On the Structuring of Distributed Systems:

The Argument for Mobility

By

Todd Papaioannou

A Doctoral Thesis

Submitted in partial fulfilment of the requirements
for the award of

Doctor of Philosophy

of Loughborough University
February 2000

For Jo

Acknowledgements

Undertaking a course of study that leads to the award of PhD is much like a journey of exploration and discovery. Although you may have some idea of where it is you wish to end up, the many rich experiences and pitfalls along the way are largely unforeseen. It is certainly an experience that I would recommend to anyone who believes they are capable. That is not to say, however, that it is a course suitable for everyone. The road to travel is long and tough, and many fall by the wayside.

My own journey has been one of academic learning and self-discovery. During my course of study, I have enjoyed incredibly the process of scaling new heights of knowledge, of cutting a trail where others may have never been, of using and pushing my mind to attack and answer the big questions. During this journey my mind has been refined to a sharpness and focus hitherto unforeseen to me, and I feel I am now able to wield my mind as a tool, in all situations. This has allowed me to look within, and understand exactly whom I am. In addition, my character has grown and expanded with a wealth of new experiences that have served to polish it.

I feel lucky to have undertaken my research in a relatively new field, where the boundaries and rules have not been defined yet. This has afforded me an academic freedom that many students do not enjoy, and allowed me to follow an academic path out of the reach of many. This type of work cannot be done alone in isolation though, and I would like to take this opportunity to thank those who have made it possible for me to get this far.

Firstly I would like to thank all my family, especially Jill, Les and Yannis, for their continued support throughout my many years of study. Without their help, I would have been unable to complete my work. I hope my completion goes some way to repaying their trust and support.

To study for a PhD requires a suitable environment and support in which to do so. Most important in providing this has been my supervisor Dr. John Edwards. John deserves special credit for having the patience to guide a determined and unconventional student, even when many of the proposed ideas were contrary to his own philosophies. I am sure the experience must have been challenging, but I believe

that we have both learnt greatly from it. I would also like to thank the other members of the MSI Research Institute for providing a stimulating social environment, and in particular Ian Coutts and Paul Clements for their support in hearing and critiquing my research philosophies as they developed.

In addition, I would like to offer my thanks to many people around the world who have had some input or influence over the course of my study. In particular, my friend and colleague Nelson Minar, who has been a trusted source of advice throughout the journey, and Dr. Danny Lange who has been an excellent mentor and font of wisdom. Also, the members of the agents, mobility and dist-obj mailing lists have provided an invaluable service as a community of peers amongst which to discuss my research. Many of the ideas expressed in this thesis have been shaped and refined in those forums.

One cannot work on anything exclusively for so long and so hard, without the need for respite. I have many friends to who I owe thanks, who have allowed me to relax, rage, or lose myself, away from grindstone. Some deserve special mention. Firstly, my best friend Darren May, who has been there from the early years and will be there at the end. Also, my friends Derek Woods and Andy Grant who have been my partners in many misdemeanours at Loughborough through the years.

Lastly, but most importantly I would like to thank my partner, Joanna Henderson, whose unswerving love, support and companionship have allowed me to concentrate my efforts on achieving my goals. She truly is a wonderful person and I count myself extremely lucky to be with her.

Abstract

The last decade has seen an explosion in the growth and use of the Internet. Rapidly evolving network and computer technology, coupled with the exponential growth of services and information available on the Internet, is heralding in a new era of *ubiquitous computing*. Hundreds of millions of people will soon have pervasive access to a huge amount of information, which they will be able to access through a plethora of diverse computational devices. These devices are no longer isolated number crunching machines; rather they are on our desks, on our wrists, in our clothes, embedded in our cars, phones and even washing machines. These computers are constantly communicating with each other via LANs, Intranets, the Internet, and through wireless networks, in which the size and topology of the network is constantly changing. Over this hardware substrate we are attempting to architect new types of distributed system, ones that are able to adapt to changing qualities and location of service. Traditional theories and techniques for building distributed systems are being challenged. In this new era of massively distributed computing we require new paradigms for building distributed systems.

This thesis is concerned with how we structure distributed systems. In Part I, we trace the emergence and evolution of computing abstractions and build a philosophical argument supporting mobile code, contrasting it with traditional distribution abstractions. Further, we assert the belief that the abstractions used in traditional distributed systems are flawed, and are not suited to the underlying hardware substrate on which contemporary global networks are built. In Part II, we describe the experimental work and subsequent evaluation that constitutes the first steps taken to validate the arguments of Part I.

The experimental work described in this thesis has been published in [Clements97] [Papaioannou98] [Papaioannou99] [Papaioannou99b] [Papaioannou2000] [Papaioannou2000b]. In addition, the research undertaken in the course of this PhD has resulted in the publication of [Papaioannou99c] and [Papaioannou/Minar99].

Contents

List Of Tables

List of Figures

Preface

Mobile Code is a new and generally untested paradigm for building distributed systems. Although garnering many plaudits and continually increasing in popularity, the technology and research field remain relatively immature. So far, most research has focused on the creation of mobile code frameworks, and as yet, there is no conceptual framework with which to contrast results. Equally, there is no clear understanding of the new abstractions offered by this paradigm. Further, many conclusions drawn about the technology remain qualitative and subjective. This dearth of quantitative results means as yet it has not been possible to evaluate the potential of both the technology and the paradigm.

It is against this backdrop that the work described in this thesis has been conducted. Before an accurate and informed decision about the suitability of mobile code technology can be made, a fuller appreciation of the paradigm is required. It is the author's opinion that the central essence of a new paradigm is the abstraction it offers to the designer. Therefore, the contribution of this thesis addresses the issues of *understanding* and *evaluating* the design abstractions offered by mobile code.

The first part of this thesis is concerned with building an *understanding* of the abstractions offered by mobile code, and the implications of using them. Certainly, it would be impossible to undertake this research without a context within which to analyse the new paradigm. To this end, we trace the emergence and evolution of abstractions employed throughout the history of computing, in an attempt to understand the reasons behind the existence of contemporary traditional distribution abstractions. We also build a philosophical argument supporting mobile code, contrasting it with traditional distribution abstractions. Further, we assert the belief that the abstractions used in traditional distributed systems are flawed, and are not suited to the underlying hardware substrate on which contemporary global networks are built.

In chapter one, we review the history of computing, and the abstractions that have been employed within this field. We begin our journey by examining the early years of computing, and trace the consecutive developments that have shaped the evolution of our present day computing landscape. We build a picture of the key phases in this

evolution, and the gradual layering of abstractions, one atop another, that characterises evolution in this area.

In chapter two, we return to focus more directly on the emergence of distribution. In examining today's distribution mechanisms we show that the fundamental abstraction in these systems is one of *location transparency*. The chapter demonstrates that the emergence of location transparency is a result of the layers of abstraction found beneath it. We argue that by using the location transparency abstraction we are attempting to impose an unsuitable abstraction onto the underlying computational substrate.

In chapter three, we begin our examination of the new design abstractions offered by Mobile Code. We discuss what makes mobile code systems different from contemporary ones and characterise these new abstractions as embodying *local interaction*. Finally, we argue that by employing this new paradigm we are using an abstraction more wholly suited to the underlying computational substrate, and thus to building distributed systems. This chapter concludes our philosophical argument concerning the structuring of distributed systems.

The philosophical argument built in Part I is extensive, and a full experimental investigation is beyond the scope and timescale of a PhD. Therefore, in Part II we take the initial steps required to validate the arguments expressed in Part I. If Part I was concerned with understanding the mobile code abstraction, then Part II is concerned with *using and evaluating* it. The experimental work is conducted by applying the new paradigm to a real world manufacturing system application, based on data derived from an industrial case study.

In chapter four, we present the rational for the experimental research undertaken in this thesis, and describe how it will support the arguments made in Part I. Further, we describe the technical issues involved with implementing mobile code abstractions, and discuss some of the advantages claimed for this new technology. Lastly, we review several of the better-known mobile code frameworks available to researchers, before presenting IBM's Aglet Software Development Kit, the framework used in our experimental work.

In chapter five, we describe a case study undertaken in the UK. The case study has been used to generate a real-world model of the Sales Order Process (SOP) of a manufacturing enterprise that is used in the subsequent implementation work. In addition, several requirements of the company were identified which will be used in later chapters as “scenarios for change” with which to test and measure our experimental implementations.

In chapter six, we describe the creation of two prototype mobile code systems. Their common parts and differences are discussed, along with the supporting tools that have been created.

In chapter seven, we begin our evaluation of the two prototype systems. Firstly, we describe the process through which we have generated several tangible software metrics. We then evaluate the prototypes through the “scenarios for change”, and reflect on what has been learnt.

In chapter eight we conclude the research undertaken in this thesis, and discuss the implications of the work, and avenues for further investigation.

Part I

Understanding

1 Abstraction

1.1 Introduction

Computers are fulfilling an increasingly diverse set of tasks in our society. They are silently assuming many mundane but key tasks, providing seamless assistance to support our lifestyles. They control our car engines, our environmental climate and even our toasters. Increasingly, sophisticated hardware is the supporting substrate for increasingly complex software. Yet despite major advances in our understanding of the construction of software, building flexible and reliable systems remains a considerable task. Increasingly powerful abstractions are employed by software engineers in an attempt to reduce the cognitive complexity of such tasks.

The emergence of computing abstractions has been instrumental in defining today's computing landscape. To fully understand its present day shape, we must first understand the forces and issues that influenced its evolution. This chapter presents a brief history of computing and the levels of abstraction developed and employed within this field, and discusses the emergence of each abstraction.

1.2 A Brief History of Computing Time

> "In the beginning there was binary. And 'lo, von Neumann did say 'that's too damn tough to understand! Can't we make it any simpler?'"

In the 1940's, the mathematician John von Neumann pioneered research into formalising the basic architecture for a computing machine. The Von Neumann architecture specified a computer in terms of three main components:

- **A Memory**: a large store of memory cells that contain data and instructions
- **An Input/Output unit**: to enable interaction and feedback with the user
- **A Central Processing Unit (CPU)**: responsible for reading and writing instructions or data from the memory cells or from the I/O unit

During execution, the CPU takes instructions and data from the memory cells one at a time, storing them in local cells known as registers. The instructions cause the CPU to manipulate the data via arithmetic or logic operations, before assigning any results back to memory. Thus, the execution of instructions results in a change in the *state* of

the machine [Burks46]. The three components of a computer are able to interact via a communications bus (see Figure 1).

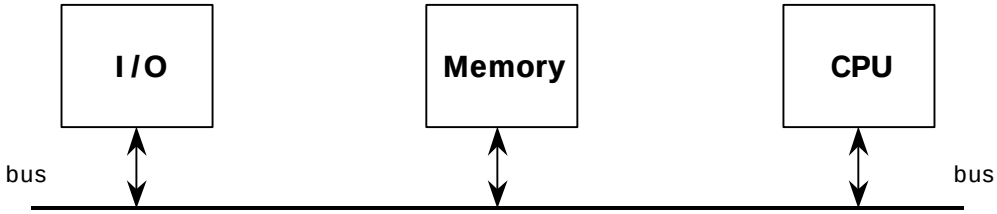

Figure 1. The von Neumann Computer Architecture

Von Neumann's research was based on the earlier theoretical work of Church and Turing on state machines [Church41] [Turing36]. Importantly though, it established a hardware architecture for a computing machine that would serve as a reference platform for decades to follow. Although we are generally accustomed to thinking of computers as extremely complex machines, the central architecture itself is quite simple. At the most basic level Harel states:

> "A computer can directly execute only a small number of extremely trivial operations, like flipping, zeroing, or testing a bit" [Harel87]

Nonetheless, von Neumann had taken the first step along a long path of evolution that would culminate in the computer systems we take for granted today. This evolution could not have taken place without advances in hardware design and manufacture, however, for the scope of this thesis we are interested only in the abstractions and technologies that have evolved to support the construction of software.

Since its creation, the von Neumann architecture has fundamentally influenced the way we think about and build our computing systems. Most contemporary programming languages can be viewed as abstractions of the underlying von Neumann architecture. These languages retain as their computational model that of the von Neumann architecture, but abstract away the details of execution. The sequential execution of language statements (instructions) changes the state of a program (computational machine) through assignment and manipulation of variables (memory cells). These languages, known as *imperative languages*, have developed through the addition of layers of increasingly high levels of abstraction [Ghezzi98]. In the next section we examine the emergence and evolution of imperative languages,

and discuss the ascending tower of abstractions that we use to construct software systems.

1.3 Procedural Abstractions

Programming a computer to perform a particular task in the early years of computing was extremely difficult and time consuming [MacLennan87]. The von Neumann architecture provided a computational model that programmers could use to manipulate physical memory locations. Nevertheless, this was still an arduous task, as each memory location was identified by a long binary string. Humans do not naturally think in binary, and programming in this manner was not only complex but also prone to error [Hopper68].

To alleviate the inherent difficulties with working in binary a new family of languages, known as assembly languages [Harel96], were developed. Assembly languages served as a primitive form of abstraction, which masked the architecture of the underlying hardware. With this new abstraction, programmers were able to specify memory locations symbolically, rather than with an unwieldy binary string.

The creation of assembly languages was the next step towards unlocking the full potential of the computer. Using them, programmers were no longer concerned with the location of individual registers and memory cells. They were able instead to program with symbolic representations of their computing machines. From here, it was a relatively simple matter to begin constructing repeatable computing algorithms from assembler symbols [Wexelblat81]. These algorithms became a layer of abstraction above the assembly symbols, which themselves were a layer of abstraction above the hardware. Quickly, the pattern for computing evolution had been defined: it would evolve through the gradual layering of ever subtler and complex levels of abstraction. Each layer abstracting away the minutiae whilst retaining as their underlying computational model the von Neumann architecture. Figure 2 shows the abstractions of assembly languages, and then computing algorithms layered over the underlying von Neumann computational model.

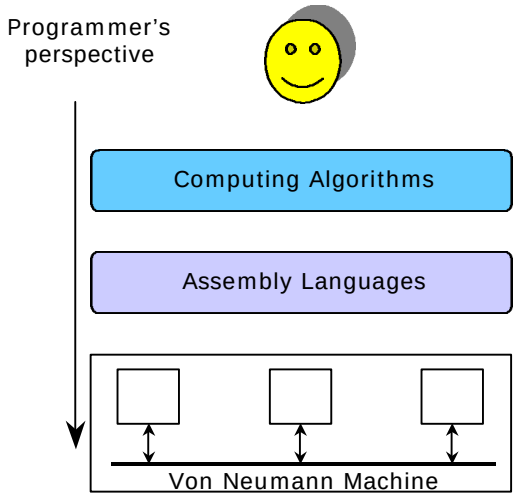

Figure 2. Early Layers of Abstraction

These early layers of abstraction were a considerable improvement in the way computer programs were constructed. However, even more significant improvements in the usability of computers would occur with the arrival of programming languages.

A programming language is a formal notation for describing algorithms for execution by a computer [Ghezzi98]. They provide abstractions to overcome the complexities involved in constructing a software program, so that a programmer does not need to be capable of manually producing the many machine level instructions that are required to get a computer to perform a particular task. The first types of programming languages developed were known as pseudo code languages.

Pseudo codes arose because in some instances programmers found that the hardware specific instructions available on their particular computing architecture were not sufficient to support the range of operations they required. Pseudo codes are machine instructions that differ to those provided by the native hardware on which they are being executed. They are invariably executed within an interpreter [MacLennan87], a software simulation of a computational machine, a virtual machine, whose machine language is the pseudo codes. The virtual machine would normally offer facilities that were not available in the real computer, for example, new data types (e.g. floating point) or operations (e.g. indexing). Ergo, pseudo codes added yet another, higher layer of abstraction, and were the initial steps taken in moving towards a tool that allowed a programmer to construct software in a language that bore no resemblance to its machine code representation [Hopper68]. Unfortunately, pseudo code languages

were hampered by slow execution speeds, since the interpreter had to first convert the codes to native instructions prior to execution. To overcome this inefficiency a new tool known as a compiler was produced. A compiler is a computer program that translates programs specified in high-level languages, for example pseudo codes, into the native hardware's assembly language [Harel93]. The program need only be translated once, but could be executed at native speeds many times, which was a distinct advantage over programs that had to be interpreted every time.

The advent of compilers led to the creation of new programming languages, known as 1st generation languages. The best known of these are IBM's Mathematical FORmula TRANslating system (FORTRAN) [IBM56], COmmon Business Oriented Language (COBOL) [DoD61], and ALGOrithmic Language (ALGOL) [Perlis58] which appeared in the mid to late 1950's respectively. These languages allowed a programmer to use a mathematical notation in order to solve a problem. FORTRAN and ALGOL were defined as tools for solving numerical scientific problems, those that required complex computations on relatively simple data, for example simulating numerically the effects of a nuclear reaction. COBOL was developed as a tool for solving business data-processing problems, those that required computations on large amounts of structured data, for example a payroll application. It was able to satisfy the needs of the bulk of the applications of the day, and its success has meant it remains in use over thirty years after its introduction [Wilson93].

The advent of compilers and 1st generation languages meant it was possible to develop computer programs without any knowledge of how your program was actually transformed into the native instruction set required by the machine upon which it was intended to execute; the translation was automatically performed by the compiler. One of the most important concepts embodied in the abstractions offered by 1st generation languages was the separation of a program into two distinct parts. The description of the data contained within the program was known as the *declarative* part, and the program logic that controlled the execution of the program and manipulation of the data was known as the *imperative* part.

Once begun, the development of programming languages progressed rapidly, and soon 2nd generation languages would emerge. These new languages were generally descendants of 1st generation languages, influenced by the lessons learnt in the early

years. They are characterised by offering a much higher level of structured flow control to the programmer whilst simultaneously introducing new techniques to aid the composition of computer programs. Typical of this set of languages is ALGOL 60 [Naur63]. The product of a committee, ALGOL 60 introduced major new concepts such as syntactic language definition [Backus78], the notion of block structure [Wilson93] and recursive programming [Ghezzi98]. Further improvements to structured flow in languages such as loops, conditional statements, sequential constructs and subroutines [Harel93] meant that some of the hardware-influenced instructions prevalent in 1st generation languages, such as the infamous GOTO[1] statement [Dijkstra68], could be removed.

By the 1970's it was becoming clear that the need to support reliable and maintainable software had begun to impose more stringent requirements on new programming languages [Ghezzi98]. Programming language research in this period emphasised the need for eliminating insecure programming constructs. Among the most important language concepts investigated in this period include: strong typing [Cardelli85], static program checking [Abadi96], module visibility [Parnas72a], concurrency [Ben-Ari90] and inter-process communication [Simon96]. Greater significance was now placed on building reliable software, and the term *software engineering* [Naur68] was used to describe an emerging methodology for dealing with the full lifecycle of software development, from specification to production. In general, it is fair to say that 3rd generation languages built on the previous generation by working at improving the software engineering principles inherent, and enforced by the languages. Some important examples of 3rd generation languages are Euclid [Lampson77], Mesa [Geschke77] and CLU [Liskov81]. The development of these languages was directly influenced by the need to improve systems programming [Wilson93], the creation of operating systems and tools such as compilers, and to produce verifiable programs.

In the last half of the 1970's new languages such as Pascal [Jensen85] [ISO90b] and C [Kernighan78] were developed. Both offered the programmer power, efficiency, modularisation and availability on a wide array of platforms. With Pascal though, Wirth aimed to create a language that would also be suitable for teaching

1 Strangely, the much maligned GOTO statement continues to exist in many languages

programming as a logical and systematic discipline, thus encouraging well-structured and well-organised programs. C on the other hand combines the advantages of a high level language with the facilities, flexibility and efficiency of an assembly language. However, to ensure the degree of flexibility required by systems programmers C does not include type checking, meaning that it is much easier to write erroneous programs in C than in Pascal [Wilson93]. Both languages continue to be widely and successfully employed today.

1.3.1 Commentary

When von Neumann first specified his computing architecture, he set the direction in which our computing landscape would evolve. Since then, we have evolved through the gradual layering of increasingly powerful abstractions upon each other. The progressive development of programming techniques that ascended via early unwieldy bit strings, through assembly mnemonics, pseudo codes, compilers and three generations of programming languages signified the first phase of our evolution. In this phase programmers were gradually lifted out of the mire, and spared the task of remembering the location of each cell or register they wish to use. They were now able to specify programs in powerful and efficient languages, without requiring any hardware specific knowledge of the computer they were using. By progressively exploring and building up the layers of abstraction, the computer had been transformed from a slow and cumbersome behemoth to a powerful, flexible tool.

In this thesis we term this period of computing the *procedural abstraction* phase. It is characterised by the development of new computing abstractions and new techniques for controlling program structure and flow. Figure 3 illustrates the individual layers of abstraction discussed in the previous section. Each box roughly represents the beginning of each abstraction, and is intended to depict the progressive layering of abstractions as programming languages were developed. Certainly each box should not be interpreted as a finite lifetime for each abstraction. For example, assembler continues to be heavily used in modern military aircraft systems [Bennet94].

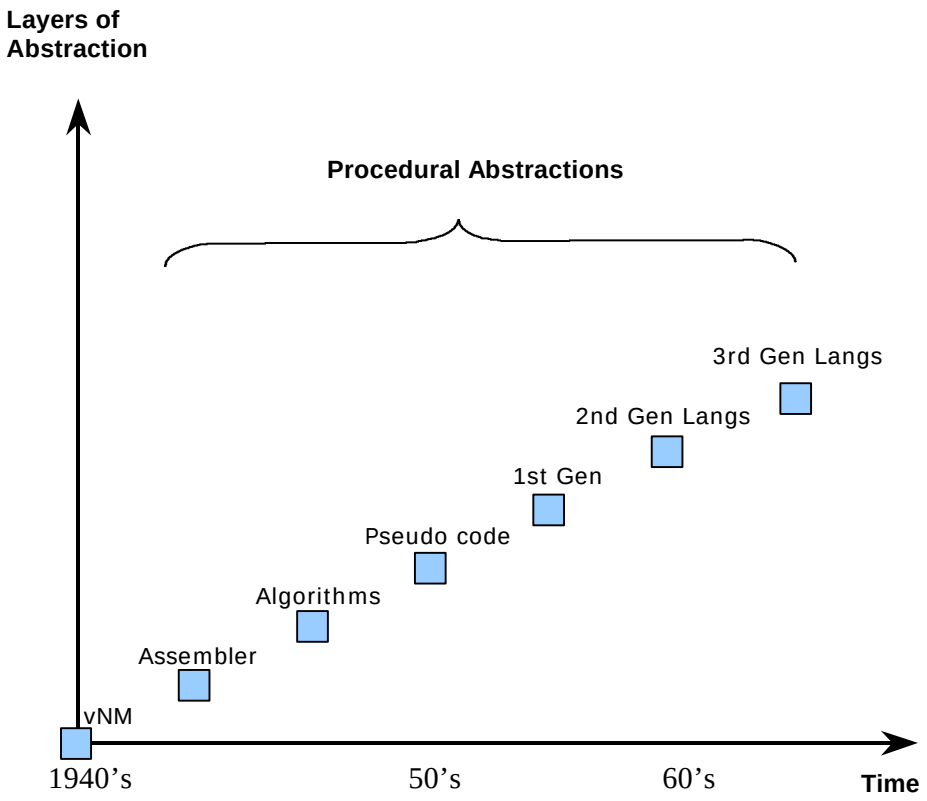

Figure 3. The layers of abstraction in the Procedural Abstraction Phase

1.4 Programming Abstractions

> "Show me your [code] and conceal your [data structures], and I shall continue to be mystified. Show me your [data structures], and I won't usually need your [code]; it'll be obvious." [Raymond98] citing and re-interpreting [Brooks95]

The mid to late 1970's saw a new trend develop within the world of computing. Supported by more powerful tools and languages programmers began to build increasingly large and complex programs [DeRemer76]. These programs were no longer standalone edifices, capable of performing a single task. Rather, they were *systems*, capable of a multitude of tasks.

The sheer size of these systems meant that for reasons of clarity and maintenance it was becoming increasingly important to organise programs into discrete modules [Knuth74]. With the Modula-2 language [Wirth77], Wirth attempted to extend Pascal

with modules and while not wholly successful the experiment was an indication of the possible advantages [Wilson93]. Language researchers soon realised that it was not only advantageous to separate programs into discrete modules, but also to conceptually encapsulate data and logic within larger entities. Such encapsulations were known as abstract data types [Hoare72] and enabled the programmer to specify new data types in addition to those primitives already supported by the language. For these new abstractions, programmers could define operations through which they could be manipulated, while the data structure that implements the abstraction remained hidden. Information or data hiding [Parnas72a] ensures that the internal data of a new type will only be manipulated in ways that are expected. The late 1970's and early 80's saw an explosion of new programming abstractions, such as type extensions [Wirth82], concurrent programming [Andrews83] and exception handling [Goodenough75]. Again, the motivation was to make software more maintainable in the long term. A resulting synthesis of many of these new techniques is the language Ada [DoD80], which can be viewed as the state-of-the-art for that time.

The 1980's saw the arrival of Object-oriented Programming (OOP), the origins of which can be traced back to Simula 67 [Birtwistle73]. An object is an encapsulation of some data, along with a set of operations that operate on that data. Operations are invoked externally by sending messages to the object [Blair91]. Thus, each object is an abstraction that both encapsulates and acts upon its logic and data respectively. This allows a programmer to view their system as being composed of conceptually separate entities, or objects. The OOP abstraction also builds on the previously discussed advances in modularity, data abstraction and information hiding, by including facilities for software reuse [Ghezzi98]. Newly created objects in the system are not implemented from scratch, rather they may inherit pre-existing behaviour from a parent object, and implement only the required new behaviour. OOP initially became popular through the success of Smalltalk [Goldberg83], but was more widely accepted with the advent of C++ [Stroustrup92], an extension of C. Other popular OO languages include Dylan [Apple92], Emerald [Raj91], Modula-3 [Nelson91] and more recently Java [Gosling96].

1.4.1 Commentary

In this thesis we term this ascendance from building programs, to architecting systems as the *programming abstraction* phase. It is characterised by the development of new techniques for modularity, data abstraction and software reuse, and would result in systems that were easier to change and maintain [DeRemer76] and were more reliable [Horowitz83]. In Figure 4 below we see the programming abstraction phase continue the gradual layering of abstractions.

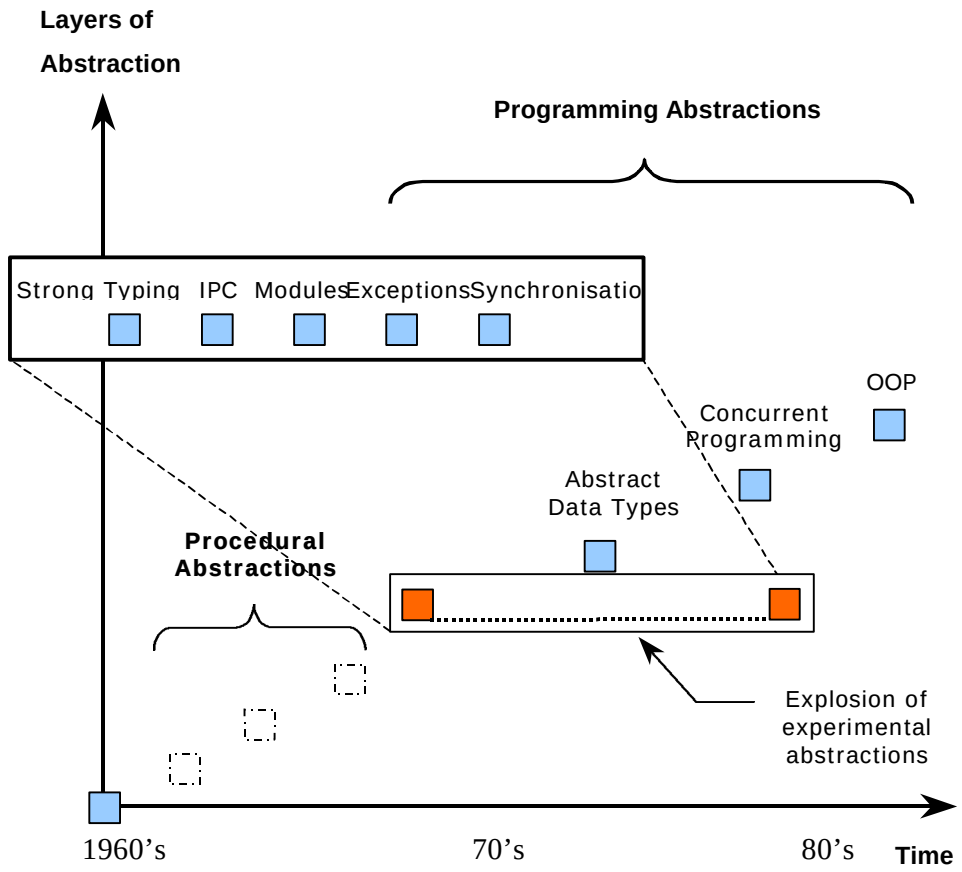

Figure 4. Layers of abstraction in the Programming Abstraction Phase

1.5 The Far Side

So far, we have concentrated solely upon the ascending layers of abstractions that are present and supported by imperative or procedural languages. These languages employ the von Neumann architecture as their underlying computing model, and are greatly influenced by the necessity for efficient execution.

With the decreasing costs of computer hardware, however, radically different designs of computing machine have become possible. This has opened up the possibility that other computational models could be found, and that it may be possible to design the computer hardware to fit the model, rather than the other way round [Wilson93]. As early as the 1960's there have been attempts to define programming languages whose computational models were based upon well-characterised mathematical principles, rather than on efficiency of implementation [Ghezzi98]. These alternative camps can be split into functional and logic programming languages.

Functional languages use as their basis the theory of mathematical functions, and they differ greatly from imperative languages as they do not support the concept of variable assignment. Assignment causes a change in value to an existing variable, whereas the application of a function causes a new value to be returned. This has important implications for the problem of concurrency, since in an imperative language it is possible to refer to a variable or object that has been reassigned without your knowledge. In a functional language, a function may be called at any time, and will always return the same value for a given parameter [Hudak89]. Further, since variables cannot be altered by assignment, the order in which a program's statements are written and evaluated does not matter; they can be evaluated in many different orders. Thus, programs can be modified as data and data structures can be executed as programs. The key concept in functional programming is to treat functions as value, and vice versa [Watt96].

The archetypal functional programming language is generally considered to be LISP [McCarthy60], which was developed in the late 1950's. It is based upon the theory of recursive functions and lambda calculus, work that was developed in the early 1940's by Church [Church41]. Since its creation LISP has become one of the most widely used programming languages for artificial intelligence and other applications requiring symbolic manipulation [Pratt84], for example symbolic differentiation, and has spawned a plethora of individual dialects. As with the imperative camp, there have been several other implementations of functional languages during the following years, for example APL [Iverson62], ML [Milner90], Miranda [Turner85] and Haskell [Thompson96]. Latterly, the competing dialects of LISP were unified in Common LISP [Bobrow88].

Another variant in the field of programming languages are those defined as logic programming languages. The main difference between functional and logic programming languages is that programs in a pure functional programming language define functions, whereas pure logic programming defines relations [Ghezzi98]. Logic programming languages first appeared in the late 1970's and are based on the principles of first order predicate calculus [Mendelson64] and eschew all relation to the underlying machine hardware. In contrast to other styles of programming, a programmer using a logic language is more involved in describing a problem in a declarative fashion than in defining details of algorithms to provide a solution [Callear94]. The knowledge about a problem and the assumptions about it are stated explicitly as logical axioms [Kowalski79]. This problem description is then used by the language's computational machine to find a solution. To denote its distinctive capabilities, in this case a computational machine that can execute a logical language is often referred to as an inference engine. Synonymous with logic programming, and the ancestor of all logic languages is PROLOG [Clocksin87].

1.5.1 Commentary

The genres of functional and logic programming languages are an important contribution to our computing landscape. Both are declarative languages and are characterised as being independent of the underlying hardware upon which they are executed; they are abstractions that are not influenced by the von Neumann architecture. However, to achieve this independence efficiency has been sacrificed [Wilson93]. This, and the fundamental change of programming mindset required for those accustomed to the imperative style has been detrimental to their widespread acceptance and deployment outside of the artificial intelligence and expert systems communities.

Perhaps most revealing in the functional vs imperative language debate is the 1978 Turing Award lecture given by John Backus [Backus78]. In this, and his paper, Backus argues that conventional programming languages are fundamentally flawed in their design since as they are inherently linked to the underlying von Neumann architecture. Backus goes on to demonstrate the advantages of functional languages over imperative ones, and further introduces a new functional language, FP. His

assertion is that the underlying abstractions we use are important, and can affect the way we think, use and build computer systems and software.

1.6 Conceptual Abstractions

In the last decade, software engineering has been scaling new heights of abstraction. Program development has undergone a tremendous revolution; in the way that programs are entered into the computer, and the way programs are assembled from existing parts [Ghezzi98]. Programmers are now able to use integrated development environments and libraries of predefined modules to rapidly compose software systems visually [Zak98].

