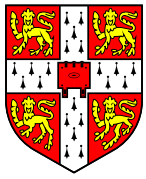
Nomadic Pict: Language and Infrastructure Design for Mobile Computation

Paweł Tomasz Wojciechowski

Wolfson College
University of Cambridge



A dissertation submitted for the degree of
Doctor of Philosophy

March 2000

Abstract

Mobile agents — units of executing computation that can migrate between machines — are likely to become an important enabling technology for future distributed systems. We study the distributed infrastructures required for location-independent communication between migrating agents. These infrastructures are problematic: the choice or design of an infrastructure must be somewhat application-specific — any given algorithm will only have satisfactory performance for some range of migration and communication behaviour; the algorithms must be matched to the expected properties (and robustness demands) of applications and the failure characteristic of the communication medium. To study this problem we introduce an agent programming language – *Nomadic Pict*. It is designed to allow infrastructure algorithms to be expressed clearly, as translations from a high-level language to a lower level. The levels are based on rigorously-defined process calculi, which provide sharp levels of abstraction. In this dissertation we describe the language and use it to develop a distributed infrastructure for an example application. The language and examples have been implemented; we conclude with a description of the compiler and runtime system.

To Maria and Zdzisław

Preface

Except where otherwise stated in the text, this document is the result of my own work and is not the outcome of work done in collaboration.

This dissertation is not substantially the same as any that I have submitted for a degree or diploma or other qualification at any other university.

No part of this dissertation has already been or is being concurrently submitted for any such degree, diploma or other qualification.

This dissertation does not exceed sixty thousand words, including tables, footnotes and bibliography.

Acknowledgments

This research was supported by a scholarship from the Wolfson Foundation. Without this support and Prof. Jerzy Brzeziński at Poznań University of Technology, who after my graduation made me want to do research into distributed algorithms, I could not even have gone to England.

I am grateful to Ken Moody for introducing me to the problem of locations and failures in process calculi, and for many stimulating and enjoyable discussions. I would also like to thank Ken for supervising my thesis and his continuous support during years of my study.

I would like to express my special gratitude to Prof. Robin Milner, whose lectures about the π-calculus I was able to attend shortly after coming to Cambridge in the fall of 1995. They gave me a lot of insight into problems which were new to me and appeared to be a great source of inspiration in my research work.

I owe especial thanks to Peter Sewell, who not only taught me a great deal about the theory and semantics of programming languages, but also, in the midst of a busy schedule, he took time out to answer all my endless questions and provided me with friendly conversation even as I was depriving him of his work and his peace. Part of this research was done in collaboration with Peter.

I consider myself very fortunate, to have been introduced to Benjamin Pierce, a co-author of Pict, while he was visiting the Computer Laboratory at the University of Cambridge in the academic year 1995/96. Some of the early ideas of the nomadic π-calculi resulted from the discussions with Benjamin and Peter during our brief meeting at Indiana University in May of 1997.

I would also like to thank Cédric Fournet and Asis Unyapoth. They have contributed valuable comments on parts of this work at various stages of its completion.

I am very grateful too to past and present members of the Opera Group, which is part of the Systems Research Group, and members of the Theory and Semantics Group, who not only showed much interest and understanding of the true nature of my work, but also created an interesting and stimulating environment.

Finally, I would like to thank Ken Moody and Peter Sewell who read what I was scribbling with extraordinary sympathy and care, cheering me on and holding me to the very highest standards.
Thanks to the staff in the Computer Laboratory at the University of Cambridge for helping me with the administrative aspects of being a PhD student.
Thanks to my mates and fellows at Wolfson College and friends from various societies for many enjoyable discussions, social events, and sharing time in Cambridge.
More personally, thanks to my family and best friends, especially to Ania, for her friendship and happiest memories. Finally, all the time I was at home, and away, I was kept upright by my parents, who reconciled themselves with patience and good grace to a son who chose to live far away, and always looked after my well-being — showing so much kindness and helping me in various important ways.

Publications

Aspects of the work described in this dissertation feature in the following publications:

- Paweł T. Wojciechowski and Peter Sewell. Nomadic Pict: Language and Infrastructure Design for Mobile Agents. In *IEEE Concurrency*. The Computer Society's Systems Magazin, April-June 2000. This is an extended version of the paper below.

- Paweł T. Wojciechowski and Peter Sewell. Nomadic Pict: Language and Infrastructure Design for Mobile Agents. This appeared in the *Proceedings of ASA/MA'99 (First International Symposium on Agent Systems and Applications/Third International Symposium on Mobile Agents), Palm Springs, CA, USA*, October 1999.[1]

- Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location-Independent Communication for Mobile Agents: A Two-Level Architecture. In Henri E. Bal, Boumediene Belkhouche, and Luca Cardelli, editors, *Internet Programming Languages (ICCL'98 Workshop)*, volume 1686 of *Lecture Notes in Computer Science*, pages 1–31. Springer, 1999. Also appeared as Technical Report 462, Computer Laboratory, University of Cambridge, April 1999.

- Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location Independence for Mobile Agents. This appeared in the *Proceedings of ICCL'98 Workshop on Internet Programming Languages, Chicago, USA*, May 1998. It is largely superseded by the paper above.

[1] Awarded *Best Paper Overall of Conference*

Contents

List of Figures

Chapter 1

Introduction

Mobile agents, units of executing computation that can migrate between machines, have been widely argued to be an important enabling technology for future distributed systems [CHK97, VT97]. They introduce a new problem, however. To ease application writing one would like to be able to use high-level *location independent* communication facilities, allowing the parts of an application to interact without explicitly tracking each other's movements. To provide these above standard network technologies (which directly support only location-dependent communication) requires some distributed infrastructure, problematic in three ways. Firstly, the distributed algorithms needed are delicate. Secondly, flexible structuring mechanisms are required to support clean factorisation of a system into its high-level application component and the infrastructure implementation. Thirdly, the choice or design of an infrastructure must be somewhat application-specific — any given algorithm will only have satisfactory performance for some range of migration and communication behaviour; the algorithms must be matched to the expected properties (and robustness and security demands) of applications.

We are addressing these issues in the context of a mobile agent programming language, designed and implemented as part of this thesis. The language, called *Nomadic Pict*, is based on a small core calculus – the Nomadic π-calculus – that has a clear rigorous operational semantics, tightly related to real network communication. This permits infrastructure algorithms to be expressed precisely and concisely in an executable form, aiding design and supporting ongoing work on correctness and robustness proofs.

The language has a two-level architecture. The low level consists of well-understood, location-dependent primitives, including communication and agent migration. The high level, in which applications can be written, extends these with location-independent communication. An infrastructure can be expressed as an implementation of the high-level primitives in terms of

the low-level language; only the low level need be supported by a widespread runtime system (the distributed parts of the infrastructure can be deployed dynamically, on application start-up, using agent migration).

The ease of writing infrastructure algorithms, and the fact that an arbitrary infrastructure can be provided for an application at compile time, make it straightforward to experiment with a wide range of infrastructures for applications with different migration and communication patterns.

The content of the thesis is as follows. In chapter 2, after discussing a variant of the asynchronous π-calculus, we describe the design of the Nomadic π-calculus, giving its operational semantics. In the end of the chapter, we describe related calculi and two other non-calculi models. In chapter 3, we present Nomadic Pict. Firstly, we motivate our decision to design and implement another programming language. Then, we describe the language in more detail, introducing enough of the syntax and idioms to be able to understand translation encodings. Finally, we compare our design with two other related programming languages. In chapter 4, we describe different plausible infrastructures for location independence in the presence of mobility, expressing two simple infrastructure algorithms as translations from high- to low-level Nomadic Pict. In chapter 5, we discuss various applications of mobile agents, matching example applications with suitable infrastructures. Then, in chapter 6 we discuss a small example application and the design of an infrastructure suited to it in more detail. The focus is on demonstrating the benefits of a multi-level architecture based on clearly defined levels of abstraction. We begin with a simple centralised algorithm, which is further extended to improve scalability and providing support for disconnected operation. In chapter 7, we describe the current implementation of the Nomadic Pict compiler and runtime system.

In §1.1 we present background work, describing process mobility, distributed objects, and mobile agent systems. We focus on the problems of migration and communication transparency in these systems. Then, in §1.2 we introduce our work and outline the contribution of this thesis to research on the semantics of programming languages and design of communication infrastructures for systems with mobility.

1.1 Mobility and Location Independence

The last few years have seen much interest in systems and applications which support and use different forms of mobility, such as mobility of processes, devices, and agents (see, e.g. a collection of representative papers on each of these, published in [MDW99]). A major challenge to the success of em-

ploying mobility in the global network is the lack of widespread distributed infrastructure. The infrastructure should offer support for connecting devices while or after they physically move, and allow processes, objects or agents to visit remote sites and bind to local resources. For example, in collaborative applications, mobile devices such as palmtop computers can be used to maintain continuous communication channels within a group of people; mobile agents can migrate to a stable part of the network and act on behalf of the users if the connection with the mobile device was likely to be broken or interrupted. The agents and processes executing on mobile or stationary computers may want to maintain communication while moving.

In order to ease application writing, the infrastructure should support some forms of location-independence for programs and devices migrating from one location to another. The principle of *location-transparency* means that all names are independent of their location, migrating processes can request identical kernel services wherever they reside, distributed objects can be invoked without knowing their physical location, etc. Ideally, this means that communication with a migrating entity can proceed as if there had been no mobility. In practice, however, this is an (unattainable) ideal, especially in wide-area networks. The issues of speed, latency, failure semantics, heterogeneity, and physical disconnection mean that agent or object behaviour must adapt with location. Arguably, this adaptation should not be entirely at infrastructure level since the knowledge about the application cannot be fully exploited. The application programmer should be aware of mobility and distribution at a low level whenever it is hard or inefficient to drive adaptation within the infrastructure. In this context, we are interested in the infrastructure support for *location independence* which allows mobile devices to be transparently reconnected after restarting at a new location, objects to be accessed after migration to a new host machine, and mobile agents to receive messages while they move on the network, as long as the only failures which happen in the system are transient failures and there are no network partition or any administrative boundaries.

The need for a suitable infrastructure to support location independence appears inevitably in mobile computing. Traditionally, the Internet Protocol (IP) assumed that an address is mapped to a static physical location, i.e. the first few octets of an IP address specified a network, and the last part of the address specified a host interface on that network. Since a mobile computer can be moved around and connected to different networks, keeping the IP address static required other protocols for routing communication to and from the mobile computer (such as *Mobile IP*, initially proposed in [IDJ91, IJ93]).

Examples of traditional infrastructures supporting location independence include network services like directory assistance (e.g. X.500 protocol), distributed object managers (e.g. CORBA, DCOM, OLE, and OpenDoc) and automatic brokers, such as Publish and Subscribe Service in Mac OS and ToolTalk. Directory assistance allows a program to find a desired service. Distributed object managers provide transparent access to a distributed collection of objects; messages are automatically routed to the destination object even if the sender does not know the object's network location. Automatic brokers provide both functions by first identifying an appropriate recipient for a message and then forwarding the message.

Below, we describe background work on migration and communication transparency in process migration, distributed objects and mobile agents.

1.1.1 Process Migration

In [MDW99], Milojičić, Douglis and Wheeler summarise the key concepts of process migration and give an overview of the most important implementations. The basic idea of process migration is to move an executing process from one node to another (a node here can be a processor or host machine) in order to, e.g. balance load distribution. Below, we describe examples of systems with process migration; they are assumed to run in a local area network.

Traditional process migration mechanisms rely on support provided by the underlying operating system. For example, in MOSIX [BGW93] and Sprite [OCD+87], the distributed operating system provides a *single system image* that allows access to system resources (e.g. files) in the same way irrespective of the process' physical location, i.e. a process running on any node can access any other node's resources transparently (we assume the notion of ownership of resources by a node). The system can automatically migrate any process — migration is transparent to processes and to the users.

Other systems provide higher level mechanisms. For example, in [MZDG93] Milojičić et al. present a task migration mechanism built on top of the Mach microkernel system [ABB+86]. In Mach, a task is an execution environment that provides the basic unit of resource allocation. A task consists of a virtual address space and protected access to system resources via ports. A task may contain one or more threads. A port is implemented as a kernel-protected communication channel. File systems and other traditional abstractions are implemented in Mach by user-space system daemons. Since ports are location independent, a task and all its ports can be easily moved from one machine to another. All tasks which previously communicated with the moved task can continue to do so because they reference a task only by its location-independent ports and communicate via messages to these ports.

Although, task migration in Mach has been implemented in user space, some modifications to the kernel were necessary.

Condor [LS92] avoids the complexity of kernel-based migration and supports migration of UNIX processes entirely as a user-space mechanism, i.e. the process state must be exported into user space and then transferred. When a process is about to migrate, the system produces a core file for the process, which is sent to the new node. However, not all types of applications can migrate in this way, e.g. processes are not eligible for migration if they use signals or inter-process communication (IPC).

Some operating systems with process migration, such as Charlotte [ACF87], Amoeba [TvRvS+90, Tan92], and Mach provide support for transparent communication of processes (or tasks in Mach) irrespective of the current process location. Maintaining communication in the presence of migration has turned out to be one of the most complex components in these systems. Complex algorithms are required, e.g. to prevent messages sent during migration being discarded or received out of order. Typically, the algorithms involve some form of forwarding through proxies.

The object-based language and system Emerald [JLHB88] takes another approach to mobility. Firstly, Emerald has language support for the notion of location and for mobility. Secondly, mobility is *fine-grained*, i.e. the unit of distribution and mobility is an object, which can be a small data object (e.g. integer) as well as an active object which contains a process. The active object mobility subsumes both process migration and data transfer. In the next section, we describe the mechanism of location-independent invocation of Emerald objects. Besides object mobility, processes executing native machine code can also be moved on the fly, but only between computers that have the same architecture. (This restriction was later lifted when full heterogeneous mobility was implemented [SJ95].)

Process migration provides several benefits, e.g. it enables: load distribution (by migrating processes from a node which is overloaded to a less loaded one), fault resilience (by migrating processes from a node that may have partial failures), improved system administration (by migrating processes from a host machine that is about to be shutdown), and improved data access locality (by migrating processes to the host of data). However, process migration has not achieved widespread use. One reason for that is the complexity of supporting migration in operating systems originally designed as stand-alone. Another reason for the failure of process migration to become commonplace commercially is the rapid improvement of processing capabilities and other resources. For example, reclaiming processing power by migration in order to improve interactive performance is now a less obvious requirement, since interactive performance is now less likely to be affected by CPU-intensive and memory-intensive processes.

1.1.2 Distributed Objects

In order to ease the writing of distributed applications, good programming abstractions are required, providing forms of location independence. In the vision of "objects all the way down", one could imagine a single natural object-oriented design for a given application, regardless of whether that application will be deployed in a local or distributed context. The failure and performance issues are tied to the implementation of the underlying system components, and consideration of these issues is left out of the application design. The interface to a remote object is just like the interface to objects used locally.

In [WWWK97], Waldo et al. argue that such a unified view of local and remote objects was mistaken. Local and distributed computing are different in many ways. Distributed systems require that the programmer be aware of network latency, have a model of remote data access and data distribution different from the model of local memory access, and take into account issues of concurrency, inherent indeterminacy, and partial failure. Neglecting these differences has led to systems which are either not robust and reliable in a distributed context, or offer unnecessary complication of the local object implementation (since all local objects in this unified view must be treated as potentially remote). The conclusion which can be drawn from this polemic paper is that in order to realise a true distributed object system, a suitable infrastructure is required. A commonly accepted technology to build such an infrastructure was unavailable at that time.

