Native Code Process-Originated Migration in a Heterogeneous Environment*, in Proceedings of the 18th Annual Computer Science Conference, 1990, p. 266-270
- [StrongArm, 00] *The Intel StrongArm processor for portable devices*, <http://developer.intel.com/design/strong/>
- [Sunray, 00] *The Sunray System produced by Sun Microsystems*, <http://www.sun.com/sunray>
- [Tanenbaum, 92] A. S. Tanenbaum *Modern Operating Systems*, Prentice Hall, Upper Saddle River, N.J., 07458, 1992.
- [TCL, 94] J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, Copyright 1994, 480 pp.
- [WabaSoft Inc., 00] *WabaSoft Inc.*, <http://www.wabasoft.com>
- [Waldo et. al., 94] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. *A note on distributed computing*. Technical Report TR-94-29, Sun Microsystems Laboratories, 1994.
- [WebDAV, 00] *Web Distributed Authoring and Versioning*, <http://www.webdav.org>
- [Weiser, M. 91] M. Weiser *The Computer for the 21st Century* in Scientific American, September 1991.
- [XML, 00] *Extensible Markup Language*, <http://www.w3.org/XML/>
- [Zayas, 87] E. R. Zayas *Attacking the process migration bottleneck*, in Proceedings 11th ACM Symposium on Operating System Principles, 1987, p. 13-24