Recent developments such as Components [Sun97] allow developers to view their systems with a larger granularity than objects. Components may be large, for example a Request Broker consisting of hundreds of objects, or as a small as a GUI widget consisting of only a few objects. In addition, techniques such as Software Patterns [Gof93] enforce a rigid literary methodology for expressing the essence of a recurring software abstraction. A pattern may be viewed as a monograph on the particular abstraction, and describes the many facets required to consistently select and use an appropriate abstraction, what issues are involved and when not to use this pattern. It is a distillation of knowledge gained by many experts over the years. Aspect Oriented programming [Kiczales97], Actors [Agha97], and Agent Oriented Programming [Wooldridge99] are examples of techniques that attempt to remove any notion of hardware from the abstraction. In fact, one may view them as attempts to personify software. In particular, the autonomous agent community appears to be having much success with its approach, allowing designers to view and build systems in a new manner, with new perspectives [Jennings et al98].

These new abstractions are no longer merely based on technological developments in language or compiler design. They are conceptual abstractions, allowing the software designer to view their system at a level completely removed from any of the underlying hardware issues. Figure 5 is the culmination of this chapter’s examination of the gradual layering of abstractions. It illustrates chronologically all three phases of abstraction we have identified: procedural, programming and conceptual, and how each individual abstraction has been layered over those preceding it.

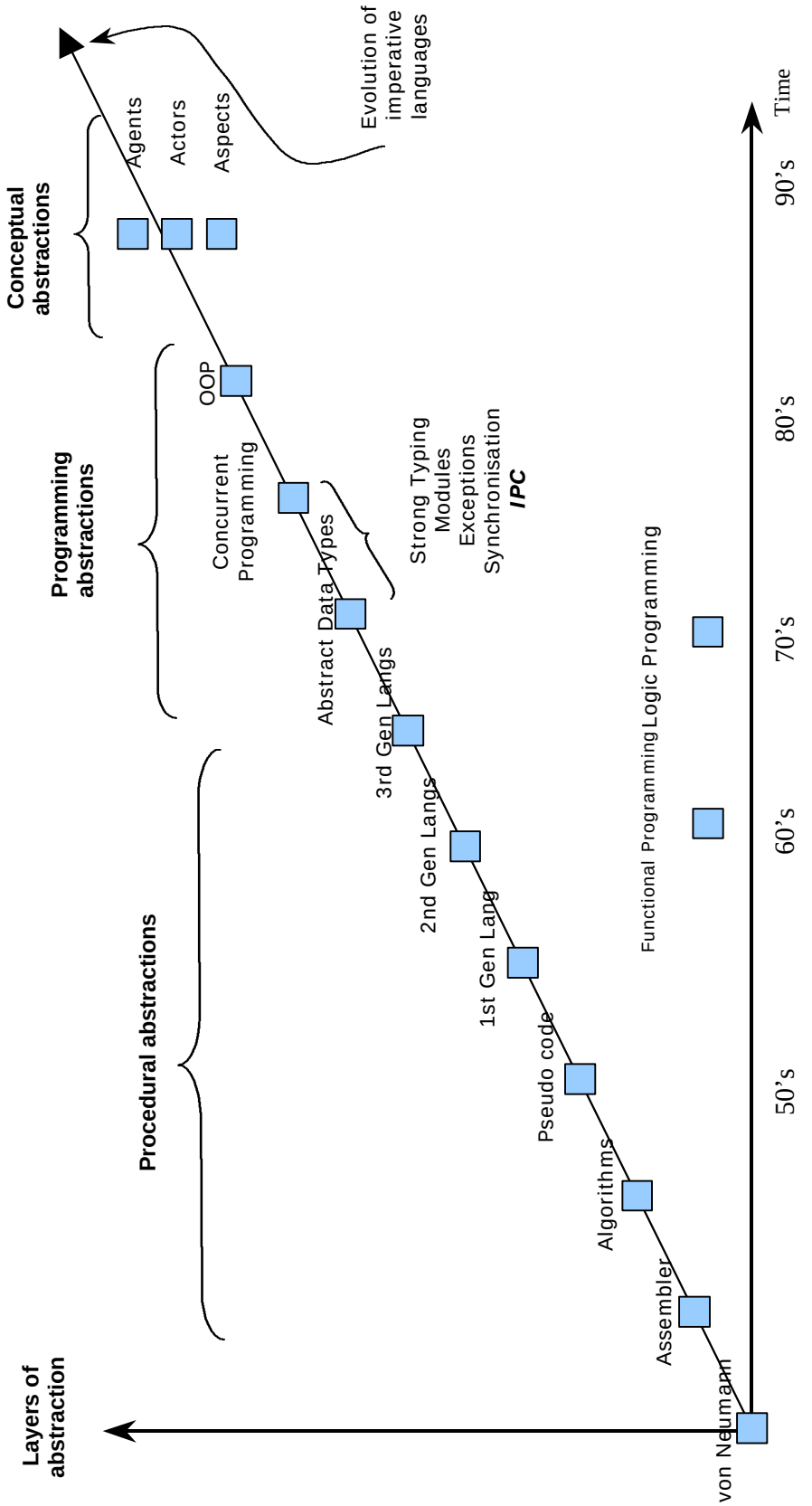

Figure 5. The full Layers of Abstraction diagram

1.6.1 Commentary

The computers we build are no longer merely high-powered calculating machines; they are useful tools that can be both incredibly flexible, and stubbornly inflexible at the same time. Our on-going affair with computers has been characterised by our attempts to harness their power, and apply them to ever more diverse situations. This affair has been tempered, however, by the complexity inherent in a computing system. The complexity involved has forced us to continually refine the languages and tools we use to build software systems. In our efforts to understand and use the technology we abstract away the details, pasting on ever more elaborate facades to hide us from the true complexities involved in creating software. Gradually we have layered increasingly complex abstractions over those lying beneath, until it is no longer even a requirement to be aware of those early abstractions. Modern day programmers have rapid development tools and libraries with which to build software. They employ conceptual abstractions that bear no resemblance to underlying hardware upon which their creations will be executed. These layers of abstraction mean that modern day programmers are not required to be aware of the abstractions that lie below, that they depend on to deliver their creation.

1.7 Concluding Remarks

> "Each successive language incorporates, with a little cleaning up, all the features of its predecessors plus a few more" [Backus78].

> "Appropriate abstractions and proper modularisation help us confront the inherent complexities of large programs" [Ghezzi98]

Abstractions are an immensely powerful tool. They allow us to manage the complexity of a situation, and to rationalise about it by removing those details we consider inessential. Further, as we attain understanding of complex issues, we construct additional layers of abstraction over those beneath, continually ascending. If we are to consider abstractions that exist within these layers we must understand the reasons for their existence, and the base abstractions that support the grand edifice.

This chapter has presented a brief history of our progress up the computing abstraction tower. It has examined the chronological development of computing

architectures and programming languages, and presented a brief explanation of their existence. Latterly, the discussion continued by examining more recent programming and conceptual abstractions and their position in the Tower of Abstractions. Although the functional and logic programming camps offer us a declarative alternative they are in the minority. The overwhelming majority of languages in use today are imperative. They are powerful abstractions whose roots are found in the pioneering work of John von Neumann in the first half of this century. Our computing evolution has been characterised and dictated by the von Neumann architecture. It has influenced the design of all imperative languages to follow, and therefore those abstractions subsequently attained by using the languages.

The aim of this thesis is to understand the mobile agent abstraction, a new technology and abstraction for building distributed systems. The review in this chapter has provided a context and history in which new and existing abstractions can now be reviewed. In the next chapter we examine the abstractions currently used in building contemporary distributed systems.

2 Towers of Babel

2.1 Introduction

In the 1970's, networking began to emerge as an important aspect of computer systems. Driven by applications in the military and airline industries, computer systems were connected and inter-operation became widespread [Cerutti83]. During the 1980's, distributed computing became a vital aspect of many computer systems. In the early 2000's, we are beginning to see the emergence of ubiquitous computing: characterised as a massive heterogeneous "sea" of disparate computational devices, with varying connection bandwidths and an ever-changing topology of connections [Weiser91].

This chapter examines the emergence of distribution and discusses the path of its evolution. In examining today's distribution mechanisms we show that the fundamental abstraction in these systems is one of *location transparency*. Further, we demonstrate that the emergence of location transparency is a result of the layers of abstraction found beneath it. We argue that by using this approach we are attempting to impose an unsuitable abstraction onto the underlying hardware substrate.

2.2 The Advent of Distribution

Before the invention of computers, processing information was both slow and tedious [Rose90]. The advent of computers has transformed the world, and the way in which we work with information [Simon96]. However, using and storing this information in isolation, like any expensive resource, is inefficient [Peters85]. Ergo, unless our computers are to exist in isolation, we require methods that allow computers to meaningfully interact [Cerutti83], and ways of transferring information between them. Communication networks, which interconnect computers and allow them to work in concert, are a common solution to this problem [Sloman87].

However, merely physically connecting computers is not enough to achieve logical interaction in its own right. Computers must adhere to a common set of rules or protocols for defining their interactions [Rose90]. By connecting separate computers, we make it possible for the programs executing on those computers to interact. When

processes on separate computers interact, we term the whole a distributed system. In the next section, we examine the software architectures used in building networks, which ultimately support any communication between networked computers.

2.3 Distributed Communication

> A network is an interconnected collection of two or more autonomous computers [Tanenbaum96].

Distributed computing as we understand it today is a far cry from the limited facilities of early distributed systems, such as remote job entry handlers [Boggs73]. Their role however was simple - to allow scarce and expensive information and resources to be shared by users. Ever since computer users began accessing central processor resources from remote terminals over 40 years ago, computer networks have become more versatile, more powerful and inevitably more complex [Green80].

At the heart of distributed computing are communication networks. They are the infrastructures that support information flow between computers. The initial development of such networks was fostered through experimental networks such as ARPANET [Roberts70] [Cerf74] and CYCLADES [Pouzin73]. ARPANET, which went live in December 1969, was initially motivated by the requirements of the US Military for a communications network that could survive a nuclear war [Tanenbaum96]. This early work established the procedures for connecting computers and facilitating their interaction. Just physically connecting computers was not sufficient to ensure successful interaction though. Two computers wishing to communicate must adhere to a common set of rules for defining their interactions. This rule set is termed a *protocol*, and is an agreement between the communicating parties on how communication is to proceed [Rose90].

To reduce their design complexity, network architectures are organised as a series of layers or levels of abstraction, each built upon the preceding one. Whilst the number and nature of these layers may differ between architectures, their purpose is similar: to offer services to the higher layers, shielding them from the details of how the offered services are actually implemented [Tanenbaum96]. Each layer has its own particular

communication protocol, and collections of protocols defined in terms of a common framework are known as a *protocol suite* or *stack* [Rose90].

In early computer systems, it was common for each application found on a computer to employ its own protocol stack. This communication support was usually built into the application, and was not available for use by any other applications. This approach therefore had the inherent disadvantages of duplicated functionality and inefficient resource usage. To alleviate this undesirable situation, research focused on providing communication mechanisms at the operating system level through the provision of shared communication suites [Sloman87].

Although a vast improvement, facilities provided by the operating system were invariably specific to the particular type of computer on which they were executing. In the mid 1970s, computer vendors began to develop their own network architectures, to enable communication between their own ranges of machines. Important examples of this period are the Internet model [Metcalfe76] [Comer91] that emerged from ARPANET [McQuillan77], IBM's Systems Network Architecture (SNA) [McFadyen76] [Cypser78] [Gray83] and Digital's DECnet [Wecker80] [Malamud91]. This meant however, that since each suite was developed for the vendors' own machines, they were usually composed of proprietary (closed) protocols. This situation posed two considerable problems:

- Systems from competing vendors were not able to interoperate
- The communication specification was controlled by a single organisation

Since the vendors controlled the protocol specification, they also had the power to change the specification at their discretion [Cerutti93]. Understandably, this made third party developers very nervous in adopting and working to a standard whose specification might be changed at any given moment. Although subsequent publishing of the protocol specifications aided their widespread adoption, the issue remained [Rose90]. Further, as each proprietary communication suite evolved, systems from competing manufacturers became even more incompatible.

The splintered evolution of incompatible communication suites forced the computing community to realise that standards were required to enable interaction between different types of computer [Mullender93]. In 1977, the International Standards

Organisation (ISO) began working towards defining a non-proprietary (open) suite of protocols. The resulting standard is known as the ISO Open Systems Interconnection (OSI) reference model [Zimmermann80] [ISO83] [OSI84] [STA87], and is jointly defined by ISO and the International Telecommunications Union (ITU-T)[2]. Most of the proprietary suites that preceded the OSI model have since undergone modification and are now considered as specialised incarnations of the OSI model.

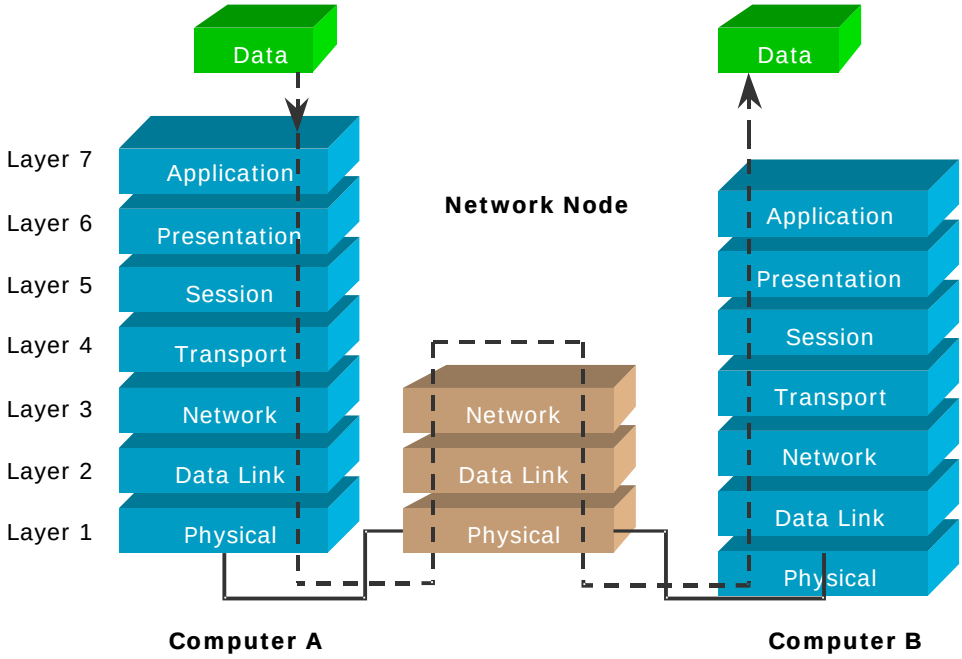

Figure 6. The OSI Reference Model

The OSI Reference model is structured into seven layers that represent the logical sequence of functions carried out when messages are constructed for transmission, dispatched, and then dismantled on arrival [Simon96]. It also serves to provide a common basis for the co-ordination of communication systems standards development and to allow existing standards to be placed into perspective [Sloman87]. An example of the OSI Reference Model is shown in Figure 6. Data at Host A is translated by the OSI stack into a form that can be communicated over the wire. It is then sent over the wire (perhaps via some network nodes), before it is reconstituted at Host B by the corresponding protocol suite, before finally being made available to the destination application.

[2] Formerly the Consultative Committee for International Telegraph and Telephones (CCITT)

Of particular interest to this thesis is Layer 7 – the Application layer. The Application layer is the highest level of abstraction defined in the OSI model and is ultimately responsible for managing the communications between applications. It provides programming primitives that a developer is able to use to access the communication facilities offered by the full protocol suite.

2.3.1 Commentary

In the previous section, we have briefly examined the emergence of communication protocols, and protocol suites, that support distributed computing. Their role and existence has been vital in ensuring we are able to successfully network our computers. In themselves, protocol suites form a hierarchy of abstractions. They provide a mechanism for translating a signal on the wire up through the layers of abstraction until at the application layer the information can be manipulated via programming primitives. These primitives bear little resemblance to their representation 'on the wire' but a developer is able to call upon the communication facilities with relative ease. The advent of the OSI model, and particularly the Internet incarnation of that model, has made communication between distributed computers much simpler. There are now a number of well-known and widely deployed communication suites in existence [Tanenbaum96].

The OSI model, and the many incarnations of protocol suites in existence are important in that they allow computers to communicate in an agreed manner. They do not address how a distributed application may be constructed. These suites are only the enabling infrastructure. Further techniques and technology are required. In the next section, we examine the emergence of distributed systems and concentrate on developments within the application layer of the OSI model.

2.4 Distributed Systems

> "A distributed system is one in which several autonomous processors and data stores supporting processes and/or databases interact in order to co-operate and achieve an overall goal. The processes co-ordinate their activities and exchange information by means of information transferred over a communications network." [Sloman87]

To understand the evolution of distributed systems, we must briefly return to examine the history of computing systems. As discussed in Chapter 1, the end of the procedural abstraction phase indicates a paradigm shift in the way software was constructed. Instead of just building monolithic standalone programs that ran in isolation, it became evident that building systems composed of smaller co-operating programs was a more effective way to construct software. Software architects began to divide their systems into discrete elements. These elements were programs in their own right, and became known as processes. A process is a running program that consists of an environment for execution and at least one thread of control [Coulouris94]. They are smaller, more manageable entities that still execute within the same computational machine, but are separately autonomous[3].

Dividing monolithic software systems into distinct processes had advantages for manageability, but meant a method was required that would allow executing processes to communicate with each other. Finding a solution to this problem became a widely researched issue with many languages gaining new facilities and programming primitives. These new facilities became known as Inter Process Communication (IPC) [Cashin80] [Fukuoka82].

2.4.1 Inter Process Communication

An early method for communication between separate processes was a unidirectional stream of bytes, known simply as a *pipe* [Coulouris94]. On a UNIX machine, for example, a pipe can be used to join the `ls` and `more` commands, e.g. '`ls –l | more`'. The output of the `ls` process is piped as input to the `more` process.

Pipes were designed as a method for linking chains of simple data-transforming programs. Initially though, they did not support networked communication, and were not able to handle large volumes of data[4] [Tanenbaum96]. A further drawback was that the pipes were bound to a specific source and target process (`ls` and `more` respectively in the above example). *Named pipes* subsequently overcame this latter limitation, allowing pipes to exist independently of any particular process.

[3] With respect to the other processes. The operating system still controls all of the processes.

[4] Local files are able to overcome this problem.

Since all interacting processes are local to each other in IPC, it is also possible to use the computer's RAM to implement a *shared memory* facility - a common region of memory addressable by all concurrent processes. Shared memory has become an important technique for use between communicating local processes. Unfortunately, there is no inherent synchronisation in this mechanism and it is easy for one process to write a value to memory for storage, and have another process overwrite it with a new value, or even erroneous data. To combat this problem, new techniques for synchronisation between processes were developed such as semaphores [Dijkstra68b], monitors [Hoare74] and sequences [Reed79].

A further communication mechanism developed was known as a *Message queue*. Message queues allow any process to write to a named queue and for any process to read from the queue. Synchronisation is inherent in the read/write operations and the message queue, which between them can support asynchronous communication between many different processes [Simon96]. Messages are distinguished by a unique identifier or message type, but are limited by being able to hold relatively small amounts of data. Table 1 lists the early IPC communication facilities, and details their advantages and disadvantages.

Method	Advantages	Disadvantages
Pipes	Simple to use; easy to chain multiple pipes;	No network support; insecure communication;
Named Pipes	Can exist unconnected to a process;	As above;
Local Files	Can handle large volumes of data; Simple to use;	Synchronisation problems; inefficient due to repeated disk access;
Shared Memory	Very fast; very efficient;	Cannot handle large volumes of data; no inherent synchronisation
Message Queuing	Inherent synchronisation; unique identifiers;	Can only hold relatively small amounts of data;

Table 1. Inter Process Communication Facilities

As the use of these facilities proliferated, it became increasingly useful to provide them as standard components of the operating system. This was normally achieved

by providing programming primitives that system builders could then employ [Coulouris94]. An early and well-known example are the IPC primitives provided in the BSD 4.x [Leffler89] versions of the UNIX [Ritchie74] operating system. These are implemented as a software layer over the underlying transport and network layers and are based on *socket pairs*, one belonging to each of a pair of communicating processes. Sockets provide a simple way of programming distributed applications using indirect message passing communication [Simon96].

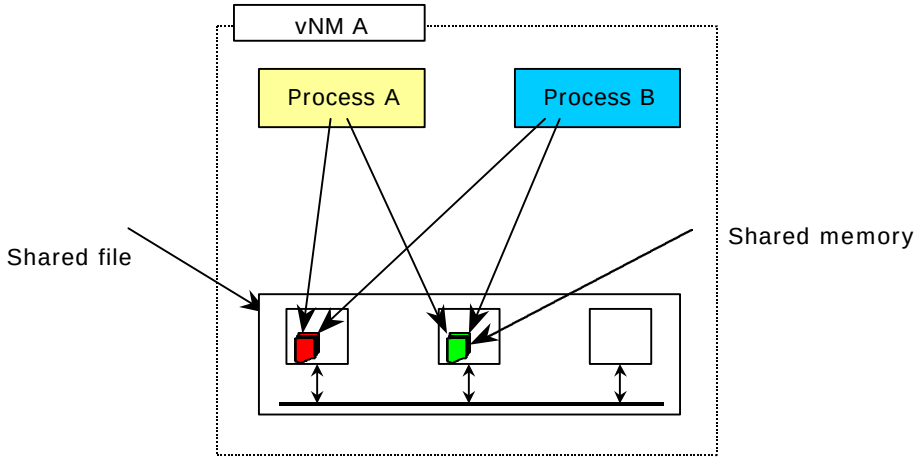

Figure 7. Inter Process Communication

In Figure 7 we see an example of IPC. Two processes are communicating by using a combination of the techniques mentioned in Table 1. By employing both local files and shared memory an optimum balance can be struck between volume of data and speed of access. Importantly, these techniques are ideal for communicating processes that exist within the same von Neumann machine.

2.4.1.1 Commentary

IPC was successful because it provided:

- simple yet effective facilities
- facilities designed for the local computing context
- facilities that were able to take advantage of local resources, e.g. memory and file space

The major factor in the success of IPC however, stemmed from the abstraction it embodies. The IPC abstraction takes full advantages of the constituent elements of the von Neumann architecture. Therefore, it is ideally suited to the underlying

hardware upon which it is used. IPC was only useful, however, for communication between processes that are executing within the same computing machine. As computer networks increased in number and size, resources were scattered even further. This distribution of resources meant that it was increasingly useful for a process on one machine to be able to access a process or resource that was located on another. Unfortunately, the existing IPC mechanisms were designed for communication between local processes only. They were complex and difficult to use in a networked manner. There was therefore a clear need for a simple mechanism to allow two networked machines to interact.

In a seminal paper, Birrel and Nelson [Birrel84] described a new mechanism, Remote Procedure Calls (RPC), which they built for the Cedar [Teitelman84] programming environment to allow remote communication.

2.4.2 Remote Procedure Calls

At their simplest, Remote Procedure Calls (RPC) are a mechanism that facilitate a request/reply interaction between two distributed processes [Simon96]. This is similar to the traditional mechanism of procedure calls [Harel93] found in high-level programming languages. The fundamental difference is that the calling procedure executes in one computing machine, and the called procedure executes in another [Cerutti93], whilst data is exchanged between the two communicating parties.

Birrel and Nelson's goal was to provide a mechanism through which remote processes could interact. They also aimed to make this mechanism *transparent* to the programmer by ensuring it was syntactically similar, and as simple for the programmer to use as ordinary procedure calls [Simon96]. Consequently, the mechanism for RPC was modelled *directly* on the IPC facilities found in the Mesa programming language [Mitchel79]. Indeed, so successful were they that RPC has no distinction in syntax between a local and a remote procedure call [Colouris94].

During an RPC call there are five separate modules that interact to enable the call. They are the client, the client-stub, the RPC communications package (RPC Runtime), the server-skeleton and the server (see Figure 8). When the client wishes to call a procedure that exists on a remote machine, it invokes the appropriate method in the client-stub. To the client, this resembles a normal local procedure call. The

client-stub then assembles one or more data packets that include the target procedure and the required arguments. These packets are then passed to the local RPC Runtime, which transmits them to the remote Runtime. On receipt, these packages are passed to the server-skeleton, where they are unpacked and passed to the target procedure in the server. Once this procedure has been executed, any results are packaged up and the process repeated in reverse. RPC is synchronous in nature, so while the server procedure is executing, the client is suspended, awaiting the result. The RPC Runtime (or request broker) establishes a client/server relationship between the interacting parties, removing the need for each party to be aware of the other's location.

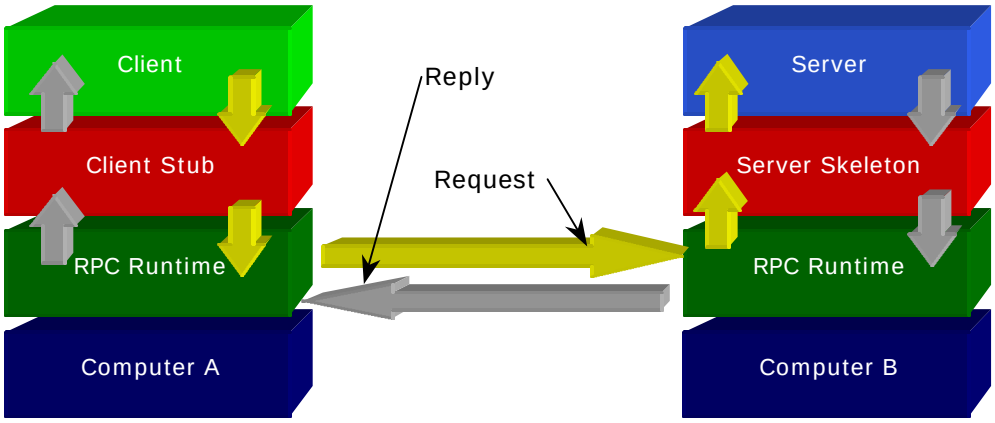

Figure 8. A Remote Procedure Call

Many RPC systems have subsequently been built, and they fall into two categories:

1] The RPC mechanism is integrated with a particular programming language that includes a notation for defining interfaces between communicating processes

2] A special purpose interface definition language that is used for describing the interfaces between clients and servers

In the first instance, languages such as Cedar, Argus [Liskov88] and Arjuna [Shrivastava89] achieve close language integration so that the requirements of remote procedure calls are handled by the language constructs themselves. The second instance includes examples such as Sun RPC [Sun89] and the Matchmaker interface language [Jones86], which have the advantage of not being tied to a specific language environment. This is achieved by having a platform neutral language that can be used to specify the names of procedures, and their required arguments, which the server is making available to the client. These specifications are known as interfaces, and are specified with an Interface Definition Language (IDL) [OMG99].

Due to its request/reply nature RPC is an extremely good way of doing Client/Server application work [Crichlow88]. Client/Server is a particular paradigm for distributing a system, where the server is a manager of one or more resources and a client is a user of that resource. The paradigm was used extensively in the 1970's to structure operating system level process interaction [Simon96] [Walsh85], and is still in extensive use today. One of the best contemporary examples being the World Wide Web [Berners-Lee92].

2.4.2.1 Commentary

The major tenets of RPC can be summarised as:

- The syntax for calling a local or remote procedure is identical
- The location of a resource is transparent to the programmer and user
- Communication is synchronous, and engenders the client/server paradigm

The early 1980's saw many breakthroughs in the distributed systems arena. Some were influenced by earlier theoretical propositions, such as communication between sequential processes [Hoare78], which were now being supported by the increasingly widespread adoption of the OSI networking suite. There were also attempts to incorporate RPC into existing programming languages, such as CONIC [Kramer83], whilst new programming languages that included distribution facilities were also developed, for example SR [Andrews82]. Again, so many proprietary and differing RPC solutions meant that the computing landscape became fractured.

In the same way that the chaos of competing, incompatible and proprietary communication protocols necessitated the creation of the OSI model, the need for a standardised model for distributed applications was recognised. In 1987, ISO began work on a Reference Model for Open Distributed Processing (RM ODP) [Brenner87] [Hutchison91] [ISO92].

2.4.3 RM-ODP

The RM-ODP model provides a framework for ODP standardisation and for the specification of systems using ODP standards [Cerutti93]. RM-ODP was an attempt to unify proprietary RPC systems, and distributed application creation. As a model, it describes in detail the application layer of the OSI model (see Figure 6). The driving

objective behind its creation was to develop a distribution infrastructure that would compliment and support the existing computing infrastructures.

Transparency Type	Proposed Advantages
Access Transparency	Enables local and remote information objects to be accessed using identical operations
Location Transparency	Enables information objects to be accessed without knowledge of their location

Table 2. Network Transparency

Like the OSI model, RM-ODP was purely a reference model. Its specification however, extends the concepts of transparency first visited by RPC, and identifies eight separate forms of transparency. These are discussed further by [Colouris94], but for the purpose of this thesis, it is suffice to demonstrate that transparency is a fundamental tenet of the RM-ODP model. We are only concerned with access and location transparency, collectively known as *network transparency* (see Table 2). Their presence or absence most strongly affects the utilisation of distributed resources [Colouris94].

Since its specification there have been a number of distributed infrastructures created that are based upon the RM-ODP model. These include the Open Software Foundation (OSF)'s Distributed Computing Environment (DCE) [OSF92], the Computer Integrated Manufacturing – Building Integrated Open SYStems framework (CIM-BIOSYS) [Gascoigne94], Sun's Remote Method Invocation (RMI) [Sun98], Microsoft's Distributed Component Object Model (DCOM) [Redmond97] and the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) [OMG94]. Some of the more recent infrastructures integrate RPC with the object paradigm in an attempt to combine the benefits of the latter, in terms of modularity, with the established communication mechanism of the former [Picco98].

2.4.3.1 Commentary

In a manner similar to the process observed in Chapter 1, the abstractions that have been created to support the construction of distributed systems have gradually been layered upon each other, continually reaching ever higher.

In Figure 9 we see the evolution of distribution abstractions. IPC first came into existence as an abstraction to enable communication between processes executing within the same computer, or von Neumann machine (vNM). So successful was this abstraction that Birrel and Nelson designed RPC in an attempt to enable remote and local calls to appear identical. Out of the confusion of proprietary RPC implementations, the RM-ODP model was born, which in turn has led to contemporary distribution infrastructures such as CORBA or RMI.

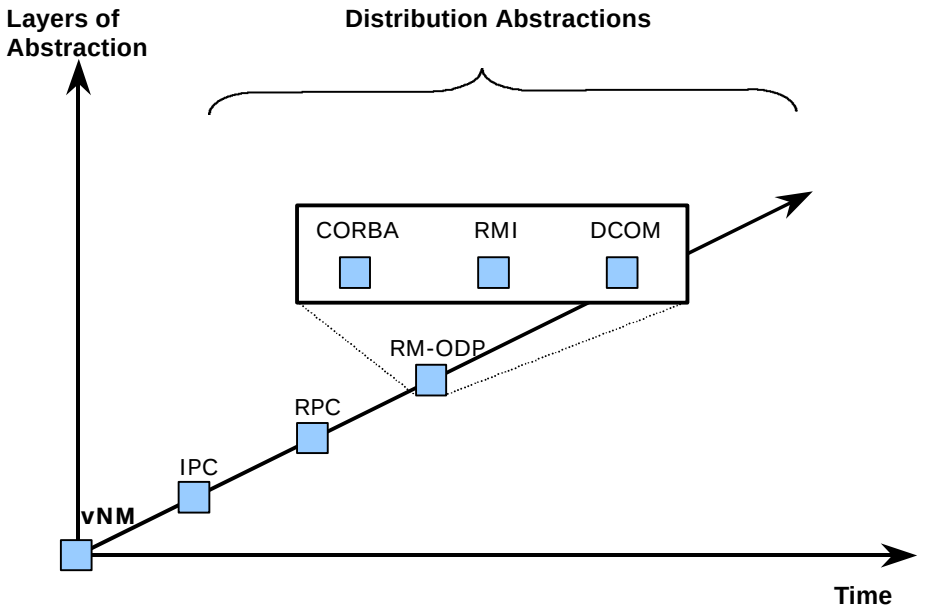

Figure 9. The evolution of Distribution Abstractions

By following the location transparency abstraction, contemporary distribution infrastructures in effect attempt to provide a virtual von Neumann machine. That is, by trying to fool every component in the system that they exist within the same address space, the overall effect is the creation of a virtual machine. Figure 10 shows an example of a distributed system built with the RM-ODP abstraction. The request broker provides a “plane of transparency” to the interacting processes.