In traditional RPC systems, such as DCE, and object based RPC systems, such as DCOM [EE98] and CORBA [OMG91], we use an interface definition language to define interfaces, where method calls are specified in terms of the primitive data types, object references, and structures of these entities. These interfaces are then compiled, for any given implementation language, with the result being stub and skeleton files. Once these are augmented with code that needs to be provided by a programmer, they can be compiled for the target operating system and architecture. The kind of information passed between the client and server cannot change without both the participants being updated simultaneously, since each must know exactly what is transmitted by the wire protocol.

We can use traditional distributed object frameworks to realise the distributed system as a whole as a set of cooperating objects. Unfortunately, the way in which those objects cooperate and communicate is decidedly non-object-oriented [Wal99]. In order to explain this point, we should recall the definition of the object-oriented paradigm. The principles that are at the core of object-oriented programming are the following: (1) the independence of an

object's interface from the implementation of the object, (2) the binding of behaviour with data, (3) polymorphism, i.e. the ability to describe an object in terms of the necessary conditions on the object, allowing an object to have multiple forms. Polymorphism is only available if the full objects, including implementations, can be passed between client and server. Unfortunately, the traditional distributed object frameworks cannot download and execute implementations because they do not allow real objects to be passed as arguments from one location in a distributed system to another, only data[1]. This precludes a common (in the non-distributed context) style of programming, when the information passed between objects as a parameter or a return value in a method call is another object. A revolution has been brought in by infrastructures that make it possible to communicate objects, not just object references. Below, we characterise the design choices made within the examples of Emerald, Network Objects and the Java environment.

Emerald [JLHB88] introduced mobile objects with safe, static subtyping and location-independent invocations. In Emerald, programmers use a single object definition mechanism with a single semantics for defining all objects. This includes small, local data-only objects, and active, mobile, distributed objects. Objects have unique network-wide names. The location of the active object may change over time, as an object migrates from one machine to another. The Emerald compiler is capable of analysing the needs of each object and generating an *appropriate implementation* from the same piece of source code, depending on the context in which it is compiled. For example, an array object whose use is entirely local to another object will be implemented differently from an array that is shared globally. Altogether with other features, such as the idea that new objects and new classes (subtypes as well as supertypes) can be added to a system at any time, Emerald can be seen as precursor of Network Objects and the Java environment, described below.

The need for semantic support for mobility, distribution, and abstract types led the Emerald group to design a new language. While method invocation is location-independent, language primitives can be used to find and manipulate the location of objects. Emerald uses *call-by-object-reference* parameter passing semantics for all object invocations, local or remote. However, the programmer may decide that an object should be moved based on knowledge about the application. For example, on remote invocations a parameter passing mode called *call-by-move* permits an invocation's argument object to be moved along with the invocation request. A *call-by-visit* mode

[1]Passing functions (code) has long been exploited in distributed implementations of functional languages (e.g. Facile described in §3.3.1).

does the same but the argument object returns to the source of the call. Also, the compiler may decide to move an object along with an invocation (e.g. small immutable objects which can be copied cheaply, such as integers or strings, are moved in this way).

Systems like Emerald successfully handle object distribution and hide the distinction between local and remote objects in small networks. But the failure modes of local and remote objects are inherently different, and performance is radically reduced by distribution and scale. This means that object placement is absolutely critical to application performance and robustness — and the tools that are currently available to automate this placement are minimal [Bla99]. (A closely related problem of the static estimation of potentially mobile functions and channels in a Facile-like language has been studied in [Kir99].) A simplified and more flexible approach to object-based distributed computing has been taken by the designers of Network Objects and Java.

Network Objects [BNOW95] is a distributed programming system designed for Modula-3. Network objects are not mobile, but the system makes it easy to communicate objects either by copying or by reference. The objects have no implicitly associated thread of control. The main features of the design are distributed typechecking, transparent invocation, powerful marshalling, efficient and convenient access to streams, and distributed garbage collection. We characterise some of these features briefly. The system provides typechecking via the *narrowest surrogate rule*, which allows a programmer to release a new version of the service (a separately compiled program) as a subtype of the old version, which supports both old and new clients and ensures type safety. Remote invocations are syntactically identical to local ones. A client invoking a method of an object need not know whether the object is local or remote. Marshalling (of argument values and results into a sequence of bytes sent between programs) relies on a general-purpose mechanism called *pickles*. Pickles perform efficient and compact marshalling of arbitrary complicated data types. The facility can be used to distribute data and computation by object copying. Network objects are always sent by reference and other objects are always sent by copying. On the other hand, object mobility would allow the same object to be either sent by reference or moved. The authors of Network Objects argue, however, that this extra flexibility does not seem to be worth the substantial increase in the complexity of mobile objects.

The Java programming language and Java environment [GJS97, LY97] provide an object-oriented layer on top of the heterogeneity of the distributed system. Rather then thinking in terms of particular architectures and operating systems as in CORBA or DCOM, the Java environment provides a virtual machine that is (more or less) the same everywhere. This allows code

to be moved from one machine to another, irrespective of the underlying architecture. Applets were the first application of the Java technology to a large scale network. By supplying code that implemented a well-known interface, and making sure that browsers of the World-Wide Web knew how to recognize such code, Java allowed active content to be added to the World-Wide Web. Different implementations for the same interface can be offered and moved dynamically into the browser that made the calls to that interface. However, in order to support the sending of whole objects rather than just code, an additional layer of distributed infrastructure is required.

The Java Remote Method Invocation system (RMI) [WRW96] provides the next step, allowing real objects (both code and data) to be passed from one Java virtual machine to another. RMI uses the standard Java object serialization mechanism to pass objects. Arguments that are references to remote objects are passed as remote references. If an argument to a method is a primitive type or a local (non-remote) object, a copy is passed to the server. Return values are handled in the same way, but in the other direction. RMI lets one pass and return full object graphs for local objects and references to remote objects. The mechanism allows the passing of subtypes to methods declared to use a supertype. If the receiving virtual machine does not have the code associated with the actual class of the object that it receives, the code for that class is downloaded, verified, and dynamically loaded into the receiving virtual machine. RMI does not allow thread mobility.

Jini [AWO+99] forms another layer of infrastructure on top of the Java environment (including RMI), adding a component, called the Lookup Service, that allows services to advertise themselves, and a simple protocol that allows these services and clients wanting to find a service to first find a Lookup Service. By using Jini, matching services and clients can be virtually automatic and transparent. The Jini architecture has been designed for subnets that enable multicasting, restricting the use of Jini to local-area networks. However, some enhancements to overcome this problem were proposed by the Jini community and realised in practice, in order to allow multicast messages to be "tunneled" to other domains through a hierarchical set of daemons.

Java offers typed bytecode and bytecode verification. This is a real advance over systems like Emerald, since one can obtain a class that someone else has compiled and be sure that it is type-safe. However, the Java virtual machine institutionalizes a particular object model, and some authors argue that it is a "technical, commercial, and cultural mistake". In their opinion, a better approach would be a virtual machine that is language-neutral – for example, a virtual RISC processor, which would "evenhandedly and compatibly enable any number of high-level languages for distributed computing" [Whi98]. This could also be done in a type secure way by using a typed assembly language (see, e.g. Morrisett et al. [MCG+99]).

1.1.3 Mobile Agents

Mobile agents are units of executing computation that can migrate between machines (in a local- or wide-area network) and act on behalf of their users or other agents. The attributes of agents such as being autonomous, goal-driven, etc. are investigated in the area of Distributed Artificial Intelligence (DAI). Here, we are only interested in the ability of agents to migrate freely, i.e. to suspend execution at some point, move the whole state of computation to another location, and resume execution at this new location. Agent mobility combines features known from mobile code (such as in Java), object migration (moving encapsulated code and data), and process migration (moving the thread of execution). Additionally, issues of security (such as authentication and authorisation) have to be solved to migrate and execute agents safely.

Many mobile agent systems have been built in Java, using Java support for code mobility; they include Aglets [LOKK97], Voyager [Obj97, Gla98], Concordia [WPW98], Mobile Objects and Agents (MOA) [MLC98], and Mole [SBH96]. TACOMA [JvRS95] and Agent Tcl [Gra95] are systems in which agents can be written using a scripting language Tcl. Some languages and runtime systems have been designed expressly to support mobile agents or mobile computation, such as Telescript [Whi96], the Join Language [FGL+96, CF99], and Nomadic Pict, described in this thesis.

A number of existing mobile agent systems provide a form of location independence; we briefly review some of them below. Comparisons are difficult, in part because of the lack of clear levels of abstraction and descriptions of algorithms — without these, it is hard to understand the performance and robustness properties of the infrastructures.

The Join Language provides location-independent messages using a built-in infrastructure, based on forwarding pointer chains that are collapsed when possible. Voyager supports location-independent messages, both synchronous and asynchronous messages and multicasts, again using forwarding pointer chains that are collapsed when possible. A directory service is also provided. The Mobile Object Workbench [BHDH98] provides location independent interaction, using a hierarchical directory service for locating clusters of objects that have moved. There is a single infrastructure, although it is stated that the architecture is flexible enough to allow others. The infrastructure work of Aridor and Oshima [AO98] provides three main forms of message delivery: location-independent using either forwarding pointers or location servers, and location dependent (they also provide other mechanisms for *locating* an agent). Mobile Objects and Agents (MOA) supports four schemes for locating agents; these are used as required to deliver location-independent messages. Stream communication between agents is also described, with

communicating channel managers informing each other on migration. The MASIF proposal [MBB+98] also involves four locating schemes, but appears to build communication facilities on top. This excludes a number of reasonable infrastructures; it contrasts with our approach here, in which location-independent message delivery is taken as primary (some infrastructures do not support a location service).

1.2 Thesis Contribution

Different forms of mobility, such as process migration, distributed objects, and mobile agents, require specific distributed infrastructures and novel forms of language and runtime support — for interaction between migrating entities, responding to network failure and reconfiguration, support of disconnected operation, binding to resources, managing security, etc. Although the problems which designers of systems with mobility have to solve are often very similar (e.g. maintaining communication between moving entities, resource discovery, etc.), the complexity of plausible implementations and heterogeneity of target environments make it hard to directly transfer and reuse results developed for a particular system.

1.2.1 Observations

Distributed infrastructures are somewhat application-specific. Different applications may require different forms of support for mobility. For example, a non-stop system manager would require thread mobility in order to be able to move all running processes from a node which is (or may soon be) partially faulty to another node. In mobile computing, the desire to support small devices which have limited CPU and memory capabilities (such as PDAs and mobile phones) will require some light-weight infrastructures designed for a particular application. In the example of an "active home", home devices could be plugged into a home LAN using a Jini architecture (which uses code mobility). However, in the vision of ubiquitous computing on the whole Internet, the infrastructure would have to be even more delicate. The problem of scale is not the only one which has to be solved. The wide-area network is more asynchronous and less predictable and manageable than a local-area network, delays and bandwidth fluctuations are unpredictable, failures harder to detect (since remote machines are not under a centralised management and normal disconnected operation of some remote device cannot be distinguished from faulty behaviour). Thus, it may not be easy to extend an infrastructure originally designed for a LAN to a wide-area net-

work by just introducing proxy servers and replicating servers to improve scalability and availability. For example, it may be required to use a *network event notification service* as a building block of the infrastructure, instead of explicitly attempt to detect failures using time-outs. The service would use time-outs only on neighbouring servers and local clients, rather than on processes that are several hops away. It would propagate information about network events to clients that subscribed for it. Also, pervasive computing on the Internet will require good support for (optional) code mobility, disconnected operation, naming and location independence (e.g. to enable binding to local resources after reconnecting a mobile computer at some other point of the network).

It seems unlikely one can build a world-wide distributed infrastructure on the whole wide-area network which could efficiently address all these requirements for all types of applications (it may even not be desirable due to security reasons). Instead, a wide-area network should offer some mechanism to enable many different distributed infrastructures to co-exist on top of some architecture-independent, perfectly scalable and loosely coupled medium. Some light-weight infrastructures could possibly be spawned dynamically with the application — they would form another layer of the legacy system. For example, in chapter 5 we describe a few potential mobile agent applications; they all use only a very limited pattern of migration and communication (we generally do not envisage free-roaming on the whole Internet), thus encouraging the use of infrastructures that are specifically tailored for the application.

The design of distributed applications for wide-area networks (WANs) may require a new model of computation (and so a new kind of programming language). In a LAN we could successfully use some transparent distributed object system in order to build a distributed application, but the intent to use the same application in a WAN, across firewalls and along links with large and highly unpredictable message latency, would require some less transparent way of accessing objects and more asynchrony in the computational model.

1.2.2 Problem Statement

Mobile agent communication primitives can be classified into two groups. At a low level, there are *location dependent* primitives that require a programmer to know the current site of a mobile agent in order to communicate with it. If a party to such communications migrates, then the communicating program must explicitly track its new location. At a high level, there are *location independent* primitives that allow communication with a mobile agent irrespective of its current site and of any migrations of sender or

receiver. Location independent primitives may greatly simplify the development of mobile applications, since they allow movement and interaction to be treated as separate concerns. Their design and implementation, however, raise several difficult issues. A distributed infrastructure is required for tracking migrations and routing messages to migrating agents. This infrastructure must address fundamental network issues such as failures, network latency, locality, and concurrency; the algorithms involved are thus inherently rather delicate and cannot provide perfect location independence. Moreover, applications may be distributed on widely different scales (from local to wide-area networks), may exhibit different patterns of communication and migration, and may demand different levels of performance and robustness; these varying demands will lead to a multiplicity of infrastructures, based on a variety of algorithms. These infrastructure algorithms will be exposed, via their performance and behaviour under failure, to the application programmer — some detailed understanding of an algorithm will be required for the programmer to understand its robustness properties under, for example, failure of a site. Below, we sketch some of the assumptions, which laid the foundation for the Nomadic Pict language.

1.2.3 Project Foundations

A good level of abstraction is needed for our language primitives - high enough not to fuss with marshalling and unmarshalling of data which need to be sent between agents, but low enough to have clear handle on the algorithms which are used in real mobile agent systems. An intuitive example of such abstraction is provided by the Remote Procedure Call (RPC) systems. At the application level, we use transparent method invocations for client-server computing (they involve parameters which are passed to the method and results which are returned to the method caller), and at the low stub level, we have a wire protocol encoded for transferring data from one location in the distributed system to another (which involves messages containing application data and control information, such as acknowledgments).

The need for clear understanding and easy experimentation with infrastructure algorithms, as well as the desire to simultaneously support multiple infrastructures on the same network, suggests a two-level architecture of the language—a low-level consisting of a single set of well-understood, location-dependent primitives, in terms of which a variety of high-level, location-independent communication abstractions may be expressed. This two-level approach enables one to have a standardized low-level runtime system that is common to many machines, with divergent high-level facilities chosen and installed at run time. It also facilitates simple implementation of the location-

independent primitives (cf. protocol stacks).

For this approach to be realistic, it is essential that the low-level primitives should be directly implementable above standard network protocols. The Internet Protocol (IP) supports asynchronous, unordered, point-to-point, unreliable packet delivery; it abstracts from routing. We choose primitives that are directly implementable using asynchronous, unordered, point-to-point, reliable messages. This abstracts away from a multitude of additional details—error correction, retransmission, packet fragmentation, etc.—while still retaining a clear relationship to the well-understood IP level. It is also well suited to the process calculus presentation that we use in 2.2. More controversially, we also include agent migration among the low-level primitives. This requires substantial runtime support in individual network sites, but not sophisticated distributed algorithms—only one message need be sent per migration. By treating it as a low-level primitive we focus attention more sharply on the distributed algorithms supporting location-independent communication. We also provide low-level primitives for agent creation, for sending messages between agents at the same site, for generating globally unique names, and for local computation.

Many forms of high-level communication can be implemented in terms of these low-level primitives, for example synchronous and asynchronous message passing, remote procedure calls, multicasting to agent groups, etc. For the work presented in this dissertation we consider only a single representative form: an asynchronous message-passing primitive similar to the low-level primitive for communication between co-located agents but independent of their locations and transparent to migrations.