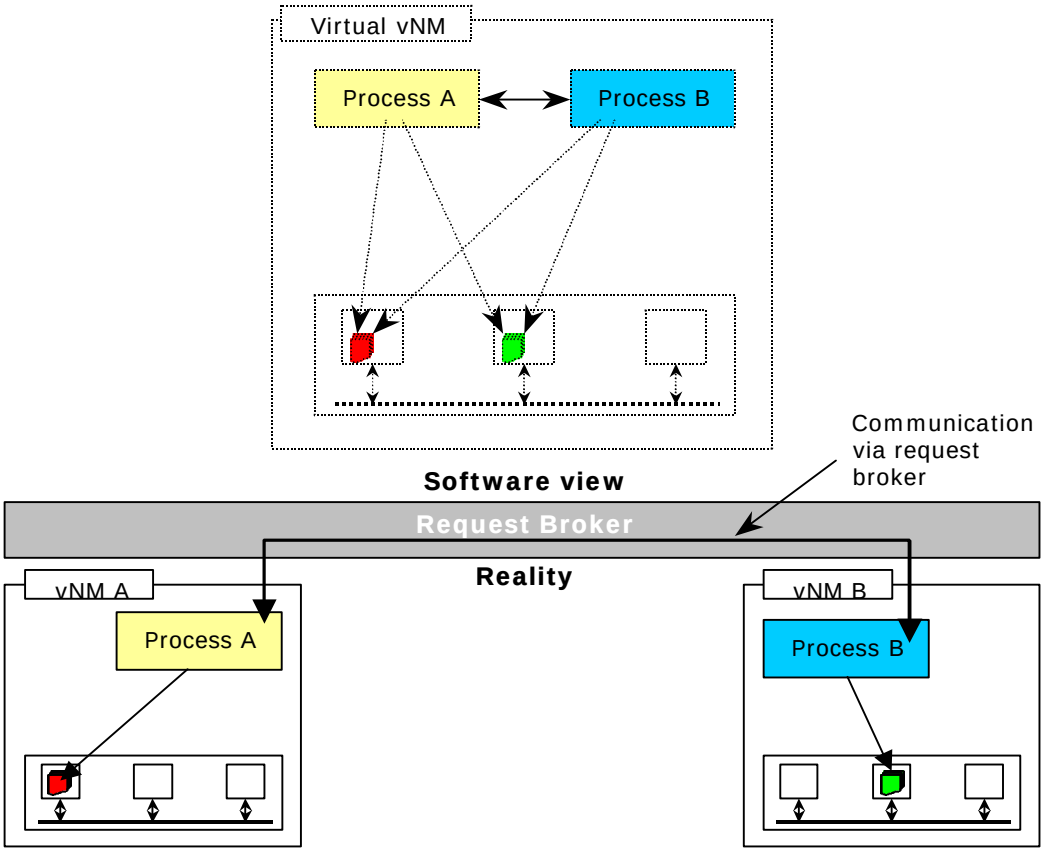

Figure 10. Request Broker providing location transparency

In reality, processes A and B exist within two complete separate vNMs, as do the resources they share. However, the infrastructure attempts to create the illusion that they exist within the same vNM. It also ensures that any required resources appear to each process as if they were in their local computing machine, thus achieving the location transparency described above.

We have now examined the emergence of contemporary abstractions and infrastructures for distribution. If we are to compare and contrast them with the Mobile Code abstraction then they must be generically categorised.

2.5 Characterisation of Traditional Distribution Architectures

So far in this chapter, we have discussed the history and emergence of contemporary distribution infrastructures. Although vendor specific (with the exception of CORBA), these infrastructures are competing implementations of the same generic type of distributed system. They share a common heritage and are each instantiations

of the RM-ODP abstraction, which itself can be traced back to RPC. For example, CORBA IDL is directly modelled on RPC.

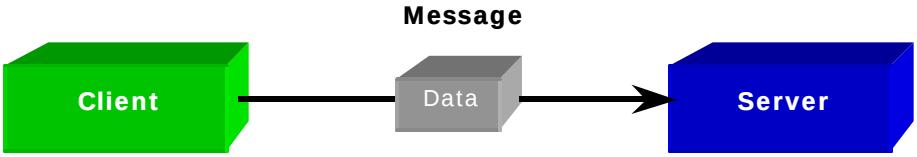

Figure 11. Mobile Data in a Traditional Distributed System

In this thesis, these systems will be characterised as distributed system infrastructures whose fundamental tenet for distribution is one of *location transparency*. They achieve this by allowing distributed systems to interact via an intermediary communications bus. The bus (or request broker) establishes a client/server relationship between the interacting parties, removing the need for each party to be aware of the other's location. The underlying communication mechanism supporting distribution will be characterised as *mobile data*.

2.6 Commentary

We have seen in Chapter 1 that modern day computing abstractions can trace their ancestry back to the original von Neumann architecture. As each abstraction has emerged, bringing with it new facilities and technologies, it has added a new layer to the continually ascending edifice. At their root though, the von Neumann architecture remains, influencing modern day designs even from the past. It is the base abstraction, the underlying model for our computational machines. As each new abstraction is layered onto the others, it must take into account those that preceded it.

When Birrel and Nelson first designed RPC in 1984, their intention was to allow the programmer to access and communicate with processes on remote machines, in the same easy manner in which they were able to access local processes. They wished to make calls to remote processes appear identical to those made locally, thereby making the location of the process transparent to the programmer (and ultimately the user). It should not matter if the process was being executed locally or on a machine on the other side of the world, it would appear exactly the same in both cases.

This phase in the development of distributed systems is pivotal. RPC was directly modelled on IPC, which had been an extremely successful mechanism for enabling

processes to communicate, and so Birrel and Nelson's intentions were not without merit. However, IPC had evolved by extending the abstractions offered by existing programming languages and by taking advantage of local facilities such as memory or file space, each fundamental constituents of the vNM. IPC therefore was a perfect abstraction for communication between processes executing in the same computational machine, i.e. in the same von Neumann machine.

RPC on the other hand attempts to mask any details of location from communicating processes. In effect, blurring the demarcation between separate vNMs to make local and remote calls look identical. The technique required to achieve this is complex; for two processes to communicate, a set of five separate modules is required (see Figure 8). Nonetheless, this technique was successful for the time, and the central tenet of the abstraction, location transparency, became one of the underlying principals for the RM-ODP model, and consequently most contemporary distribution infrastructures. Part of the reason behind the success of RPC is because it is perfectly suited to building client/server software systems. At the time, business software was predominately hosted on centralised mainframe computers, computer networks were predominately LANs or WANs and the number of personal computers was dramatically lower than today. Equally, concurrent programming was slowly becoming a reality and objects were only just gaining momentum. Thus, is it is not difficult to see why the RPC abstraction was employed successfully for the types of software system being constructed at the time. Further, it follows that such a successful technique would be used as the baseline for newer distribution infrastructures such as CORBA. These new infrastructures take this issue further, creating what in effect is a virtual vNM, where the illusion is created that all components in the system exist within the same computational machine (see Figure 10).

Since that time, the nature of the environment in which these distributed systems exist has been changing. Fuelled by the Microsoft vision of a PC on every desk, personal computers have taken over many of the responsibilities that used to be the domain of the mainframe. The network has also seen a dramatic enlargement with the explosion of the Internet, but has also suffered from quality of service issues. Object-oriented programming has fundamentally changed the way we view software systems, moving

us away from the synchronous single threaded model, to one that includes asynchrony, multi-threading, encapsulation and component reuse. In short, many of the assumptions made in the creation of RPC have now become erroneous. For example, RPC implicitly assumes that the network is 100% reliable, and thus that remote procedures will always be available. Anyone who has used the Internet will attest this as a fallacy.

By 1994, the first strong doubts over the validity of the RPC approach were being raised. In a seminal paper, Waldo *et al* [Waldo94] argue that objects[5] acting in a distributed system are intrinsically different to those in a local system and therefore must be treated very differently. They identify four major problem areas when comparing local and distributed systems (see Table 3).

Problem	Details
Latency	• Can be up to a difference of 4-5 orders of magnitude • Most obvious • Least worrisome
Memory Access	• Unable to use pointers • Because memory is both local and remote, call types have to differ • No possibility of shared memory
Partial Failure	• Is a defining problem of distributed computing • Not possible in local computing
Concurrency	• Adds significant overhead to programming model • No programmer control of method invocation order

Table 3. Problems of a Distributed System

In particular, partial failure is identified as an extreme problem for distributed computing. Sloman had earlier expressed the view that:

> "If the programmer is to take advantage of location transparency, this means that the behaviour must be the same in both cases [local and remote]. This can be costly and difficult to achieve, especially in the face of failures" [Sloman87]

[5] This applies equally to processes and procedures, etc

In addition, even before the Waldo paper, Nelson himself had suggested that:

> "If the aim is to provide location transparency then we must aim to provide the same behaviour as in the case of a failure in a local procedure call, although this can be costly." [Nelson81]

In Figure 12, we see a software system built with the RM-ODP abstraction distributed over three vNMs. Each component has access to certain resources, but of course, there is no way for the component to tell if the resource is local (within the same vNM) or remote. In the case of remote resources, the request broker is required to support the illusion that they are indeed local, by providing the relevant connections "behind the scenes". This is depicted by the lines flowing through the plane of transparency. From this very simple hypothetical system, it is evident just how many lines cross the boundaries of vNMs. At each crossing, the system is subject to the types of problem identified in Table 3.

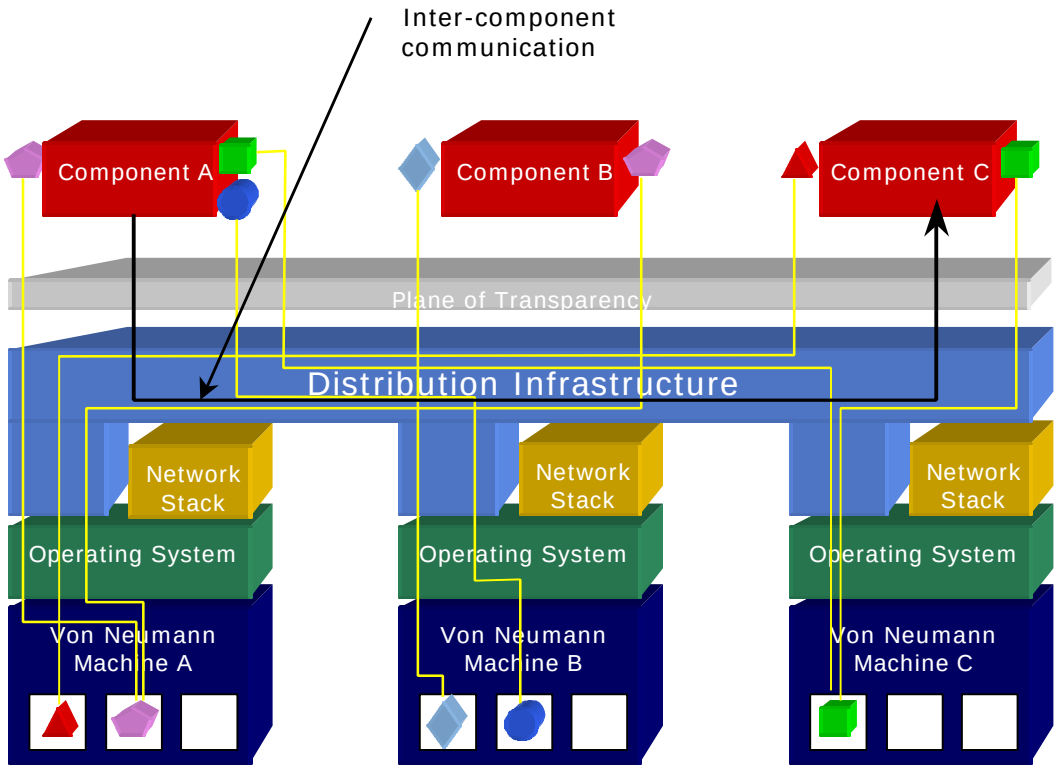

Figure 12. Back flips required by ORB to ensure location transparency

The central thesis of the Waldo paper is that local and remote computing are just plain different, and should be treated as such. They argue that distributed systems should be built with the premise that there are two distinct types of objects: local objects and

remote objects. Although Waldo *et al* identify the key differences between local and distributed computing, their discussion of why these make distributed computing different are pragmatic. The differences are eloquently stated, but there is no reason given for exactly *why* these differences are evident, just that they are – and that the two types of computing should be treated differently. In this part of the thesis, we go further and present an argument as to the cause of these differences.

We have seen that IPC was an ideal abstraction for interacting processes within the same vNM. Its success was built on the fundamental elements of a vNM, i.e. a single memory (that could be shared), a single CPU and local files (I/O). RPC attempts to take this effective abstraction and make it apply to many vNMs, by making location transparent. This is similar to many contemporary distribution infrastructures. Indeed, the stated goal of the Millennium experiment undertaken at Microsoft Research is:

> "… to eliminate completely the distinction between distributed and local computing … by raising the level of abstraction so that programmers are not even aware of distribution" [MSR98]

However, practice has shown that this approach is fraught with difficulties [Waldo94], and the discontinuation of this project serves as a clear indication.

Certainly then, there are two diametric views as to how we may build reliable distributed systems.

1] Use an abstraction that completely removes any knowledge of location
2] Use an abstraction that views remote and local objects as completely different

This thesis supports the assertions of Waldo *et al*, i.e. that we should treat local and remote objects differently. However, we go further and argue that the fundamental reason that RPC, and thus contemporary distributed systems based on the RM-ODP abstraction, suffer from the problem mentioned above is because of the underlying abstraction they embody. The RPC abstraction pays little regard to the supporting layers beneath it; rather it attempts to strike out on a new course of its own and is unsuitable for the underlying hardware substrate. Instead of continuing the long line of abstractions that have served so well, RPC attempts to impose an abstraction that is perfect for one vNM onto many. It pays little attention to the underlying hardware

abstraction, which as we have seen is the vNM. RPC has broken the abstraction tower, and it is this fact that causes the acute problems associated with distributed systems that Waldo et al have identified. While the RPC approach has been, and continues to be, useful under certain circumstances, it no longer supports the type of distributed system we wish to build in today's networks with current software engineering techniques and technologies.

2.7 Concluding Remarks

> "It can be argued that RPCs should not be entirely transparent as their semantics and performance differ from those of local procedure calls." [Colouris94]

> "… a number of distributed systems have attempted to paper over the distinction between local and remote objects [and failed]. These failures have been masked in the past by the small size of the systems." [Waldo94]

As computers have become more prevalent, and the resources they represent the lifeblood of business, we have developed methods for connecting computers and enabling them to communicate with each other. Once communication was achieved it was only natural that we pursue techniques for building software systems that span multiple hosts, allowing us to harness the additional power and multiple resources made available.

In this chapter, we have examined the emergence of distribution, and traced the evolution of abstractions used to build networks. Networks are an essential constituent of distribution, they enable communication between computers. They are the substrate over which distributed systems can be built. Next, we have examined the evolution of abstractions used in contemporary distributed systems. We have seen how RPC attempts to extend the extremely successful IPC abstraction, ultimately leading to the *location transparency* abstraction, embodied in many contemporary distributed infrastructures. In effect, these infrastructures attempt to create a virtual von Neumann machine. This approach has been shown to be unreliable.

The central thesis in this chapter is that by attempting to create the illusion that all components exist within the same machine, location transparency is breaking the

layers of abstractions upon which computing has been built since the dawn of computing. The abstraction is unsuitable for the underlying computational machine upon which it must execute. We need new techniques and abstractions for distributed computing that do not break our layers of abstraction, rather they continue to appreciate what has preceded them, and are suited to the underlying computational machine. In the next chapter, we review mobile code, a new technology that promises to fulfil these requirements.

3 Mobility

3.1 Introduction

Code mobility is not a completely new idea. There have been several widely used and successful mechanisms for moving code around a network previously employed, perhaps the best known being the PostScript language [Adobe85] that is used to control printers.

Recently though, mobility has been examined from a different perspective, and has become a burgeoning topic for discussion in mainstream distributed systems research. Mobility currently boasts a flourishing research community dedicated to investigating the potential of this new paradigm [Mobility99]. So far in this thesis, we have built an argument against using location transparency, the abstraction embodied in contemporary distributed systems. We have identified the need for new abstractions for distribution, which are entirely suited to the underlying computational machine, and are able to distinguish between local and remote resources.

In this chapter, we conclude Part I of the thesis, the philosophical argument concerning the abstractions employed in building distributed systems. We begin by reviewing mobile code abstractions and examining the differences between systems built with these abstractions and contemporary distributed systems. Finally, we discuss what makes mobile code systems different, and why the abstractions they embody are more suited to distribution than location transparency.

3.2 A Brief History of Code Mobility

There have been previous examples of code mobility. One of the earliest being remote batch job submission [Boggs73]. Employed at the time of hugely expensive central mainframes, batch job submission allowed users to submit code for execution on the server. Although working at a very basic level, this technique was a mainstay of computing life when both processor time and core resources were scarce. In effect, batch job submission allowed computation to be moved from one location to another to take advantage of local resources, although the movement required manual intervention by the user.

This basic concept was the seed for further research, and out of it grew projects such as Accent [Rashid81] and RIG [Rashid86], which culminated in the MACH [Accetta86] operating system. These were experiments in building distributed operating systems, which attempted to present the same abstractions regardless of the underlying hardware substrate. Latterly, this work has been embodied in migratory systems such as Locus [Thiel91] and Cool [Lea93], which support process and object migration respectively. Both systems provide mobility at the operating system level, and therefore any migration is transparent to the user and system programmer. As argued in Chapter 2 though, complete transparency can be counter-productive. Certainly, the designers of Emerald [Jul88] concur, as they offer the programmer explicit control over migration, as well as automatic migration.

Thus far, the techniques described have been positioned at the operating system level and are particularly useful when dealing with small scale distributed systems. They do not tend to be suitable for large-scale networks and systems, particularly those of the scale of the Internet, and have mainly been used for techniques such as load balancing [Picco98]. Although process migration never took off as a commercial reality, the research was widely regarded as successful [Milojicic99].

The notion of mobile computation at a higher level of abstraction was first suggested in “Objectworld” [Tsichritzis85], a hypothetical computing environment geared towards information dissemination in which all objects could be mobile. This, and the ideas embodied in migratory systems have spawned a new field of research that is investigating similar solutions but on a much larger scale and at a higher level of abstraction. This field has many names, amongst them mobile code systems, mobile object systems, active networks and mobile agents. For the remainder of this thesis, we use the terms interchangeably unless explicitly stated otherwise. Unfortunately, there is still no consensus among the mobility research community as to what exactly each term refers to, or a standard definition for each to which everyone subscribes. Therefore, in this thesis we define a mobile agent as:

> “a software agent that is able to autonomously migrate from one host to another in a computer network.” [Papaioannou99]

The notion of a mobile agent was first established in 1994 with the release of a white paper by White [White94] that described a computational environment known as "Telescript" [White96]. In this environment, executing programs were able to transport themselves from one node to another in a computer network, in order to interact locally with resources at those nodes. Telescript was never a commercial success, but it did generate a lot of academic interest.

Since that time, this field has exploded in popularity, with a plethora of new frameworks and infrastructures appearing almost continually [MAL99]. This profusion of experimental frameworks is reminiscent of the explosion of new programming languages in the early days of computing (see Chapter 1) and is indicative of a new and immature research field. Although we review some of the more popular mobile code systems in the next chapter, to fully understand this new paradigm we must first examine the differences between contemporary and mobile code based distributed systems.

3.3 The Differences

In Chapter 2, we saw that the central tenet and abstraction of contemporary distributed systems is *location transparency*, with inter-component communication being achieved via an intermediary communications broker. For both the programmer and the system components, this abstraction provides no notion of location. Instead, the *distribution infrastructure* enforces a "plane of transparency" in an attempt to create a virtual computational machine above the network layer. The abstraction hides any details of the underlying hardware, and attempts to create the illusion that every component of a distributed system exists within the same computational machine. Unfortunately, this approach is subject to the many problems identified by Waldo *et al* (see Section 2.6). This thesis argues that the location transparency abstraction is fundamentally flawed, as it breaks the Tower of Abstractions by attempting to impose an unsuitable abstraction on the underlying computational substrate.

Distributed systems built around the tenet of mobile code are quite different. Instead of masking the physical location of a component, mobile code infrastructures make it evident. These systems embody a completely different abstraction. Each node in the network has an *Executing Environment* (EE) through which components are able to

access the facilities of the network layer. These facilities can then be used to communicate with other remote components as normal. However, if components require access to a resource that is not located at their current host, or wish to interact locally with another component, they are able to migrate to the new host. In Figure 13, we see examples of the mobile code paradigm. Component A is in communication with Component B, both of which have references to local resources. However, in contrast to contemporary distributed systems, A requires explicit knowledge of the location of B so that they may communicate. There is no request broker to mediate the communication. Component C is separate, and demonstrates the mobility aspect of this approach. Instead of communicating with a data source across the network, C is able to migrate to the data source's host, and interact with it locally. In a contemporary system, C would not even be aware that the data source resided on a different host.

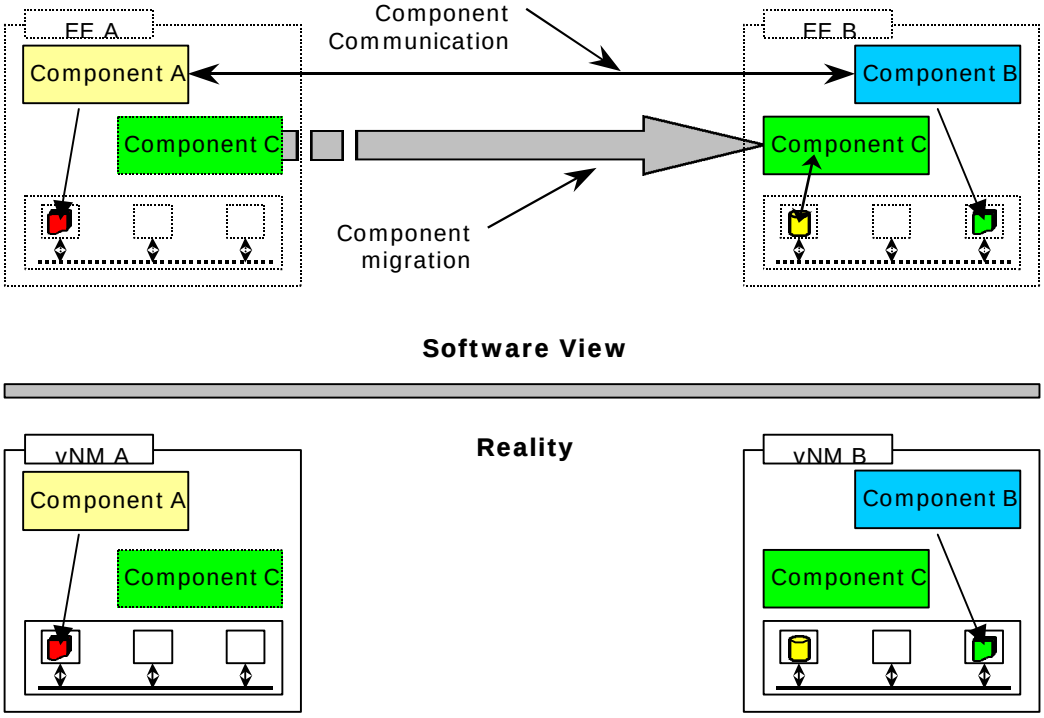

Figure 13. Communcation across the network, and mobile agent migration.

The major differences between mobile and contemporary distributed systems are well described by Picco [Picco98] and are summarized here:

- **Code mobility is geared for Internet-scale systems –** systems such as Emerald and Locus were designed with small-scale networks in mind. Thus, they assume high

bandwidth, reliable networks, small latency, trust, and homogeneity. Mobile agents on the other hand are built with the opposite criteria in mind.

- **Programming is location aware –** mobile agent systems provide an abstraction in which the notion of location is available to the programmer and the constituent components of the system.
- **Mobility is a choice –** migration is controlled by the programmer or at runtime by the agent, instead of being triggered transparently by the system.
- **Load balancing is not the driving force** - process and object migration operating systems were primarily designed to assist with resource and load balancing. Mobile agents are used to design systems supporting flexibility, autonomy and disconnected operation.

Mobile code is a powerful programming abstraction offering many possibilities. To fully appreciate and employ successfully, it is important to understand all the nuances of the different architectural abstractions afforded to the system designer. In the following sections, we describe the different flavours of the mobile code paradigm.

3.4 Mobile Code Design Abstractions

To discuss differences in design abstraction we require a context in which to examine each abstraction. Further, we must define common concepts that may be used to perform our analysis. In the following examples, *Components* are the constituent parts of a software system. They execute within an execution environment at a particular *Host*. Components may contain *Logic*, an encapsulation of the knowledge required to perform a certain *Task*. Completion of this task may also require access to a *Resource*. Components may interact with each other via *Message* passing, in which each message may contain pure data, logic or both. In addition, components are able to migrate to a new host if they so desire. Examples of each abstraction are shown in Figure 14.

3.4.1 Remote Computation

In remote computation, components in the system are static, whereas logic can be mobile. For example, component A, at Host H_A, contains the required logic L to perform a particular task T, but does not have access to the required resources R to complete the task. R can be found at H_B, so A forwards the logic to component B, which also resides at H_B. B then executes the logic before returning the result to A. This is how the aforementioned remote batch entries [Boggs73] work.

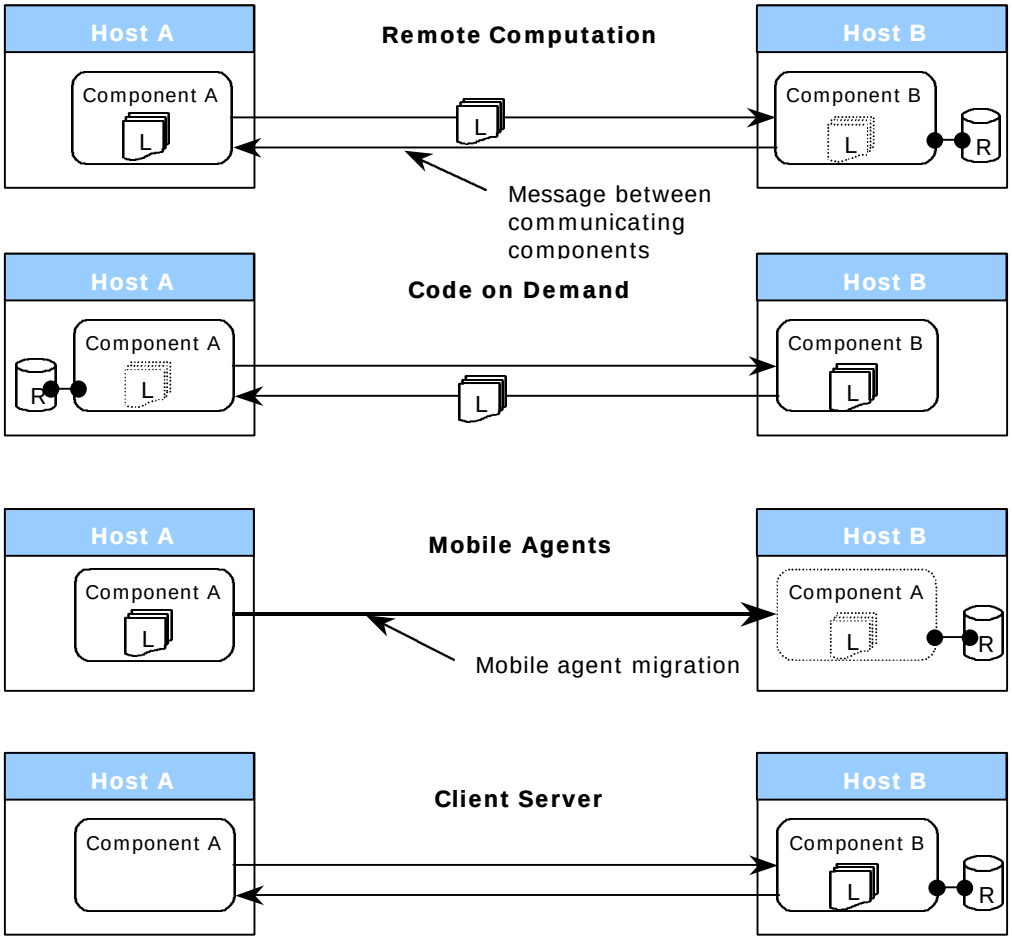

Figure 14. Examples of the different mobile code abstractions.

3.4.2 Code on Demand

In Code on Demand, component A already has access to resource R. However, A (or any other components at Host A) has no idea of the logic required to perform task T. Thus, A sends a request to B for it to forward the logic L. Upon receipt, A is then able to perform T. An example of this abstraction is a Java applet, in which a piece of code is downloaded from a web server by a web browser and then executed.

3.4.3 Mobile Agents

With the mobile agent paradigm, component A already has the logic L required to perform task T, but again does not have access to resource R. This resource can be found at H_B. This time however, instead of forwarding/requesting L to/from another component, component A itself is able to migrate to the new host and interact locally

with R to perform T. This method is quite different to the previous two examples, in this instance an entire component is migrating, along with its associated data and logic. This is potentially the most interesting example of all the mobile code abstractions. There are currently no contemporary examples of this approach, but we examine its capabilities in the next section.

3.4.4 Client/Server

Client/Server is a well known architectural abstraction that has been employed since the first computers began to communicate. In this example, B has the logic L to carry out Task T, and has access to resource R. Component A has none of these, and is unable to transport itself. Therefore, for A to obtain the result of T, it must resort to sending a request to B, prompting B to carry out Task T. The result is then communicated back to A when completed.

3.4.5 Subtleties of the Mobile Agent abstraction

Although all of the mobile code abstractions are ostensibly similar, there are some fundamental differences, which have substantial implications for which particular abstraction to employ. In this section, we highlight one of the key issues that differentiate the abstractions, multi-hop mobility. Multi-hop mobility refers to the ability of a mobile agent to migrate to more than one host, taking action at successive hosts in order to fulfill some goals. The destination of the next host may only be determined at the present host, and does not have to be known at the outset of the journey. In contrast, the other mobile code abstractions are utilized at best as mobile messengers, that do not continue to further hosts once they have performed their tasks, or at worst as techniques for shipping code around a network. For example, let us hypothesize a situation where a BookAgent has queried all StoreFrontAgents and is unable to fulfil its Order. It then has to contact the WarehouseAgent to ask whether a copy can be allocated from there, or when the next copy will arrive. In a contemporary client/server architecture, this would require many calls to remote processes before the task had been complete. Each time a call is made across the network the system runs the risk of the Waldo problems. On the other hand, a mobile agent is able to migrate from host to host, and interact with the StoreFrontAgents locally, before finally arriving at the host of the WarehouseAgent. Once there, it can

begin a new dialogue with the WarehouseAgent to establish when the required book will become available. This scenario is depicted in Figure 15 below.

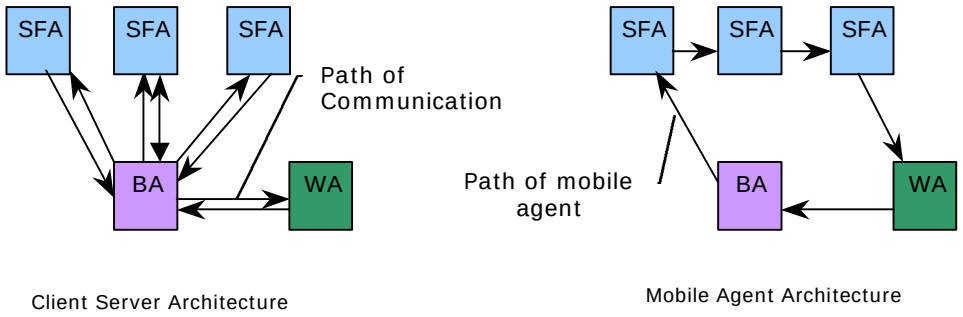

Figure 15. Network routing of Client/Server and Mobile Agent architectures

From these diagrams, it is evident that a mobile agent architecture involves less recourse to network communication than a client/server architecture in this particular scenario. In addition, each time the mobile agent is using the network it is to transport itself, not make a remote call to a component on another machine. If we imagine that each interaction entailed more than a simple request/reply dialogue then the client/server diagram would quickly become littered with communication arrows, whilst the mobile agent one would remain identical. The ability to move the computation to the data source and continue locally is one of the biggest advantages of mobile agents.

3.5 Characterisation of Mobile Agent Systems

Although we have examined several abstractions that are part of the mobile code family, the one with the greatest potential is undoubtedly the mobile agent abstraction. In this thesis, mobile agent systems will be characterised as enabling distributed systems by supporting *local interaction* and mobile *logic and data*.

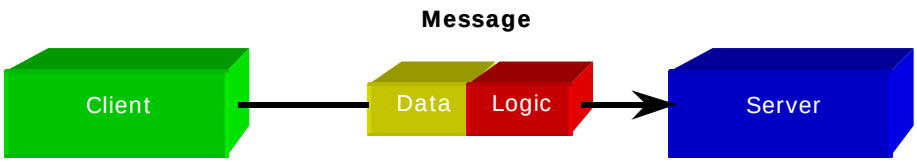

Figure 16. Mobile logic *and* data in the Mobile Agent Abstraction

This is very different to the characterisation in Section 2.5 of the messaging in a distributed system built with the location transparency abstraction.

3.6 Commentary

In Chapter 1, we traced the evolution of computing from the early work of von Neumann through to the present day. We followed the emergence of computing abstractions, and saw how those we employ have been gradually layered upon each other, forming a continually ascending tower of abstractions, whilst retaining as their underlying computational model and base abstraction the von Neumann machine.

In Chapter 2, we examined the emergence of distribution. We saw how RPC attempts to extend the successful abstraction of IPC onto many computational machines by promoting *location transparency*, an abstraction that would manifest itself in distributed systems built around the tenets of RM-ODP. Ultimately, distributed systems built with this abstraction suffer from several major problems (see Table 3). We have argued and demonstrated that this is due to the location transparency abstraction breaking the Tower of Abstractions that has been built to enable and support computing. In short, we argue that location transparency is an unsuitable abstraction for distribution for the underlying computational model.

In this chapter, we have reviewed a new paradigm, with new abstractions, that potentially fulfils the requirements for a distribution abstraction put forward earlier in Chapter 2. Our requirements may be summarised as follows.

A distribution abstraction:

- that remains faithful to the underlying von Neumann machine
- that does not break the tower of abstractions
- that is able to differentiate between local and remote components

It is precisely these requirements that the mobile code paradigm fulfils. As we have seen, its central tenet is one of *local interaction*. Components in a distributed system that wish to communicate are able to transport themselves across the network so they may interact locally at the same host. In addition, components are also able to communicate by exchanging messages across the network.

In each case, the core abstraction remains faithful to the underlying von Neumann machine and the Tower of Abstractions. Instead of attempting to remove location from the abstraction, and build a virtual computational machine, mobile code makes

location evident. It is a central aspect of the abstraction, and enables designers to make a judgement on how components might communicate. Indeed, the *execution environment* of a mobile code system may itself be viewed as an additional virtual computational machine being added to the Tower, but it remains consistent with the underlying base abstraction. By ensuring that any protracted communication is done locally, components are able to return to the successes of IPC by taking advantage of the core facilities of the vNM, e.g. shared memory and files. Instead of attempting to achieve distribution by imposing an unsuitable abstraction across many machines, mobile code simply layers a new abstraction upon the existing tower; a time honoured route to success. In fact, we argue that local interaction as embodied in mobile code systems should be viewed as a successful adaptation of IPC to distribution.

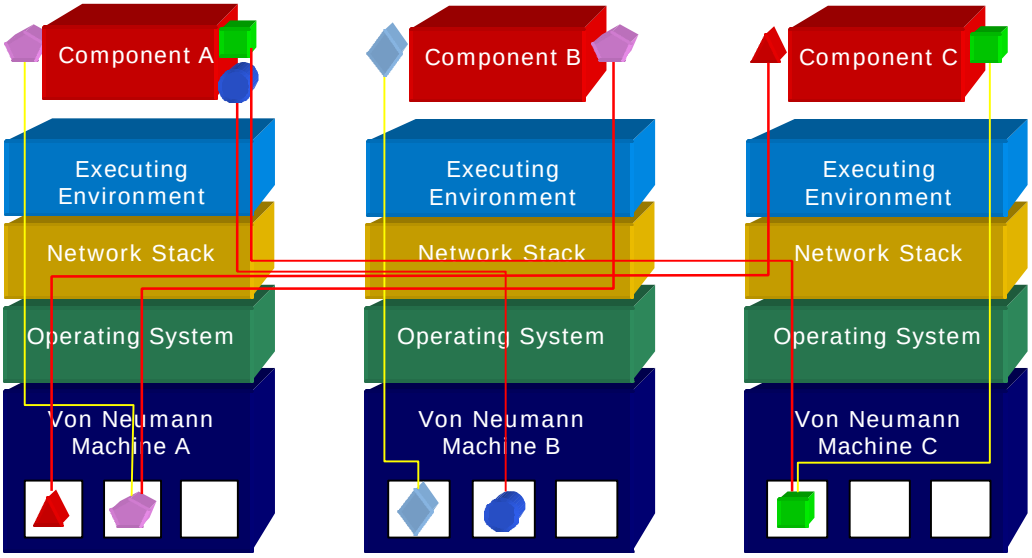

Figure 17. A distributed system built with mobile code

In Figure 17, we see the same hypothetical distributed system that was first encountered in Chapter 2. However, this time the system has been built with the mobile code paradigm. Again, each process has access to certain resources, but this time there is clear knowledge of the location of each resource, i.e. in which vNM it resides. Local references are shown in yellow, whilst remote references are shown in red. Knowledge of the location of a resource, allows each component to make a judgement about the type of reference it holds to that resource. In comparison to the RM-ODP version of this model, there is no illusion being created by the “plane of transparency”. While network references may still suffer from the problems depicted in Table 3, the components themselves are aware that this is a potential problem. In

addition, if a component decides it would be beneficial to be located at the same host as a resource it may migrate to take advantage of local interaction. For example, in the case of component C, when it has finished interacting with the green cube, it may migrate to vNM A to communicate locally with the red triangle.

The major conceptual difference between the two distribution abstractions is clear, *location*. With location transparency, location is removed from the abstraction and a virtual computational machine is created which attempts to create the illusion that all components in a system reside within the same address space. The illusion, however, can be shattered by any number of problems associated with trying to create a rock solid abstraction across the network.

In contrast, local interaction makes location evident and components are able to make a judgement themselves about how to communicate with other components. It is this fundamental difference that the author believes is vitally important. In Chapter 1, we discussed how abstraction is an immensely powerful tool. It allows us to manage the complexity of a situation, and to rationalise about it by removing those details we consider inessential. It is the author's belief that when it comes to distribution, location is a vital piece of information. We are no longer attempting to build distributed systems in networks in which location can be papered over, in which the size of the system can mask the fallacies in the paradigm. We are now building large systems in which the network is unreliable, in which the topology of the network or availability of resources may change rapidly. In such an environment, information about location becomes essential. If we examine perhaps the most successful distributed system of all time, the Internet, we see that location is central to its success. The URL [Berners-Lee92b] abstraction is purely a reference to location, but has been fundamental to the evolution and success of the web. We must learn from these lessons.

3.7 Concluding Remarks

> "Keep design as simple as possible, but no simpler" [Einstein39]
>
> "A designer knows that he has arrived at perfection not when there is no longer anything to add, but when there is no longer anything to take away" [Antoine de Saint-Exupery]

We have seen throughout Part I of this thesis how important abstraction is to computing. It is the central essence of an idea or design. Abstractions allow us to remove the details and focus on the essence of a situation. Any specific example of a technology is merely an instantiation of the abstraction. The majority of the history and evolution of computing has been concentrated on the development of new abstractions. Our current abstractions for distribution have proved limiting and unreliable. We require new abstractions to support distributed computing on a hitherto unforeseen scale. Mobile Code systems are one such solution.

In Part I of this thesis we have built a philosophical argument concerning the abstractions used in building distributed systems. It is our belief that the location transparency abstraction, as embodied in the RM-ODP model, is fundamentally unsuited to the underlying hardware substrate. Instead of attempting to utilise the strengths of preceding abstractions, location transparency enforces a "plane of transparency" whose purpose is to create the illusion of co-location and to mask any details of distribution from components in the system. The abstraction views location as a detail that can be removed.

Local interaction on the other hand remains faithful to the core abstraction, and makes use of the core facilities embodied in IPC. Instead of masking location, it makes it evident. Communicating components are aware if they are local or remote to each other, and are able to make a judgement about how to communicate. By utilising the strengths of the von Neumann machine *and* the network, the local interaction abstraction allows us to build distributed systems that do not suffer from the Waldo [Waldo94] problems.

The central argument of Part I is that local interaction should be the abstraction of choice for building distributed systems. In hindsight, we should view location transparency as an evolutionary blip, a wrong fork in the road. If we are to build successful distributed systems in the myriad of new networks, we must be bold and admit our mistakes of the past.

Part II

Using and Evaluating

4 Mobility in the Real World

4.1 Introduction

Mobile Code is a new and generally untested paradigm for building distributed systems. Although garnering many plaudits and continually increasing in popularity, the technology and research field remain relatively immature [Picco98]. To date, most research has focused on the creation of mobile code frameworks, and as yet there is no consensus on a conceptual framework with which to compare results. Further, there is no clear understanding of the new abstractions offered by this paradigm. Part I of this thesis aspires to address the conceptual deficiencies of the research field by offering a philosophical argument and critique of mobility.

In Part II we begin our study of mobility in the real world. In later sections of the chapter, we will see that there are many advantages claimed for mobile code systems. Unfortunately, these claims remain qualitative and subjective in their nature. The dearth of quantitative results, however, means it has not yet been possible to properly evaluate the potential of either the technology or the paradigm. In the last year a trickle of results is beginning to validate some of the claims [Papastavrou99] [Picco98b], and these results are certainly important in establishing the credibility of mobile code systems. Nonetheless, it is the author's belief that these types of improvement are optimisations, or incremental improvements. The true benefit of the paradigm is in the type of software architecture that can be built. In support of our arguments presented in Part I, in Part II we provide an insight into how well mobile code architectures respond to real world pressures.

4.2 Research Motivation

In Part I, *Understanding*, we have presented an argument built around a philosophical understanding and critique of the abstractions used to build distributed software systems. The central thesis is that contemporary distributed systems built with the location transparency abstraction are fundamentally flawed and that we require new abstractions for distribution. Our proposal is that a new abstraction, local interaction, is better suited to the underlying hardware substrate upon which distributed systems are built. To demonstrate this we have traced the emergence and evolution of

computing, and the abstractions that exist in this field, beginning with the early pioneering work of John von Neumann. We believe that Part I contributes to raising the level of conceptual understanding surrounding the mobile code paradigm, especially when examined in the wider context of the different abstractions embodied by distributed systems.

Although we believe the essence of any technology is the core abstraction it embodies, we understand that pure academic reasoning is never sufficient to make a valid judgement about a new technique or technology. What is required is first hand experience. Therefore, in addition to our philosophical argument, we aim to support these arguments by investigating the application of mobile code in the real world. We wish to demonstrate the feasibility of actually building distributed systems with this technology. Certainly, the arguments presented in Part I are extensive, and a full experimental investigation is beyond the scope and timescale of a PhD[6]. Instead, we must shorten our horizons and take the first steps along the long path of validation. Part II is therefore a report on our experiences of *Using and Evaluating* mobile code in the real world.

As we have seen, the technology base in the field of mobile code remains immature. Whilst the plethora of new frameworks continues to increase, the amount of real distributed systems built with this technology remains low [Milojicic99]. Although abstractions are the central essence of a paradigm, the technological instantiation of that abstraction must successfully embody it. To support our argument of Part I, we must prove that mobile code can be used to build real world systems. Thus, our research motivation is to investigate and use mobile code, as it would be in the real world, and to analyse the issues involved and the lessons that can be learnt.

In Chapter 3, we described the choice of design abstractions available to the system architect who wishes to employ mobile code. These were Remote Computation, Code on Demand, Mobile Agents and Client/Server. Since many examples of Code on Demand currently exist [Hopson96], and Client/Server architectures are an extremely well known approach, we feel these abstractions are of less interest to this study. Therefore, the implementation described in this thesis will encompass prototype

[6] Indeed, an entire academic career could be pursued with these arguments!

systems of the Remote Computation and Mobile Agent abstractions. We have gained an understanding of each abstraction, and have been able to compare the two. For ease of use, and because of the conceptual abstraction they support, from herein we refer to the former as the Mobile Object system, and the latter as the Mobile Agent system.

4.2.1 Research Objectives

As the software systems that underpin industry have become ever more complex and interlinked, the inherent flexibility of the underlying software designs has been compromised. On the small scale and under the right circumstances software systems can be extremely responsive, flexible and easy to change, for example the existence of the requisite skills. Therefore, matching a change in business practice should not be a problem. However, when examined in the large this is not the case. As observed by Cox:

> "There was a time when the virtue of software over physical media like paper and pencil was in its very responsiveness … Although this may be to some extent true for small projects (program building), it is not (and has never been) true for ambitious undertakings (system building). In fact, software systems are usually the least responsive element in many organisations today. The organisation as a whole is able to adapt more fluidly than the software upon which it has grown dependent." [Cox87]

Recent experience has shown that attempts to create large scale supporting infrastructures have resulted in complex monolithic systems that are the least flexible element within an enterprise [Barber98]. Most companies require a change in their software at some point, and so software change is one of the most important issues currently facing the software industry [Booch94]. A software system will have a limited lifetime if it cannot be altered to accommodate a change in the business process it is intended to support.

This issue is well known to the software engineering community, and in this thesis we refer to it as System Agility. There already exists a substantial body of work relating to the issue of system agility, e.g. [ICSE'99], and the full variety of issues is vast. We

cannot hope to consider them all in our experimental study, so we initially select two broad but vitally important factors on which to focus:

1] How easy the system is to understand
2] How easy it is to modify

These are still broad issues, with many factors contributing to each, so we refine our focus even further. To represent each facet, we have selected the specific issues of Semantic Alignment (SA) and Component Coupling (CC). System integration and agility has been one of the main issues of research at the MSI Research Institute for nearly a decade, and therefore SA and CC augment the research undertaken by other members of the institute [MSI99] [Coutts98b]. In the next sections, we briefly review both concepts.

4.2.2 Semantic Alignment

The ability to communicate ideas clearly and effectively was a concern for the human race even before written records began [Pinker95]. Whenever two people talk, they have only an approximate understanding of each other. When they speak the same language, share intellectual assumptions, and have common backgrounds and training, the alignment may be closer. As these factors diverge, there is an increasing need to put effort into constant calibration and readjustment of interpretations, since ordinary language freezes meanings into words and phrases, which then can be "misinterpreted" (or at least differently interpreted). Clear communication requires a shared understanding of the meaning of terms; and this understanding is known as Semantic Alignment [Clark96]. While this term has its roots in linguistics, it is also applied to software engineering. For example, if information is being shared between two company databases that have a table for "employee," they are apparently in alignment. However, if one was created for facilities planning and the other for tax accounting, they may not agree on the status of part-time, off-site, on-site contract, or other such “employees.”

A software system is invariably built to support a business process. Therefore, in the context of system agility we define Semantic Alignment as:

> "Semantic Alignment refers to how successfully a software system embodies the real world business process is it intended to support, i.e. how well the software models the real world."

For example, if in the real world a business process contained the concepts of Apples, Oranges, Potatoes and Tomatoes, but the software model only contained the concept of Food, then this system would not be as successfully aligned as a system that contained the concepts of Fruit and Vegetables.

4.2.3 Component Coupling

Component Coupling was first defined in the 1970's by Constantine and Yourdon [Yourdon79]. It is a technique for measuring the inherent maintainability and adaptability of a software system, both of which are important issues that directly affect the overall agility of a software system. In short, component coupling measures the dependencies between two software components, i.e. how many times a component depends on the functionality of another object to perform its role. It is considered desirable to limit the number of inter-object dependencies in a system, since this not only affords greater flexibility to the designer during construction, but also ensures the system remains easy to change in the future. Therefore, the objective of a designer is to limit these dependencies, thus making the system "loosely" coupled, so that objects can be interchanged or updated more easily.

The benefits of loose coupling are potentially huge and include [Clark96]:

- Higher component reuse
- Higher productivity
- More robust systems, since failures cascade less
- Fewer bugs, as increased reuse means what is reused needs less testing.
- Complex systems become easier to alter, due to higher component reuse.
- Easier component enhancement, modification and bug fixing

Coupling is usually associated with cohesion [Yourdon79], which is a measure of the inter-relationships between functions of a single component. Since our study is to examine distributed systems, we feel cohesion is of secondary interest in this case. Therefore, we concentrate on how component coupling is affected by the choice of mobile code abstraction, and define coupling as:

> "A measure of the external dependencies of a component defined by the number of links that component has to other components within a software system."

4.3 Research Statement

The main aims of the research undertaken in Part II can be summarized as follows:

1] To demonstrate mobile code can be used to build real world software systems

We describe the construction of two prototype mobile code systems. They are used to investigate the effectiveness of the two selected abstractions in building real world distributed systems. To simulate real world software problems the prototypes are constructed to support the Sales Order Process of a UK manufacturing enterprise. This real world business process was identified during an industrial case study (for further details see Chapter 5).

2] To learn how mobile code responds to real software problems

Merely building proof-of-concept systems is a worthy exercise, but systems in the real world very rarely fulfil all the requirements of a business for any length of time. In the majority of cases, the capabilities of a software system will need to be later upgraded to support new functions or features, usually due to a change in a business process. In addition to their creation, we aim to evaluate the prototypes with respect to the issues of understanding and changing a system that currently confront system designers. To achieve this we have extracted several "Scenarios for Change" from data collected during our case study, which will be used to evaluate how well the prototypes respond to change. Three common and related problems facing the software industry today have been identified as candidates for examination. These are:

- **System agility** – how well a system responds to change
- **Semantic alignment** – how well a system embodies the business process it is intended to support
- **Component coupling** - how intermeshed the components of a software system are

From the experiments, we hope to gain an insight into how successful mobile code systems are when subjected to the kinds of pressures prevalent in industry.

However, before proceeding with the construction of the prototype systems, it is important to first examine the technical issues associated with using mobile code. To support our philosophical understanding, we must also appreciate the requirements and consider the limitations of mobile code infrastructures before employing them. For the remainder of this chapter we focus on issues relating to *mobility in the real world*.

4.4 Technical Issues and Enabling Technology

We have seen in Chapter 3 that distributed systems built with mobile code technology usually consist of execution environments that are hosted at different nodes of a network. Mobile agents are able to migrate between these hosts in order to interact locally with static resources and other static agents resident at the hosts. This hosting and migration can be achieved through several different mechanisms, and combinations thereof. In this section, we examine several of the key issues and decisions that must be taken when implementing and using a mobile agent framework.

4.4.1 Strong vs Weak Mobility

The terms strong and weak mobility refer to the method and nature of the mobile agent migration. In strong mobility, the entire computational entity, i.e. its code, data, execution state and program counter migrate to the new host. There are two ways of achieving this, firstly by true migration and secondly by remote cloning. With true migration, the mobile agent is suspended before being transferred in its entirety to the new host. Upon arrival, the agent is restarted and is able to continue its execution at exactly the point at which it was suspended. Remote cloning on the other hand achieves migration by stopping the entity at the first host before creating a copy at the new host. Indeed, some might argue that since computers can only copy and delete [Cox98], both methods are actually the same. Some important examples of mobile agent frameworks that exhibit strong mobility include Agent Tcl [Gray97], Ara [Peine97] and Telescript [White].

Weak mobility on the other hand is only able to migrate the code associated with the entity across the network. Any state or non-constant data that is required by the entity must be packaged up for travel before migration. The onus of this packaging is

placed upon the programmer at design time. Weak mobility is generally easier to achieve technically, especially with programming languages such as Java available, but is burdened by its limitations when complex applications are considered. The programmer must be fully aware of any data that may be required after migration and take care to package it, or it will be lost. The majority of (if not all) mobile agent frameworks based on Java are weakly mobile (see Section 4.6. for examples)

4.4.2 Interpretation vs Compilation

By their very nature, mobile agents are inherently distributed [Clements97]. As such, they must be executable across a variety of platforms and operating systems to achieve their full potential, although in a closed and privately controlled network they may benefit from homogeneity. Their true advantage however, comes from being able to migrate and continue functioning in a heterogeneous network of systems. This advantage is implementation dependent and has greatly influenced the way in which mobile agent systems are created. To enable heterogeneous execution it is usual for these frameworks to be written in some type of script or bytecode that can subsequently be interpreted, usually by a dedicated executing environment. Indeed, the spiralling popularity of Java, combined with its platform independence, has made it the de facto language for mobile agent systems. Interpretation removes the need to recompile an agent at a new host and instead places the onus on merely ensuring an environment exists at the new host that is capable of uniformly executing the agent on arrival. Most examples of this type of system have a server or some type of executing environment in which the mobile agents are executed [Lange98][Gray97]. Interpretation does of course have the previously discussed limitation of execution speed, but this is often seen as a minor trade-off, due to the ease in which portability is achieved.

Compilation is not particularly popular in the field of mobile agents, since it forces the sending machine to be aware of the platform and hardware architecture of the receiver, so that it may choose the appropriate compiler or the appropriate library of native code. As the number of different platforms being supported increases the complexity is wont to spiral out of control. Compilation does however have the advantage of speed of execution. Some examples are [Knabe96] [UCI96].

4.4.3 Resource Management

When a mobile agent migrates to a remote host, any references it has to local resources are likely to become invalid. Before execution can be resumed, all its references must be evaluated and reassigned. This problem can be overcome in a number of ways:

- **Copy** - If the resource can be copied, then the mobile agent can take a copy of the resource with it to the new host.
- **Move** - The mobile agent can take the only copy of the resource along with it.
- **Network reference** - If the resource is static, then the reference can be changed into a network reference that points back over the network to the resource.
- **Reference removal** – If the reference is no longer needed, or cannot be accessed remotely via a network reference, it can be removed.
- **Rebinding of reference** – If another copy or instance of the resource, or a similar resource, is found at the new host, the reference can be rebound to it.

Which tactic to adopt is often determined by the nature of the resource in question, and the programming language being employed. For example, it would be nonsensical to copy or move an entire database to a new host.

4.4.4 Security

Security is one of the most emotive issues raised when discussing mobile agent systems. It is often quoted [Johansen99] as the major reason mobile agent systems have not taken off in the mainstream. There is currently a wealth of research being done on this particular subject [Vigna98]. A brief summary of the most important security issues are describe below in Table 4.

The work described in this thesis is concerned with private networks, in which all the hosts and agents are trusted and their origins known. Thus, the only class of applicable attack is that of a third party eavesdropping on a transmission. This could be overcome by the usual cryptographic techniques employed in such exchanges as email, for example. Therefore, the issues of security are considered external to the scope of this thesis.

Attacked	Type of Attack	Explanation
Host	Host compromised by arriving agent	An incoming agent may try to access and corrupt the host's local files, resources or even try stopping the server in a denial of service attack.
	Host compromised by external third party	Someone who wishes to bring down the host may send a huge number of agents to the host to tie up all the resources, or even crash the host
Agent	Agent is compromised by the new host	If the host is untrusted it may try to access private information, e.g. a credit card number, a password, etc, for later use, or replay.
	Agent is compromised by another agent	During an inter agent conversation the other agent again tries to access private information, or to crash the agent to stop it fulfilling its task
	Agent is compromised by a third party	Since some inter agent comm'n takes places over the network a third party may try to alter exchanged messages for their own benefit, e.g. to recommend their host instead of another, or to reveal content of agent
Network	Network compromised by incoming agent	An incoming agent attempts to flood the network with copies of itself

Table 4. Summary of mobile agent security issues

4.4.5 Communication

Communication among mobile agents in a network can take several different forms. Since there is no guarantee that there is actually another agent at the present node, the most basic inter-agent communication usually begins by using the executing environment to pass messages to another agent. This can be achieved directly, if the agent's identity is known, or can be broadcast to the entire node. Once the presence of the agent is established, communication can then proceed more privately with both agents being involved in a one-to-one dialogue.

Mobile agents are also able to communicate over the network, in a similar way to traditional Internet applications, such as ftp, telnet, etc. Once again, the initial establishment of a dialogue between agents is achieved via the hosting executing environments. Communication with remote mobile agents does have associated problems, caused by the mobility of the agent. Passing messages between two agents requires some type of address, which refers to the receiving agent's location.

Obviously, this can cause problems if the receiver is able to move to a new location, as the address is no longer valid. New techniques for overcoming this particular problem are in the early phases of research and development, but include multicast messaging, where a message is broadcast to the entire network, instead of just to the local node.

At the higher levels of abstraction, communicating mobile agents will usually do so by purely message passing. However, at lower levels of abstraction, for example communicating mobile objects, some sort of remote procedure call mechanism is usually provided, that allows objects to interact in the same manner as contemporary systems.

4.5 Advantages Claimed for Mobile Code Systems

In the previous section, we examined several key technical issues that shape how we may utilise and implement mobile code infrastructures. Simply understanding the technological issues however, will not allow us to make an informed judgement of this new technology. We must also understand what advantages mobility might bestow upon distributed systems built with this new paradigm.

So far, there have been many advantages claimed for mobile agents [Chess97][Lange99]. These claims are usually in the form of qualitative assessments but unfortunately, very few quantitative measures exist to support these claims. However, a summary of some of the more frequently quoted and accepted claims are described in the following sections.

4.5.1 Bandwidth Savings

Distributed systems by their nature are required to communicate over the network. This communication can sometimes be in the form of multiple consecutive interactions between two components, for example, a query client and a database. This type of data querying can result in heavy network traffic. Mobile agents are able to overcome this problem by relocating to the host of the database. Instead of shipping data back and forth across the network, they are able to migrate the required business logic to the data source. Once in situ, they can perform any required queries and process the returned information without saturating the network. After

processing, they are able to continue with their work, transporting merely the result to a new host, if it is in fact needed.

4.5.2 Reducing Latency

Many manufacturing and robotic systems must be controlled in real time. Controlling these systems through a factory wide network can be affected by latency and data timeliness. Mobile agents are able to overcome this problem by migrating to be local to the process and control it in real time, thus bypassing the problems of latency.

4.5.3 Disconnected Operation

As the amount of Internet traffic increases, the response from the telecommunications companies in installing new carrier infrastructure is immense [Kotz99]. Nevertheless, this effort may still not be enough to satisfy the expanding base of users. Moreover, many users will not have access to the high-speed bandwidth available to wealthy corporations. Currently, most home users in the UK still connect via a modem and copper telephone lines. Further, the proliferation of mobile devices, such as palm top computers, which employ wireless networks implies that many users and devices will be extremely limited in the bandwidth available to them. This disparity in quality of connection means that performing tasks that require a continuous connection to the network will be probably not be feasible financially, if not technically.

Mobile agents are a solution to this problem. A particular task can be encapsulated within a mobile agent. The agent is then dispatched to a host that is part of the network backbone, and enjoys massive bandwidth access. Once there, the mobile agent is able to carry out its task in the resource rich environment before returning home. A further advantage of this paradigm is that since the mobile agent is now independent of the device, the device can go offline, or even be switched off, before again connecting later for the agent to return with the results.

4.5.4 Increased Stability

One of the major problems with distributed systems is failure, and the identification of the particular type of failure. Traditional distributed systems are built with the philosophy that the network is permanent, and any failure is unexpected. When it does happen it is very difficult to tell whether the network has failed, the machine that

was hosting the component you were communicating with has died or the component itself has frozen.

One of the underlying philosophies behind mobile agents is that the network is not a permanent resource. By building software with mobile agents, distributed systems can be less dependent on the network, since the underlying tenet is local interaction. Discovering the nature of a failure in a local context is a much easier proposition, and so systems built this way can be more stable. Mobility can also be used to achieve replication for fault tolerance, and support robust distributed systems. If a host is being shut down, or experiencing problems, an agent is able to react to this by migrating to a new host where it can continue with its operations.

4.5.5 Server Flexibility

In contemporary distributed systems, when data is exchanged between communicating hosts, each host owns a copy of the code that is required to package outgoing and interpret incoming messages. As protocols are evolved to better support efficiency and security, the effort required to upgrade protocols becomes immense. By using mobile agents, the protocols can be encapsulated within the agents, and removed from the servers. Thus, if a protocol requires an upgrade the mobile agent population can be upgraded gradually as and when required, instead of the entire server base.

Further, since mobile agents are able to carry around their own code, the distributed system can become more flexible since the mobile agent is not merely limited to the functions a server predefines. It is able to bring along new or improved code and can extend the functionality of the server in which it is executing.

4.5.6 Simplicity of Installed Server Base

An additional advantage of relocating the computational logic and protocols within the mobile agent is that the installed servers become much simpler. Effectively, a server becomes merely an executing environment for hosting mobile agents. As this requires far less functionality pre-engineered into the software from the outset, it can help with preventing legacy. Further capabilities can be added by mobile agents at a later date.

4.5.7 Support distributed computation

Mobile agents are inherently distributed, and as such can be a fundamental enabler for distributed computation. However, they are also heterogeneous, often separated from both hardware and software dependencies by their executing environment. This means they are an ideal technology for integrating disparate legacy systems that have dependencies already.

4.5.8 Commentary

The advantages we have seen described for mobility are certainly exciting. Whilst very few quantitative results exist to verify the claimed advantages, the overall picture painted is one of a completely new paradigm for building distributed systems. Such is the excitement that many research labs have already begun to produce mobile code infrastructures [Lange98] [Concordia]. In later years, this initial group may become known as 1st generation infrastructures.

As the mobile code research field has matured, a few quantitative measures are beginning to be published [Picco98b]. Papastavrou *et al* [Papastavrou99] have shown that using mobile agents to perform your database queries locally can have a dramatic affect on system performance. Johansen has shown that bandwidth usage can indeed be reduced by significant levels by using mobile agents when compared to traditional client/server architectures [Johansen99].

It is the author's belief, however, that the majority of advantages discussed in the previous sections are merely optimisations. Many of these advantages could be achieved with contemporary distributed systems, for example by redesigning communications protocols. The true advantage of this new paradigm is the types of distributed system that can be built: ones that do not suffer from the Waldo problems. In the next section, we review some of the well-known frameworks to see how these new abstractions are manifesting themselves.

4.6 Survey of Mobile Agent Systems

The rapid explosion of interest in this field of research means that there are a large number of new mobile agent frameworks appearing, almost continually. The Mobile Agent list [MAL99] currently numbers the known packages at 64. In this section, we

review some of the better-known frameworks and analyse how they embody the mobile code abstractions discussed in Chapter 3.

4.6.1 Java

Although not marketed as a mobile agent framework, the Java [Gosling96] Development Kit does provide enough native facilities to support weakly mobile code. This should not be a surprise since the original goal of Java's designers was to provide a portable, easy to learn, network aware object-oriented language. To ensure portability, Java was designed to be platform independent. Instead of compiling Java into native instruction codes, it is compiled into an intermediary format known as bytecodes. The bytecodes can then be interpreted on *any* platform that has a suitable java interpreter; the interpreter is known as the Java Virtual Machine (JVM) [Lindholm99]. By having the intermediary bytecode stage, Java is an ideal language for weak code mobility. The most widely known examples of Java's mobile code capabilities are probably applets and servlets [Hopson96], mobile snippets of code that can be transferred over the network in an asynchronous manner. Applets and servlets should not be viewed as mobile agents however, since they are merely single-hop pieces of code that contain no notion of autonomy. They do embody the Remote Computation (RC) and Code on Demand (CoD) design abstractions (see Section 3.4).

Inherent platform independence supported through interpretation has made Java an extremely popular choice among mobile agent framework implementers. One might even argue it is the de facto language. These facilities in conjunction with its security model [Gong99] and object serialisation [Sun98b] make it a particularly useful technology base from which to begin.

4.6.2 D'Agents

Developed at Dartmouth College, D'Agents [Rus97] is one of the new breeds of mobile agent framework. In its first incarnation as Agent Tcl [Gray97], D'Agents employed a Tcl [Ousterhout94] interpreter, extended to support strong mobility. When an agent wishes to migrate to another machine it need only call a single function, `agent_jump`, which triggers the interpreter to package up the complete state of the agent and send it to a destination machine. Strong mobility has always been a design goal of the Dartmouth Group and recently, D'Agents has been updated to be a

multi-language framework and now supports strong mobility in Java. However, this facility has come with a price; in order to support strong mobility in Java the D'Agents team had to modify the JVM, which means that the framework will only work with the specialised JVM. With the current rate of change in the Java world, this means that the D'Agent interpreter can quickly become out of date.

4.6.3 Mole

Mole [Straßer96] was the first mobile agent framework developed in Java, and was initially released in 1995 by the IPVR group of Stuttgart University. Mole supports weak mobility only, a choice the designers justify in [Baumann97]. Interestingly, the Mole group assert that their choice of weak mobility was to avoid the problems of using a modified JVM that quickly became out of date. Their goal was to provide a pervasive framework the worked 'out-of-the-box' with any standard JVM. This is in contrast to the D'Agents group and demonstrates the generally unexplored nature of the research field. Whether strong or weak mobility is the correct methodology remains an open question within the mobility community.

Mole provides the notions of *places*, the executing environment, where *user agents* are able to meet and communicate. They can interact with the underlying operating system resources via *service agents*, which are always stationary. Mole supports a number of communication mechanisms including *badges*, *sessions* and *events*. An ascending hierarchy of increasingly anonymous and wider scope of influence mechanisms, they are fully described in [Baumann97].

4.6.4 Hive

Hive is a distributed agents platform, a decentralized system for building applications by networking local system resources, and taking advantage of mobile code [Minar99]. Its designers, a group at the MIT Media lab, are using it to provide the infrastructure for connecting their many Things That Think [Gershenfeld99] research initiatives. Hive is built using the standard Java features of object serialisation and interpretation used by so many mobile agent frameworks and therefore supports weak mobility.

The Hive architecture consists of the following three abstractions: *cells*, *shadows* and *agents*. A cell is the executing environment in which agents are hosted. Cells also contain shadows, which are placeholders for local resources, for example a display or printer. The designers of Hive have made particular efforts to address the problems of agent description and Hive supports both a syntactic and semantic ontology.

Inter-agent communication in Hive has been achieved by using RMI as the communication mechanism. This allows the methods of Hive agents to be executed remotely. While this approach is simple, and uses built in capabilities of the Java language, it has the disadvantages of loss of control and security. In the author's opinion, it also blurs and lowers the abstraction level of the mobile agent to one of merely a mobile object. If an agent's methods can be called and executed remotely, then any notion of autonomy for the agent has been lost. Hive thus embodies a hybrid abstraction, drawing elements from the autonomous agents research arena, and from contemporary RPC distributed systems. This hybrid abstraction has caused the Hive team some considerable headaches in achieving their goals [Minar99b]. This is a shame, since the ontological descriptions supported by Hive are superior to many if not all of the other frameworks reviewed.