1.2.4 Contribution

The main contribution of this dissertation is the design and implementation of Nomadic Pict, a concurrent programming language with thread mobility. The language introduces a new model of concurrent mobile computation in a wide-area network. It assumes the underlying environment is very asynchronous and loosely coupled — the runtime system generally does not depend on any distributed infrastructure — all (application-specific) infrastructure algorithms are executed by the Nomadic Pict virtual machine as normal applications above the network level. Although the desire to have a runtime system implementation purely local may seem to be absolute in the real world, it appeared very useful for prototyping purposes. The implementation of the Nomadic Pict runtime system is very light-weight (even messages are encoded as little agents).

We have used our language to design some non-trivial *infrastructure algorithms* for mobile agent systems. The language has been used to prototype a small example application and the design of a suitable communication infrastructure for it. The infrastructure design, which required good scalability and support for disconnected operation, is an interesting problem in its own right. Our experience of using Nomadic Pict has been positive — the sharp levels of abstraction have aided the design of algorithms. It was also possible to include in this thesis an almost complete specification of the algorithms, expressed as Nomadic Pict encodings.

The Nomadic Pict language allows distributed algorithms to be expressed precisely and unambiguously, but in a compact and clean way; this enables a good understanding of the infrastructure algorithms. The language primitives have simple but powerful semantics. The communication primitives are designed to express easily fundamental concepts of synchronisation and concurrency in the presence of mobility. A polymorphic type system and the notion of agents allow simple objects to be expressed. Although we focus here on distributed infrastructures for location-independent communication between mobile agents, our language can be used for specifying virtually any kind of distributed infrastructure (e.g. distributed garbage-collectors for distributed programming languages).

The work may contribute to future design of specific infrastructures, and also to future industrial languages for distributed programming.

Chapter 2

Model of Mobile Computation

The purpose of this chapter is to introduce the Nomadic π-calculus, a new model of mobile computation. The calculus is a formal extension of the asynchronous π-calculus, also presented in this chapter. The Nomadic π-calculus is designed to model computations with the use of mobile agents. The calculus identifies two levels of abstraction. At a low level there are location dependent primitives that require a programmer to know the current site of a mobile agent in order to communicate with it. At a high level there are location independent primitives that allow communication with a mobile agent irrespective of its current site and of any migrations. Implementation of these requires delicate distributed infrastructure. Our calculus serves as a foundational core for Nomadic Pict language design. We use our language as a concise and precise notation for specifying the infrastructure algorithms.

In our work we deal with *communication* aspects of using mobile agents. By analogy to process communication in distributed operating systems we can think of two low-level methods of communication between agents: message passing, where agents send and receive messages over communication channels, and the concept of logically shared memory, where agents interact by means of a finite collection of shared variables. Shared memory, simulated in distributed systems by message passing, has potential to make it easier to write distributed applications. Unfortunately, it requires a distributed infrastructure which does not scale well (in particular if we allow migration). Therefore, we are primarily concerned with a message passing method as more fundamental in distributed systems. The message-passing style of communication can be efficiently implemented just above IP protocols. Other abstractions at the same level, such as streams which are useful in many agent applications, can be incorporated into our model by simply extending the current calculus.

The message-passing communication of processes and mobile agents can be best understood in terms of a tiny model capturing the essential features of communication and synchronisation, but expressive enough to reason about complex interactions built on top of it. Many different models of sequential and concurrent computation have been proposed. For example, the lambda-calculus invented by Church in the 1930s [Chu32, Chu41], has proven to be a good model of purely functional computation (the programming paradigm where the only observable properties of an expression are its behaviour when applied to arguments). Concurrency, distribution, and recently mobility, have introduced new models of computation, most notably *(state-based) automata models* and *process calculi* (sometimes also called *process algebras*).

Process calculi denote processes and actions in concurrent systems by using algebraic expressions and sets of algebraic operators. They are built around three basic principles [Pie97]: modelling interaction via communication in terms of message passing rather than shared variables, using a small set of basic primitives to specify behaviour of the system, deriving useful algebraic laws for manipulating expressions written using these primitives. We focus here on one process calculus, the π-calculus of Milner, Parrow, and Walker [MPW92, Mil91].

In §2.1 we present a very simple version of the π-calculus, describe its semantics, first informally by comparison to message passing in operating systems, and formally by giving operational semantics. In §2.2 we describe Nomadic π-calculus — a formal extension of the above. We conclude the chapter by showing related work.

2.1 Asynchronous π-Calculus

The π-calculus of Milner, Parrow, and Walker [MPW92, Mil91] is a model of concurrent computation. Below, we present a simple *asynchronous*, choice-free version of the calculus (the asynchronous π-calculus was first proposed by Honda and Tokoro [HT91], and Boudol [Bou92]). The π-calculus has two kinds of entities — concurrent *processes* and communication *channels* (identified by globally unique *names*). The calculus allows communication between concurrent processes by an output and an input (on the same channel) in parallel. One of its goals is to demonstrate that in some sense it is sufficiently powerful to allow only names to be the content of communications. Names have no structure, while the syntax of processes is as follows.

2.1.1 Syntax

Take an infinite set $\mathcal{N}$ of *names* of channels, ranged over by $x, y, \ldots$ etc. The *process terms* are then those defined by the grammar in Figure 2.1[1][2].

P, Q	::=	$()$	nil
		$P \mid Q$	parallel composition of P and Q
		$x!v$	output v on channel x
		$x?p \rightarrow P$	input from channel x
		$x?*p \rightarrow P$	replicated input from channel x
		new x **in** P	new channel name creation

Figure 2.1: Syntax of the π-Calculus

The term $()$ represents an inactive process, which cannot perform any action. The form $P \mid Q$ means that P and Q are concurrently active, and can also communicate. Intuitively, a process term $x!v$ sends the name v on channel x. A process term $x?w \rightarrow P$ waits to receive a name on x, substitutes w in P by this name after reception, and continues with P. Placing the restriction operator **new** x **in** before a process expression P ensures that x is a fresh channel in P — i.e. messages sent and received on x will never be mixed with messages sent on any other channel created elsewhere, even if that channel would happen to be named x too. In $x?w \rightarrow P$ the 'formal parameter' w binds in P; in **new** x **in** P the x binds in P. We will work up to alpha renaming of bound names so as to avoid name clashes (in the same way as we are allowed to rename formal parameters and their occurrences in a function definition). We write $\{a/x\}P$ for the process term obtained from P by replacing all free occurrences of x by a, renaming as necessary to avoid capture. We assign parallel composition the lowest precedence among the operators. Substitutions have precedence over the operators of the calculus.

[1] Here we have adopted the language concrete syntax, instead of a more concise, mathematical style usually found in the literature on process calculi.

[2] An original definition of the π-calculus also includes a choice operator $+$; the expression $P+Q$ denotes an external choice between P and Q: either P is allowed to proceed and Q is discarded, or vice versa; we drop the full choice here as it is not very useful for programming in our language (input-only choice, which seems more useful, can be encoded as a library module).

2.1.2 Informal Semantics

We first try to describe the semantics of π-calculus communication informally, using the communication metaphors of message-passing in network architectures. In network architectures and distributed systems (described, e.g. in [CDK94]), processors share only a communication network. We can think of a π-calculus *channel* as an abstraction of a physical communication network. It provides a communication path between processes, a means for data to be transferred between them. A *process* here is a running program in the sense understood in the field of operating systems; it consists of an environment for execution together with a thread of control. Communication is accomplished when one process sends a message to a channel and another (concurrent) process acquires the message by receiving from the same channel. The message can only be received after it has been sent (causal ordering).

One of the arguments of a send operation must specify an identifier denoting the message destination address. This identifier must be known to any process that wishes to send to this address. In the Internet protocols, destination identifiers for messages are specified as the Internet address of the host computer (as in IP) or a pair of the Internet address and a fixed port number attached to the host computer on which a receiving process runs (as in UDP). In the π-calculus, however, there is no notion of physical machines, channels are linked directly to processes, and channel *names* are used for communication addresses. Therefore, perhaps a better intuition would be provided by distributed operating systems such as Amoeba [MvR92], in which messages are transmitted directly to processes, or to communication ports that are attached to processes (a *communication port* here is one of several alternative points of entry to the receiving process). Messages are addressed by specifying port *names*.

The semantics of π-channels is however different from semantics of communication ports, as shown below. A communication port is a message destination that can have many senders, but has exactly one receiver. In the π-calculus, many concurrent processes can share the same channel for input, although only one process will succeed in receiving a message sent by some other process[1]. Here v is received on x by either P or Q

$$x!v \mid x?u \rightarrow P \mid x?w \rightarrow Q$$

[1]Some operating systems provide the ability to send a message to groups of destinations (either ports or processes) identified by a group identifier. A message addressed by such a group identifier will be received by all group members. There is no similar operation in the π-calculus.

There can be many outputs on the same channel competing for the same input — only one will succeed, introducing nondeterminism. For example process P can receive either a or b on x

$$x!a \mid x!b \mid x?u \rightarrow P$$

If we want to model process P which is always ready to receive a new message (similar to a process listening on the communication port), we can use a replicated input

$$x!a \mid x!b \mid x?*u \rightarrow P$$

Π-calculus names can be dynamically generated and communicated between processes; every process which has obtained a channel name x can use it for unrestricted communication (in particular the process can *read* from x). This allows modelling of systems with evolving connectivity structures. Pure port names can only have output capability. Thus, there is no simple analogy between π-calculus channels and communication ports as described above.

A port has usually a message queue to store incoming messages. Sending processes add messages to the queue and the receiver process removes messages from the queue. The send and receive operations in operating systems can include *synchronisation* of the receiving operation with the sending operation, so that the sending or receiving process is prevented from continuing until the other process makes an action that frees it (much as semaphore operations on shared variables). In the *asynchronous* form of communication, the sending process is allowed to proceed as soon as the message has been copied to a local buffer and the transmission of the message proceeds in parallel with the sending process. The receive operation can have *blocking* or *non-blocking* variants. In the original π-calculus, the communication is *synchronous* — the input and output processes synchronize at every message. Here, we only consider a variant with asynchronous communication and blocking input, where the writer can continue computation after sending value to a channel, but the reader is always blocked if there is no message to read. This style seems more practical in distributed systems. The asynchronous version is known to be powerful enough to encode the synchronous message passing discipline of the π-calculus (see [HT91, Bou92]).

The issue of channel implementation is hidden from the model. The calculus assumes that channels are *global* (each process which obtained a channel name can read messages sent to the channel by any other process) and *unbounded* (i.e. they are never full; therefore an output will never be blocked due to overfilling the channel). There is no guarantee on the order

of message delivery, much like in connectionless transport protocols, such as UDP, in which neither the transport nor the network layer is required to perform any sequencing of data packets. The UDP service does not guarantee that all messages sent are actually received at the destination, however it makes its 'best effort' to deliver each message. It is a responsibility of higher-level services to support reliable delivery. In the π-calculus, we assume that messages are never lost or duplicated[1].

Although we have used an analogy to message-passing, we should not, however, forget that the π-calculus is simply a model of concurrent *computation* — there is neither the notion of process locations in the calculus, nor the physical separation between the sender and receiver. Communication (or computation) is assumed to be error-free; process failures cannot be expressed in the calculus. However, many other π-calculi (including the Nomadic π-calculus) identify the problem of distribution, and they do it in the context of network communication. In the Nomadic Pict language, bare π-calculus, as described here, is used as a means for expressing local concurrent computation within an agent.

2.1.3 Operational Semantics

The operational semantics of π-calculus expressions and operations on channels is usually defined as a *reduction* relation, as in the lambda-calculus. We say that P reduces to Q, written $P \longrightarrow Q$, if P contains two parallel subprocesses that can communicate on the same channel to become the corresponding subprocesses of process Q. For example, in the expression $x!a \mid x?u \rightarrow R$, first two subprocesses can communicate, the value a is being sent along the channel x, reducing the whole expression to $() \mid \{a/u\}R$. We normally drop inactive processes $()$. In order to illustrate substitution $\{a/u\}R$, let R be $y!u$. Then the data value a is substituted for the bound variable u in R as follows:

$$x!a \mid x?u \rightarrow y!u \quad \longrightarrow \quad y!a$$

A replicated input $x?*p \rightarrow P$ can be used to construct a server which after reading a value from x, is ready to accept a new input on x; it loosely

[1]Nomadic π-calculus (described below) introduces messages which can be misaddressed and discarded ('lost'), and timed inputs to model situations where the potential sending process has crashed or the expected message has been lost. A *timeout* specifies an interval of time after which the input operation will give up its action.

behaves as an arbitrary number of parallel copies of $x?p \rightarrow P$:

$$\begin{array}{rcl} x!a \mid x?*u \rightarrow y!u & \longrightarrow & \{a/u\}(y!u) \mid x?*u \rightarrow y!u \\ & \longrightarrow & y!a \mid x?*u \rightarrow y!u \end{array}$$

More importantly, the replicated input term allows the encoding of infinite computations (similarly to the way in which *recursion* is used for this purpose in pure functional languages). For example, a process term $x?*u \rightarrow (y!u \mid x!v)$ responds to a message on x by sending the message on y and 'triggering' another copy of itself by sending another message on x, thus leading to an infinite computation (here a continuous stream of v's on y).

$$\begin{array}{rcl} (x?*u \rightarrow (y!u \mid x!u)) \mid x!v & \longrightarrow & (x?*u \rightarrow (y!u \mid x!u)) \mid \underline{y!v} \mid x!v \\ & \longrightarrow & ... \\ & \longrightarrow & (x?*u \rightarrow (y!u \mid x!u)) \mid \underline{y!v \mid ... \mid y!v} \mid x!v \end{array}$$

The data values sent on channels are just names. In particular, a name received on a channel can then be used itself as a channel name for output or input. The strength and subtlety of the calculus comes from the dynamic character of name scoping. A restricted name can be sent (exported) outside its original scope (this is known as *scope extrusion*). For example, in the expression $(x?y \rightarrow y!u) \mid (\textbf{new } z \textbf{ in } (x!z \mid z?v \rightarrow v!a))$, we create a new fresh channel z, which will be exported outside its original scope, and used for communication. Names x, u, and a are free (they have been either created or imported by some process which our expression is part of). Initially, the scope of z is limited to the second branch of the parallel, and z is unknown in the first branch. One can then pass z to the first branch on x — outside the scope of the initial **new** z **in** binder which must therefore be moved (with care, to avoid capture of other instances of z), obtaining the term **new** z **in** $(z!u \mid z?v \rightarrow v!a)$. From now on, z can be used as a communication channel between both branches and our term reduces to **new** z **in** $u!a$ which is structurally equivalent to $u!a$.

The operational semantics can be defined in two steps, by giving a definition of a *structural congruence* (written $\equiv$) and the binary reduction relation $\longrightarrow$ over process terms. The structural congruence relation formalizes the intuition that we can always rearrange a reducible process term such as to enable reduction (we can change the order of parallel compositions, enlarge the scope of bindings, or garbage-collect null processes and names which will be no longer used). Rules for our simple π-calculus are grouped in Figure 2.2.