4.6.5 Voyager

ObjectSpace's Voyager platform is a one-size-fits all communication infrastructure. At the time of writing Voyager currently supports EJB [Sun99], CORBA, DCOM, and RMI. In its early days ObjectSpace promoted the capability of Voyager to take existing CORBA IDL classes and "virtualise" them, effectively making them weakly mobile. This was a major selling point for Voyager, but recently the company has been playing down these capabilities [Glass99]. Voyager should really be viewed as a Java based messaging broker that has some added capabilities from the mobile agent field. This allows programmers to create network applications by choosing between traditional and mobile distribution technologies, and has been a widely successful product.

4.6.6 Jini

Jini [Arnold99] is Sun Microsystem's proposed architecture for embedded network applications. It is built using Java and RMI in much the same way as Hive. Jini

provides simple mechanisms that enable devices to plug together to form an impromptu distributed system. Each device provides services that other devices in the system may use. These devices provide their own interfaces, which Sun claims "ensures reliability and compatibility". Much to the chagrin of the Hive team, Jini is a very similar framework, although it does not have the shadow/agent conceptual split. Most important however is that Jini's creators do not consider location to be an important part of the abstraction. Where a particular service resides in the network is not of importance to Jini, the interfaces and lookup services are intended to handle this sort of issue. Further, Jini only supports single-hop mobility, and as such can be categorized as embodying merely the CoD abstraction. This continued support of the location transparency abstraction and only a basic mobile code abstraction are surprising as Waldo is one of the authors of the Jini specification.

4.6.7 Aglets

The Aglet Software Development Kit (ASDK) [Lange98] has been developed by IBM's Tokyo Research Labs, and was one of the first and most publicised Java based mobile agent frameworks released. The core abstractions supported by the ASDK are that of an *aglet*, a *proxy* and a *context*.

An aglet is a mobile autonomous agent, whose structure can be considered to consist of two distinct parts, the aglet core and the aglet proxy. The core is the heart of the aglet and contains all of the aglet's internal data and logic. It provides interfaces through which the aglet may communicate with its environment. The aglet core is then encapsulated by an aglet proxy that acts as a shield against any attempt to directly access any of the aglet's private internals, and can hide the real location of the aglet from malicious aglets.

The aglet context is the executing environment in which the aglets exist. It provides an interface to the underlying operating system through which aglets are able to access core facilities, and gain references to other aglets' proxies. The context also manages the lifecycle of an aglet. Since the ASDK only provides weak mobility, this lifecycle is one of the ASDK's most valuable features since it allows the programmer to describe behaviour an aglet should perform in reaction to certain events, for example, the shutdown of the current host, or a request to migrate to a new host. This

lifecycle is supported through an event-based scheme that is well known in the window system programming world. Aglets implement a number of event handling methods that can be customized by the programmer. These methods cover all the important events in the life cycle of an aglet (creation, dispatch, arrival, deletion, etc.). For example, if you move an aglet it will be notified upon leaving its host and upon arrival at the new host. Of all the frameworks reviewed, Aglets enforces the mobile agent abstraction and metaphor most strongly. In contrast to Hive, all communication between aglets is via messaging. On receipt of a message, an aglet is able to decide what to do with the message, and when, thus sustaining the autonomy of the agent.

4.6.8 The Mobile Agent Graveyard: Telescript and Odyssey

Developed by General Magic Telescript [White96] was an object-oriented programming language designed for the development of Personal Intelligent Communicators (PICs). PICs were defined as being handheld palmtop-like devices with little memory and low bandwidth capability. Telescript was the first of its kind to appear and ground breaking in the facilities it offered.

Telescript was an interpreted language that supported strong mobility. There were actually two levels of the language: *High Telescript,* the actual language used for implementation, and *Low Telescript,* a Postscript like language which could be interpreted better by the top level executing environment, the *engine.*

Other abstractions supported by Telescript included *agents,* mobile agents that were able to migrate on a single command of `go`; *places,* stationary processes that provide interfaces to services, and were normally inhabited by agents; *tickets,* objects that describe an agents journey; *permits,* objects that define the capabilities and resource constraints of an agent.

There is an important programming paradigm difference between Aglets and Telescript that demonstrates the differences between strong and weak mobility: Telescript is focused on process migration that allows you to "go" in the middle of a loop and resume the execution in the middle of that loop on another machine. Aglet developers must consider how to deal with migration of non-static data.

Sadly, Telescript is no longer available, having gone to the Mobile Agent Graveyard[7]. Odyssey was General Magic's attempt to revive its flagging fortunes with a Java based mobile agent framework that resembled Telescript. It never made it out of beta.

4.7 Choosing a Mobile Agent Framework

Whilst there are an increasing number of mobile agent frameworks, when the study described in this thesis began the choice was limited to perhaps half a dozen. From those available, IBM's Aglet framework was selected. It would be appealing to be able to demonstrate a methodology employed for selecting the framework, but there is none. The Aglets package was chosen due to the connections of Danny Lange, the inventor and chief architect of Aglets, to researchers at MSI. However, in defence, several important factors support the choice of the ASDK:

- it was one of the first to use the Java programming language;
- it contains the notion of agent itinerary which systems such as Telescript did not support;
- it is being proposed for submission to the Object Management Group (OMG) Mobile Agent Facility RFP;
- it includes a fine grained security model
- aglets has proven to be an extremely popular framework in the mobile agent community for its clear agent abstractions and lifecycle facilities

Actual mobility in the ASDK is enabled by the provision of two facilities:

- the Agent Transfer Protocol (ATP)
- the Java Agent Transfer and Communication Interface (J-ATCI).

The ATP is an application level protocol for distributed agent based information systems and facilitates migration of the aglets over a network. Based on the naming conventions of the Internet, ATP uses the Universal Resource Locator (URL) [Berners-Lee92b] for specifying host locations, whilst maintaining a platform independent protocol for enabling the transfer of mobile agents between networked computers. Although this protocol has been released with the ASDK, its domain of use is by no means exclusive to aglets, as it offers the opportunity to handle mobile

[7] It lives on though, through furtively copied gold CD's!

agents from any programming language and a variety of agent systems, as long as they implement the protocol interfaces.

Reinforcing the ATP at a higher communication level is J-ATCI, an independent agent protocol enabling agents to move and communicate within a network. J-ATCI is a simple and flexible programming interface that enables programmers to develop platform independent agents without having to build into them the necessary protocols for wire communication. By ensuring a native implementation of the J-ATCI designers can expect their agents to function on any platform. The J-ATCI has also been submitted to the OMG.

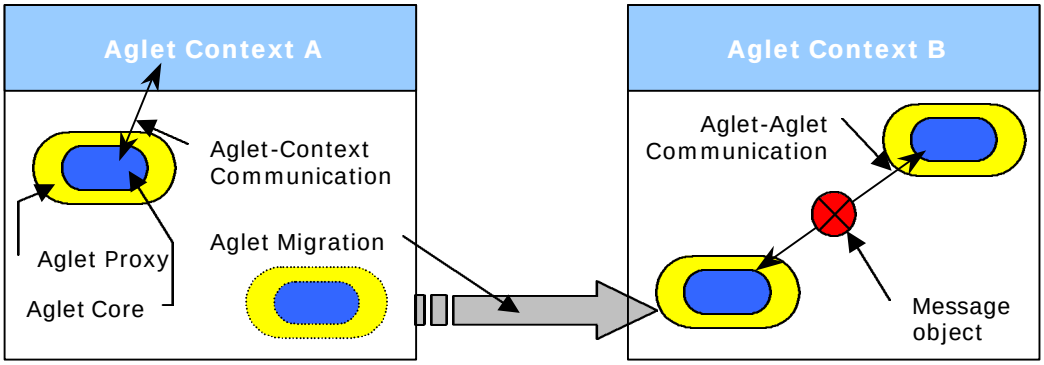

Figure 18. The Aglet Environment

4.8 Concluding Remarks

Pure academic thought might have been encouraged in the classical world, but in ours, we require facts too. To support the philosophical argument of Part I, we construct two prototype distributed systems with mobile code technology. To evaluate the systems we have identified several issues that are constantly engaging the software industry: system agility, semantic alignment and component coupling. The business process our systems are intended to support has been extracted from an industrial case study. The prototypes will be subjected to several *Scenarios for Change*, which will allow us to gain an insight into how well they perform.

This chapter also contains a review of the technical issues involved with implementing the mobile code abstractions, a summary of many of the claimed advantages for mobile code and a roundup of several of the more established mobile code infrastructures. In the following chapters, we report on the implementation and

evaluation of our prototypes. Before that however, we describe the case study that was used to generate a business model and process for the prototypes to support.

5 I.T.L. : An Industrial Case Study

5.1 Introduction

This chapter describes the industrial case study undertaken in the course of the research described in this PhD. It was performed at Instrument Technology Ltd (ITL), a high performance vacuum component manufacturer based on the south coast of the UK, in Q1 1997. In the next section, we discuss the methodology and the objectives of the case study.

5.2 Why a case study?

> "A case study is an exploration of a question or phenomenon when little is known in advance, and where the situation may be complex." [Yin94]

Case studies are able to examine processes within a specific context, draw on multiple sources of information, and relate a story, usually in a chronological order. In case studies, we are able to ask: "How or why does this occur?" We can create a rich, textured description of a social, economical or infrastructural process [Scanlon97]. This information can give an insight into how to gain answers to more specific questions, or produce conceptual models of a business process.

It has already been shown that the mobile code community recognises the lack of real world examples of their technology [Picco98] [Milojicic99]. We aim to prove that mobile code can be used to build real software systems. Therefore, the scope of this particular study was to gain an insight into I.T.L. and identify a suitable business process. The extraction of an industrial process model would provide a suitable reference around which the subsequent prototype implementations could be built. Further, the case study allows us to generate real world scenarios that can be used to evaluate the prototype systems after their construction.

When performing a case study it is extremely important to select an appropriate methodology [Jones97]. To achieve our objectives, the methodology selected was to carry out a qualitative, exploratory case study. Qualitative studies are particularly useful in attempting to answer questions such as 'Why?' or 'How?' [Strauss90], while exploratory studies are those that attempt to gain an initial insight into a situation.

Together they allow the examiner to create a 'snap-shot' in time of a particular process or situation. The methodology was considered appropriate, as it was capable of fulfilling our requirements:

1] To produce an SOP model,
2] which was based on a real world example,
3] upon which a set of experimental scenarios could be based.

The models generated from the case study are presented and discussed later in the chapter, following an overview of I.T.L.

5.3 Who are I.T.L.?

Instrument Technology Limited (I.T.L.) is a British manufacturing company based in East Sussex. It has been established for over twelve years, and usually performs steadily. A recent diversification in product range had reaped benefits however, and at the time of the case study, the company had shown a growth in turn-over from £500k to nearly £10m in five years, whilst concurrently developing an extensive, global customer and distributor base. More recently, the company has been affected by the crash of the Asian tiger economies.

5.3.1 What does I.T.L. do?

I.T.L.'s core business is manufacturing high performance vacuum components, primarily for the semi-conductor industry. The scope of the product range ensures that there are few other companies in the world that manufacture a greater diversity of standardised vacuum components. At the time of the case study, there were over 2,000 modular products and almost 7,000 items in the product catalogue. In an interview with the managing director [Barlow97] it became clear that these figures were expected to increase. The company has been quick to recognise the trend towards customer-driven specialised services and part production. This is supported by an extremely flexible design service offering almost unlimited choice to customers, who are able to submit their own specifications for product manufacture. Co-existing with the standardised product group is the specialised vacuum chamber division, which builds intricate, high pressure chambers and vacuum chambers, usually for advanced research facilities such as CERN.

5.3.2 How does I.T.L. work?

Until 1997, I.T.L. perceived[8] its largest market to be in the UK and Export direct sales, in which they have a substantial market share. However, the emphasis for the company is now shifting to much larger, more lucrative contracts with several international OEM's. Deals with a number of multinationals have consolidated previously successful working relationships, and ensured good market standing for I.T.L., which is now emerging as a global “player” in the vacuum component market. For direct sales, a network of Sales agents deals with the promotion and marketing of brand products. The network encompasses Europe, the Far East and Central and Southern Africa, with several more slated for adoption in the short term. All orders are still supplied from I.T.L.’s headquarters in the UK. OEM partners are offered exceptional configurability in delivery and service. For example, specialised packaging, branding or invoicing.

Figure 19. An overview of I.T.L. around the world.

I.T.L. now perceives the greatest potential for sustained growth in expanding its network of Sales Agents into new markets, whilst attempting to broker new OEM deals with further American companies [Barlow97]. Consolidation with its oriental partners has also brought new opportunities in reducing manufacturing costs, and the company is investigating the viability of investing in new manufacturing facilities in the Far East.

[8] The term “perceived” used here is factually correct, at the time of writing no one at I.T.L. was able to give exact figures for any of their markets.

Finally, I.T.L. has settled on a long-term strategy of expanding its global presence. In doing so, I.T.L. has realised that it will no longer be economical to continue with centralised stock control since transportation of its products is expensive. Ergo, the company is considering adding new stock control centres or warehouses at globally strategic locations.

5.3.3 Commentary

With the increasingly extensive portfolio of products and parts, the configurability that I.T.L. offers to its customers, coupled with the long term strategy of expansion and the need to remain responsive in the market place, it is clear that I.T.L. requires a high degree of flexibility from both its business practices and the supporting IT infrastructure.

I.T.L. is also hoping to expand both its network of Sales Agents and its stock control centres. This requires a radical change in the company's business practices. It must transform from a central and localised operating model to a distributed one. The pitfalls and problems associated with transformations of this kind are well documented [Peters82] [Hammer93] [Goldman95].

Equally, as the Asian Tiger economies example demonstrates, I.T.L. is competing in a fluctuating market. Responding to such problems as, for example, changing suppliers or meeting 'Just In Time' (JIT) manufacturing requirements mean the company must strive to remain agile. Here, agility is considered the ability to respond quickly to market pressures. For example, both up and downturns in orders, adding or removing suppliers, adding or removing sales agents, etc.

It was our aim to generate a process model from a real company. This would then form the basis for our implementations, and would allow us to evaluate their performance when subjected to the kinds of pressures a real software system may experience. From the case study, it is clear that I.T.L. is a prime example of a manufacturing enterprise facing the very real pressures of remaining agile and competitive. The requirements of I.T.L. can be summarised as:

- It requires a high degree of flexibility in its IT infrastructure
- It must be able to add new sales agents quickly

- It needs to add new stock control centres
- It must be able to upsize and downsize with equal ease

5.4 Process Modelling

Having established I.T.L. was a suitable candidate upon which to base our implementations it was important to identify a suitable business process. The requirements of I.T.L. listed in the previous section all pertain to the Sales Order Process (SOP). Indeed, the SOP plays a pivotal role in any business that relies on constant orders for survival, and involves links to customers, distributors and suppliers throughout the world. This is a perfect process to support with a distributed software system, and therefore, the decision was taken to use I.T.L.'s SOP as the process model.

Understanding the internal process of a company can be complex. A simple but effective tool that is often used for this purpose is a data flow diagram (DFD) [DeMarco78]. Using DFDs, the core business processes of I.T.L. were modelled in an attempt to understand how I.T.L. responds to a new order (see Figure 20). In this diagram, the many processes are defined by the senior management figures that are responsible for those particular areas. Each core process is surrounded by a dotted line for further clarification.

From this rather complex diagram, it is possible to extract the core business processes and represent them in a higher level, abstract view. Figure 21, the Abstract Process Model (APM), shows this simplified view and depicts the interactions between the each process upon receipt of a new order. The decision branch shown in Figure 21 has been intentionally omitted from Figure 20 for reasons of clarity. By examining the interactions between the major components of the APM a basic visual model was generated to represent the entire process. This can be seen in Figure 22. To better understand this model we will walk through an example of a new order being placed.

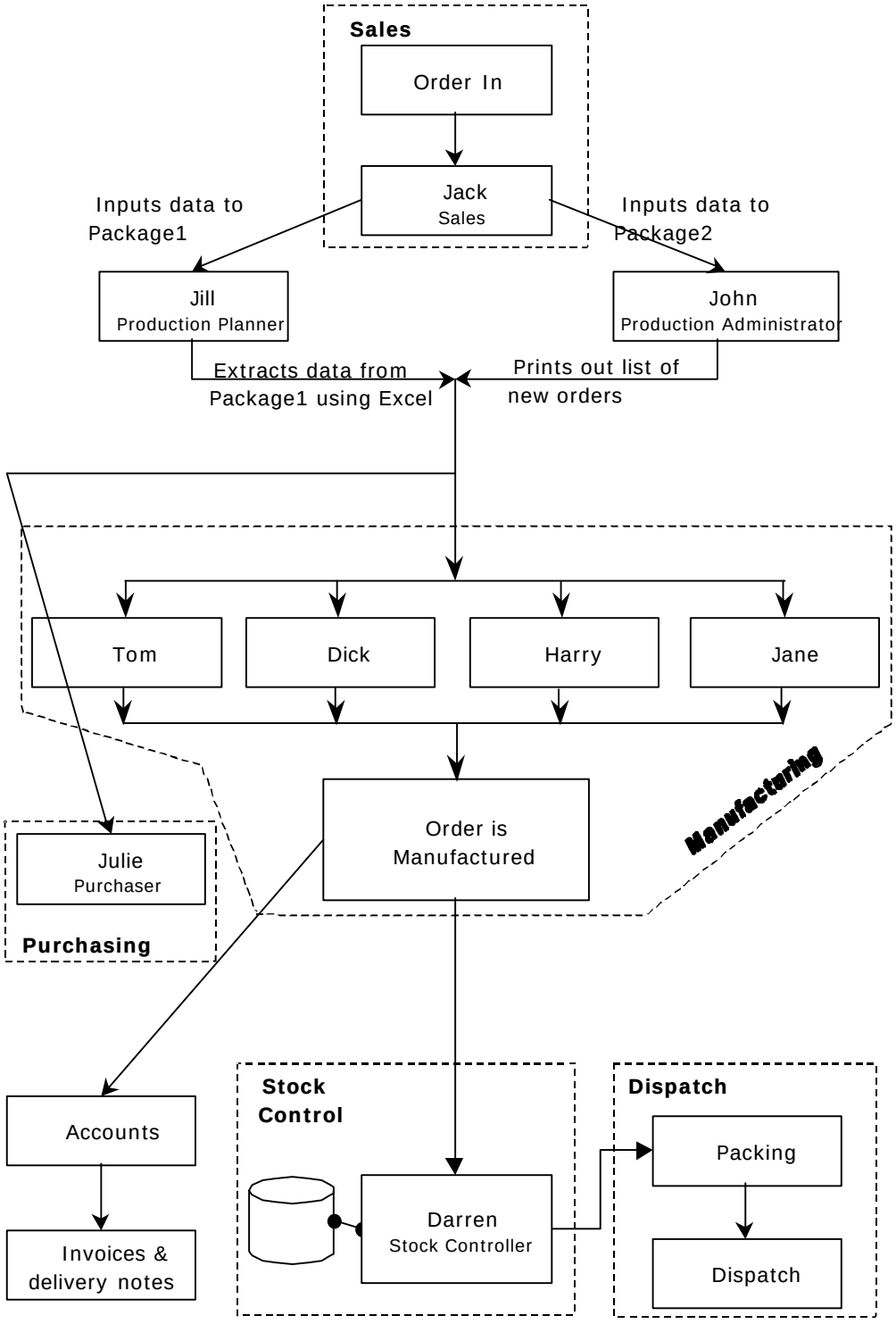

Figure 20. Information flow through I.T.L. on receiving an order

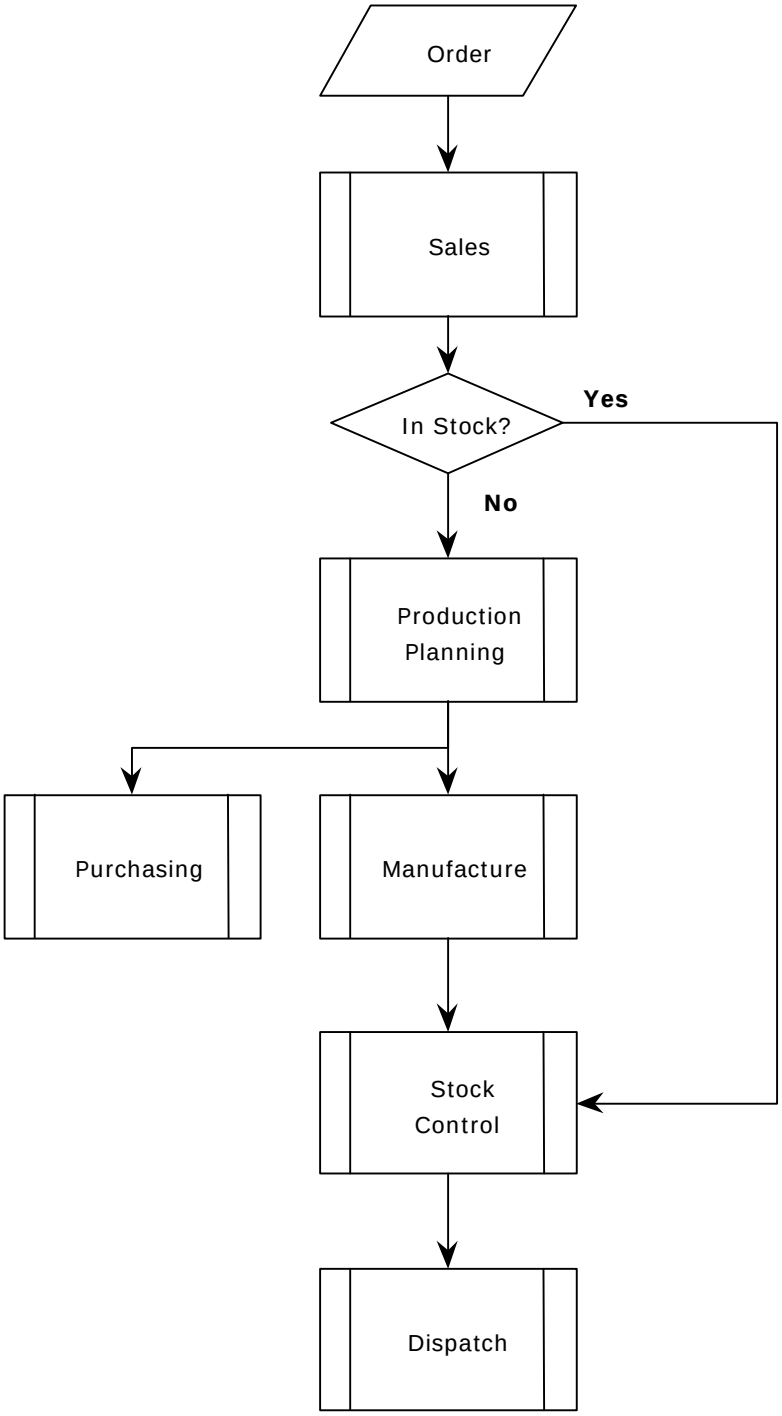

Figure 21. Abstract Process Model

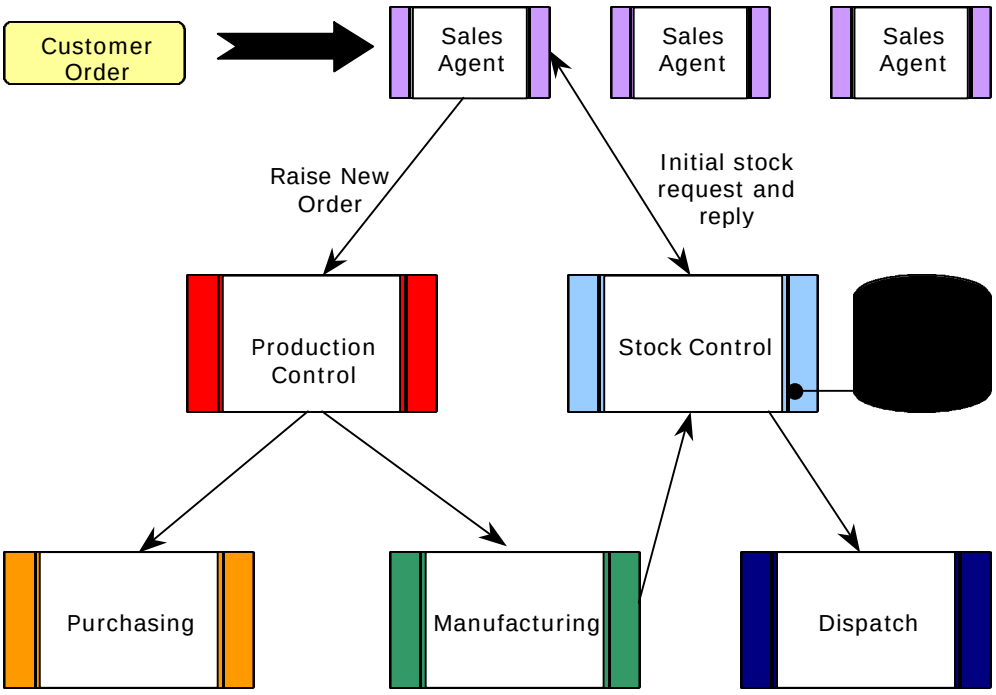

Figure 22. The Sales Order Process

5.4.1 A Walkthrough

A new Customer Order is placed with a Sales Agent. The Sales Agent then interrogates Stock Control to see if the order can be fulfilled from the existing stock. If it can, a new Order is raised and the items are allocated to that order number before being dispatched to the customer, along with an invoice.

If the items are not in stock, then the order is passed to production control where again, an Order is raised. Accompanying this Order is a new Works Order for the required manufacturing of the requested products, or product parts. The Works Order is then passed to manufacturing for completion, and if necessary purchasing for replacement of raw materials. Once the product or parts are completed, they are booked into Stock Control before being checked out again for dispatch. The standard delivery time at I.T.L. is three weeks, unless the order is being specially manufactured to specifications submitted by the customer.

5.4.2 Refining the Model

Implementation of two software systems to support in full the Sales Order Process of a manufacturing enterprise is beyond the scope and time frame of a PhD. Therefore,

we decided to concentrate on the interactions of sales agents handling order requests and the stock control centres. These particular facets are fundamental to the SOP as a whole, and are intrinsically associated with the issues of building distributed software systems. Thus, these processes form the major components of the subsequent prototype implementations.

The Production Control process was removed from the model since scheduling is an entire field of research in its own right and was deemed external to the objectives of this thesis. In addition, the greyed out areas of Dispatch and Manufacturing represent processes that were considered of secondary importance to the requirements identified in Chapter 4. These would make excellent candidates for investigation and expansion in any future work. The finalised model used in the implementation can be seen in Figure 23.

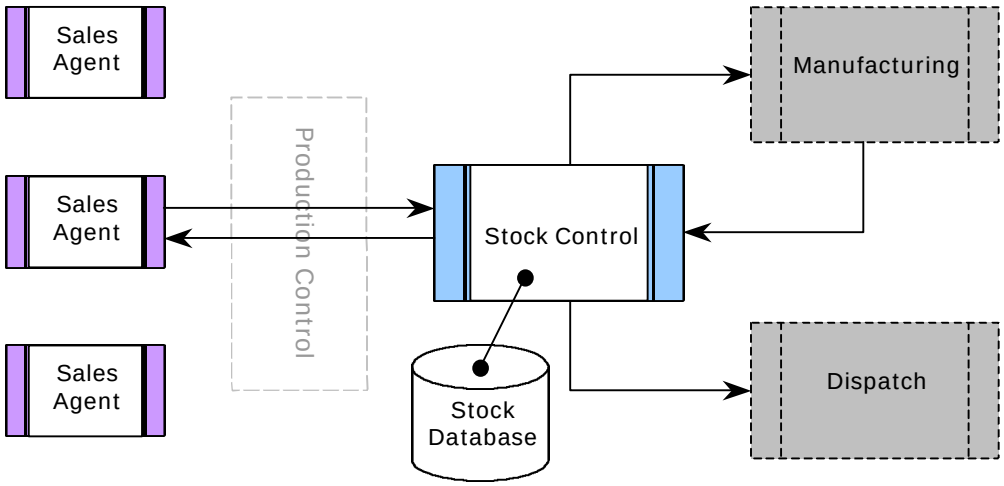

Figure 23. Modified Sales Order Process model

5.5 Concluding Remarks

Gaining an insight from a real world manufacturing enterprise is an invaluable tool for developing a model from which to base experimental work. This chapter has presented the case study undertaken at the vacuum component manufacturer Instrument Technology Ltd. Examination of I.T.L.'s core business processes has yielded a high level abstract model based around the Sales Order Process. This model will be used as the basis for the prototype implementations described in the next chapter. In addition, the company's background and operations were examined,

resulting in the identification of a set of requirements that I.T.L. had of their software system. These are summarised below.

I.T.L.:

- requires a high degree of flexibility in its software systems
- must be able to add new sales agents quickly
- needs to add new stock control centres
- must be able to remove new additions with equal ease.

In the next chapter we describe the implementation of our two prototype systems.

6 Implementation

6.1 Introduction

It has been stated that the field of mobile code research lacks examples of real world applications [Picco98]. Therefore, the work in Part II of this thesis has been undertaken with that fact in mind. In support of our philosophical argument for mobile code, we wish to demonstrate the feasibility of actually building real distributed systems with this technology.

In the previous chapter, we described the generation of a Sales Order Process model, which we aim to support with mobile code technology. We have further refined the model to focus our investigative work on those aspects that depend on distribution by choosing to concentrate on the interactions of sales agents dealing with order requests and the stock control centres.

In this chapter, we describe the implementation of our two prototype systems, a mobile object version of the business model and a mobile agent version. First, we begin by presenting a top down view of the implemented SOP model, before going on to discuss the common parts of the two prototype systems and detail their differences.

6.2 The Model

Figure 24 depicts the implemented mobile agent model of the SOP. The fundamental operation of the process is as follows: following an enquiry from a customer to a SalesAgent (SA), an OrderAgent (OA) is dispatched to the StockControlAgent (SCA) where it requests the fulfilment of its order by passing an Order object. The StockControlAgent, which is resident at a distribution point, queries the stock database to see if enough products are in stock. If there are enough products, the StockControlAgent then returns a DeliveryDate object to the OrderAgent. The OrderAgent then returns and reports to its parent SalesAgent, which is then able to notify the customer of the delivery date.

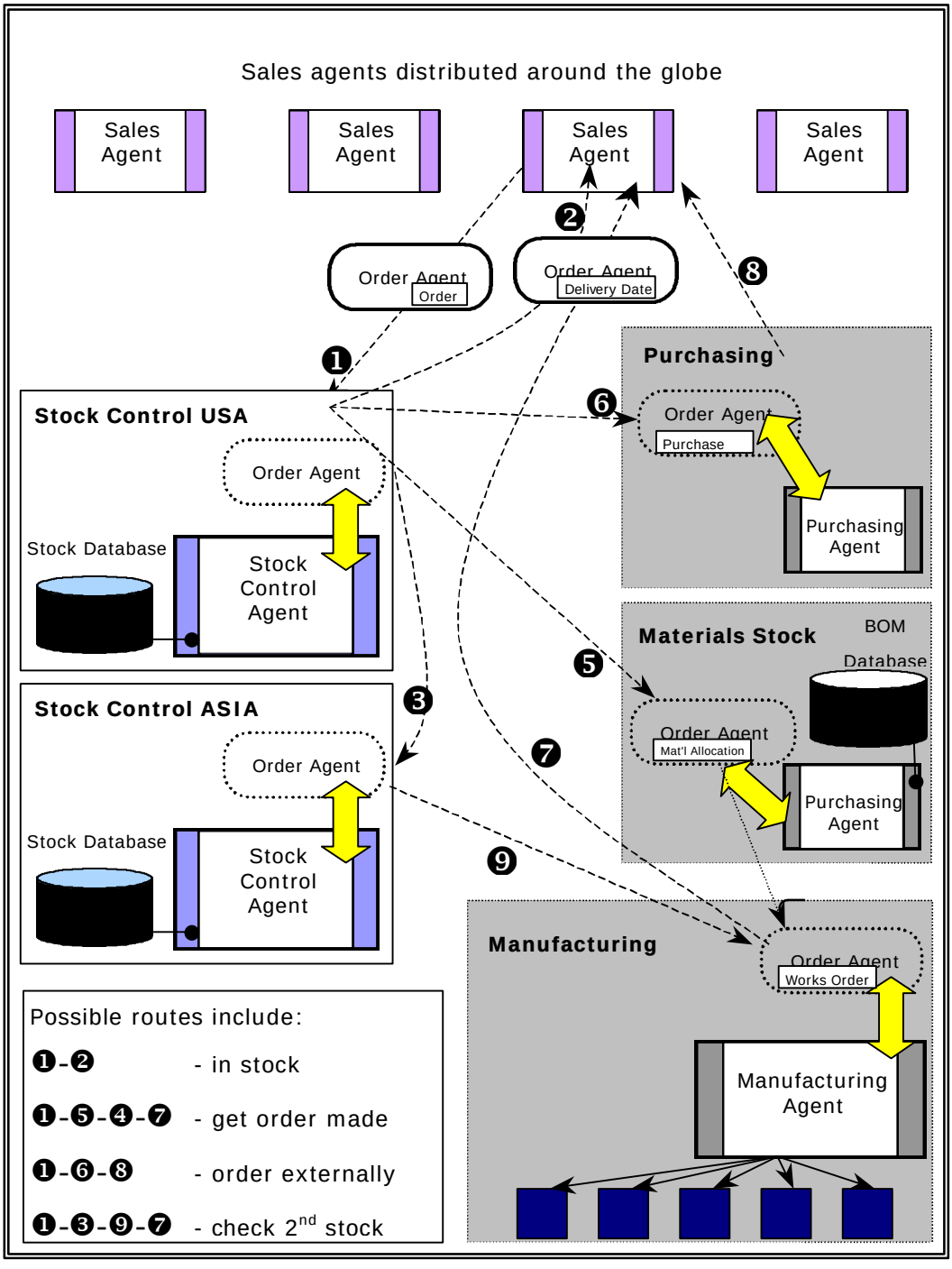

Figure 24. Agent Sales Order Process Model – with example routes for OrderAgents

If there are not enough products in stock to satisfy the order, the OrderAgent migrates to the manufacturing plant where it uses the Product ID encapsulated in the Order object and queries the BOM database for a list of sub-parts or raw materials required. This is then encapsulated within the OrderAgent, which is dispatched to manufacturing to deliver it, before returning to the SalesAgent with a DeliveryDate

object containing a standard delivery date. If there are not enough raw materials in stock, agents within the manufacturing plant server generate a PurchaseOrderAgent that encapsulates details of all the required materials.

The mobile object model is very similar to that described above, the key difference being that the results from stock database queries are gathered from remote StockControlAgents by a mobile OrderObject guided by a specific itinerary. Instead of processing this information locally to the data source, it is returned to the SalesAgent for processing. At arrival, the OrderObject delivers the results before being terminated. If further excursions are necessary, the SalesAgent creates new mobile objects and dispatches them as required. The mobile object does not make autonomous decisions based on the acquired information.

6.3 The Bestiary

The implementation work described in this thesis was undertaken using IBM's Aglet Software Development Kit [Lange98], a mobile agent development framework that was extensively described in Section 4.6.7. This framework has been used as the base upon which to implement the two different versions of the SOP model. Each major process has been embodied as an agent, and there is quite a large overlap in commonality between the two systems. Similar amongst both models are the static agents consisting of SalesAgents, StockControlAgents, ManufacturingAgents, PurchasingAgents and DispatchAgents. As one might expect, there are also mobile components to the systems, and it is here that each system differs from the other. In the mobile agent system, there are OrderAgents, whilst in the mobile object system there are OrderObjects. Generically, we will refer to these as the Order components of the systems. This is primarily, although not entirely, where the distinction between the Remote Computation and Mobile Agent abstraction is evident. It should be noted that in a static analysis of the system, the mobile Order components are a single entity in the design. However, during execution the number of migrating mobile components in the system would be significantly more than the number of static components. In the following sections, we discuss each agent type and its relationship to other agents.

6.3.1 OrderAgents

OrderAgents represent the mobile components in the Mobile Agent system. The agents discussed in this paper can be classified in line with Franklin and Graesser [Franklin96] as goal oriented, communicative, and mobile i.e.:

- **Goal oriented** – they do not simply act in response to the environment
- **Communicative** – they are able to communicate with other agents
- **Mobile** – they are able to transport themselves from one host to another.

On creation, each OrderAgent is given a copy of a new `Order and` an `Itinerary` that contains details of which hosts they must visit to enquire about completion of their Order. Encapsulated within the `Itinerary` are `Tasks`, which the OrderAgent carries out on arrival at a new host. Once the OrderAgents have been given an `Order`, they are then responsible for completion of that order. Some example program listings of an OrderAgent and a Task can be found in the Appendices.

After creation, the OrderAgents migrate to the first host in their `Itinerary` to interact with the resident StockControlAgent. This interaction will involve the OrderAgent querying the StockControlAgent as to whether the Order it is carrying can be satisfied by the levels of stock currently held. The actual stock database is queried by the StockControlAgent; the OrderAgent does not interact with it. The OrderAgent processes the results returned by the StockControlAgent. If the relevant stock is available the OrderAgent asks the StockControlAgent to book out the stock to its Order number before returning to the SalesAgent that created it to report on the delivery date, whilst the StockControlAgent sends a message to the DispatchAgent with details of where to send the products. If the stock levels at the first StockControlAgent are unsatisfactory, the OrderAgent is able to migrate to the next host in its list to begin the process again. However, if no StockControlAgents are able to satisfy the Order then the OrderAgent will proceed to the ManufacturingAgent to request production of the relevant components. Although this behaviour remains unimplemented, it is intended that the ManufacturingAgent would then interact with some scheduling software system to ascertain an estimate on the required time for manufacture that the OrderAgent could use to report to the SalesAgent. Currently, this communication consists of a simple message and acknowledgement from the ManufacturingAgent.

The valid outcome for the goal of the OrderAgent is reporting a delivery date for the order to the SalesAgent. If all else fails, it will return and report that it has failed, allowing the SalesAgent to begin the process again. In the future, this may also include reporting an allocation for raw materials, an internal works order number and time to manufacture. While not complex, OrderAgents usually make up the majority of the agent population in the system, although this is dependent on the number of enquiries received by the SalesAgents. Potentially, there could be hundreds of mobile OrderAgents migrating through the network, attempting to fulfil their own particular `Order`. Since OrderAgents require no interaction with a user, they have no Graphical User Interface (GUI).

6.3.2 Order Objects

OrderObjects are the mobile components of the Remote Computation system. However, in contrast to the mobile agent system, it is more appropriate to view the mobile objects as mobile messengers. Initially they appear to perform the same function as the OrderAgents described above, and in many respects, this is true. On creation, the OrderObjects are given an `Itinerary` and an `Order` and are dispatched to the first host on their list. There, they again query the StockControlAgent to establish whether the order may be fulfilled at that host. Although OrderObjects are still able to migrate to a data source and take advantage of local interaction and all the advantages that brings, they do not contain the business logic to autonomously process any results. They merely add them to their records before migrating to the next host in the `Itinerary`. Once all hosts in the list have been visited, and all stock databases queried, the OrderObjects return to their origin to report the findings to their parent SalesAgent, after which they are terminated. In this system, the processing of the results is performed by the SalesAgent, which creates a new OrderAgent and dispatches it to one of the hosts to commit the stock to the Order. Again, during execution there may be many hundreds of mobile OrderObjects instantiated within the system.

6.3.3 SalesAgents

SalesAgents are static agents that are responsible for generating Order components, giving them an `Order` and `Itinerary`, and sending them out into the network so they

may interact with StockControlAgents. SalesAgents are the human users' main interaction with the SOP system and therefore they have a GUI with which the sales person can create a new Order. SalesAgents are more complex than the Order components, since they must keep track of current orders, but they still remain "slim" and can be manifest as a client for sales persons working on terminals or NetPCs, or be hosted on a laptop for travelling sales persons.

In the mobile agent version the only logic contained within these agents is that required to create a new OrderAgent, with its accompanying `Order` and `Itinerary`. They are capable of maintaining a list of spawned OrderAgents, and thus are aware of which Orders have been fulfilled. In the mobile object version, they also contain the business logic required to process the results returned by their slave OrderObjects.

6.3.4 StockControlAgents

The StockControlAgents are another example of static agents within the systems, but as they do not interact with human users, they have no user interface. They are responsible for handling all requests for products and materials made by the Order components, and act as custodians for the information contained in the stock databases. As such, they are a communications bridge between the data sources and the other agents in the system. All requests for stock levels and allocation must be made through the StockControlAgents.

Manufacturing enterprises are usually supported by a heterogeneous mix of hardware and software, with many different types of database systems employed at any given time. When designing StockControlAgents so they may connect to such a variety of database systems it became apparent that some of the required features of these agents were particular to each database, whilst others were generic and could be applied to any StockControlAgent. In the initial stages of the implementation, the StockControlAgents had been using text files as their storage medium, modelled on MICROS records. Many new database systems no longer use text files however, so it was later decided to improve their capability to allow them to communicate with any ODBC enabled database. ODBC is an industry standard for database access. The work on this problem has yielded a common design that can be used as a base pattern

and applied to all StockControlAgents [Papaioannou99]. The DataQueryAgent is discussed later in Section 6.4.1.

6.3.5 ManufacturingAgents, MaterialsAgents, PurchasingAgents and DispatchAgents

These particular agent types have been classified as having secondary importance to this initial study. Currently all three are represented in the SOP systems by “dumb” static agents. By dumb we mean that they are merely communicative and possess no internal logic to perform any particular tasks. They are able to simply acknowledge communication from other agents, and represent a definite avenue for further investigation and research. However, their presence in the systems allows us to begin to explore the issues involved with multi-hop mobile agents vs the client/server paradigm.

6.4 Considering Lifecycle and Maintenance Issues

The implementations described in this thesis are proof-of-concept systems. They are used in our experimental work to demonstrate that real world software can be built with mobile code systems. In addition, we wished to measure the degree of flexibility, coupling and semantic alignment offered by the mobile code abstraction. Further, to fully consider the support provided for building real world systems we examine the full lifecycle phases of software systems. These include issues relating to design, implementation, runtime and maintenance. The resulting knowledge and supporting tools and are discussed in the next sections.

6.4.1 DataQueryAgent: A Proto-Pattern for Database Query

A major goal of the work described in this thesis has been to build agile software systems. For the software architectures implemented in this study to achieve this throughout their lifetimes, they must be capable of querying a variety of new or legacy databases. Investigation into this problem has generated an effective and reusable proto-pattern that can be used to build agent database query systems [Papaioannou98]. The DataQueryAgent, shown in Figure 25, can be decomposed into several constituent parts, which are described in the following sections.

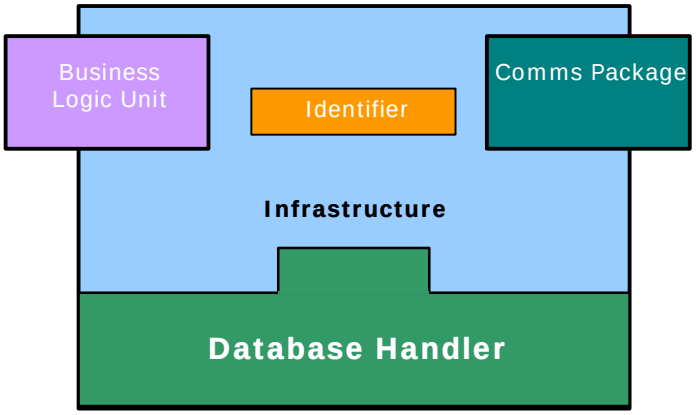

Figure 25. DataQueryAgent Architecture

6.4.1.1 The Infrastructure

The infrastructure provides the system creator with the facilities to communicate with, and manage the lifecycle of agents in the system. The environment in which the agent will execute normally dictates the infrastructural requirements, although they are usually accessible through the framework libraries or via class inheritance. For example, in our implementations these facilities are attained by extending the abstract `Aglet` class.

6.4.1.2 The Identifier

The Identifier plays an essential role in system security and traceability. Whilst it is more usual for mobile agents to carry an Identifier, static agents must also be able to prove their credentials. In future implementations, we imagine that StockControlAgents would be able to generate PurchaseOrderAgents and WorksOrderAgents in order to fulfil unsatisfied orders. Part of the parent's Identifier would be handed to these child agents, as proof of their origin on dispatch to another host.

6.4.1.3 The Communication Package

The Communication Package handles the incoming communication from querying agents and translates this into a format the Business Logic Unit or Database Handler components are able to understand. Inter-agent communication methods vary between different agent environments, as do the communication protocols and requirements of differing agent solutions. In some examples, simple `String` matching is sufficient for simple communication. However, interactions that are more

complex may require an attempt at semantic level communication. The use of Agent Communication Languages (ACL's) such as KQML [Labrou96] is typical of the more advanced approaches that are being proposed to solve these problems. To handle the requirement for a variety of communication methods, the Comms Package can be interchanged by the software designer with respect to their particular requirements.

6.4.1.4 Business Logic Unit

The Business Logic Unit is used to understand communication and queries from other agents, and generate a course of action to fulfil those requests. In the SOP scenario, when an OrderAgent is dispatched by the SalesAgent, it encapsulates an `Order` object. Upon arrival at the StockControlAgent, it will attempt to fulfil that order, a task that in itself can require some simple logic. For example, for simplicities sake an `Order` object only contains descriptions of the full products that are expected. Although the OrderAgent may only be aware that it requires one hundred widgets by Tuesday, the StockControlAgent may include some logic that translates this request into one where a widget must be supplied with a grommet and two nuggets. Thus, the Order actually requires one hundred widgets and grommets, plus two hundred nuggets. More probably, the StockControlAgent will query another database to retrieve the Bill of Materials for the product. Since all the OrderAgents will require this same logic, it is clear that including it as part of the DataQueryAgent is the best solution. By keeping the size of the `Order` and the encapsulated logic low, the size of the OrderAgent is kept small, reducing network traffic.

6.4.1.5 The Database Handler

The Database Handler deals with connecting to a database, retrieving information from it, updating it, or even switching databases transparently to the requesting agent. It works in tandem with the Business Logic Unit to fulfil the request of a querying agent. The Database Handler ensures that the DataQueryAgent is capable of interfacing with many different types of data source.

The examples shown in Figure 26 address a large percentage (but by no means all) of the real world situations and the methods currently being employed to query databases within a manufacturing enterprise. Connecting to a new type of database ostensibly requires only the production of a new Database Handler. However, we make no

claims about the ease of this task. It is understood that access to a database is not all that is required; there remains the difficult problems of understanding the schema used in the new database before specific information can be retrieved. Work towards this goal can be seen in the efforts of the EDI and STEP/PDES community.

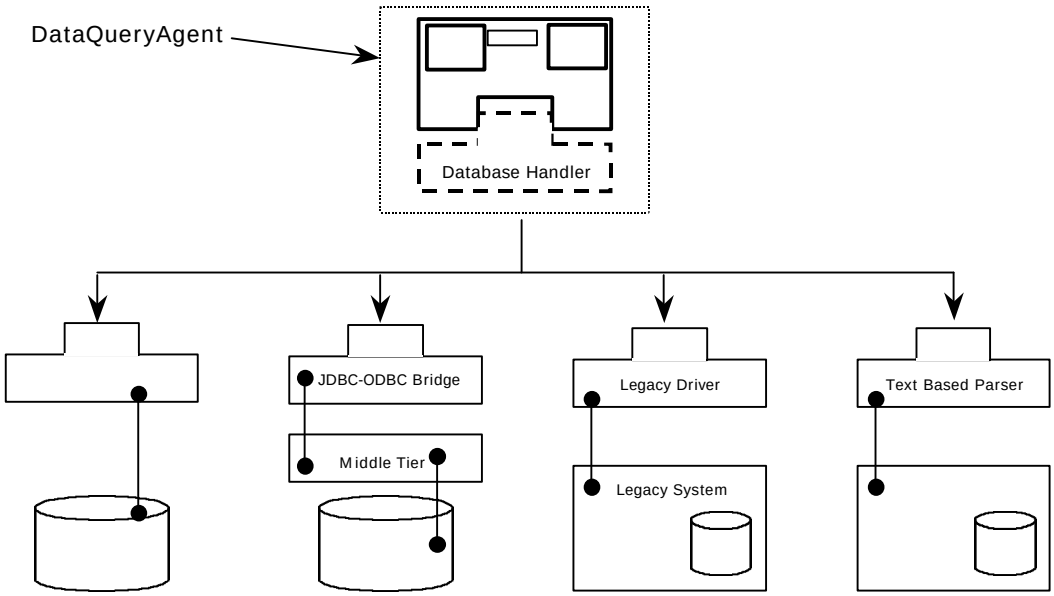

Figure 26. The DataQueryAgent with examples of different DataHandler modules

6.4.2 The Data Connector Tool

When constructing the StockControlAgents for our implementations, using the DataQueryAgent pattern, it became apparent that the most arduous task involved was in making the connection to a database. Whilst on the surface a relatively simple task, there are several variables that must be configured correctly, and a number of JDBC interfaces that must be used accurately. To alleviate the problems this caused, the DataConnector tool was produced to automate some of these tasks.

The DataConnector Tool is a Java program, with a user interface that allows the user to insert the required parameters for connection to a JDBC compliant data source. The validity of these parameters can be repeatedly tested, using the refresh, update and test facilities, until the correct configuration is achieved. Once a satisfactory connection has been made, this data is then exported by serialising it to disk. Each StockControlAgent can then be given a reference to the file that contains the particular information they require to connect to their specific database.

6.4.2.1 Benefits of DataConnector

The biggest advantage in using this tool is the ability to test connections to a database and server across the network, or even the Internet. If a virtual enterprise were to decide to use mobile agent technology as a tool for rapid integration, it is likely that one of the collaborators (or their systems administrator) will have some prior experience in using the technology. The DataConnector tool allows a single administrator to test all the required database connections between the relevant systems, and produce a set of connection information files that can be forwarded to the respective sites. Moreover, if the agent environments and servers have already been set up, a Messenger agent could deliver the files, and the DataHandlers could be completed and initialised automatically. The lightweight nature of a connection information file means that continued use of the agent system would allow an administrator to build up a set of predefined files for various configurations that would accelerate the speed with which new collaborators or data sources could be added in the future, increasing the system agility and responsiveness of the enterprise.

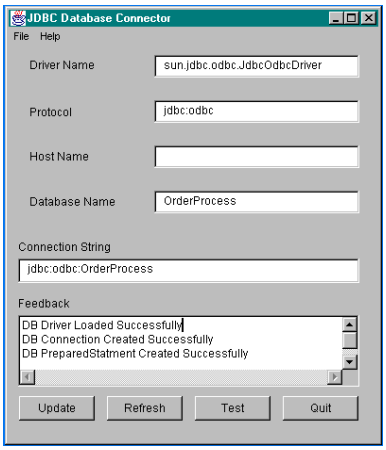

Fig 27 Screenshot of DataConnector

6.5 Concluding Remarks

In this chapter, we have described the realisation of our Sales Order Process model. We have produced two prototype implementations in order to evaluate the mobile object and mobile agent abstractions. The major processes identified in the overall business logic of the SOP have been embodied as agents in these systems, which comprise a mixture of static and mobile agents. Each individual type of agent created has been reviewed and discussed and their relationships examined.

The major difference between the two systems is the physical and conceptual location of the business logic associated with processing stock query results. In the mobile object version, this logic remains in the SalesAgent and is in an analogous position to where it would be found in a traditional client/server system. In the mobile agent

version, this logic is encapsulated within the mobile OrderAgent. In the former, the processing of the results must take place after all the data has been returned to the client, whilst in the latter the decision can be made locally to the data source by the mobile agent.

At the start of this chapter, we mentioned that part of the rational for this study was to demonstrate the feasibility of building real distributed systems with this new technology. We have accomplished that. We have built two prototype Sales Order Process software systems, based on a real world model, with mobile code technology. In addition, through consideration of the lifecycle and maintenance issues of these systems we have developed a proto-pattern to assist in the modular creation of DataQueryAgents. Supporting this pattern is a small tool, the DataConnector tool, which allows system administrators to rapidly connect DataQueryAgents to their data sources.

During the case study, described in Chapter 6 we also established several real world requirements for such systems. These have been identified as “scenarios for change” that can be used to evaluate how well each prototype responds to the types of pressures experience by real world software systems. The evaluation process and results are described in the next chapter.

7 Evaluation

7.1 Introduction

The previous chapter described the implementation of two mobile code systems. The rational for their construction was to evaluate the mobile object and mobile agent abstractions, in an attempt to understand exactly what each has to offer, and how that might affect how we build distributed systems. In this chapter, we evaluate how successfully each prototype responds to the *scenarios for change* that were generated from data collected in the case study of I.T.L, and report on the lessons learned and insights gained during these experiments.

7.2 Generating Useable Metrics

Evaluating software architectures is a notoriously hard task [Whitmire97]. There are very few established techniques or measurements for gathering data, and although software engineering as a discipline strives to emulate the classical sciences, we are still a long way off. Instead of formal equations, we have methodologies for developing metrics. They include: the Quality Function Deployment approach [Kogure83], the Software Quality Metrics approach [Boehm76] [McCall77] and the Goal Question Metric (GQM) approach [Basili94] [Solingen99]. Basili's GQM methodology was selected to evaluate the systems as it enjoys widespread popularity and support within the software engineering community.

In the next sections, we present an overview of the GQM methodology, and the principle goals, questions and metrics identified for the systems.

7.2.1 The Goal

The GQM methodology is based upon the assumption that to gain a practical measure one must first understand and specify the goals of the software being measured, and the goals of the measuring process. More specifically, it is important to specify what is being evaluated, what task it should fulfill and from what perspective to view the measurements. Once this framework has been established, it is possible to direct investigation and measurement towards the data that defines the goals operationally. The generated framework is also useful when interpreting the data.

The overall goal of our evaluation can be stated as:

> "To evaluate each prototype system from the industrialist's perspective, with respect to satisfying the industrial motivations to support system agility"
> (see section 5.5)

7.2.2 The Questions

Having stated the goal, the process is continued by generating a broad set of questions that may provide some indication of the individual issues encapsulated by the main goal. The objective is to generate as many questions as possible, including redundant or invalid questions. As the process continues, it is usual to develop a hierarchical set of questions that can subsequently be narrowed. This refined set can then be answered through tangible measurements made on the system.

To this end two workshops were held, one at MSI, Loughborough University, and one in the Computer Science Department of Reading University. In order to evaluate the prototypes with respect to the issues identified in section 5.5, the initial questions focused on system complexity (how easy is it to understand), and system agility (how easy is it to change). The results of these workshops were a large and varied set of questions, with many superfluous or duplicate entries. This is an expected part of the Basili methodology. Table 5 lists the focused set of questions that remained after refining.

7.2.3 The Metrics

After several iterations of refinement, and some healthy pruning, a set of usable software metrics remained that could be used to evaluate the two mobile code systems. These are shown in Table 6.

On their own, most of the generated metrics are extremely narrow in their focus. However, through combination, it is possible to arrive at some useful measures of a software system. In the following sections, we examine how these metrics can be used to evaluate the implemented systems, and discuss how well each prototype performs.

Generated Questions	Metric Number
How well does the system support change?	
How easy is it to understand the system?	
How many business entities map onto data abstractions	(1)
How many business processes map to software methods	(2)
Which real world entities that are mobile are also mobile in the system	(3)
Which real world entities that are static are also static in the system	(4)
How many components are there in the system	(5)
How many lines of code are there	(6)
How many comments are there	(7)
How easy it was to modify the system?	
How many conceptual entities must be changed - for example requirement a)	(8)
How many objects must be changed	(9)
How many src files must be changed	(10)
How many interactions must be changed	(11)
How many components are there in the system relative to the size	(5) + (6)
How many real world entities map to a software component	(1)+(2)+(3)+(4)
How many components must be changed	(9)
How many interactions must be changed	(11)
How many inter-entity connections are there	(12)
How many methods of the object are public	(13)

Table 5. Questions generated using the Basili GQM Method

Metric	Nature of metric
(1)	Identify information-based abstractions in the real world. Compare with info based abstractions in the software
(2)	Identify process-based abstractions in the real world. Compare with processes evident in the software.
(3)	Identify mobile elements of the real world, compare with mobile elements in the software
(4)	Identify static elements of the real world, compare with static elements in the software
(5)	Count the components
(6)	Count lines of code
(7)	Count comments, and get ratio of comments/method
(8)	Count num changes to entities for each requirement
(9)	Count num changes to objects for each requirement
(10)	Count num changes to interactions for each requirement
(11)	Count how many files are changed for each requirement
(12)	Count number of inter object method invocations
(13)	Count number of public methods

Table 6. Metrics Generated using the GQM Method

7.3 Evaluating Semantic Alignment

It has been demonstrated that semantic alignment between real world abstractions and components of a software system is important when attempting to build agile software systems [Coutts98b]. It is also a factor in how responsive a software system may be to change. To understand what the implications are for semantic alignment, when using mobile code, and to compare the two mobile code prototypes, we require some way of measuring how well the abstractions of the real world are embodied in software, and how well they resemble the real world model. For this, we have developed a term called Conceptual Diffusion.

7.3.1 Conceptual Diffusion

Conceptual Diffusion is defined as a measure of:

> "The degree to which a single concept or semantic abstraction in the application domain maps to the components in a software system."

Therefore, we may say that:

> ***CD* = A/B**

Where *CD* is conceptual diffusion, A is the number of concepts included in this abstraction, and B is the number of components in which this abstraction is embodied.

Conceptual diffusion can be examined at different levels of granularity to gain different perspectives on a situation. For example, in a software system that is intended to support a Sales Order Process we expect the concept of an Order to be present. On analysis, we find that in both the agent and the object systems the concept of an Order is split over four separate components. Thus, in these two systems, the concept of an Order can be said to have a conceptual diffusion rating of four (see Table 7).

Table 7 also shows the results of metrics (1) and (2). These metrics are examples of examining conceptual diffusion at a larger level of granularity. For example, metric (1) requires the identification of all the information-based concepts within the real world, and a comparison with their counterparts in the software systems. Since Order is an information-based abstraction, it is therefore included in the results of metric (1). We may use Conceptual Diffusion to gain an insight into how well concepts or abstractions are embodied in software.

Objects	Info Abstractions		Process Abstractions						SOP Logic	
	Order	Customer	SA	SCA	PC	M	P	D	MobAg	MobOb
BaseAglet			✓	✓		✓	✓	✓		
DBAglet				✓						
OrderAglet	✓								✓	✓
SlaveItin									✓	✓
SlaveDetails				✓						
SalesAglet			✓							✓
Result									✓	✓
GenericTask									✓	✓
StockCommit Task									✓	✓
DBStockRequest Task									✓	✓
NewOrderDialog			✓							
Order	✓									
OrderListEntry			✓							
OrderList			✓							
Product	✓									
ProductList				✓						
FutureLevels	✓									
OrderNumbers			✓							
SlaveList			✓							
Conceptual Diffusion	4	N/A	7	4	N/A	1	1	1	6	7

Table 7. Analysis of Conceptual Diffusion Present in Mobile Code

7.3.2 Semantic Alignment

Conceptual Diffusion in itself is a measure of how well a software system is semantically aligned with those business processes it is trying to support. As it stands however, the conceptual diffusion measure remains relatively fine grained in its perspective. It does not offer an overall view of a system, rather an insight into a particular abstraction.

To gain an overall perspective of a system, a compound metric has been devised. It is a combination of metrics (1) to (4) and is termed the Semantic Alignment Metric:

$$SA = \left\{ \frac{Is}{Ir}, \frac{Ps}{Pr}, \frac{Ms}{Mr}, \frac{Ss}{Sr} \right\}$$

where SA is semantic alignment, *I* is information based abstractions, *P* is process based abstractions, *M* is mobile components, *S* is static components, *s* denotes in software and *r* denotes in the real world. Thus, $\frac{Ps}{\mathrm{Pr}}$ is the ratio of process-based abstractions in the software to the process based abstractions in the real world.

Mobile elements	Mobile agent	Mobile object
Order	✓	✓
Products		
Materials		
Static elements	**Mobile agent**	**Mobile object**
Sales	✓	✓
Stock Control	✓	✓
Production Ctrl		
Manufacturing	✓	✓
Purchasing		
Dispatch	✓	✓

Table 8. Results of Metrics (3) and (4)

This metric can be used to analyse a system and to assess how well the software system reflects the semantics of the application domain. A comparison with the ideal alignment of {1,1,1,1} can be used as a measure to gauge how difficult it might be to understand the software, given an understanding of the application domain. Table 8 shows the results of metrics (3) and (4).

By combining the results of the first four metrics, we are able to state that:

For the Mobile Object System Semantic Alignment = {4,22/6,1/3,2/3}

For the Mobile Agent System Semantic Alignment = {4,21/6,1/3,2/3}

7.3.3 Commentary

The results of the Conceptual Diffusion and Semantic Alignment analysis show that both Mobile Agent and Mobile Object systems should be easy to understand, as the abstractions in the real world align reasonably well with the components of the software systems. The information abstractions from the real world are on average spread over four components in the implementations. When considering mobile and static component alignment, for both systems, a third of the components in the domain are modelled as mobile in the implementation, and two thirds of the static components in the domain are modelled as static elements in the implementations.

The difference in the two systems is shown when considering the semantic alignment of the business process. Here the mobile agent system is shown to have better semantic alignment than the mobile object system as the process logic for the SOP is contained *solely* within the OrderAgent and not diffused across both the SalesAgent and the OrderObject. Therefore, we can conclude that the mobile agent solution provides better semantic alignment with the real world business processes it supports.

If we consider contemporary distributed systems, we find they have no facility to support mobile components in a system. Therefore, they would be unable to implement any of the mobile abstractions. Instead, these abstractions would have to be diffused over several static components. If we consider the requirement for a stub, skeleton and IDL file, in addition to the client and server implementations, then the conceptual diffusion would be considerable. Since mobile code systems are equally adept at building static components, we can also postulate that mobile code systems

increase the semantic alignment between the real world and its supporting software systems, for any system that is not constructed from completely static components.

In addition, these new metrics are not merely restricted to use after the fact, but can be used proactively during the specification process, before any software has actually been built. Ensuring good semantic alignment of a software system before production will undoubtedly save both time and money in the long term. In particular, these metrics can be useful for identifying those components that should be mobile, and those that should be static. With increasing numbers of mobile code systems being built, this will prove an increasingly important aspect of system analysis and design

7.4 Evaluating System Agility

In order to evaluate the agility of a system it is necessary to make changes to that system. The case study of I.T.L. highlighted several real-world industrial requirements for agility that a company may have for a distributed SOP system. Using these requirements as scenarios for change, modifications to both the mobile agent and mobile object implementations were undertaken, in order to evaluate the agility of each system.

7.4.1 Change Capability

The GQM methodology enabled the derivation of several metrics that can be used to measure certain changes in a software system after modification. These measurements are specified by metrics (8), (9), (10) and (11). Individually, they enable us to measure narrow slices of change to a system. However, by combining these metrics it is possible to produce a more encompassing measure of agility. This set has been termed Change Capability, and is described by:

$$CC_{\acute{a} \rightarrow \hat{a}} = \left\{ \sum_{\acute{a}}^{\hat{a}} \ddot{a}o, \sum_{\acute{a}}^{\hat{a}} \ddot{a}s, \sum_{\acute{a}}^{\hat{a}} \ddot{a}i, \sum_{\acute{a}}^{\hat{a}} \ddot{a}\mathring{a} \right\}$$

where Change Capability CC, for a required change, is the set of the changes to the number of objects (*o*), the number of src files (*s*), the number of interactions (*é*) and the number of conceptual entities (å), between states á and â. A conceptual entity is

analogous to the abstraction or concept referred to in the previous sections. For example, it could be an Order, or a StockControlAgent. Interactions are those exchanges of information between objects, usually via method invocations, although for agents this also applies to any messaging dialogue they might enter. Changes to those interactions will usually imply changing a method signature.

Change Capability can be used to compare systems or to get a measure of the agility of the system relative to the ideal {0,0,0,0}. For the mobile object and mobile agent systems Change Capability for each requirement is summarised in Table 9.

Industrial Requirement	System	
	Mobile Agent	Mobile Object
The addition of new sales agents	{0,0,0,0}	{0,0,0,0}
The addition of new stock control centres	{3,3,1,2}	{3,3,1,2}
The removal of new additions	As A or B	As A or B
Allowing changes to the business logic of the SOP to be made easily	{1,1,0,1}	{2,2,0,2}

Table 9. Change Capability metric sets after "scenarios for change"

7.4.2 Commentary

Again, these results show that both systems are relatively easy to change. Adding new sales facilities requires only the instantiation of new SalesAgents that incurs zero changes to the system code. New stock control centres can be added through a low number of changes that are the same for both systems. The difference between the systems becomes apparent when making changes to the Sales Order Process logic. In the mobile agent system, this logic is contained *solely* in the single mobile OrderAgent, whereas in the mobile object system it is contained in both the SalesAgent and the OrderObject.

The Change Capability metric can be used by a system designer to evaluate how responsive to change their system has been after a specific change. It is possible to

deduce areas that require refactoring, or are particularly troublesome when undertaking change. For example, consider the CC set {5, 20, 20, 1}. We see that for this change, although only one conceptual entity was changed, there were twenty changes to source files, five changes to objects, and twenty changes to the interactions of those objects. Changing the signature of twenty methods in five objects to enable a change in a single entity can cause serious problems and should lead the designer to review how diffuse this particular entity actually was. Of course, this is also revealed by the Conceptual Diffusion metric.

While both implementations have demonstrated they are relatively agile, the question of whether they are more agile than a contemporary distributed system remains open. Certainly, it is unlikely that a traditional system will be any more agile than the mobile object system, since Remote Computation and Client/Server are very close in terms of the abstraction they offer. Nevertheless, we are able to assert that the mobile agent system has shown that it is more agile than the mobile object system. This increased agility was due to the reduced conceptual diffusion and improved semantic alignment that the mobile agent abstraction allows. In the next section, we pursue this matter by examining loose coupling, a central issue to building agile software systems.