Structural congruence:

$$
\begin{array}{rcll}
P \mid 0 & \equiv & P & \\
P \mid Q & \equiv & Q \mid P & \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & \\
\textbf{new } x \textbf{ in new } y \textbf{ in } P & \equiv & \textbf{new } y \textbf{ in new } x \textbf{ in } P & \\
P \mid \textbf{new } x \textbf{ in } Q & \equiv & \textbf{new } x \textbf{ in } (P \mid Q) & \text{if } x \notin \mathrm{FV}(P) \\
\textbf{new } x \textbf{ in } P & \equiv & P & \text{if } x \notin \mathrm{FV}(P)
\end{array}
$$

Renaming of bound variables

$$
\begin{array}{rcll}
x?w \rightarrow P & \equiv & x?v \rightarrow (\{v/w\}P) & \text{if } v \notin \mathrm{FV}(P) \\
\textbf{new } x \textbf{ in } P & \equiv & \textbf{new } y \textbf{ in } (\{y/x\}P) & \text{if } y \notin \mathrm{FV}(P)
\end{array}
$$

Reduction semantics:

$$
\begin{array}{rl}
c!v \mid c?w \rightarrow P \longrightarrow \{v/w\}P & \text{communication} \\
c!v \mid c?*w \rightarrow P \longrightarrow c?*w \rightarrow P \mid \{v/w\}P & \text{communication and replication} \\
\dfrac{P \longrightarrow Q}{P \mid R \longrightarrow Q \mid R} & \text{reduction under } \mid \\
\dfrac{P \longrightarrow Q}{\textbf{new } x \textbf{ in } P \longrightarrow \textbf{new } x \textbf{ in } Q} & \text{reduction under } \textbf{new} \ldots \textbf{in} \\
\dfrac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q} & \text{structural congruence}
\end{array}
$$

Figure 2.2: Operational Semantics of the π-Calculus

There is potentially a large set of different variants of the π-calculus presented here. We mention some of these calculi in the end of this chapter. There are also a number of extensions which are useful in designing a programming language based on process calculi. A natural extension is to allow tuples of names to be sent (as in *polyadic* π-calculus), or allow more general data, e.g. tuples-of-tuples and basic values such as booleans, strings and natural numbers. Another extension would be to have *recursion*, e.g. with process variable X and a recursion operator `rec` $X.P$.

Having defined a set of basic operators (syntax) and operational semantics, a notion of observational (or behavioural) equivalence and congruence can be introduced. This makes it possible to reason about the behaviour of communicating processes in a formal theoretical framework. The formal framework and proof methods developed within the π-calculus (e.g. based on bisimulation) are beyond scope of this thesis. A theory of bisimulation for the π-calculus is described, e.g. in [San96, MPW92]. Honda and Tokoro [HT91], and Amadio, Castellani, and Sangiorgi [ACS98] present two different notions of bisimulation for the asynchronous π-calculus. More about π-calculus can be found in Milner's book [Mil99], see also good introductory texts such as a chapter of the "Computer Science and Engineering Handbook" [Pie97] and the tutorials [Mil91, San99, San, Sew].

2.2 Nomadic π-Calculus

We were looking for a calculus which would lay down a foundation for a distributed programming language, suitable for describing infrastructure algorithms for mobile agent systems. The asynchronous π-calculus described above provides an abstract model of concurrent computation. The model is based on a reduced set of concepts which allows expressing the dynamic generation of names and processes, communication on abstract channels, transmission of channel names between processes, and a static scoping discipline. However, the calculus does not provide support for modelling *distributed programming*; we are not able to express in it, e.g. the notion of computer nodes, allocation of resources to these nodes, process mobility, and system failures. Therefore, we proposed the *Nomadic π-calculus*, a calculus which captures some of these formally, enough to provide a clean and efficient strategy for the programming language design. Most notably, it identifies two levels of abstraction suitable for formal reasoning about infrastructure algorithms.

In this section our two levels of abstraction are made precise by giving two corresponding process calculi, the low- and high-level Nomadic π-calculi. Their design involves a delicate trade-off — the distributed infrastructure

algorithms that we want to express involve non-trivial local computation within agents, yet for the theory to be tractable (particularly, for operational congruences to have tractable characterisations) the calculi must be kept as simple as possible. The primitives for agent creation, agent migration and inter-agent communication that we consider do not suffice to allow the required local computation to be expressed clearly, so we integrate them with those of the asynchronous π-calculus presented above. The other computational constructs that will be needed, e.g. for finite maps, can then be regarded as lightweight syntactic sugar for π-processes. The advantage of selecting the basic model of the π-calculus on which to add additional features is that we will be able to inherit and state many of the results and proof methods developed within the π-calculus theory.

The low- and high-level calculi are introduced in §2.2.1 and §2.2.2 respectively. The operational semantics of the calculi are described informally — the precise reduction semantics will be given in §2.2.3. For simplicity, the calculi are presented without typing or basic values (such as integers and booleans). Types and basic values will be introduced in chapter 3, describing the Nomadic Pict language.

2.2.1 Low-Level Calculus

We begin with an example. Below is a term of the low-level calculus showing how an applet server can be expressed. It can receive (on the channel named *getApplet*) requests for an applet; the requests contain a pair (bound to a and s) consisting of the name of the requesting agent and the name of its site.

$$
\begin{array}{l}
\mathit{getApplet}?*[a\ s] \rightarrow \\
\quad \textbf{agent}\ b = \\
\quad\quad \textbf{migrate to}\ s \rightarrow (\langle a@s \rangle \mathit{ack}!b \mid B) \\
\quad \textbf{in} \\
\quad\quad 0
\end{array}
$$

When a request is received the server creates an applet agent with a new name bound to b. This agent immediately migrates to site s. It then sends an acknowledgement to the requesting agent a (which is assumed to also be on site s) containing its name. In parallel, the body B of the applet commences execution.

The example illustrates the main entities represented in the calculus: sites, agents and channels. *Sites* should be thought of as abstractions of physical machines or, more accurately, as instantiations of the Nomadic Pict runtime system on machines; each site has a unique name. The calculus does

not explicitly address questions of site failure, network failure and reconfiguration, or security. Sites are therefore unstructured; neither network topology nor administrative domains are represented in the formalism. *Agents* are units of executing code; an agent has a unique name and a body consisting of some process term; at any moment it is located at a particular site. *Channels* support communication within agents, and also provide targets for inter-agent communication—an inter-agent message will be sent to a particular channel within the destination agent. Channels also have unique names.

The inter-agent message $\langle a@s\rangle ack!b$ is characteristic of the low-level calculus. It is location-dependent—if agent a is in fact on site s then the message b will be delivered, to channel ack in a; otherwise the message will be discarded. In an implementation at most one inter-site message is sent.

Names As in the π-calculus, names play a key rôle. We take an infinite set $\mathcal{N}$ of names, ranged over by a, b, c, d, e, f, s, x and y. Formally, all names are treated identically; informally, a and b will be used for agent names, c, d, e, f for channel names, and s for a site name. (A type system of the language will allow these distinctions to be enforced.) The calculus allows new names (of agents and channels) to be created dynamically.

Names are *pure*, in the sense of Needham [Nee89]; no information about their creation is visible within the calculus and language (in our current implementation they do contain site IDs, but could equally well be implemented by any mechanism that allows globally-unique bit strings to be created locally, e.g. by choosing large random numbers).

Values We allow the communication of first-order values, consisting of names and tuples.

u, v	::=	x	name
		$[v_1 \,..\, v_n]$	tuple ($n \geq 0$)

Patterns As in the π-calculus, values are deconstructed by pattern matching on input. Patterns have the same form as values, with the addition of a wildcard.

p	::=	$\_$	wildcard
		x	name pattern
		$[p_1 \,..\, p_n]$	tuple pattern ($n \geq 0$, no repeated names)

Process terms The main syntactic category is that of *process terms*, ranged over by P, Q. We will introduce the low-level primitives in groups.

agent $a = P$ **in** Q	agent creation
migrate to $s \to P$	agent migration

The execution of the construct **agent** $a = P$ **in** Q spawns a new agent on the current site, with body P. After the creation, Q commences execution, in parallel with the rest of the body of the spawning agent. The new agent has a unique name which may be referred to both in its body and in the spawning agent (i.e. a is binding in P and Q). Agents can migrate to named sites — the execution of **migrate to** $s \to P$ as part of an agent results in the whole agent migrating to site s. After the migration, P commences execution in parallel with the rest of the body of the agent.

$P \mid Q$	parallel composition
0	nil

The body of an agent may consist of many process terms in parallel, i.e. essentially of many lightweight threads. They will interact only by message passing.

new c **in** P	new channel name creation
$c!v$	output v on channel c in the current agent
$c?p \to P$	input from channel c
$c?*p \to P$	replicated input from channel c
if $u = v$ **then** P **else** Q	value equality testing

To express computation within an agent, while keeping a lightweight implementation and semantics, we include π-calculus-style interaction primitives described in 2.1. Execution of **new** c **in** P creates a new unique channel name; c is binding in P. An output $c!v$ (of value v on channel c) and an input $c?p \to P$ in the same agent may synchronise, resulting in P with the names in the pattern p replaced by corresponding parts of v (the output is asynchronous — note that we do not have $c!p \to Q$ in the syntax). A replicated input $c?*p \to P$ behaves similarly except that it persists after the synchronisation, and so may receive another value. In both $c?p \to P$ and $c?*p \to P$ the names in p are binding in P. The conditional allows any two values to be tested for equality.

wait $c?p \to P$, $n \to Q$	input with timeout

For implementing infrastructures that are robust under some level of failure, or support disconnected operation, some timed primitive is required. The

low-level calculus includes a single timed input as above, with timeout value n. If a message on channel c is received within n seconds then P will be started as in a normal input, otherwise Q will be. The timing is approximate, as the runtime system may introduce some delays.

$$\textbf{iflocal } \langle a \rangle c!v \to P \textbf{ else } Q \quad \text{test-and-send to agent } a \text{ on current site}$$

Finally, the low-level calculus includes a single primitive for interaction between agents. The execution of **iflocal** $\langle a \rangle c!v \to P$ **else** Q in the body of an agent b has two possible outcomes. If agent a is on the same site as b, then the message $c!v$ will be delivered to a (where it may later interact with an input) and P will commence execution in parallel with the rest of the body of b; otherwise the message will be discarded, and Q will execute as part of b. The construct is analogous to test-and-set operations in shared memory systems — delivering the message and starting P, or discarding it and starting Q, atomically. It can greatly simplify algorithms that involve communication with agents that may migrate away at any time, yet it is still implementable locally, by the runtime system on each site.

Syntactic sugar Empty tuples and tuple patterns will generally be elided, writing $c!$ and $c? \to P$ for $c![]$ and $c?[] \to P$. Multiple new channel bindings will be coalesced, writing **new** c, c' **in** P for **new** c **in new** c' **in** P. Let-declarations will be used, writing **let** $p = v$ **in** P for **new** c **in** $(c!v \mid c?p \to P)$ (where c is a name not occurring free in v or P).

Scope extrusion Channel names are first-class values and they can be freely sent to processes which are located at other agents. As in the π-calculus, names can be *scope-extruded* — here channel and agent names can be sent outside the agent in which they were created. For example, if the body of agent a is

$$\begin{array}{l}\textbf{agent } b = \\ \quad \textbf{new } d \textbf{ in} \\ \quad\quad \textbf{iflocal } \langle a \rangle c!d \to 0 \textbf{ else } 0 \\ \textbf{in} \\ \quad c?x \to x!\end{array}$$

then channel name d is created in agent b. After the output message $c!d$ has been sent from b to a (by **iflocal**) and has interacted with the input $c?x \to x!$ there will be an output $d!$ in agent a.

We require a clear relationship between the semantics of the low-level calculus and the inter-machine messages that are sent in the implementation. To achieve this we allow direct communication between outputs and inputs on a channel only if they are *in the same agent* — messages can be sent from one agent to another only by **iflocal**. Intuitively, there is a distinct π-calculus-style channel for each channel name in every agent. For example, if the body of agent a is

$$\begin{array}{l}\textbf{agent } b = \\ \quad \textbf{new } d \textbf{ in} \\ \qquad d? \rightarrow 0 \\ \qquad \mid \textbf{iflocal } \langle a \rangle c!d \rightarrow 0 \textbf{ else } 0 \\ \textbf{in} \\ \quad c?x \rightarrow x!\end{array}$$

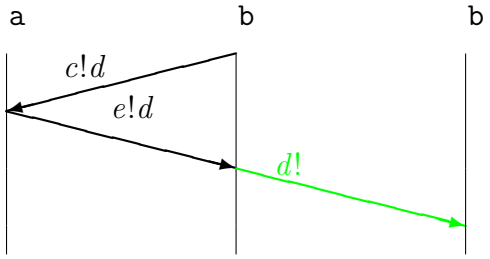
then after some reduction steps a contains an output on d and b contains an input on d, but these cannot react. At first sight this semantics may seem counter-intuitive, but it reconciles the conflicting requirements of expressiveness and simplicity of the calculus. An implementation creates the mailbox datastructure — a queue of pending outputs or inputs — required to implement a channel as required; it could be garbage collected when empty. The queue is part of an agent's state which is transferred with every move of the agent.

Communication of names between agents preserves locality of channels. For example, if the body of agent a is

$$\begin{array}{l}\textbf{agent } b = \\ \quad \textbf{new } d \textbf{ in} \\ \qquad (d? \rightarrow 0 \\ \qquad \mid \textbf{iflocal } \langle a \rangle c!d \rightarrow 0 \textbf{ else } 0) \\ \quad \mid e?y \rightarrow y! \\ \textbf{in} \\ \quad c?x \rightarrow \textbf{iflocal } \langle b \rangle e!x \rightarrow 0 \textbf{ else } 0\end{array}$$

then after some reduction steps agent b contains both an input and an output on d which can synchronise according to the scope extrusion rule of the π-calculus, as follows. In the second line, we create a fresh channel name d; the name is bound in the process put in brackets. A message $c!d$ is sent to agent a (by **iflocal**) where it can interact with the input on c. After agent a has obtained d from agent b, the name d is exported back to agent b, received on e (*outside* the original scope of d), and used for communication with the input following **new** d **in** . A sample execution is below (a grey arrow illustrates

communication inside agent b).



Syntax Summarizing, the terms of the low-level calculus are presented in Figure 2.3. Note that the only primitive which involves network commu-

P, Q	::=	
	agent $a = P$ **in** Q	agent creation
	migrate to $s \to P$	agent migration
	$P \mid Q$	parallel composition
	0	nil
	new c **in** P	new channel name creation
	$c!v$	output v on channel c in the current agent
	$c?p \to P$	input from channel c
	$c?*p \to P$	replicated input from channel c
	wait $c?p \to P$, $n \to Q$	input with timeout
	if $u = v$ **then** P **else** Q	value equality testing
	iflocal $\langle a \rangle c!v \to P$ **else** Q	test-and-send to agent a on current site

Figure 2.3: Syntax of the Low-Level Nomadic π-Calculus

nication is **migrate**, which requires at most one (reliable) message to be sent, asynchronously, between machines. Distributed implementation of the low-level calculus is therefore straightforward, requiring no non-trivial distributed algorithms. It could be done either above a reliable datagram layer or above TCP, using a lightweight layer that opens and closes streams as required. In the current implementation of Nomadic Pict we build upon TCP connections.

Two other useful forms of location-dependent output are expressible in the calculus given.

$\langle a \rangle c!v$	output to agent a on the current site
$\langle a@s \rangle c!v$	output to agent a on site s

The execution of an output $\langle a \rangle c!v$ in the body of an agent b will either deliver the message $c!v$ to agent a, if agent b is on the same site as a, or will silently discard the message, if not. The execution of an output $\langle a@s \rangle c!v$ in the body of an agent will either deliver the message $c!v$ to agent a, if agent a is on site s, or will silently discard the message, if not. We regard these as syntactic sugar for

$$\textbf{iflocal } \langle a \rangle c!v \rightarrow 0 \textbf{ else } 0$$

and

$$\textbf{agent } b = (\textbf{migrate to } s \rightarrow (\textbf{iflocal } \langle a \rangle c!v \rightarrow 0 \textbf{ else } 0)) \textbf{ in } 0$$

(where b is fresh) respectively. Since the primitives fail silently if a is not where expected, they are usually used only where `a`'s location is predictable. In an implementation, the first is implementable locally; the second requires only one asynchronous network message. Note that one could optimize the case in which the second is used on site s itself by trying **iflocal** first:

$$\begin{array}{l} \textbf{iflocal } \langle a \rangle c!v \rightarrow \\ \quad 0 \\ \textbf{else} \\ \quad \textbf{agent } b = (\textbf{migrate to } s \rightarrow (\textbf{iflocal } \langle a \rangle c!v \rightarrow 0 \textbf{ else } 0)) \textbf{ in } 0 \end{array}$$

2.2.2 High-Level Calculus

The high-level calculus is obtained by extending the low-level calculus with a single location-independent communication primitive:

$\langle a@? \rangle c!v$	location-independent output to agent a

The intended semantics of an output $\langle a@? \rangle c!v$ is that its execution will reliably deliver the message $c!v$ to agent a, irrespective of the current site of a and of any migrations.