7.5 Evaluating Loose Coupling

To build loosely coupled systems, components of that system should not be linked directly to form a complex network of interactions and inter-dependencies. Instead, they should remain distinct abstractions, embodying the concept of their real world equivalents. Components can then be assembled into a software system, with no prior knowledge of each other.

7.5.1 Evaluating Coupling in Mobile Code Systems

We have already seen in the preceding sections that distributed systems built with mobile code are able to minimise conceptual diffusion. This enables an extremely good alignment between real world processes and their supporting software counterparts. On examination of the static software entities in our systems, for example SalesAgents, StockControlAgents, ManufacturingAgents, etc, we find that

they are fully decoupled from each other. During execution of the system, there is no communication or interaction between any of the static components. Any communication that does take place within the systems is between static and mobile entities. Until a mobile entity alights at a host and attempts to interact with a static one, there is no coupling between any of the components. This is significant, since the system only experiences tighter coupling during a dialogue between components, i.e. when a mobile entity wishes to communicate with a static one. Of course, this dialogue depends upon prior knowledge on the part of the mobile entity as to what language the other agent understands, be it a syntactic dialect, or a more complex semantic conversation. In a private, controlled system however, this knowledge will always be available. In addition, since there are very few types of component that are mobile it is simple to alter the interactions, by updating the mobile agent population. Research is being undertaken so a dialogue may be established with no foreknowledge [Martin99]. Although this is currently in the static, intelligent agents domain, in time it will naturally be applied to that of mobile agents.

7.5.2 Commentary

Our prototype systems have demonstrated extremely low, if not non-existent, component coupling until runtime. Contemporary distributed systems such as CORBA do support loose coupling in the same inherent manner [Coutts98b]. Components in these systems that wish to communicate require implicit knowledge of each other's interfaces. These interfaces are the central aspect of building distributed systems with traditional technology.

> "You should be able to look only at the IDL and know precisely how to implement against it." [Vinoski99]

Therefore, even if the key conceptual abstractions remain embodied in large grained components, for these components to interact they must be aware of each other *a priory*, and inevitably end up intermeshed with each other. The work of Coutts and Edwards has shown that it is possible to build loosely coupled systems with traditional technology by employing additional design patterns and forethought. The author believes that being required to follow this enforced route is simply increasing the cognitive complexity of building distributed systems. Something that is already an onerous task.

This circumstance arises since location transparency, the abstraction employed in contemporary distributed systems, does not support loose coupling inherently. Distributed systems built with this abstraction rely on component interface signatures for identification, and to facilitate communication. Coutts and Edwards [Coutts98b] have demonstrated that with further software architectures a certain degree of loose coupling can be achieved. Their use of the Mediator pattern has one drawback however – all components that wish to interact must do so via the Mediator. The strength of this approach is also its main weakness. By enforcing a policy of mediation, the distributed system is also subjected to centralised control, and thus the Mediator is a single point of failure. Building distributed software systems with a single point of failure is known as a bad technique.

In a contemporary distributed system the concept of physical location is hidden. However, for two components to interact there must be some form of identification involved. This identification manifests itself through the interface types of the interacting components. Therefore, in reality the purpose of identification by interface is to enable the location of a component that can provide the required services. The core information in the task of locating a component is no longer physical location, rather it is the interface. Although the major tenet of this abstraction is location transparency, it is clear that the task of locating components remains. It has merely been replaced by an alternative method. Of course, practitioners of contemporary distributed systems argue that location transparency as provided by the abstraction is for the benefit of those who build and use the system. This may be the case, but we must also consider the implications of using this abstraction on the supporting technology, i.e. the distribution infrastructure.

Distributed System Technology	Locator Requirement	Dialogue Requirement
Traditional Technology	Interface	Interface
Mobile code systems	Location	Interface

Table 10. Requirement of Distributed Systems

On the other hand, components in distributed systems built with the local interaction abstraction do not rely on interface signatures to be located. Instead, they employ physical location as the information required for location. This is an important difference. By retaining location as the locator, the mobile code abstraction divorces the distribution mechanism from the dialogue constraints. This is shown in Table 10.

This separation has important implications for how tightly coupled a system might be. By divorcing distribution from dialogue, distributed systems can be much more loosely coupled until runtime. At the outset, all that two components who wish to communicate must know about each other is their respective locations. It is only when they actually wish to interact that they become more tightly coupled. The difference to contemporary technologies is in the timing of when it is required.

The implications of this subtle change are fundamental. System agility is affected by the coupling of components within a system, and in this respect, we argue that local interaction does indeed support looser coupling than traditional distribution technologies. By divorcing the mechanism for distribution from the dialogue, components in a system can be loosely coupled right up until the moment of interaction. Although once engaged in dialogue the components become tightly coupled, the moment of coupling has been delayed. Therefore, we may conclude that mobile code systems are more loosely coupled, and this looser coupling enables improved system agility when compared with traditional distribution technology.

The important issue to understand is why there are such marked differences between the abstraction offered by current distribution technologies and that offered by mobile code. In chapter one we examined the history of computing and saw how the computing landscape we inhabit today has been formed through the gradual layering of ascending abstractions. This is not a problem, since abstractions are an extremely useful tool for reducing the complexity of a situation, removing the minutiae so one might contemplate the problem at hand with clarity. However, what is important about abstraction is the importance of using an appropriate one. One that is able to accurately describe the real situation, without losing any important information.

It has been the author’s belief that the major tenet of RM-ODP systems, that of location transparency, is fundamentally flawed in this respect. The first notion of this

abstraction arose when Birrel and Nelson attempted to take the extremely successful abstraction of IPC, and apply it to many networked machines, in order to make local and remote calls look identical. This philosophy has prevailed and been extended so that we currently employ an abstraction that attempts to make every object or component in a distributed system believe they are executing in the same computing machine. However, by attempting to "shoehorn" an abstraction that was perfectly suited for the underlying hardware, i.e. a single von Neumann machine, onto many computing machines an important piece of information has been lost from the abstraction – location. Waldo et al identify several problems of distributed systems but do not offer a clear reason for these problems. We propose that it is due to the loss of location from the distribution abstraction. Identification of components in the network can no longer be achieved via their location, instead they must be identified by their interface signatures.

The assertion of the author is that although this technology can indeed build successful distributed systems, the drawbacks do not warrant the effort. The price for using the interface as a locator is tightly coupled systems that are difficult to change. Instead of enabling location transparency, mobile code systems enable local interaction, an abstraction ideally suited to single von Neumann machines. By using physical location as a locator, mobile code systems are able to separate the issues of distribution from the issues of dialogue, and thus these systems are more loosely coupled. Additionally, they provide improved semantic alignment, and thus reduce the cognitive complexity of the system.

Employing the correct abstraction can have fundamental consequences to building distributed systems. Instead of a flat plane of components that all believe they are in the same host, the mobile code abstraction removes this opacity of RM-ODP and exposes the rich network environment.

7.6 Concluding Remarks

Evaluating software systems is never an easy task. The evaluation in this thesis has been undertaken following Basili's GQM methodology. Using this technique a set of tangible metrics was developed to assist in the evaluation of the two mobile code systems. The motivation for the experimental work carried out in this thesis was to

demonstrate the feasibility of actually building distributed systems with mobile code technology, and to investigate the implications for system agility when using this new paradigm.

We initially examined the issue of semantic alignment and compared our two prototype systems. The experimental work has shown that by reducing the conceptual diffusion in a system, the mobile agent abstraction is able to offer improved semantic alignment with the business process it is intended to support when compared to the mobile object system. The difference is barely significant in our systems, but could easily be magnified in a full size system. In the process of this evaluation, two software metrics have been developed to assist the system designer in identifying which components, if any should be mobile.

On examination, system agility is a harder issue to resolve. The experimental work has shown that mobile code systems are relatively agile, with the mobile agent abstraction being slightly more so than the mobile object abstraction. The differences in each implementation with respect to agility are identical to the differences in semantic alignment. This is due to lower conceptual diffusion in the mobile agent system, something that is enabled by the autonomy of the agent metaphor.

When looking at loose coupling we see no difference between the mobile object and mobile agent prototypes. However, in general component coupling in these systems is extremely low. This is in marked contrast to distributed systems built with the location transparency abstraction. Although our work does not shed any further quantitative light onto this matter, our observations do support the argument made in Part I of this thesis: that location transparency is fundamentally flawed. Our conclusion is that this is further exacerbated by combining the information used for location of components with that required for a dialogue. Local transparency on the other hand separates these two issues, and is thus able to build more loosely coupled systems that are more responsive to change.

8 Conclusions

Building distributed systems is not a new endeavour. We have been doing so for as long as we have been networking computers. However, the types of system being built, and the nature of the underlying network are evolving beyond the wildest dreams of the early network pioneers. Networks are becoming pervasive in society, and the dream of ubiquitous computing is finally being realised. These new networks bring new requirements for how we build distributed systems. We can no longer guarantee network reliability or even topology. Our existing technologies and infrastructures are beginning to creak under the strain.

This thesis has been concerned with how we build distributed systems. Instead of focusing merely on the technology used to implement them, we have also focused on the abstractions employed in their construction. These immensely powerful concepts allow us to manage the complexity of a situation, by removing those details we consider inessential. After all, the central essence of any paradigm is the abstractions it embodies. The major contributions of this thesis have been:

- An extensive philosophical argument and critique of abstractions for distribution
- The demonstration of the feasibility of building real-world distributed systems with mobile code infrastructures
- The creation of the new software metrics of Conceptual Diffusion, Semantic Alignment and Change Capability
- Quantitative comparisons of the Mobile Agent and Remote Computation abstractions

In Part I, *Understanding*, we traced the emergence of abstractions in computing, and built a philosophical understanding and critique of the abstractions used to construct distributed software systems. The central thesis of this work is that by employing the location transparency abstraction, and attempting to create the illusion that all components exist within the same computational machine, contemporary distributed systems are fundamentally flawed as they break the Tower of Abstractions by attempting to impose an unsuitable abstraction on the underlying computational substrate. We have demonstrated that location transparency was a wrong fork in the evolutionary road of distribution. Our proposal is that a new abstraction, local interaction (embodied in mobile code infrastructures), that returns to the core

successes of the von Neumann computational machine is a more suitable abstraction with which to build distributed systems in today's ubiquitous networks. Removing location from the abstraction has proven detrimental to the agility of systems built with this technology, since the issues of distribution have become tied with those of dialogue. Whilst we advocate the use of abstraction, we believe that location transparency loses essential information when employed. We believe that Part I of this thesis contributes by raising the level of conceptual understanding surrounding the mobile code paradigm.

The arguments presented in Part I are extensive, and a full experimental investigation was deemed beyond the scope and timescale of a PhD. Instead, our horizons were shortened to encompass the first steps along the long path of validating the argument. Part II, *Using and Evaluating,* is therefore a report on our experiences of mobile code in the real world. To date, the mobile code research arena has remained relatively immature, and the dearth of real systems has hampered its development. With this in mind, our experimental work was based upon a business process model generated from an industrial case study. We reported on the creation of two prototype systems that embodied the Mobile Agent and Remote Computation abstractions, part of the mobile code family of abstractions. In this, we have achieved our first aim; to demonstrate the feasibility of building real world distributed systems with mobile code. We also wish to comment on the relative merits of each prototype.

In the course of the experimental work, we subjected our systems to real world pressures in the form of Scenarios for Change, also generated from the case study. During the subsequent evaluation, we developed several metrics using the Basili GQM methodology. The metrics of Conceptual Diffusion, Semantic Alignment and Change Capability have proved to be useful techniques for evaluation that can be used during both the specification process, and post construction. In addition, we have tried to consider the full lifecycle of our systems, an exercise that has produced several supporting tools and proto-patterns.

Our evaluation of the two mobile code prototypes draws us to conclude that the mobile agent abstraction is the more useful to employ. From our experiments, we observe that mobile agents enjoy increased semantic alignment and system agility when compared to the remote computation abstraction. The differences in each

implementation arise due to the lower conceptual diffusion of the mobile agent system, something that is enabled by the autonomy of the agent metaphor.

We believe that this thesis is a beginning, an initial monograph on abstractions for distribution. It is clear that location transparency is unsuitable for some types of system we wish to build, and that mobile code offers a viable alternative. This is not to say that all distributed systems should be built with mobile code. Mobile agents offer us a solution for networks where topology, quality of service and varying bandwidth are the core issues. We should appreciate the nuances of each abstraction, so that we may apply them in the correct situation.

8.1 Future work

As has been mentioned, the arguments made in Part I are extensive, and their scope beyond that which can be considered in the timescale of a PhD. This is not to say we have not contemplated what would be required. The experiments described in this thesis have been a first step. We have demonstrated the viability of mobile code, and our results indicate that the mobile agent abstraction supports good system agility. The question of whether mobile code technology is superior to contemporary technology remains open. It is very difficult to compare the two, since the maturity levels of the technologies differs greatly. Distributed systems built around the RM-ODP model have been around for over a decade with much industry development, whilst mobile agent systems have been around merely a few years.

We believe the next stage of validation for our philosophical argument would be to undertake a course of research to directly compare Mobile Agents with RM-ODP. To avoid the differences in technology maturity, we envisage building each abstraction from the ground up. A clean room implementation of both abstractions would allow a more valid and comprehensive comparative analysis. Further, it is clear that software patterns and software metrics evolve throughout their lifetime. Through use, practitioners are able to refine them. We believe additional software metrics would support this investigative work.

As has already been mentioned, an obvious avenue for future work would be to continue the SOP implementation undertaken in this thesis. The current model

embodied in our prototypes has many areas where it can be expanded. Increasing the size and complexity of our systems would allow us to reapply the scenarios for change. A comparative study with our current results would be a valuable exercise to ascertain how much of an effect size and complexity has on system agility. We should also be searching for collaborative partners on other continents to truly test how successfully each system supports distribution.

Finally, the creation of a modelling language that includes the facility to specify mobile components would be an invaluable addition to the system designer's toolbox. Current modelling languages, such as UML [Booch97], do not include the concept of mobility. Extending *de facto* industry methodologies is a sure fire way to ensure widespread adoption of new ideas and technologies.

8.2 Commentary

Using mobility is not just about what the technology can do for you. It is also about a fundamental change of mindset. By removing the conceptual block that is the *plane of transparency*, distributed systems designers can begin to appreciate the rich environment that is the network. If we remain faithful to the Tower of Abstractions, and employ the network as our communications infrastructure, we draw on the strengths of the von Neumann machine *and* the network suite, whilst divorcing the issues of distribution from those of dialogue.

In hindsight, it is easy to illustrate the reasons our computing evolution meandered down the location transparency fork. Recently an expanding community has realised there are problems with this approach. As a software engineering community in the large, we must be brave enough to face up to those problems, and admit our mistakes. It is better to attack the problem as early as possible, than build ever more elaborate software constructs to support a dying abstraction. The ideas generated during the work undertaken in this thesis have allowed the author to view distribution from a different perspective. Local interaction is beginning to establish itself as a valid tool for building earthbound distributed systems, but it has already been considered for perhaps the ultimate distributed system - a space based network [Papaioannou99c]. There can be no question of location transparency being employed when the distances involved in this type of network are considered!

Mobile agents have shown considerable early promise. The future they depict is one of a rich network environment, inhabited by an ecology of autonomous agents. Nodes in the network become islands of resources, on which agents may alight to take advantage of resources locally. The population consists of mobile and static agents, all enjoying some level of autonomy, ranging from simple task specific instructions, to complex autonomous agent architectures. The mobile agents live in the network, able to migrate, clone, sleep, wake, but in reality insert a new layer of abstraction over the underlying computation substrate. They act for other agents, or their human owners. The static agents are brokers for immovable resources such as printers or databases. In this virtual ecology, we see the glimpses of our future computing.

List of Publications

Clements, P.E., Papaioannou, T. and Edwards, J.M., "Aglets: Enabling the Virtual Enterprise", Proceedings of the 1st International Conference on Managing Enterprises - Stakeholders, Engineering, Logistics and Achievement, ME-SELA '97, Wright, Rudolph, Hanna, Gillingwater and Burns (eds), Mechanical Engineering Publications, Loughborough University, July 1997,pp 425-432, ISBN 1-86058-066-1

Papaioannou, T., Edwards, J.M., “Mobile Agent Technology Enabling the Virtual Enterprise: A Pattern for Database Query”, in notes of Agent Based Manufacturing Workshop, part of the International Technical Conference Autonomous Agents '98.

Papaioannou, T., Edwards, J.M., “Using Mobile Agents To Improve the Alignment Between Manufacturing and its IT Support Systems”, International Journal of Robotics and Autonomous Systems, 27, pp 45-57, 1999.

Papaioannou, T., Edwards, J.M., “Mobile Agent Technology in Support of Sales Order Processing in the Virtual Enterprise”, in [Camarinha-Matos et al]

Papaioannou, T., “Mobile Agents: Are They Useful for Establishing a Virtual Presence in Space?”, in notes of Adjustable Autonomy Symposium, part of the AAAI Spring Symposium Series, Stanford University, 1999.

Papaioannou, T., Minar, N., “Mobile Agents in the Context of Competition and Cooperation”, Proc. of MAC3 workshop, part of Autonomous Agents ‘99 conference, Seattle, 1999.

Papaioannou, T., Edwards, J.M., “Manufacturing Systems Integration and Agility: Can Mobile Agents Help?”, accepted for publication in Journal of Integrated Computers-Aided Engineering, IOS Press. To appear in January 2001 Issue.

Papaioannou, T., Edwards, J.M., “Towards Understanding and Evaluating Mobile Code Systems”, accepted for publication in Journal of Autonomous Agents and Multi-Agent Systems, Kluwer Academic Publishers. To appear in 2000.

References

Abadi96 Abadi, M., and Cardelli, L., “A Theory of Objects”, Monographs in Computer Science, Springer-Verlag, Berlin, 1996.

Accetta86 Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A., Young, M., “MACH: A New Kernel Foundation for UNIX Development”, Proc. Summer USENIX Conference, pp 93-112, 1986.

Adobe85 Adobe Systems Inc., “The Postscript Language Reference Manual”, Addison-Wesley, 1985.

Agha97 Agha, G., “Abstracting Interaction Patterns: A Programming Paradigm for Open Distributed Systems”, in Najm, E. and Stefani, J.B., Eds, “Formal Methods for Open Object-based Distributed Systems”, Chapman & Hall, 1997

Andrews82 Andrews, G.R., “The distributed programming language SR – mechanisms, design and implementation”, Software Practice and Experience, Vol 12, pp 719-753, 1982

Andrews83 Andrews, G., Schneider, F., “Concepts and Notations for Concurrent Programming”, ACM Computing Surveys, 15, pp 3-43.

Apple92 Apple Computers, “Dylan, an Object Oriented Dynamic Language”, Apple, Cupertino, CA, 1992.

Arnold99 Arnold, K., Wollrath, A., O’Sullivan, B., Sheifler, R., Waldo, J., “The Jini Specification”, Addison-Wesley, 1999.

Backus78 Backus, J., "Can Programming be Liberated from the Von Neumann Style?”, Comm. ACM 21 (8), pp. 613-641.

Ball98 Ball, K., McClain, D., Minium, D., 1997, "Enterprise Enablement for Java Applications", XDB SystemsReferences

Barber98 Barber, M., Weston, R., "BPR Scoping Paper", IJPR, 1998.

Barlow97 Interview with the Managing Director of I.T.L., Mr David Barlow, 1997.

Basili94 Basili, V.R., Caldiera, G., Rombach, H.D., (1994), “The Goal Question Metric Approach”, Encyclopedia of Software Engineering, pp 528-532, Wiley and Sons.

Baumann97 Baumann, J., Hohl, F., Rothermel, K., “Mole – Concepts of a Mobile Agent System”, Technical Report No 1997/15, Faculty of Computer Science, Stuttgart, Germany, 1997.

Ben-Ari90 Ben-Ari, M., “Principles of Concurrent and Distributed Programming”, Prentice-Hall, Englewood Cliffs, NJ, 1990.

Bennet94 Bennett K.H., Ward M.P., 'Using Formal Transformations for the Reverse Engineering of Real-time Safety Critical Software' Proc. Second Safety-Critical Systems Symposium, Birmingham, 1994, pub. Springer-Verlag, ISBN 0-387-19859-8, pp. 204 –223

Berners-Lee92 Berners-Lee, T.J., Cailliau, R., Groff, J.-F., Pollerman, B., "World-Wide Web: The Information Universe.", in *Electronic Networking: Research, Applications and Policy*, Vol 2 (1), pp 52-58, Westport CT: Meckler Publishing.

Berners-Lee92b Berners-Lee, T., Fielding, R., Masinter, L., "Uniform Resource Identifiers (URI): Generic Syntax", available at http://www.ietf.org/rfc/rfc2396.txt

Birrel84 Birrel, A.D., Nelson, B.J., "Implementing remote procedure calls", ACM Transactions on Computer Systems, Vol 2, pp 39-59, 1984

Birtwistle73 Birtwistle, M. G., Dahl, O. J., Myhraug, B., Nygaard, K., "Simula Begin", Petrocelli/Charter, New York, 1973.

Blair91 Blair, G.S., et al. "Object-Oriented Languages, Systems and Applications", Pitman, London UK, 1991, cited in [Coutts98]

Bobrow88 Bobrow, D.G., De Michiel, L.G., Gabriel, R.P., Keene, S.E., Kiczales, G, and Moon, D.A., "Common LISP object system specification", ACM SIGPLAN Notices, 23, September, 1988.

Boehm76 Boehm, W., Brown, J.R., Lipow, M., "Quantitative Evaluation of Software Quality", Proc. 2nd International Conference on Software Engineering, 1976, pp 592-605.

Boggs73 Boggs, J.K., "IBM Remote Job Entry Facility: Generalised Subsystem Remote Job Entry Facility", IBM Technical Disclosure Bulletin, 752, August 1973.

Booch94 Booch, G., "Object Oriented Analysis and Design with Applications", Redwood City, CA: Benjamin/Cummings, 1994

Booch97 Booch, G., Rumbaugh, J., Jacobsen, I., "Unified Modelling Language Semantics and Notation Guide 1.0", Rational Rose Software Corporation, CA, 1997.

Brener87 Brenner, J.B., "Open distributed processing", ICL Technical Journal, Vol. 5 (4), pp 613-637, 1987

Brooks95 Brooks, F.P. Jr, "The Mythical Man-Month: Essays on Software Engineering", Addison-Wesley, Reading, MA, 1995.

Burks46 Burks, A.W., Goldstine, H.H., von Neumann, J., “Preliminary Discussion of the logical Design of an Electronic Computing Instrument”, U.S. Army Ordinance Dept. Report, 1946.

Callear94 Callear, D., “Prolog Programming for Students”, Ashford Colour Press, England, 1994.

Camarinha-Matos98 Camarinha-Matos, L. M., Vieira, W., “Using Multiagent Systems and the Internet in Care Services for the Ageing Society”, appearing in [Camarinha-Matos et al], 1998.

Camarinha-Matos et al Camarinha-Matos, L. M., Afsarmanesh, H., Marik, V., eds. "Intelligent Systems for Manufacturing: Multi-Agent Systems and Virtual Organisations", Kluwer Academic Publishers, 1998, ISBN 0-412-84670-5

Cardelli85 Cardelli, L., and Wegner, P., “On understanding types, data abstraction, and polymorphism.”, ACM Computing Surveys, 17 (4), pp 471-522, 1985.

Carrot97 Carrot, A.J., Wright, C.D., West, A.A., Harrison, R., "Creating a distributed object-oriented integration framework for machine design and control ”, First International Conference on Managing Enterprises-Stakeholders, Engineering, Logistics & Achievement (ME-SELA ‘97) at Loughborough University, 22-24 July 1997.

Carver91 Carver, G. P., Bloom, H.M., "Concurrent Engineering through Product Data Standards", U.S. Department of Commerce, May 1991.

Carzaniga97 Carzaniga, A., Picco, G.P., Vigna, G., “Designing Distributed applications with Mobile Code Paradigms”, Proc. 19th International Conf. On Software Engineering (ICSE’97), 1997, Taylor, R., Ed., ACM Press, pp 22-32.

Cashin80 Cashin, P.M., “Inter-Process communication”, Bell-Northern Research Report, May 1980.

Cerf74 Cerf, V. and Kahn, R., “A protocol for Packet Network Interconnection”, IEEE Trans. on Communication, Vol. COM-22, pp 637-648, 1974.

Cerutti83 Cerutti, D., Pierson, D., “Distributed computing environments”, McGraw-Hill, 1993

Cheong83 Cheong, V.E., “Local Area Networks”, Wiley and Sons, 1983.

Chess97 Chess, D., Harrison, C., Kershenbaum, A. "Mobile Agents: Are They A Good Idea ?", in "Mobile Object Systems, Towards one programmable Internet", Edited by Vitek, J., Tschudin, C., Springer-Verlag Lecture Notes in Computer Science 1222, 1997, ISBN-3-540-62852-5.

Chomsky59 Chomsky, N., “On Certain Formal Properties of Grammers”, Information and Control, 2 (2), pp 137-167, 1959, cited in [Coutts98]

Church41 Church, A., "The calculi of lambda conversion.", Annals of Mathematics Studies, 6, Princeton University Press, Princeton NJ, 1941.

Clements97 Clements, P.E., Papaioannou, T. and Edwards, J.M., "Aglets: Enabling the Virtual Enterprise", Proceedings of the 1st International Conference on Managing Enterprises - Stakeholders, Engineering, Logistics and Achievement, ME-SELA '97 , Wright, Rudolph, Hanna, Gillingwater and Burns (eds), Mechanical Engineering Publications, Loughborough University, July 1997, pp 425-432, ISBN 1-86058-066-1.

Clocksin87 Clocksin, W.F., Mellish, C.S., “Programming in Prolog”, 3rd edition, Springer-Verlag, 1987.

Comer91 Comer, D., “Internetworking with TCP/IP Volume I: Principles, Protocols, and Architectures”, 2nd Edition, Prentice Hall, 1991

Coulouris94 Coulouris, G., Dollimore, J., Kindberg, T., “Distributed Systems: Concepts and Design (2nd Edition)”, Addison-Wesley, 1994

Coutts98 Coutts, I., A., "An Infrastructure to Support the Implementation of Distributed Software Systems", doctoral thesis (to be published), Loughborough University, 2001.

Coutts98b Coutts, I.A., Edwards, J.M., “Support for Component Based Systems: Can Contemporary Technology Cope?”, in [Camarinha-Matos et al], 1998.

Cox87 Cox, B. J., "Object Oriented Programming - An Evolutionary Approach", Addison-Wesley, Wokingham, UK, 1987, cited in [Coutts98]

Cox98 Cox, B. J., Opinion expressed in private correspondence via email, 1998.

Crichlow88 Crichlow, J.M., “An Introduction to Distributed Parallel Computing”, Prentice-Hall, 1988.

Cypser78 Cypser, R., “Communications Architectures for Distributed Systems”, Addison-Wesley, 1978.

DeMarco78 T. DeMarco, *Structured Analysis and System Specification,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978

DeRemer76 DeRemer, F., Kron, H.K., “Programming in the Large Versus Programming in the Small”, IEEE Transactions on Software Engineering, SE-2 (2), pp 80-86, 1976.

Dijkstra68 Dijkstra, E.W., "Goto Statement Considered Harmful", Comm. ACM, 24, pp. 147-148, 1968.

DoD61 Department of Defense, "COBOL, Revised Specification for a Common Business Oriented Language", 196.

DoD80 Department of Defense, "Ada Programming Language", Report MIS-STD-1815, Washington D.C., 1980.

DoD80b USA Department of Defence, “Reference Manual for the Ada Programming Language”, Proposed Standard Document, 1980.

Einstein39 Einstein, A., Infeld, L., “The Evolution of Physics”, 2nd Edition, Simon and Schuster, NY, 1960.

Franklin96 Franklin, S and Graesser, A., 1996, "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents", Proceedings of the 3rd Int. Workshop on Agent Theories, Architectures, and Languages, Published as Intelligent Agents III Springer-Verlag , Berlin, 1997, pp 21-35.

Fukuoka82 Fukuoka, H., “Interprocess communication facilities for distributed systems: a taxonomy and a survey”, Research Report, Georgia Institute of Technology, GIT-ICS-82/06, 1982.

Gascoigne94 Gascoigne, J.D., “CIM-BIOSYS Integrated System Implementation Toolset”, MSI Research Institute, Loughborough University, England, 1994.

Gershenfeld99 Gershenfeld, N., “When Things Start to Think”, Henry Holt & Company, 1999, ISBN 0805058745. in [Minar99]

Geschke77 Geschke, C. M., Morris, J. H. Jr., Satterthwaite, E. H., "Early experiences of Mesa", Comm. ACM, 20 (8), pp. 540-553, 1977.

Ghezzi98 Ghezzi, C., Jazayeri, M., “Programming Language Concepts”, 3rd ed., Wiley and Sons, 1998.

Glass99 Keynote speech given by Graham Glass, CTO of ObjectSpace at Autonomous Agents 99, Seattle, May 1999.

GoF93 Gamma, E., R. Helm, R. Johnson, & J. Vlissides, ``Design Patterns: Abstraction and Reuse of Object-Oriented Designs", *Proceedings, ECOOP '93*, Springer-Verlag, 1993.

Goldberg83 Goldberg, A., Robson, D., “Smalltalk-80: the Language and Its Implementation”, Addison-Wesley, Reading, MA, 1983.

Goldman95 Goldman, S.L., Nagel, R.N., Preiss, K., “Agile Competitors and Virtual Organisations”, Van Nostrand Reihold Publishing, 1995.

Gong99 Gong, L., “Inside Java 2 Platform Security: Architecture, API Design, and Implementation”, Addison-Wesley, 1999. ISBN: 0201310007

Goodenough75 Goodenough, J.B., “Exception handling: Issues and proposed notitation”, Comm. ACM, 16 (12), pp 683-696, Dec. 1975.

Gosling96 Gosling, J., Joy, B., Steele, G., "The Java Language Specification", Addison-Wesley, Reading, MA, 1996.

Gray83 Gray, J.P., Hansen, P.J., Homan, P., Lerner, M.A., Pozefsky, M., “Advanced program-to-program communication in SNA”, IBM Systems Journal, Vol. 22 (4), pp 298-318, 1983.

Gray97 Gray, R., “*Agent Tcl: A flexible and secure mobile-agent system*”, PhD thesis, Dept. of Comp Sci, Dartmouth College, June 1997.

Green80 Green, P.E. Jr, “An Introduction to Network Architectures and Protocols”, in [IEEE80].

Hammer93 Hammer, M., Champy, J., “Re-engineering the corporation”, Nicholas Braedly Publishing, London, 1993.

Harel87 Harel, D., “The science of computing: exploring the nature and power of algorithms, Addison-Wesley, USA, 1987.

Harel93 Harel, D., “Algorithmics: The Sprit of Computing 2nd Editions”, Addison-Wesley, 1993.

Hoare72 Hoare, C.A.R., “Notes on data structuring”, Structured Programming, Academic Press, pp 83-174, 1972

Hoare74 Hoare, C.A.R., “Monitors: An operating system structuring concept”, Comm. ACM, Vol. 17 (10), pp 549-557, 1974

Hoare78 Hoare, C.A.R., “Communicating sequential processes”, Comm. ACM, Vol. 21 (8), pp 666-677, 1978

Hodgson97 An interview with the Head of IT at I.T.L., Mr Richard Hodgson, 1997.

Hopper68 Hopper, G. M., Keynote Address at the inaugural History of Programming Languages conference, June 1-3, 1978, cited in [Wexelblat81].

Hopson96 Hopson, K.C., Ingram, S.E., Chan, P., “Developing Professional Java Applets”, Sams Publishing, 1996, ISBN: 1575210835

Horowitz83 Horowitz, E., “Fundamentals of Programming Languages”, Springer-Verlag, 1983.

Hudak89 Hudak, P., “Conception, Evolution and Application of Functional Programming Languages”, ACM Computing Surveys, Vol 21, pp 359-411, 1989.

IBM56 IBM Corporation, "Programmer's Reference Manual, The FORTRAN Automatic Coding System for the IBM 704 EDPM", 956.

ICSE99 Proceedings of 21st International Conference on Software Engineering, “Preparing for the Software Century”, ACM PRES 1999, ISBN: 1-58113-074-0

IEEE80 IEEE Transactions on Communications, Special Issue on Computer Network Architectures and Protocols, Vol. 28 (4), April 1980.

ISO83 International Standards Organisation, “Basic Reference Model for Open Systems Interconnection”, ISO 7498, ISO, 1983

ISO90 International Organisation for Standardisation. Pascal. Technical report ISO 7185. ISO Geneva, 1991.

ISO92 International Standards Organisation, “Basic Reference Model of Open Distributed Processing, Part 1: Overview and guide to use”, ISO/IEC JTC1/SC212/WG7 CD 10746-1, ISO, 1992

Iverson62 Iverson, K.E., “A Programming Language”, Wiley and Sons, 1962.

Jennings98 Jennings, N.R., Sycara, K.P. and Wooldridge, M., “A roadmap of agent research and development.”, in Autonomous Agents and Multi-Agent Systems, 1, pp 7-38, Kluwer Academic Publishers, 1999.

Johansen99 Johansen, D., interview in [Milojicic99], IEEE Concurrency, 1999.

Johansen99b Johansen, D., “Mobile Agent Applicability.”, In, Proceedings of the Mobile Agents 1998, Springer-Verlag LNCS series, Stuttgart, 9-11 September, 1998. Also in, Journal of Personal Technologies, Springer-Verlag, Vol 2, No. 2, 1999.

Jones83 Jones, M.B., Rashid, R.T., “Mach and Matchmaker: kernel and language support for object-oriented distributed systems”, ACM SIGPLAN Notices, Vol. 21 (11), pp 67-77

Jones97 Jones, M. Dr., Given in a presentation at the EPSRC Methodology Workshop held at Cambridge University, UK, 1997.

Jul88 Jul, E., Levy, H., Hutchinson, N., Black, A., “Fine-grained Mobility in the Emerald System”, ACM Transactions on Computer Systems, Vol 6 (2), 1988, pp 109-133.

Kernighan78 Kernighan, B.W., Ritchie, D.M., “The C Programming Language”, Prentice Hall, 1978.

Kiczales97 Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J-M., Irwin, J., “Aspect-Oriented Programming”, Proc. European Conf. on OOP (ECOOP), Springer-Verlag LNCS 1241, 1997

Knabe96 Knabe, F.C., “Language and compiler support for mobile agents”, PhD thesis, Carnegie Mellon University, 1996.

Knuth74 Knuth, D.E., “Structured programming with goto statements”, ACM Computing Surveys, 6 (4), pp 261-301, Dec. 1974

Kogure83 Kogure, M., Akao, Y., “Quality Function Deployment and CWQC in Japan”, Quality Progress, October 1983, pp 25-29.

Kotz99 Kotz, D., Gray, R.S., “Mobile code: The Future of the Internet”, in [Papaioannou/Minar99], 1999.

Kowalski79 Kowalski, R.A., “Logic for Problem Solving”, North-Holland, Amsterdam, 1979.

Kramer83 Kramer, J., Magee, J., Sloman, M., Lister, A., “CONIC: an integrated approach to distributed computer control systems”, IEE Proceedings, Part E, Vol 130 (1), pp 1-10, 1983.

Labrou94 Labrou, Y., Finin, T., “A Semantics Approach for KQML – a General Purpose Communication Language for Software Agents”, in proc. 3rd Int’l Conf. On Information and Knowledge Management (CIKM’94), 1994.

Lampson77 Lampson, B., Mitchell, J., Satterthwaite, E., "Report on the programming language Euclid", SIGPLAN Notices, 12 (2), Feb 1977

Lange98 Lange, D.B., Oshima, M., "Mobile Agents with Java: The Aglet API", World Wide Web Journal, 1998.

Lea93 Lea, R., Jacquement, C., Pillevesse, E., “COOL: System Support for Distributed Object-Oriented Programming”, Comm. ACM, Vol 36 (9), 1993, pp 37-46.

Leffler89 Leffler, S., McKusick, M., Karels, M. and Quartermain, J., “The Design and Implementation of the 4.3 BSD Unix Operating System”, Addison-Wesley, 1989

Lindholm99 Lindholm, T., Yellin, F., “The Java Virtual Machine Specification 2nd Edition”, Addison-Wesley, 1999. ; ISBN: 0201432943

Liskov81 Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, J.C., Sheifler, R. and Snyder, A., “CLU Reference Manual”, Springer-Verlag, 1981.

Liskov83 Liskov, B., Sheifler, R., “Guardians and actions: linguistic support for robust distributed programs”, ACM TOPLAS, Vol 5 (3), pp 381-404, 1983

Liskov88 Liskov, B., “Distributed Programming in Argus”, Comm. ACM, Vol. 31 (3), pp 300-12, 1988

MacLennan87 MacLennan, B. J., "Principles of programming languages (Second Edition)", Holt, Rinehart & Winston, 1987.

MAL99 The Mobile Agents List, a repository of mobile agent systems, available at: http://www.informatik.uni-tuttgart.de/ipvr/vs/projekte/mole/mal/mal.html

Malamud91 Malamud, C., “Analysing DECnet / OSI Phase V”, Van Nostrand Rheinhold, NY, 1991

Martin99 Martin, D. L., Cheyer, A. J., and D. B. Moran, "The open agent architecture: A framework for building distributed software systems," Applied Artificial Intelligence, vol. 13, pp. 91--128, January-March, 1999.

McCall77 McCall, J.A., Richards, P.K.z, Walters, G.F., “Factors in Software Quality”, Rome Air Development Centre, RADC TR-77-369, 1977.

McCarthy60 McCarthy, J., "Recursive functions of symbolic expressions and their computation by machine.", Comm ACM, 3 (4), pp 184-195, 1960.

McFayden76 McFayden, J.H., “Systems network architecture: An overview”, IBM Systems Journal, Vol 15 (1), pp 4-23, 1976.

McQuillan77 McQuillan, J., Walden, D., “The ARPA network design decision”, Computer Networks, 1 (3), pp 243-289, 1977

Mendelson64 Mendelson, E., “Introduction to Mathematical Logic”, Van Nostrand Reinhold, 1964.

Metcalfe76 Metcalfe, R.M., Boggs, D.R., “Ethernet: Distributed Packet Switching for local computer networks.”, Comm. ACM, 19 (7), pp 395-404, 1976

Milner90 Milner, R., Tofte, M., Harper, R.M., “The Definition of Standard ML”, MIT Press, Cambridge, MA, 1990.

Milojicic99 Milojicic, D., “Trend Wars: Mobile Agent Applications”, IEEE Concurrency, pp 80-90, July-September, 1999.

Minar98 Minar, N., “Designing an Ecology of Distributed Agents”, Masters Thesis, Media Lab, MIT, 1998.

Minar99 Minar, N., Gray, M., Roup, O., Krikorian, R., Maes, P., (1999), “Hive: Distributed Agents for Networking Things”, Proceedings of. ASA/MA ‘99.

Minar99b Private email correspondence with Nelson Minar, Hive team lead and chief architect, Dec 1999.

Mitchel79 Mitchel, J.G., Maybury, W., Sweet, R., “Mesa Language Reference Manual (V5.0)”, Tech. Report CSL-79-3, Xerox PARC, Palo Alto, CA.

Mobility98 Frequently Asked Questions (FAQ) of The Mobility List, 1998.Available at http://mobility.lboro.ac.uk/faq.html

Mobility99 The Mobility Mailing List – de facto mailing for discussion of mobility. Home page at: http://mobility.lboro.ac.uk

Molina98 Molina, A., Flores, M., Caballero, D., "Virtual Enterprises: A Mexican Case Study", published in [Camarinha-Matos98], 1998, pp159-170.

MSI99 MSI Research Institute, Final Report of the EPSRC Grant entitled “Manufacturing Software Interoperability: Steps towards Interoperating Distributed Objects”, EPSRC Grant No GR/M02446 (GR/K50504), Duration 01/05/95 – 30/04/99, 1999

Mullender93 Mullender, S.J. (ed), “Distributed Systems 2nd Edition”, ACM Press, 1993.

Naur63 Naur, P. et al (Eds.), "Revised Report on the Algorithmic Language ALGOL 60", Comm. ACM, 6, pp. 1-17, 1963.

Naur78 Naur, P., “The European Side of the Last Phase of the Development of Algol 60”, SIGPLAN Notices 13 (8), pp 15-44, 1978.

Nelson91 Nelson, G., “Systems Programming with Modula-3”, Prentice-Hall, Englewood Cliffs, 1991.

OMG94 Object Management Group, “The Common Object Request Broker: Architecture and Specification”, OMG Inc., 492 Old Connecticut Path, Framingham, MA, USA, 1994

OMG99 Object Management Group, “IDL Syntax and Semantics”, OMG Inc., 492 Old Connecticut Path, Framingham, MA, USA, 1999, available at: http://www.omg.org/pub/orbrev/drafts/revised_99-08-01.idl

OSF92 OSF, “Introduction to OSF DCE”, Prentice-Hall, 1992

OSI84 Reference Model of Open Systems Interconnection for CCITT Applications, Malaga-Torremolinos, 1984

Osório98 Osório, A.L., Nuno, O., Camarinha-Matos, L., "Concurrent Engineering in Virtual Enterprises: the Extended CIM-FACE Architecture", published in [Camarinha-Matos98], pp171-184.

Ousterhout94 Ousterhout, J.K., “Tcl and the Tk toolkit”, Addison-Wesley, 1994.

Papaioannou/ Minar99 Papaioannou, T., Minar, N., “Mobile Agents in the Context of Competition and Cooperation”, Proc. of MAC3 workshop, part of Autonomous Agents ‘99 conference, Seattle, 1999.

Papaioannou98 Papaioannou, T., Edwards, J.M., “Mobile Agent Technology Enabling the Virtual Enterprise: A Pattern for Database Query”, in notes of Agent Based Manufacturing Workshop, part of the International Technical Conference Autonomous Agents '98.

Papaioannou99 Papaioannou, T., Edwards, J.M., “Using Mobile Agents To Improve the Alignment Between Manufacturing and its IT Support Systems”, International Journal of Robotics and Autonomous Systems, 27, pp 45-57, 1999.

Papaioannou99b Papaioannou, T., Edwards, J.M., “Mobile Agent Technology in Support of Sales Order Processing in the Virtual Enterprise”, in [Camarinha-Matos et al]

Papaioannou99c Papaioannou, T., “Mobile Agents: Are They Useful for Establishing a Virtual Presence in Space?”, in notes of Adjustable Autonomy Symposium, part of the AAAI Spring Symposium Series, Stanford University, 1999.

Papaioannou2000 Papaioannou, T., Edwards, J.M., “Manufacturing Systems Integration and Agility: Can Mobile Agents Help?”, accepted for publication in Journal of Integrated Computers-Aided Engineering, IOS Press. To appear in 2000.

Papaioannou2000b Papaioannou, T., Edwards, J.M., “Towards Understanding and Evaluating Mobile Code Systems”, accepted for publication in Journal of Autonomous Agents and Multi-Agent Systems, Kluwer Academic Publishers. To appear in 2000.

Papastavrou99 Papastavrou, S., Samaras, G., Pitoura, E., “Mobile Agents for WWW Distributed Database Access”, in Proc. IEEE International
Conference on Data Engineering (ICDE99), 1999.

Parnas72a Parnas, D.L., “A technique for software module specification with examples”, Comm. ACM, Vol 15 (5), pp 330-336, 1972.

Parnas72b Parnas, D.L., “On the criteria to be used in decomposing systems into modules”, Comm. ACM, Vol 15 (12), pp 1053-1058, 1972.

Peine98 Peine, H., Stolpmann, T., “The Architecture of the Ara Platform for Mobile Agents”, in [Rothermel97], pp 50-61.

Perlis58 Perlis, A., Samelson, K., "Preliminary Report - International Algebraic Language", Comm. ACM 1 (12) pp. 8-22, 1958.

Peters82 Peters, T., Waterman, R.H., Jr, “In search of Excellence”, HarperCollins, 1982.

Peters85 Peters, T., Austin, N., “A Passion for Excellence – the Leadership difference”, HarperCollins, 1985.

Picco98 Picco, G.P., “Understanding, Evaluating, Formalizing, and Exploiting Code Mobility”, PhD thesis, Politecnico di Torino, 1998

Picco98b Picco, G.P., Baldi, M., “Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications”, In Proceedings of the 20th International Conference on Software Engineering (ICSE'98), Kyoto (Japan), R. Kemmerer and K. Futatsugi, eds., April 1998, IEEE CS Press, ISBN 0-8186-8368-6, pp. 146-155, 1998.

Pinker95 Pinker, S., “The Language Instinct”, Harper Collins, 1995, ISBN: 0060976519.

Pouzin73 Pouzin, L., “Presentation and major design aspects of the CYCLADES computer network”, in Proc. 3rd ACM-IEEE Communications Symposium, pp 80-87, 1973

Pratt84 Pratt, T.W., "Programming Languages, Design and Implementation, 2nd edition", Prentice-Hall, 1984.

Raj91 Raj, R.K., Tempero, E., Levy, H.M., Black, A.P., Hutchinson, N.C., and Jul, E., “Emerald: A general purpose programming language”, Software-Practice and Experience, Vol 21 (1), 1991.

Rashid81 Rashid, R., Robertson, G., “Accent: a communications oriented network operating system kernel”, ACM Operating Systems Review, Vol 15 (5), pp 64-75, 1981

Rashid86 Rashid, R., “From RIG to Accent to Mach: the evolution of a network operating system”, Proc. ACM/IEEE Computer Society Fall Joint Conference, ACM, 1986.

Raymond98 Raymond, E.S., “The Cathedral and the Bazaar”, Version 1.40, 1998/08/11, http://www.tuxedo.org/~esr/

Redmond97 Redmond, F., III, "Dcom: Microsoft Distributed Component Object Model", IDG Books Worldwide, ISBN: 0764580442

Reed79 Reed, D.P., Kanodia, R.K., "Sychronisation with Eventcounts and Sequences", Comm. ACM, Vol. 22 (2), pp 3-23, 1979.

Ritchie74 Ritchie, D.M., Thompson, K., "The UNIX time-sharing system", Comm. ACM, Vol 17 (7), pp 365-375, 1974

Roberts70 Roberts, L.G., Wessler, B.T., "Computer network development to achieve resource sharing", in Proc. SJCC, pp 543-549, 1970.

Rose90 Rose, M.T., "The Open Book: a practical perspective on OSI", Prentice-Hall, 1990

Rothermel97 Rothermel, K., Popescu-Zeletin, R., Eds, "Mobile Agents: 1st International Workshop MA'97", Lecture Notes in Computer Science, Vol 1219, Springer-Verlag, 1997.

Rothermel98 Rothermel, K., Hohl, F., Eds, "Mobile Agents, 2nd Int'l Workshop MA '98", Lecture Notes in Computer Science, Vol 1477, Springer-Verlag, 1998.

Rus97 Rus, D., Gray, R., Kotz, D., "*Transportable information agents*", Journal of Intelligent Information Systems, 9:215-238, 1997

Scanlon97 Scanlon, E., "Suggestions for Case Study Research Methods", http://www.gwbssw.wustl.edu/~csd/evaluation/casestudy/caseguide.html

Shock80 Shock, J.F., "An annotated Bibliography on Local Computer Networks", XEROX Palo Alto Research Center, 1980.

Shrivastava-et-al Shrivastava, S.K., Dixon, G., Parrington, G.D., Hedayati, F., Wheater, S., Little, M., "The Design and Implementation of Arjuna", Proc. 3rd Conference on Object Oriented Programming, Nottingham.

Shroeder93 Shroeder, M, D., "A State-of-the-Art Distributed System: Computing with BOB.", In Distributed Systems, 2nd ed., S. Mullender, ed., ACM Press, 1993

Simon96 Simon, E., "Distributed Information Systems – from client/server to distributed multimedia", McGraw-Hill, 1996.

Sloman85 Sloman, M., Kramer, J., Magee, J., "The Conic toolkit for building distributed systems", Proc. 6th IFAC Workshop on Distributed Computer Control Systems, California, Pergamon Press, 1985

Sloman87 Sloman, M., Kramer, J., "Distributed Systems and Computer Networks", Prentice-Hall, 1987

Solingen99 Solingen, R.V., Berghout, E., (1999), "The Goal/Question/Metric Method",McGraw Hill, ISBN 0-07-709553-7

SSA95 System Software Associates Inc., "BPCS Client/Server Distributed Object Computing Architecture", Technical Report, 1995.

Stallings87 Stallings, W., "Handbook of Computer Communications Standards", Vol 1, Macmillan, NY, 1987

Stamos86 Stamos, J.W., "Remote Evaluation", Technical Report TR-354, MIT, 1986

Straßer96 Straßer, M, Baumann, J., Hohl, F., (1996), "Mole - A java Based Mobile Agent System", in Proc. ECOOP'96 workshop on Mobile Object Systems.

Strauss90 Strauss, A. & Corbin, J. (1990) "Basics of Qualitative Research", Newbury Park, CA.: Sage Publications.

Stroustrup92 Stroustrup, B., "The C++ Programming Language", 2nd edition, Addison-Wesley, Reading, MA, 1992.

Sun89 Sun Microsystems Inc., "NFS: Network File System Protocol Specification", Tech. Report RFC 1094, file available for anonymous ftp from ftp://nic.ddn.mil, directory /usr/pub/RFC, 1989

Sun97 Sun Microsystems Inc., "JavaBeans Component Framework Specification", Revision 1.01, JDK 1.1, July 1997.

Sun98 Sun Microsystems Inc., "Java Remote Method Invocation Specification", Revision 1.50, JDK 1.2, October 1998.

Sun98b Sun Microsystems Inc., "Object Serialisation Specification", JDK 1.1, October 1998, available at http://java.sun.com/products /jdk/1.1/docs/guide/serialization/spec/serialTOC.doc.html

Sun99 Sun Microsystems Inc., "Enterprise Javabeans Specification", Version 1.1, 1999, available at http://java.sun.com/products/ejb/docs.html

Tanenbaum96 Tanenbaum, A.S., "Computer Networks – 3rd Edition", Prentice-Hall, 1996.

Teitelman84 Teitelman, W., "A tour through Cedar", IEEE Software, Vol. 1 (2), pp 44-73, 1984

Thiel91 Thiel, G., "Locus Operating System, a transparent system", Computer Communications, Vol 14 (6), 1991, pp336-346.

Thompson96 Thompson, S., “Haskell: The Craft of Functional Programming”, Addison-Wesley, 1996.

Tsichritzis85 Tsichritzis, D., “*Objectworld*”, Office Automation, Springer-Verlag, 1985

Turing36 Turing, A.M., “On Computable Numbers, with an application to the Entscheidungsproblem”, Proc. London Mathematical Society, Vol 2 (42), pp 230-265, 1936.

Turner85 Turner, D.A., “Miranda: A non-strict functional language with polymorphic types”, in Functional Programming Languages and Computer Architercture, Lecture Notes in Computer Science 201, pp 1-16, Springer-Verlag, 1985.

UCI96 “MESSENGERS: A Distributed Computing Environment for AutonomousObjects” UCI Technical Report: TR-96-20, 1996, available from http://www.ics.uci.edu/~bic/messengers/

Vigna98 Vigna, G., ed, “Mobile Agents and Security”, LNCS Vol 1419, Springer-Verlag, 1998.

Vinoski99 Vinoski, S., Chief Architect at Iona Technologies Inc, comment made in dist-obj mailing list. Thu, 15 Jul, 1999.

Waldo94 Waldo, J., Wyant, G., Wollrath, A., Kendall, S., “A note on distributed computing”, Sun Microsystems Technical Report SML 94-29, 1994.

Walsh85 Walsh, D., Lyon, B., Sager, G., Change, J.M., Goldberg, D., Kleiman, S., Lyon, T., Sandberg, R. and Weiss, P., “Overview of the Sun Network File System”, Proc. of the Winter Usenix Conference, 1985.

Watt96 Watt, S., “Pride and prejudice: four decades of LISP”, in [Woodman96], pp 235-254, 1996

Wecker80 Wecker, S., “DNA: the digital network architecture”, in [IEEE80]

Weiser91 Weiser, M., “The Computer for the 21st Century”, Scientific American, Vol 265 (3), pp 94-104, 1991.

Wexelblat81 Wexelblat, R.L., ed, "History of Programming Languages", ACM Monograph Series, Academic Press, 1981.

White94 White, J.E, “Telescript technology: the foundation for the electronic marketplace”, White Paper, General Magic, Inc. USA, 1994.

White96 White, J., “Telescript Technology: Mobile Agents”, In Software Agents, Bradshaw, J., Ed., AAAI Press/MIT Press, 1996.

Whitmire97 Whitmire, S.A., “Object-oriented design measurement”, Wiley and Sons, 1997, ISBN:0-471-13417-1

Wilson93 Wilson, L.B., Clark, R.G., “Comparative Programming Languages”, Addison-Wesley, 1993.

Wirth77 Wirth, N., “Modula: A language for modular programming”, software Practice Experience, 7 (1), Jan 1977.

Wirth82 Wirth, N., "Type Extensions", ACM Transactions on Programming Languages and Systems, 10 (2), pp. 204-214, February 1988.

Wong97 Wong, D., Paciorek, N., Walsh, T., DiCelie, J., Young, M., Peet, B., “Concordia: An infrastructure for collaborating mobile agents”, in Proc. First Int’l Workshop on Mobile Agents ’97, Springer-Verlag, 1997

Woodman96 Woodman, M., “Programming Language Choice, Practice and Experience”, Thomson Computer Press, 1996.

Wooldridge99 Wooldridge, M., Jennings, N.R., Kinny, D., “A Methodology for Agent-Oriented Analysis sand Design”, in Proc. 3rd Annual Conf. On Autonomous Agents, Eds, Etzioni, O., Müller, J.P., Bradshaw, J.M., ACM Press, 1999.

Wright96 Wright, D.T., Burns, N.D., "Impact of Globalisation on organisational Structure and Performance", Proc. of the Organisational Management Division, International Association of Management 14th Annual Conference. Toronto, Canada, August 2-6, 1996, pp. 58-63

Yin94 Yin, R.K., “Case Study Research”, Sage Publications, 1994

Yourdon79 Yourdon, E., Constantine, L.L., “Structured Design”, Prentice-Hall 1979.

Zak98 Zak, D., “Programming With Microsoft Visual Basic 6.0”, Microsoft Press, 1998.

Zimmerman80 Zimmerman, H., “OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection, in [IEEE80].

Appendices

Appendix A

Program listing of an example OrderAgent:

```
package uk.ac.lboro.todd.aglets.mascenario;

import uk.ac.lboro.todd.aglets.mascenario.tasks.*;
import uk.ac.lboro.todd.aglets.*;
import uk.ac.lboro.todd.aglets.order.*;
import uk.ac.lboro.todd.aglets.utils.*;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import java.net.URL;
import java.util.Date;

/**
 * A simple QueryAglet that can be created by a Master and tasked
 * with tracking down the stock levels of a product from a list of
 * hosts.
 *
 * @version   2.1   10/11/98    Changed from a properties lookup for
 *                              the DataSource to multicast messaging
 *  version   2.0   21/10/98    Most of the required logic has now
 *                              been refactored and shifted to the
 *                              Task classes. Allows for far more
 *                              modularity.
 *  version   1.2   18/10/98    Query can now handle missing data
 *                              sources and also the addition of
 *                              subsequent tasks, after the
 *                              completion of the first one.
 *  version   1.1   08/10/98    Query aglet is now able to complete
 *                              Itinerary and request a retraction
 *                              Removed MakeRequest and added it to
 *                              StockRequestTask. Makes more sense.
 *  version   1.01  25/09/98    Added capability to create with
 *                              details and receive an Itinerary.
 *  version     1.00        23/09/98    First attempt.
 *
 * @author      Todd Papaioannou
 */

public class QueryAglet extends BlindAglet {

    // Our data variables
    AgletProxy dataProxy = null;
    ResultSet resSet = null;
    AgletProxy mProxy = null;
    SlaveItin itin = null;
    Order order = null;

    // Do some tasks when the aglet is created
    public void onCreation(Object init) {

        // Pass up the hierarchy
```

```
        super.onCreation(init);

        SlaveDetails det = (SlaveDetails)init;

        // Must make a note of the master here
        mProxy = det.getMaster();

        // Initialise our important internals
        resSet = new ResultSet(getAgletID());
        order = det.getOrder();

        // Add our own listener and adapter
        addMobilityListener(
            new MobilityAdapter() {

                int counter = 0;

                // Using this as a safety check in case we get caught
                // in a loop in the same host
                public void onArrival(MobilityEvent event) {

                    if (counter > 1)
                        System.out.println("ACounter = " +
                              new Integer(counter).toString());
                    counter++;

                    if (counter > 3) {
                        System.out.println("Self destructing!");
                        try {
                            event.getAgletProxy().dispose();
                        } catch (Exception e) {
                            System.out.println(e.toString());
                        }
                    }
                }

                public void onDispatching(MobilityEvent event) {
                    counter = 0;
                }

                public void onReverting(MobilityEvent event) {
                    appendMessage("Being retracted by Master to
                                                    homebase.");
                }
            }

        ); /* End of Adapter */

    }

    // Test run
    public void run() {

        //System.out.println("\nInto run");

        // Just a safety check, in case of delay
        while (itin == null) {
            for (int i = 0; i < 3; i++) {

                waitMessage(1 * 1000);
            }
```

```
        }

        // Do we have an itinerary and is this the last stop?
        if ((itin != null) && itin.atLastDestination()) {

            // Let's get a reference to the final Task object.
            GenericTask task =
                        (GenericTask)itin.getTaskAt(itin.size()-1);

            try {
                task.finishTasks(itin);
            } catch (Exception e) {
                System.out.println(e);
            }
        }
    }

    /**
     * Handle ourselves being killed gracefully
     *
    public void onDisposing() {

        // Clear up and get rid of our itinerary
        itin.clear();
        removeMobilityListener(itin);
    }*/

    /**
     * Returns true if the current host is our origin
     */
    public boolean atHome() {

        if (getAgletInfo().getOrigin().equals(getAgletContext(). \
            getHostingURL().toString()))
            return true;
        else
            return false;
    }

    /**
     * Allows a slave to contact it's master and ask for a
     * retraction. Useful since the Master has no idea where the
     * Slave might have ended up.
     */
    public void returnHome() {

        try {
            Message msg = new Message("RetractMe");
            msg.setArg("url", getAgletContext().getHostingURL());
            msg.setArg("id", getAgletID());
            mProxy.sendOnewayMessage(msg);
        } catch (InvalidAgletException iae) {
            System.out.println("1 " + iae.toString());
        } catch (Exception e) {
            System.out.println("2 " + e.toString());
        }
     }

    /**
     * Find out who is the data source in this context
     */
```

```
    public boolean whoSource() {

        try {
            ReplySet set = getAgletContext().multicastMessage
                                        (new Message("DataSource?"));

            // Give any sluggards a chance
            while (!set.isAnyAvailable())
                waitMessage(1*10);

            FutureReply future = set.getNextFutureReply();
            Object reply = future.getReply();
            AgletID aid = (AgletID)reply;
            dataProxy = getAgletContext().getAgletProxy(aid);

        } catch (NotHandledException ex) {
            System.out.println(ex);
            dataProxy = null;
        } catch (MessageException ex) {
            System.out.println(ex);
            dataProxy = null;
        }

        if (dataProxy != null)
            return true;
        else
            return false;
    }

    /**
     * Attempt to handle any incoming messages
     */
    public boolean handleMessage(Message msg) {

        if (msg.sameKind("Itinerary")) {
            itin = (SlaveItin)msg.getArg();
            appendMessage("Itinerary received, starting trip.");
            itin.startTrip();
        } else {
            System.out.println(msg.toString());
            return false;
        }

        return true;
    }

    /**
     * Override super class method to allow for easy redirection
     * during testing.
     */
    public void appendMessage(String text) {
        System.out.println("[" + getName() + "] " + text);
    }

    /**
     * Return the current order we are dealing with
     */
    public Order getOrder() {
        return order;
    }
```

```
    /**
     * Return our current result set
     */
    public ResultSet getResults() {
        return resSet;
    }

    /**
     * Allow someone to try to clear our result set
     */
    public void clearResults() {
        resSet = null;
    }

    /**
     * Return a reference to our Master's proxy
     */
    public AgletProxy getMasterProxy() {
        return mProxy;
    }

    /**
     * Return a reference to the DataAglet's proxy
     */
    public AgletProxy getDataProxy() {
        return dataProxy;
    }

    /**
     * Return a reference to our Itinerary
     */
    public SlaveItin getItin() {
        return itin;
    }

} /* End of Class */
```

Appendix B

Program listing of an example Agent Task:

```
package uk.ac.lboro.todd.aglets.mascenario.tasks;

import uk.ac.lboro.todd.aglets.*;
import uk.ac.lboro.todd.aglets.utils.*;
import uk.ac.lboro.todd.aglets.mascenario.*;
import uk.ac.lboro.todd.aglets.order.*;

import com.ibm.aglet.*;
import com.ibm.agletx.util.*;
import java.net.URL;

/**
 * StockRequestTask - a task that allows an agent to make a request
 * to a DataSource aglet. The request is encapsulated within the
 * Order object the slave carries around with it.
 *
 * @version   2.1   04/11/98    First attempt with MA instead
 *                              of MO's. Added evalResult().
 * @version   2.0   21/10/98    Massive refactoring of the
 *                              code. Very little of the behaviour of
 *                              the Aglet relies on code in run()
 *                              The addition of finishTasks allows
 *                              for a much simpler and more modular
 *                              approach to design of Slave agents.
 * @version   1.11  08/10/98    MakeRequest has been added from
 *                              QueryAglet. Makes more sense.
 * @version   1.10  08/10/98    StockRequest now fully functional
 * @version   1.00  28/09/98    First attempt.
 *
 * @author      Todd Papaioannou
 */

public class StockRequestTask extends GenericTask {

    /**
     * Use this to allow us a better view of what goes on at a host
     */
    static boolean pause = true;

    // Our owner aglet
    QueryAglet qag = null;
    Result result = null;

    /**
     * The actual work associated with this Task.
     */
    public void execute(SeqItinerary itin) throws Exception {

        // Find out who the data source is
        AgletProxy proxy = itin.getOwnerAglet();
        qag = (QueryAglet)proxy.getAglet();
        URL currentHost = qag.getAgletContext().getHostingURL();

        // Is this the last desination?
        if (itin.atLastDestination() == false) {
```

```
            // We must still have some tasks to do.
            // Is there a data source handy?
            if (qag.whoSource() != true) {
                qag.appendMessage("No damn data source!");

                // Are we actually at the last address?
                if (!currentHost.toString().
                          equals(itin.getAddressAt(itin.size()-1)) ) {

                    // Make it easier to see what's actually going on
                    if (pause)
                        qag.waitMessage(2*1000);
                    qag.appendMessage("Proceeding to next stop on \
                                                          Itinerary");
                }
            } else {
                qag.appendMessage("Found a data source.");

                // Get our info from the data source
                makeRequest();
                qag.appendMessage("Finished Request, evaluating \
                                                            results.");
                evalResult();
            }
        }
    } // End of execute

    /**
     * Make a request for an Order to be checked.
     */
    public void makeRequest() {

        try {

            Object reply = qag.getDataProxy().sendMessage(
                  new Message("Order", new NamedOrder(qag.getName(),
                                                qag.getOrder())));
            result = (Result)reply;

        } catch (InvalidAgletException ex) {
            System.out.println(ex);
        } catch (NotHandledException ex) {
            System.out.println(ex);
        } catch (MessageException ex) {
            System.out.println("[ERROR] Make Request Failed because \
                                       of:\n" + ex.getException());
            System.out.println(ex);
        }

        // Let's put some artificial pausing in. Looks good for the
        // humans!
        if (pause) {
            for (int i=0; i < 160; i++) {
               System.out.print(".");
            }
            System.out.println("\n");
        }

    } // End of makeRequest
```

```
    /**
     * Can this host satisfy our order?
     */
    private void evalResult() {

        boolean success = false;

        if (result.getIndicator() == Result.YES) {

            qag.appendMessage("We have a RESULT!");
            qag.appendMessage("Result " + result.getHost() + " will \
                                              satisfy this order.");
            success = true;
        }

        // Add this result to our set for future reference.
        qag.getResults().addResult((Result)result);

        if (success) {
            commitOrder();
        } else {
            qag.appendMessage("Current host cannot satisfy order. \
                                              Going to next host.");
        }
    }

    // This routine allows us to attempt to commit and order
    private void commitOrder() {

        try {

            String reply = (String)qag.getDataProxy().sendMessage(
                  new Message("Commit", new NamedOrder(qag.getName(),
                                                  qag.getOrder())));

            // We have successfully committed the Order
            if (reply.equals("Committed")) {

                qag.appendMessage("Order successfully committed.");
                qag.getMasterProxy().sendOnewayMessage(new Message
                     ("Committed", qag.getOrder().getOrderNumber()));
                qag.appendMessage("Tasks have been completed. \
                                             Disposing of myself.");

                // Kill ourselves
                qag.dispose();

            } else if (reply.equals("OutOfStock")) {
                qag.appendMessage("Out of Stock!");
                qag.getMasterProxy().sendOnewayMessage(new Message
                    ("OutOfStock", qag.getOrder().getOrderNumber()));
                qag.dispose();
            } else {
                qag.appendMessage("Something messed up! Getting rid \
                                                      of myself.");
                qag.dispose();
            }

        } catch (InvalidAgletException ex) {
            System.out.println(ex);
        } catch (NotHandledException ex) {
```

```
            System.out.println(ex);
        } catch (MessageException ex) {
            System.out.println("[ERROR] Make Request Failed because \
                                    of:\n" + ex.getException());
            System.out.println(ex);
        }

    }

    // Must define this since it's abstract
    public void finishTasks(SeqItinerary itin) throws Exception {
    }

} /* End of Class */
```