2.2.3 Reduction Semantics

The informal descriptions of the primitives in §2.2.1, 2.2.2 can be made precise by giving them an operational semantics. We adopt a *reduction semantics*, defining the atomic state-changes that a system of agents can undergo by reduction axioms with a structural congruence, following the style of [BB92, Mil92].

The process terms of the calculi in §2.2.1,2.2.2 only allow the source code of the body of a single agent to be expressed. During computation, this agent may evolve into a system of many agents, distributed over many sites. The reduction relation must be between the possible states of these systems, not simply between terms of the source calculi; we express such states as *configurations* Γ, P. Here Γ is a *location context* that gives the current site of any free agent names; P is a term of the (low- or high-level) calculus extended with three new forms.

$@_a\, P$	P as part of agent a
new $a@s$ **in** P	new agent name a, currently at site s
$\mathbf{wait}_t\ c?p \to P,\, Q$	input with timeout at t (UTC)

Configurations may involve many agents in parallel. The form $@_a\, P$ denotes the process term P as part of the body of agent a, so for example $@_a\, P \mid @_b\, Q$ denotes P as part of the body of a in parallel with Q as part of the body of b. It will be convenient to allow the parts of the body of an agent to be syntactically separated, so e.g. $@_a\, P_1 \mid @_b\, Q \mid @_a\, P_2$ denotes $P_1 \mid P_2$ as part of a in parallel with Q as part of b. Configurations must record the current sites of all agents. For free agent names this is done by the location context Γ; for the others, the form **new** $a@s$ **in** P declares a new agent name a, which is binding in P, and records that agent a is currently at site s.

We now give the detailed definitions. Process terms are taken up to alpha-conversion throughout. Structural congruence $\equiv$ includes the axiom

$$@_a\,(P \mid Q) \equiv @_a\, P \mid @_a\, Q$$

allowing the parts of an agent a to be syntactically separated or brought together, and the axiom

$$@_a\, \mathbf{new}\ c\ \mathbf{in}\ P \equiv \mathbf{new}\ c\ \mathbf{in}\ @_a\, P \qquad \text{if } c \neq a$$

allowing channel binders to be extruded past $@_a$. It is otherwise similar to a standard structural congruence for an asynchronous π-calculus, with

scope extrusion both for the new channel binder **new** c **in** P and for the new agent binder **new** $a@s$ **in** P. In full, it is the least congruence satisfying the following axioms.

$$\begin{array}{rcll} P & \equiv & P \,|\, 0 & \\ P \,|\, Q & \equiv & Q \,|\, P & \\ P \,|\, (Q \,|\, R) & \equiv & (P \,|\, Q) \,|\, R & \\ P \,|\, \textbf{new } c \textbf{ in } Q & \equiv & \textbf{new } c \textbf{ in } P \,|\, Q & \text{if } c \text{ not free in } P \\ P \,|\, \textbf{new } a@s \textbf{ in } Q & \equiv & \textbf{new } a@s \textbf{ in } P \,|\, Q & \text{if } a \text{ not free in } P \\ @_a \,(P \,|\, Q) & \equiv & @_a \, P \,|\, @_a \, Q & \\ @_a \, \textbf{new } c \textbf{ in } P & \equiv & \textbf{new } c \textbf{ in } @_a \, P & \text{if } c \neq a \end{array}$$

A configuration is a pair Γ, P, where the location context Γ is a finite partial function from $\mathcal{N}$ to $\mathcal{N}$, intuitively giving the current site of any free agent names in P, and P is a term of the (low- or high-level) extended calculus. The initial configuration, for a program P of the (low- or high-level) unextended calculus, to be considered as the body of an agent a created on site s, is:

$$\{a \mapsto s\},\ @_a \, P$$

We are concerned only with configurations that can arise by reduction of initial configurations for well-typed programs. In these, any particle (i.e., **agent**, **migrate**, output, input, **if**, or **iflocal**) will be under exactly one @ operator, specifying the agent that contains it. (In this presentation of the Nomadic π-calculus we do not give a type system, and so leave this informal.) Other configurations have mathematically well-defined reductions but may not be easily implementable or desirable, for example

$$\Gamma,\ @_a \, (c?b \rightarrow @_b \, P)$$

receives an agent name and then adds P to the body of that agent.

We define a partial function match, taking a value and a pattern and giving (where it is defined) a finite substitution from names to values.

$$\begin{array}{rcl} \text{match}(v, \_) & = & \{\} \\ \text{match}(v, x) & = & \{x \mapsto v\} \\ \text{match}([v_1 \,..\, v_m], [p_1 \,..\, p_m]) & = & \text{match}(v_1, p_1) \cup \ldots \cup \text{match}(v_m, p_m) \\ \text{match}(v, [p_1 \,..\, p_m]) & & \text{undefined, if } v \text{ is not of the form } [v_1 \,..\, v_m] \end{array}$$

The natural definition of the application of a substitution from names to values to a process term P is also a partial operation, as the syntax does not allow arbitrary values in all the places where free names can occur. We

write $\{v/p\}P$ for the result of applying the substitution $\mathrm{match}(v,p)$ to P. This may be undefined either because $\mathrm{match}(v,p)$ is undefined, or because $\mathrm{match}(v,p)$ is a substitution but the application of that substitution to P is undefined.

The reduction axioms for the low-level calculus are as follows.

$$\begin{array}{lcll}
\Gamma, @_a\,\mathbf{agent}\ b = P\ \mathbf{in}\ Q & \longrightarrow & \Gamma, \mathbf{new}\ b@\Gamma(a)\ \mathbf{in}\ (@_b\,P | @_a\,Q) & \\
\Gamma, @_a\,\mathbf{migrate\ to}\ s \to P & \longrightarrow & (\Gamma \oplus a \mapsto s), @_a\,P & \\
\Gamma, @_a\,\mathbf{iflocal}\ \langle b\rangle c!v \to P\ \mathbf{else}\ Q & \longrightarrow & \Gamma, @_b\,c!v | @_a\,P & \text{if } \Gamma(a) = \Gamma(b) \\
 & \longrightarrow & \Gamma, @_a\,Q & \text{if } \Gamma(a) \neq \Gamma(b) \\
\Gamma, @_a\,(c!v | c?p \to P) & \longrightarrow & \Gamma, @_a\,\{v/p\}P & \\
\Gamma, @_a\,(c!v | c?{*}p \to P) & \longrightarrow & \Gamma, @_a\,(\{v/p\}P | c?{*}p \to P) & \\
\Gamma, @_a\,\mathbf{if}\ u = v\ \mathbf{then}\ P\ \mathbf{else}\ Q & \longrightarrow & \Gamma, @_a\,P & \text{if } u = v \\
 & \longrightarrow & \Gamma, @_a\,Q & \text{if } u \neq v
\end{array}$$

To express the reduction axioms for an input with timeout, we need to write the configuration as a triple Γ, t, P, where t is the global time UTC (Coordinated Universal Time). The reduction axioms below are to illustrate the language implementation issues — they do not currently form part of the Nomadic π-calculus operational semantics which is used for reasoning formally within our model, e.g. in [Uny]. Thus, we can drop t in all other contexts.

$$\begin{array}{lcll}
\Gamma, t, @_a\,\mathbf{wait}\ c?p \to P,\ n \to Q & \longrightarrow & \Gamma, t+1, @_a\,\mathbf{wait}_{t+n}\ c?p \to P,\ Q & \\
\Gamma, t, @_a\,\mathbf{wait}_{t'}\ c?p \to P,\ Q & \longrightarrow & \Gamma, t+1, @_a\,Q & \text{if } t \geq t' \\
\Gamma, t, @_a\,(c!v | \mathbf{wait}_{t'}\ c?p \to P,\ Q) & \longrightarrow & \Gamma, t+1, @_a\,\{v/p\}P &
\end{array}$$

The rules mentioning potentially-undefined expressions $\Gamma(x)$ or $\{v/p\}P$ in their side-condition or conclusion have an implicit additional premise that these are defined. Such premises should be automatically satisfied in derivations of reductions of well-typed programs.

Note that the only inter-site communication in an implementation will be for the **migrate** reduction, in which the body of the migrating agent a must be sent from its current site to site s.

The high-level calculus has the additional axiom below, for delivering location-independent messages to their destination agent.

$$\Gamma, @_a\,\langle b@?\rangle c!v \quad\longrightarrow\quad \Gamma, @_b\,c!v$$

Structural congruence:

$$
\begin{array}{rcll}
P & \equiv & P \,|\, 0 & \\
P \,|\, Q & \equiv & Q \,|\, P & \\
P \,|\, (Q \,|\, R) & \equiv & (P \,|\, Q) \,|\, R & \\
P \,|\, \textbf{new } c \textbf{ in } Q & \equiv & \textbf{new } c \textbf{ in } P \,|\, Q & \text{if } c \text{ not free in } P \\
P \,|\, \textbf{new } a@s \textbf{ in } Q & \equiv & \textbf{new } a@s \textbf{ in } P \,|\, Q & \text{if } a \text{ not free in } P \\
@_a\,(P \,|\, Q) & \equiv & @_a\,P \,|\, @_a\,Q & \\
@_a \textbf{ new } c \textbf{ in } P & \equiv & \textbf{new } c \textbf{ in } @_a\,P & \text{if } c \neq a
\end{array}
$$

Reduction semantics:

$$
\begin{array}{lcll}
\Gamma, @_a \textbf{ agent } b = P \textbf{ in } Q & \longrightarrow & \Gamma, \textbf{new } b@\Gamma(a) \textbf{ in } (@_b\,P | @_a\,Q) & \\
\Gamma, @_a \textbf{ migrate to } s \to P & \longrightarrow & (\Gamma \oplus a \mapsto s), @_a\,P & \\
\Gamma, @_a \textbf{ iflocal } \langle b \rangle c!v \to P \textbf{ else } Q & \longrightarrow & \Gamma, @_b\,c!v | @_a\,P & \text{if } \Gamma(a) = \Gamma(b) \\
 & \longrightarrow & \Gamma, @_a\,Q & \text{if } \Gamma(a) \neq \Gamma(b) \\
\Gamma, @_a\,(c!v | c?p \to P) & \longrightarrow & \Gamma, @_a\,\{v/p\}P & \\
\Gamma, @_a\,(c!v | c?{*}p \to P) & \longrightarrow & \Gamma, @_a\,(\{v/p\}P | c?{*}p \to P) & \\
\Gamma, @_a\,\langle b@? \rangle c!v & \longrightarrow & \Gamma, @_b\,c!v & \\
\Gamma, @_a \textbf{ if } u = v \textbf{ then } P \textbf{ else } Q & \longrightarrow & \Gamma, @_a\,P & \text{if } u = v \\
 & \longrightarrow & \Gamma, @_a\,Q & \text{if } u \neq v \\
 & & & \\
\Gamma, t, @_a \textbf{ wait } c?p \to P,\ n \to Q & \longrightarrow & \Gamma, t+1, @_a \textbf{ wait}_{t+n}\ c?p \to P,\ Q & \\
\Gamma, t, @_a \textbf{ wait}_{t'}\ c?p \to P,\ Q & \longrightarrow & \Gamma, t+1, @_a\,Q & \text{if } t \geq t' \\
\Gamma, t, @_a\,(c!v | \textbf{wait}_{t'}\ c?p \to P,\ Q) & \longrightarrow & \Gamma, t+1, @_a\,\{v/p\}P &
\end{array}
$$

$$
\frac{Q \equiv P \quad \Gamma, t, P \longrightarrow \Gamma', t', P' \quad P' \equiv Q'}{\Gamma, t, Q \longrightarrow \Gamma', t', Q'}
\qquad
\frac{\Gamma, t, P \longrightarrow \Gamma', t', P'}{\Gamma, t, P \,|\, Q \longrightarrow \Gamma', t', P' \,|\, Q}
$$

$$
\frac{(\Gamma, a \mapsto s), t, P \longrightarrow (\Gamma', a \mapsto s'), t', P'}{\Gamma, t, \textbf{new } a@s \textbf{ in } P \longrightarrow \Gamma', t', \textbf{new } a@s' \textbf{ in } P'}
\qquad
\frac{\Gamma, t, P \longrightarrow \Gamma', t', P' \quad c \notin \mathrm{dom}(\Gamma)}{\Gamma, t, \textbf{new } c \textbf{ in } P \longrightarrow \Gamma', t', \textbf{new } c \textbf{ in } P'}
$$

Figure 2.4: Operational Semantics of the Nomadic π-Calculus

Reduction is closed under structural congruence, parallel, **new** c **in** _ and **new** $a@s$ **in** _, as specified by the rules below.

$$\frac{Q \equiv P \quad \Gamma, t, P \longrightarrow \Gamma', t', P' \quad P' \equiv Q'}{\Gamma, t, Q \longrightarrow \Gamma', t', Q'} \qquad \frac{\Gamma, t, P \longrightarrow \Gamma', t', P'}{\Gamma, t, P \mid Q \longrightarrow \Gamma', t', P' \mid Q}$$

$$\frac{(\Gamma, a \mapsto s), t, P \longrightarrow (\Gamma', a \mapsto s'), t', P'}{\Gamma, t, \textbf{new } a@s \textbf{ in } P \longrightarrow \Gamma', t', \textbf{new } a@s' \textbf{ in } P'} \qquad \frac{\Gamma, t, P \longrightarrow \Gamma', t', P' \quad c \notin \mathrm{dom}(\Gamma)}{\Gamma, t, \textbf{new } c \textbf{ in } P \longrightarrow \Gamma', t', \textbf{new } c \textbf{ in } P'}$$

All rules for the Nomadic π-calculus are grouped in Figure 2.4.

2.3 Related Models

Many different models of concurrent computation have been proposed in the literature. They can be roughly classified into two groups: automata models and process calculi. Below we briefly characterise a few process calculi which are related to our work. A more elaborated analysis of two example calculi will appear in the next chapter, where we discuss the design choices in concurrent programming languages which are based on mobile process calculi. In this section, we focus on the automata models and describe two of them: I/O automata, which assume input/output-style of communication between concurrent processes, and Mobile UNITY, which extends the shared-memory model of UNITY with abstractions designed to study loosely-coupled and mobile systems. We also describe two implementations, IOA and LIME, which are formally based on these models.

We also attempt to discuss and compare some of the automata and π-calculi features (the comparison, however, should be seen as a rough approximation only, since the two automata models and process calculi represent different philosophy).

2.3.1 Related Calculi

In recent years a number of process calculi have been introduced in order to study some aspect of distributed and mobile agent computation. They include:

- The π_l calculus of Amadio and Prasad [AP94], for modelling the failure semantics of Facile [TLK96] (see also discussion in §3.3.1).
- The Distributed Join Calculus of Fournet et al. [FGL+96], intended as the basis for a mobile agent language (see also discussion in §3.3.2).

- The language of located processes and the Dπ calculus of Riely and Hennessy, used to study the semantics of failure [RH97, RH98] and typing for control of resource use by mobile agents [HR98b, HR98a].

- The calculus of Sekiguchi and Yonezawa [SY97], used to study various primitives for code and data movement.

- The dpi calculus of Sewell [Sew97a, Sew98], used to study a subtyping system for locality enforcement of capabilities.

- The Ambient calculus of Cardelli and Gordon [CG98], used for modelling security domains.

- The Seal calculus of Vitek and Castagna [VC98, VC99], focusing on protection mechanisms including revocable capabilities.

There is a large design space of such calculi, with very different primitives being appropriate for different purposes, and with many semantic choices. A thorough comparison and discussion of the design space is beyond the scope of this dissertation — a brief discussion can be found in [Sew]; here we highlight only some of the main design choices:

Hierarchy We have adopted a two-level hierarchy, of agents located on sites. One might consider tree-structured mobile agents with migration of subtrees, e.g. as in [FGL$^+$96]. The added expressiveness may be desirable from the programmer's point of view, but it requires somewhat more complex infrastructure algorithms — migrations of an agent can be caused by migrations of their parents — so we neglect it in the first instance.

Unique Naming The calculi of §2.2 ensure that agents have unique names, in contrast, for example, to the Ambients of [CG98]. Inter-agent messages are therefore guaranteed to have a unique destination.

Communication In our preliminary work [SWP98] the inter-agent communication primitives were separated from the channel primitives used for local computation. The inter-agent primitives were

$\langle a@?\rangle!v$	location-independent output of v to agent a
$\langle a@s\rangle!v$	location-dependent output
$?p \to P$	input at the current agent

These give a conceptually simpler model, with messages sent to agents rather than to channels at agents, but to allow infrastructure encodings to be expressed it was necessary to add variants and local channels. This led to a rather large calculus and somewhat awkward encodings.

2.3.2 I/O Automata

The *input/output (I/O) automaton* model [LT87, LT89] is a simple type of state machine, designed to model reactive programs interacting with their environments. Originally developed for specifying and verifying theoretical distributed algorithms, I/O automata have also been applied to practical communication services like TCP, distributed shared memory, and group communication (see [GL98] for many references).

An I/O automaton consists of a set of *actions*, a set of *states* (including a nonempty subset of *start states*), a set of *transitions*, and a set of *tasks*. The actions are classified as either *input*, *output*, or *internal*. The inputs and outputs are used for communication with the automaton's environment, while the internal actions are visible only by the automaton itself. Transitions are triples (s, a, s'), of adjacent states s and s' and action a which caused the change of states. Tasks are sets of non-input actions. A *composition* operation is defined by which I/O automata can be combined to form a larger automaton representing a concurrent system. Concurrent reactive programs are described by I/O automata that compose by synchronising an input with an output action. The *hiding* operator is used to reclassify output actions as internal so they cannot be used in further compositions.

For example, an asynchronous network architecture can be modelled as a composition of processes and communication channels. A process i is modelled as an I/O automaton which has output actions of the form $\texttt{send(m)}_{i,j}$, where j is an outgoing neighbour of i and m is a message, and inputs of the form $\texttt{receive(m)}_{j,i}$, where j is an incoming neighbour of i. The communication channels are also modelled as automata, allowing the specification of various types of channel, e.g. reliable FIFO, reliable reordering channels, channels with failures, etc. The channel automata interact with process automata by $\texttt{send(m)}_{i,j}$ and $\texttt{receive(m)}_{j,i}$ actions. For brevity, we omit here specification of actions for processes and channels.

The operation of an I/O automaton is described by its *executions*, which are alternating sequences of states and actions. As their notion of external behaviour, they use simple linear *traces* of executions, which are sequences of input and output actions. I/O automata admit a notion of *implementation* based on inclusion of sets of traces. Some properties to be proved about I/O automata are formulated as properties of their (fair) traces. Two important

special types of trace property are *safety* and *liveness* properties.

Proofs for I/O automata typically involve *compositional methods* for reasoning about collections of interacting components, *forward and backward simulation relations* for proving that one automaton implements another, and *invariant assertions* for proving that a particular property is true in all reachable states. A *forward simulation* [LV94] from automaton A to automaton B is a relation R between states of A and states of B that satisfies two conditions: (1) each start state of A is R-related to some start state of B, and (2) for each step (s_A, a, s'_A) of A and each state s_B of B such that $(s_A, s_B) \in R$, there exists an execution fragment (i.e., a sequence of steps) of B that "corresponds" to the step in a particular way. Namely, it has the same trace and leads to a state s'_B with $(s'_A, s'_B) \in R$.

IOA Language IOA [GL] is a formal language for defining and stating properties of I/O automata. Together with a design for a coordinated suite of tools, it allows the validation of distributed algorithms expressed as IOA programs, and — in a final stage — enables programs to be translated automatically into the source code of an existing programming language, thereby eliminating the need for a final coding step.

The language evolved from pseudo-languages used for describing distributed algorithms that are based on guarded commands, i.e. named, parameterized transition definitions containing *preconditions* and *effects*. An effect can be described either *operationally* by simple imperative programs, or *assertionally* in a form of a predicate relating pre- and post-states. In moving from pseudocode to a formally defined programming language, the authors of IOA have made several design choices. For example, data types are defined axiomatically, in the style used by LP (Larch Prover) [GG91] and other theorem provers. This provides a sound semantics and facilitates translation into the theorem prover input language. The programmer can also define new types using Larch. The IOA toolset supports a variety of analytic tools which range from light-weight validation, formal proof, to Java code generation for producing distributed implementations of the algorithms. All the tools are based formally on I/O automata.

At the moment, the language is not very expressive. In [GL98], it is postulated that in order to avoid complicating the semantics of IOA, any additional programming features should be made as *syntactic sugar* (i.e. there should be an unambiguous translation of the code with the additions into code without them). In [GL], Garland and Lynch discuss a number of potential extensions to IOA. It is not clear, however, how to formally define all such extensions (e.g. a module system). Currently, the language lacks

local naming conventions. For example, all action names in a composition are global. Also, all of an automaton's state variables are global to all of its transition definitions. The current version of IOA does not have explicit structures for specifying action order — however it is likely that such support will be introduced in future[GL].

2.3.3 Mobile UNITY

A number of other concurrency models have been proposed, which are based on different types of automata than presented above, with a different notion of composition and external behaviour. Here we discuss Chandy and Misra's *UNITY* model [CM88] which is based on automata that combine via shared variables instead of shared actions. Rather than dealing directly with execution sequences, the formal semantics of UNITY are given in terms of program properties that can be proven from the text. *Mobile UNITY* [RMP97] is an extension of the UNITY notation and logic with concepts designed to deal with mobility. The model has been proposed to study problems such as: *decoupling*, *context dependencies*, and *location transparency* in systems with mobility. For example, it has been used in an exercise involving the specification and verification of Mobile IP [MR97], and in modelling various forms of program mobility [PRM97].

A UNITY program consists of three sections, called: `declare`, `initially`, and `assign`. The first two define variable types and the initial program conditions (such as the initial values of data structures), the last section consists of a set of assignment statements. The types include also abstract types, such as sets and queues. The assignment statements execute atomically, and are selected for execution with an interleaving semantics in a weakly fair manner — in an infinite computation each statement is scheduled for execution infinitely often. A larger system can be composed from programs, by using either a simple union operator, or superposition. For example, if we have two programs A and B, we can use the *union* operator [] to construct a new system (denoted as A [] B), which consists of the following: the union of all program variables (in such a way that variables with the same name refer to the same physical memory), the union of all assignment statements (interleaved for execution in a fair manner), and the intersection of initial conditions. Communication between programs A and B will take place via shared variables. Note, however, that reading and writing is not synchronised — it is a fair interleaving execution, but not "in lock step". Thus, some values, e.g. written by A, may not be read by B, or B may read the same value many times. Another way to compose systems is through the use of *superposition*, which combines different components by synchronising

statements rather than sharing variables. For example, we can superimpose new statements and variables of B on an underlying program A, such that the new statements do not assign to any of the original variables of A, and each of the new statements is synchronised with some statement of A. This kind of composition helps in building layered systems, where the underlying layers are not aware of the higher layers.

In Mobile UNITY, a system consists of programs declared as in UNITY, and two additional sections: `components`, where program declarations are instantiated (i.e. free parameters of programs are bound by the instantiation), and `interactions`, which defines transient interactions among program instances. Mobile UNITY captures dynamic reconfiguration and disconnection of system components. Essentially, components are capable of continuing operation whilst disconnected. Components are *located*; a predefined location variable (say λ), initialised in the `components` section, will store the current location of a program and can be freely used in assignment statements. The type of this variable will depend on the application (e.g. location of a mobile computer and location of a software agent are of different type). The model assumes isolation of namespaces of individual components. It assumes that variables associated with distinct components are distinct even if they bear the same name. Therefore, to fully specify a variable in a global scope, its name should be prefixed with the name of the component in which it appears (e.g. $A.x$, or $A.\lambda$).

Interactions among components can be expressed in three possible ways, by using: extra statements, reactions, and inhibitions. The simplest is to use an extra assignment statement, which involves variables of different components and is accompanied by a predicate, following `when`, treated as a guard on the statement. For example, an assignment statement $receiver.x := sender.x$ `when` $sender.\lambda = receiver.\lambda$ would be executed only if components *sender* and *receiver* are co-located (the precise meaning of "at the same location" depends on semantics of the location variable λ). Since the execution of this statement is interleaved with other statements in a fair but arbitrary order, we do not have much control on whether the assignment will succeed or not while the location predicate is true. Therefore, the key new concept is a *reactive statement*, which allows for specification of location- and context-dependent side effects. It can be used, e.g. to detect and propagate changes between components. For example, we can use the following reactive statement $receiver.x := sender.x$ `reacts–to` $sender.\lambda = receiver.\lambda$, to guarantee that each and every value written to $sender.x$ will also appear at $receiver.x$ whenever the predicate following `reacts–to` is true. The set of reactive statements continues to execute until no statement would have an effect if executed.

Other programming constructs include inhibitions and transactions. The *inhibitor* provides a mechanism for constraining the scheduler when execution of some statement would be undesirable in a certain global context. The effect of `inhibit` s `when` p is a strengthening of the guard on statement s by conjoining it with $\neg p$ and thus inhibiting execution of the statement when p is true. Normally, the components in Mobile UNITY execute *asynchronously*. If necessary, one can define a transient synchronisation construct, which is a mechanism for synchronising pairs of statements when their components are co-located. The `inhibit` clauses can further be used to prohibit the statements from executing independently when the components are not co-located. The *transaction* provides a form of sequential execution. A sequence of assignment statements forming a transaction must be scheduled in the specified order with no other nonreactive statements interleaved in between.

The UNITY and Mobile UNITY languages support statement and proof of program properties (such as safety and liveness), using temporal logic reasoning. Proofs of program simple properties involve a universal quantification over the set of assignment statements; proofs of more complicated progress properties are carried out inductively to show that the program moves through a whole sequence of steps in order to achieve some goal.

The low-level primitives of Mobile UNITY presented above have been used to formally express high-level language abstractions for communication between mobile components by transient state sharing [MR98]. Based on a paradigm of shared memory, the transient sharing constructs for read-only, read-write, engagement and disengagement operations provide a mechanism for expressing highly decoupled and context-dependent systems. For example, they consider a queue of documents to be output on a printer. A laptop computer paired with the printer via some wireless communication medium can occasionally disconnect from the network (and so from the printer), and connect again, so it has to maintain a local cache of this queue. Each time the laptop re-connects, the state of the queue must be properly reconciled: updates to the queue are atomically propagated, expressed as a transient sharing of the queue. Operationally, both the laptop and printer maintain two variables: a variable representing a local view of the queue, and a history variable, introduced for detecting changes, so that only new values are propagated. The history variable models the previous state of the queue seen at the counterpart. The propagation of changes is expressed by guarded reactive statements which assign new values to variables and history variables. Additional statements are required to specify reintegration policies, which indicate what values the queue variables should take on disconnection (`disengage`) and when the connectivity is re-established (`engage`).

LIME In [PMR99], Picco et al. describe LIME (Linda in a Mobile Environment), a system designed to assist in the rapid development of dependable mobile applications over wired and ad hoc networks. The authors intend to describe the formal semantics of LIME using Mobile UNITY. In LIME, all location-independent communication between mobile agents (that can reside on mobile hosts) take place via transiently shared tuple spaces distributed across the host machines. At the application level, agents and hosts perceive movements as a sudden change of context. The set of tuples which are accessible by agents residing on a given host is altered transparently in response to the changes in the connectivity among mobile hosts. A prototype of LIME, built upon IBM's TSpaces, is under development [PMR99].

2.3.4 Brief Comparison

Below, we briefly compare I/O automata and process calculi. A similar comparison of Mobile UNITY and process calculi is difficult since the models are based on different philosophy. Instead, we briefly discuss some arbitrarily selected issues (namely modularity and expressing mobility) in the context of different models.

I/O automata and process calculi There arguably exist some similarities between process calculi and automata models. Both models use a notion of composition based on synchronising external actions. In the π-calculus, the parallel operator composes processes which may synchronise (communicate) if they share the same π-channels. A process expression beginning with an input or an output may, in general, be part of several redexes that are ready to be evaluated by a reduction step. According to the process expression chosen, reductions may yield different results. Synchronising external actions of I/O automata may also exhibit non-determinism, yielding many meaningful executions. This models the intrinsic behaviour of real concurrent programs; they may often yield different results depending on the order in which various internal events occur. The basic intuition is that executions (or traces) have a similar meaning to the chains of labelled transitions in process calculi. The I/O automaton may have some input and output actions defined by which it can communicate with an external user; this allows problems to be expressed in terms of traces at the "user interface". In the case of process calculi, we can define observational congruence, understood as some (presumably rather coarse) congruence that is clearly induced by top-level real-world observations.

However, process calculi have a very different style and syntax from I/O automata: they denote concurrent processes by algebraic expressions, and actions by a small set of algebraic operators. The syntax and computable operational semantics are formally defined. This helps in designing programming languages based on the model. The way of describing action code in I/O automata is not part of the model. Commonly natural language is used to describe states and transitions, supported by arbitrary mathematical objects, or, more recently the IOA language. However, one of the most important differences between process calculi and automata models is that in process calculi the state is implicit in the process term that one has reduced to. In I/O automata a set of states is explicit, initialised in action encodings and given in transitions which are triples of state, action, and state.

The authors of I/O automata introduce a notion of *fairness*, which specifies that all the system components (automata) get fair turns to perform steps. It rules out the possibility that some components are permanently denied turns to take steps. There is little work on formalising fairness rules for the π-calculus, but Costa and Stirling's work on fairness for CCS [CS87] seems likely to generalise to the case of π-calculus ([PT97a], section 2.9). The source of problems which, at the moment, make it difficult to define a complete theory of fair π-calculus is that, although it is not hard to define fair reductions, other semantics, e.g. "fair bisimulation", gets quite complicated. In practice, however, one can (with abstract machines, which are equipped with some operational semantics) describe the behaviour of particular schedulers, some of which enforce some fairness (see, e.g. the abstract machine of Pict [Tur96]).

In the π-calculi, proofs typically use bisimulation techniques. We say that two process expressions P and Q are *bisimilar* if every action of one can be matched by a corresponding action of the other to reach a bisimilar state. In practice, so-called *weak bisimulation* is often more useful, which relaxes the demand that the processes simulate each other's behaviour "in lock step" and instead regards arbitrarily many steps of internal communication as equivalent to a single step. It is similar to the forward simulation relation described in the I/O automata section.

Expressing Mobility Mobile UNITY has been proposed to model dynamically reconfiguring distributed systems, and attempts to address design issues raised by mobile computing. I/O automata, because of the essentially static structure of computation which can be expressed, are not very suitable for addressing these problems. An original concern of the π-calculus was to study systems with evolving connectivity structure, by enabling names of

communication channels to be freely communicated along channels. This departs from the static nature of the π-calculus' direct predecessors: CSP [Hoa78a] and CCS [Mil89]. Nevertheless, the calculus does not have the notion of physical mobility. However, many extensions of the π-calculus have been introduced in order to study some aspect of mobile agent computation (e.g. our calculus and the calculi described earlier in this section).

Modularity In the π-calculus, there is no simple way of grouping processes into named components, and so expressing boundaries between components (required, e.g. to model firewalls). Therefore, network architecture, which can easily be expressed in I/O automata by defining two classes of components (see 2.3.2) would have to be modelled by a set of plain π-processes which compose in the standard way using the parallel operator. On the other hand, the Mobile UNITY language allows a system to be declared as a collection of decoupled and mobile components which interact asynchronously. However, there are also calculi which have some forms of modularity, e.g. the Ambient calculus[CG98] used for modelling security domains. An ambient is a named cluster of processes and subambients, which moves as a group. Ambients can model a variety of concepts such as network nodes, packets, channels, and software agents. Similarly, we can group concurrent processes into named agents in the Nomadic π-calculus. Agents could possibly be used for abstracting away reactive components of complex systems. In addition, adopting into the model a tree-like hierarchy of agents would allow a hierarchy of components to be built. Some features, which are only represented in the π-calculi-like models, such as dynamic name and process generation and local name extrusion, can facilitate non-trivial reasoning about system reconfiguration and security.

Which model? A common dilemma in selecting a model is how abstract we want the model to be. A model which is too abstract is not credible, as we can lose many important details. On the other hand, if too many details are exposed, reasoning in such a model becomes tedious. Concurrent programs in Mobile UNITY are sets of assignment statements (often guarded by logical predicates). Conversely, an action style of specifying the programs seems to be closer to the intuitive informal description of how they actually execute. It seems reasonable that it should be easier to produce executable code from specifications expressed in a language which enjoys executable operational semantics, e.g. based on actions. However, specifications expressed in the Mobile UNITY logic are likely to be more concise than in the process calculi (and so easier to grasp and proofs arguably easier to carry out), but they

further abstract from details of the systems modeled. Thus, there is no definite answer to the initial question — it all depends on the context in which a model is used and some preferences, e.g. about proof techniques.

There are some tools designed for all the models described above. A variety of UNITY validation tools have been developed, e.g.: HOL-UNITY theorem prover [And92] (implemented as a library of HOL), and the UNITY Verifier [Kal94] — a symbolic model checker for finite state UNITY programs. Some tools have also been designed for the π-calculus, e.g. the Mobility Workbench [Vic94]; the workbench can efficiently check open bisimulation equivalences. Some proofs of formal specifications in I/O automata have been carried out using interactive theorem provers, such as the Larch Prover [GG91, Che98] and Isabelle [Pau94].

Chapter 3

Programming Language

In this chapter we describe the Nomadic Pict language. It is designed to allow infrastructure algorithms to be expressed as clearly as possible, as translations from a high-level language to a low level. In §3.2, we describe the principles and syntax of our language and some programming idioms which are used in the infrastructure translations. We conclude the chapter by describing Facile and the Join Language — two general-programming languages, which are based on mobile process calculi. We compare the design choices, and argue that Nomadic Pict is more suitable for the specification and experimenting with the infrastructure algorithms for mobile agent systems. We begin the chapter by presenting our motivations for building a new programming language for mobile computation.

3.1 Motivations

With the emergence of the World-Wide Web and ubiquitous computing, a number of new programming languages have been developed, such as Java and scripting languages (e.g. Tcl, Python, and Perl). These languages are usually interpreted or compiled to some architecture-independent bytecode, which can be highly portable on current and future platforms. Java, the most popular of these languages, has been used to build many mobile agent systems. It is relatively easy to use Java, due to support offered by the language for code mobility, architecture-independence, and secure program execution. Aglets [LOKK97], Voyager [Gla98], Concordia [WPW98], and Mole [SBH96] are examples of Java-based mobile agent systems. However, the metaphor "mobile agent" was first introduced in Telescript [Whi96], an object-oriented language designed specifically for mobile agent programming. A few programming languages which support mobile computation appeared as extensions of functional languages (such as Facile and JoCaml [CF99])

3.1.1 Mobility in a Wide-Area Network

The main concern in the mobile agent community has been demonstrated so far around problems of safety, security, scalability, fault-tolerance, and efficiency of code manipulation and execution. The issues of a programming language design for mobile agents appear seldom, and usually in the context of high-level communication between intelligent agents. It is true, that many of the technical problems mentioned above are still not well understood and solved, however the main problem seems to be not only technological. In spite of the potential benefits of using mobile agents in certain contexts, there are very few real agent applications in use. Mobile agent technology itself simply does not offer anything which could not be implemented in the traditional client-server model. It may however, as a set of positive factors, enhance some applications [CHK97]. Also, the importance of software mobility will grow naturally as a consequence of pervasive mobile computing (enabled by mobile computers, PDAs, etc.) However, deployment of mobile computation, especially in wide-area networks, may appear to be difficult. Mobility complicates system management, because programs are no longer bound to static locations and administrative domains. However, some authors argue that mobile agents have the potential to provide a convenient, efficient and robust programming paradigm for Internet applications, particularly when computers have only intermittent access to the network [GKN+97]. They predict that agent technology will be a critical near-term part of the Internet [KG99]. It appears to us, however, that in order to exploit this potential fully, there must be better programming language support than is offered at present.

Knabe [Kna95] pointed out desirable properties of mobile agent languages: strong typing, remote resource access, automatic memory management, security, authentication, support for manipulation, transmission and execution of code-containing objects. Java and other languages used in current mobile agent systems successfully meet some or all of these properties. However, they do not necessary offer convenient programming metaphors for building applications where agents must communicate and collaborate while migrating. The construction of such applications would be made easier by isolating the agent communication and collaboration from: low-level communication, authentication protocols, tracking the agent whereabouts, and lock-based concurrency control (if supported). One of the aims is to enable developers, who are not necessarily experts in distributed systems and mobile agents, to construct applications, but in such a way that they can still be aware of the distribution concerns. This awareness seems essential in the context of mobile agents; it facilitates the construction of applications which are efficient, scalable, and secure.

The programming languages used in current mobile agent systems are rather traditional — they are based either on the object-oriented style, or the imperative techniques of scripting languages. On this ground, system developers added new primitives to support mobility of code. However, they usually do not provide the suitable level of abstraction needed for expressing communication between mobile agents on the Internet. Communication is usually made possible through current middleware systems or RPC-like techniques. RPC is too low-level an abstraction — it does not support location independence. Middleware, such as CORBA-compliant Object Request Brokers, provide support for language-independent and location-transparent method invocation. The main problem is, however, that they cannot adequately provide scalability and flexibility, important issues in the case of mobile agent systems. The proposition of Globe [vSHBT98], a worldwide location service for distributed objects, offers better infrastructure for a wide-area network, which can better cope with (occasional) mobility of objects.

We argue, however, that the transparent-object model simply does not provide a suitable level of abstraction for expressing the (low-level) communication between mobile agents, which should not be made transparent to the application intended for a wide-area network. The reason is simply because a wide-area network is not like a local-area network: communication is very asynchronous, delays and bandwidth fluctuations unpredictable, communication may have to cross administrative boundaries and firewalls, remote sites are invisible, failures are indistinguishable from deliberate disconnected operation of some machines, mobile computers can be moved between network domains and have only intermittent access to the network, and so on. Any global distributed infrastructure which would try to hide these issues to some extent (if possible) by replication and fault-tolerant protocols, and provide a uniform transparent-object model to the programmer would dramatically slow down the whole Internet (besides it would be undesirable due to security reasons). On the other hand, communication between mobile agents is sufficiently above network protocols to make searching for new programming abstractions worthwhile.

3.1.2 Verification of Mobile Computation

There is a natural demand for the most critical parts of systems to be analysed, verified, and proven correct (*correctness* informally means compliance of program execution with our assumptions). Network protocols and distributed infrastructures are highly concurrent; testing can often be useful to find common errors, but is not capable of guaranteeing correctness. The activity of proving formally that a system is correct is called *verification*. The verification of formally described protocols can, to some extent, be au-

tomated using interactive tools, such as theorem provers. Nevertheless, the process is costly, since one has to write an input for a standard theorem prover from a program specification, and after proving it correct, must recode the program using a traditional programming language. This process is difficult, error-prone, and the verification information is not readily reusable as the system grows. The growing complexity of mobile protocols (such as Mobile IP) and prevalent programming practice, will make such an approach difficult and questionable. New methods are required, which would be easier to use and less costly.

The solution would be to verify the actual code expressed in a programming language rather than some abstraction of it, as this gives us more accurate and reliable information about the way the system is going to behave (it would also facilitate updating proofs after code revisions). The advantage is that executable code could be generated from verified specifications which are proven correct. This would save duplication of effort in all these cases where formal verification is essential, and rule out the possibility of introducing errors while coding. Unfortunately, the traditional programming languages are not very suitable for formal verification. The solution would be to use a programming language whose syntax is already quite abstract, providing a concise set of semantically-clean primitives, which are however efficiently implementable.

Although this is an ultimate goal, there are already examples of rigorous verifications on the level of actual running code, without resorting to approximate techniques. For example, Arts and Dam [AD99] demonstrate a pragmatic approach to formalisation and verification based on an example from industry, and discuss their experience with using a verification tool for Erlang programs. Erlang [AWWV96] is a functional programming language developed at Ericsson, which has been used for writing robust distributed telecommunication applications. The "core" features of Erlang include: list and number processing, dynamic process creation (also spawning processes on remote hosts), and communication. They extract, from the *real* implementation in Erlang, a fragment implementing the protocol to verify, add some additional code to provide a very simple simulated interface to parts of the system that are irrelevant for the problem at hand, and verify the program using the (interactive) verification tool. Erlang programs can be seen as a very precise, and in some sense formal, description of the algorithm (although the language semantics has not been formally defined).

In [GL98], it has been argued that the features which make a language suitable for verification and proofs (e.g. nondeterminism, simplicity, declarative style) are different from those that make it suitable for code generation (e.g. determinism, expressive power, imperative style). Nondeterminism helps to validate designs in a general form. Simplicity gives hope for simple

semantics and simple proof rules, making designs described in such a language easier to understand and verify. A declarative style would be easier to translate into the input languages of standard theorem provers. On the other hand, a deterministic language with an imperative style is easier to translate to efficient executable code. The expressiveness of language primitives makes programming easier. We believe that Nomadic Pict offers a good balance of the trade-offs mentioned above. Although, Nomadic Pict is a general programming language, it can also be used for informal but rigorous proofs, as demonstrated in [Uny].

The language offers primitives that can be efficiently implemented yet not at all far from a process calculi-like level of abstraction. The object and functional programming concepts which are useful for general programming have complex semantics, and therefore applying them to reasoning about mobile computation (where partial failures may happen) must be done with care. Our approach is as follows. We are first looking for convenient metaphors (or concepts) which are *basic* in the general model of *concurrent computation*, where each program is a set of agents or activities which *co-exist* and *interact* with each other. Then, we add to this support for mobility and physical distribution of agents. Distributed objects and functions, being the most complex metaphors, may eventually be built on top, if required. Objects and functions known traditionally from sequential programming can be seen as deterministic subsets of that more general model.

In our work we build on process calculi. Process calculi offer a small set of operators accompanied by computable operational semantics, which altogether form a clean design path for the construction of programming languages. The expressive power of a language is achieved by additional programming idioms, such as data structures, higher-order programming, and concurrent objects. They can be formed by adding a layer of convenient syntactic sugar and a static type system to a tiny core. For example, translations of high-level idioms, like functions and objects, into π-calculus processes have been given a rigorous theoretical treatment (e.g. [San99, San98], see also objects in Pict [PT95, LSN96]). The Nomadic π-calculus, an extension of the π-calculus, formed a foundation for the Nomadic Pict language design. The advantage of our calculus is that, although the model is nicely tractable theoretically, there is only a small gap between the model abstractions and the real system implementation. Therefore, we hope that any results of reasoning formally in the (quite intricate) mathematical model of mobile communicating agents should be easily transferable into the real system. This should guarantee a good level of confidence that the system and applications developed in it will meet the specification, and help in understanding what the system actually does. It may also facilitate construction of tools such as debuggers, monitors, and optimised compilers for programming with mobility.

3.1.3 Infrastructure Design and Specification

In distributed systems with mobile hosts and code mobility, chunks of distributed computation may have only intermittent access to the network; they are no longer static but can freely migrate to other physical locations while exchanging messages. Designers of mobile agent systems usually assume that mobile agents are autonomous, solitary programs which can migrate from site to site and perform tasks on behalf of their users. The agents communicate at visiting sites ("meeting places") with other resources or agents. They can also open RPC connections to services which are on remote sites. On top of that, there are mechanisms provided for *tracking* locations of mobile agents and message delivery. In our language, the act of *communication between agents* rose to the rank of a single language primitive, which may greatly simplify building applications. We have defined semantics of this primitive precisely within our calculi-based model and provided a number of implementations. We wanted to be able to express the infrastructure algorithms in a form that is *clean* and *easy to understand*. Also, we wanted to *prototype* the algorithms and simple examples of agent applications in a distributed environment. At the time of starting the project, there were few languages which supported location-independent communication as a primitive, with a formally specified semantics (e.g. Facile and the Join Language described in §3.3). These systems, however, are closed. They do not offer convenient language metaphors for location-dependent communication, and do not provide a means for expressing and supplying the infrastructure algorithms for location-independent language primitives. Therefore, we decided to implement our own light-weight runtime system and language, where this would all be possible. In our system, the agent programming abstractions can be factored into hierarchical translations of the higher into the lower level language, thus customising the agent system. An arbitrary infrastructure can be deployed dynamically, on application start-up, using agent migration — this makes it straightforward to experiment with a wide range of infrastructure algorithms for applications with different migration and communication patterns. Also, a simple message passing discipline of the low-level language, freed from the burden of marshalling and unmarshalling parameters of network and middleware protocols, makes it easy and straightforward to verify results of reasoning formally about message-passing algorithms in the presence of mobility.

Nomadic Pict has been designed to validate our model of mobile computation with highly-concurrent agents. The system is currently less suitable for building serious applications, due to the lack of sufficient support for application interoperability; we plan to extend the language to meet some of

these requirements in future. Services, traditionally provided by operating systems and some "middleware" (such as object brokers and lookup service), are assumed to be delivered as translations in Nomadic Pict. This allows complex distributed systems to be built while at the same time remaining inside a coherent, integrated framework. Therefore one can easily prototype new algorithms, because the system services can be customized and their source code is highly readable. The semantics of all external and internal services can therefore be understood within a single coherent model.

3.2 Nomadic Pict

In the following sections, we describe the Nomadic Pict language and programming idioms. We conclude the chapter presenting two general-purpose programming languages supporting code mobility, Facile and Join-calculus, which also grew up from process calculi.

3.2.1 Language Principles

We have designed and implemented Nomadic Pict as a vehicle for exploring distributed infrastructure. It builds on the Pict language of Pierce and Turner [PT97a, PT97b, Tur96], a concurrent (but not distributed) language based on the asynchronous π-calculus [MPW92, HT91, Bou92]. Pict supports fine-grain concurrency and the communication of asynchronous messages. To these Low-Level Nomadic Pict adds primitives for agent creation, the migration of agents between sites, and the communication of location-dependent asynchronous messages between agents. The high-level language adds location-independent communication; an arbitrary infrastructure can be expressed as a user-defined translation into the low-level language. The combination of low-level language and facilities for defining a translation thus embody the design principle:

> A wide-area programming language should provide a level of abstraction that makes distribution and network communication clear; higher levels should be provided and implemented using the modularisation facilities of the language. It should be possible to deploy such infrastructure dynamically.

Such a language can have a standardised low-level runtime system that is common to many machines, with divergent high-level facilities chosen and installed on demand. The levels of abstraction can be made precise by giving process calculi equipped with rigorous operational semantics. Preliminary

definitions of the (low and high-level) Nomadic π-calculi were given in chapter 2. They have since been extended to large fragments of the language, for use in correctness proofs, but are not described here (see [Uny]).

We have focussed on the simplest language that allows us to study the core problem of location-independent communication introduced in chapter 1, rather than attempting to produce an industrial-strength language. In particular, we study a single representative location-independent primitive, that of delivering a message to an agent on an arbitrary site. We believe that analogous work could be carried out for other high-level primitives, e.g. multicasts, and for many other concurrent languages.

A further simplification is the adoption of a fixed two-level architecture, rather than a general purpose module system. The utility of a rich module system for structuring communication protocols, in the absence of mobility, has been demonstrated in the FOX project [HLP98]; see also Ensemble [Hay98]. In future work we intend to integrate an ML-style module system with a Nomadic Pict language.

In this section we introduce enough of the language for the example application and infrastructures given in the following chapters. The language extends primitives of the Nomadic π-calculus with some convenient syntactic sugar and a type system. The operational semantics of the primitives have been described in §2.2.

3.2.2 Low-Level Language

Below is a program in the low-level language showing how mobile agents can be expressed; it extends the applet server in §2.2.1 with programming constructs such as types and functions. Inside the agent `a` (which is assumed to be on site `s'`) we define a function `spawn`; the function accepts two formal parameters `s` and `prompt`, and creates a new fresh agent named `b`

```
new answer : ^String
    def spawn [s:Site prompt:String] =
      (agent b =
         (migrate to s
          <a@s'>answer!(sys.read prompt))
       in
         ())
    ( spawn ! [s1 "How are you? -" ]
    | spawn ! [s2 "When does the meeting start? -" ]
    | answer ?* s = print!s
    ...
```

In the body of the agent `a`, we have two parallel invocations of the function `spawn`, creating two agents `b` (with different names) which immediately migrate to remote sites `s1` and `s2`, print a prompt on the current console, and read from a standard input. A message containing the string read from the console is sent to the spawning agent `a`. The agent `a` can receive the messages on the channel named `answer` (carrying values of type `String`) and print the reply on the screen. *Functions* (process abstractions) are syntactic sugar; they can be replaced by channel communication internal to the agent (we will explain it in §3.2.4). The sites, agents, and channels are typed; the language *types* are described below. The language is built above asynchronous messaging, both within and between sites; in the current implementation inter-site messages are sent on TCP connections, created on demand, but our algorithms do not depend on the message ordering that could be provided by TCP.

Types The language inherits a rich type system from Pict, including simple record types `[label1=T1...labeln=Tn]`, higher-order polymorphism, simple recursive types `(rec X =..X..)` and subtyping. It has a partial type inference algorithm. Pict's four basic channel types are classified as follows: `^T` (the type of input/output channels carrying values of `T`) is a subtype of both `!T` (output channels accepting `T`) and `?T` (input channels yielding `T`). That is, a channel that can be used for both input and output may be used in a context where just one capability is needed. Also `/T` (*responsive* output channels carrying `T`) is a subtype of `!T`.

Nomadic Pict adds new base types `Site` and `Agent` of site and agent names, a *variant* type for expressing variants, and a type `Dyn` of *dynamic* values (to date only partially implemented) for implementing traders. The variant type {`label1>T1...labeln>Tn`} denotes all values {`label>v:T`} such as $(label, T) \in \{(label_1, T_1), ..., (label_n, T_n)\}$.

In this thesis we make most use of `Site`, `Agent`, the base type `String` of strings, `Int` of integers, `Bool` of Booleans, the type `^T` of channel names that can carry values of type `T`, tuples `[T1...Tn]`, variants, and existential polymorphic types such as `[#X T1...Tn]` in which the type variable `X` may occur in the field types `T1...Tn`. We also use a type operator `Map` from the libraries, taking two types and giving the type of maps, or lookup tables, from one to the other (we will explain maps in §3.2.4).

Values Channels allow the communication of first-order values: names `a`,`b`,..., strings, tuples `[v1...vn]` of the n values `v1...vn`, packages of existential types `[#T v1...vn]`, and elements of variant types `{label>v}`. The language does not support passing of processes (except for the migration

of whole agents) or of higher-order functions. **Patterns** `p` are of the same shapes as values, with the addition of a wildcard.

Core Language Syntax The main syntactic category is that of *processes*.

`c!v`	output `v` on channel `c` in the current agent
`c?p = P`	input from channel `c`
`c?*p = P`	replicated input from channel `c`
`( P | Q )`	parallel composition
`( Dec P )`	local declaration
`()`	null process
`if` `v` **`then`** `P` **`else`** `Q`	conditional
`iflocal` `<a>c!v` **`then`** `P` **`else`** `Q`	test-and-send to agent `a` on this site
`wait` `c?p=P` **`timeout`** `n -> Q`	input with timeout
`<a>c!v`	send to agent `a` on this site
`<a@s>c!v`	send to agent `a` on site `s`

Declarations `Dec` are used to introduce new channels, agents, and express migration; they form a separate syntactic category.

`new` `c:T P`	new channel name creation
`agent` `a=P` **`in`** `Q`	agent creation
`migrate to` `s P`	agent migration

All bound variables (and wildcards) are explicitly typed. In practice, however, many of these type annotations can be inferred automatically by the compiler. Therefore we did not include them in the syntax above. Types are required in definitions, e.g. execution of **`new`** `c:^T` creates a new unique channel name for carrying values of type `T`.

In the language implementation, we use environments to store bindings of names to values instead of an explicit substitution. For example, an output `c!v` and an input `c?p=P` in the same agent may synchronise, resulting in `P` with the appropriate parts of the value `v` bound to the formal parameters in the pattern `p`.

3.2.3 High-Level Language

The high-level language is obtained by extending the low-level with a single location-independent communication primitive:

`c @ a ! v`	location-independent output to channel `c` at agent `a`

The low-level communication primitives are also available, for interacting with application agents whose locations are predictable. Other low- and high-level communication primitives may be added in future, e.g. in order to support stream communication.

Summarising, agents located on a particular site can freely change location by migrating to a new site. Channels created inside an agent are local within the agent but can also be used for communication between agents, providing that the sender and receiver use the same channel name (which has been defined in common lexical scope or dynamically extruded). The location-independent output will require some distributed infrastructure to deliver messages reliably. In distributed operating and "middleware" systems, this kind of infrastructure is a hard-wired part of the system. In Nomadic Pict, the distributed infrastructure for location-independent communication, and addressing schemes can be anything whatever, since all have to be explicitly encoded in the low-level language.

Expressing Encodings The language for expressing encodings of high-level language primitives allows the translation of each interesting phrase (all those involving agents or communication) to be specified and type checked; the translation of a whole program (including the translation of types) can be expressed using this compositional translation. A rudimentary module system allows encodings of any new phrases of the high-level language to be expressed as macro definitions. We can use the macros in programs writing **do** "*macroname*" *parameter* **in** P. Here, the type of *parameter* is not known until the macro definition is applied and the type information can be inferred. We omit in this section the concrete syntax of the language for expressing encodings; the example infrastructures in chapters 4 and 6 should give the idea (see also Appendix).

3.2.4 Examples and Idioms

We give some syntactic sugar and programming idioms that will be used in the translations. Most are standard π-calculus idioms; some involve distributed communication. The syntax for process abstractions, value declarations, and applications has been introduced in Pict and described informally below (see [PT97b] for a formal description of the source-to-source translations and more syntactic sugar).

Syntactic sugar (a) In Pict, we can define *process abstractions* via the declaration keyword **def**, as in

```
def f [x:T1 y:T2] = (x!y | x!y)
```

and instances are created using the same syntax as output expressions, as in `f![a b]` (`f` *is* a channel — a process abstraction is translated into a channel declaration **new** `f` and a replicated receiver on `f`). Recursive and mutually recursive definitions

```
def f [..] =  ...  g![..]  ...
and g [..] =  ...  f![..]  ...
```

are also allowed.

(b) The syntactic category of values is extended with declaration values of the form `(Dec v)`, as in

```
c!(new d:T d)
```

The complex value is always evaluated to yield a simple value, which is substituted for the complex expression; the process above creates a fresh channel `d` and sends it off along `c`, as in `(new d:T c!d)`.

(c) A declaration

```
val p=v
```

evaluates a complex value `v` and names its result. Formally, a **val** declaration `(val p=v e)` is translated using the continuation-passing translation, so that the body `e` appears inside an input prefix on the continuation channel which is used to communicate a simple value evaluated from the complex value `v`. This means that **val** declarations are *blocking*: the body `e` cannot proceed until the bindings introduced by the **val** have actually been established.

(d) In value expressions, we allow the *application* syntax `(v v1 ... v2)`. For example, we can define a process abstraction

```
def double [i:Int r:/Int] = +![i i r]
```

and then, in the scope of the declaration, write `(double i)` as a value, dropping the explicit result channel `r`, e.g. `printi!(double 2)` would compute `4` and print it out on the console, using the library channel `printi`.

(e) A functional style is supported in Pict by a small extension to the syntactic class of abstractions. For example, we can replace a process abstraction **def** `f [a1:T1 a2:T2 r:/T] = r!v`, where `v` is some complex value, by a 'function definition'

```
def f (a1:T1 a2:T2) : T = v
```

and avoid explicitly giving a name to the result channel `r`.

(f) The idiom "invoke an operation, wait for a signal (i.e. a null value `[]`) as a result, and continue" appears frequently, so it is convenient to introduce `;` (semi-colon), as in

```
(v1 ...); (v2 ...)
```

for the sequence of operations `v1` and `v2`.

(g) In Nomadic Pict we introduced *variants* and a new type `Dyn`. In the infrastructure translations, we use a variant type {`label1> T1 ... labeln> Tn`} so often, that it is convenient to introduce a new process form **`switch`**, as follows

```
c?v= switch v of
       (
         {label1> p1} -> P1
          ...
         {labeln> pn} -> Pn
       )
```

that matches a variant type value `v` with all the variants, chooses the one which has the same label as `v`, and proceeds with a process `P` of the matched variant.

(h) We can compare *dynamic values* at runtime via the process keyword **`typecase`**, as in

```
c?v= typecase v of
       s:String -> print!s
       [s:String d:^String] -> d!s
     else print!"Type not recognised."
```

where c has type `^Dyn`. Instances of dynamic values are created using `(dynamic v)`. For example, `c!(dynamic ["ala" x])` in parallel with the process term above may synchronise, resulting in "ala" being sent along the channel `x`, `c!(dynamic "ala")` would print "ala", but any other (dynamic) value sent on `c` would print an error message. The translations of **`switch`** and **`typecase`** use the value equality testing primitive.

Process Abstractions in Agent Programming A name `f` of process abstraction **`def`** `f x:T=P` in agent `a` has type `/T`. In the current implementation, the name `f` can be extruded outside `a` and used for remote invocations, as in `f@a!...`. However, we usually use standard channels for inter-agent communication (e.g. in 'procedures' described below). The type `/T` of `f` guarantees that there is exactly one receiver on `f`. Therefore it is not sensible to use `f` outside agent `a` for a *local* output because this would never synchronise. In the future, the type system may enforce types `/T` to be used within a single agent only (this would require the introduction of new types and sub-typing rules for channels which are intended to be used for inter-agent communication). We neglect it for a while so as not to complicate the type system.

However, functions from Nomadic Pict libraries are treated in a special way; they can be effectively used in all agents defined in a user program (so formally it looks as though each agent has a private copy of each library function it might ever use). Declarations of library modules precede lexically the program declarations, therefore the library function names are visible inside any agent in a normal way, just as any other names defined in the lexical scope.

A plausible extension of the Nomadic π-calculus would be a higher-order nomadic calculus, in which agents could communicate higher-order values, such as process abstractions, not just names and simple values. If efficiency is not critical, it would be very easy to extend the current implementation of Nomadic Pict to support higher-order values, by simply extending a type of standard Nomadic Pict messages to allow any higher-order value to be transmitted, and sending a local environment with messages. On the receiver side, the higher-order value would be interpreted as any other agent process.

Procedures Within a single agent one can express 'procedures' in Nomadic Pict as simple replicated inputs. Replicated inputs are useful to express server agents. Below is a first attempt at a pair-server, that receives values `x` on channel `pair` and returns two copies of `x` on channel `result`, together with a single invocation of the server.

```
new pair   : ^T
new result : ^[T T]
( pair?*x = result![x x]
| pair!v
| result?z = ... z ... )
```

This pair-server can only be invoked sequentially—there is no association between multiple requests and their corresponding results. A better idiom is below, in which new result channels are used for each invocation. The pair-server has a polymorphic type (`X` is a type variable), instantiated to `Int` by a client process.

```
type (Res X) = ^[X X]
new pair : ^[#X X (Res X)]
( pair?*[#X x r] = r![x x]
| (new result:(Res Int)  (pair![1 result] | result?z =... z ...))
| (new result:(Res Int)  (pair![2 result] | result?z =... z ...)))
```

The example can easily be lifted to remote procedure calls between agents. We show two versions, firstly for location-dependent RPC between static agents and secondly for location-independent RPC between agents that may be migrating. In the first, the server becomes

```
new pair : ^[#X X (Res X) Agent Site]
    pair?*[#X x r b s] = <b @ s>r![x x]
```

which returns the result using location-dependent communication to the agent `b` on site `s` received in the request. If the server is part of agent `a1` on site `s1` it would be invoked from agent `a2` on site `s2` by

```
new result : (Res Int)
  ( <a1 @ s1>pair![7 result a2 s2]
  | result?z = ...z... )
```

If agents `a1` or `a2` can migrate this can fail. A more robust idiom is easily expressible in the high-level calculus—the server becomes

```
new pair : ^[#X X (Res X) Agent]
    pair?*[#X x r b] = r@b![x x]
```

which returns the result using location-independent communication to the agent `b`. If the server is part of agent `a1` it would be invoked from agent `a2` by

```
new result : (Res Int)
  ( pair@a1![3 result a2]
  | result?z= ...z... )
```

Locks, methods and objects The language inherits a common idiom for expressing concurrent objects from Pict [PT95]. The process

```
new lock:^StateType
  ( lock!initialState
  | method1?*arg = (lock?state = ...  lock!state'  ...)
  ...
  | methodn?*arg = (lock?state = ...  lock!state'' ...))
```

is analogous to an object with methods `method1`...`methodn` and a state of type `StateType`. Mutual exclusion between the bodies of the methods is enforced by keeping the state as an output on a lock channel; the lock is free if there is an output and taken otherwise. For more detailed discussion of object representations in process calculi, the reader is referred to [PT95].

Finite maps The algorithms given in the following chapters involve finite maps from Nomadic Pict standard library — in the first, there is a daemon maintaining a map from agent names to site names; in the second, there are daemons maintaining maps from agent names to lock channels. The translations make use of the following constructs:

```
c!(map.make eq)
```

outputs the empty map on channel `c` (where `eq` is a comparing function over the keys),

```
(map.add m a v)
```

returns a map containing the same binding as `m`, plus a binding of `a` to `v`; if `a` was already bound in `m`, its previous binding disappears,

```
switch (map.lookup m a) of
(
  { Found> p } -> P
  { NotFound> _ } -> Q
)
```

looks up `a` in map `m`. Our calculi are sufficiently expressive to allow these to be expressed directly, in a standard π-calculus style — formally we can regard the constructs as syntactic sugar for π-calculus process terms, as in [SWP99]; in Nomadic Pict, maps are defined using the built-in library `List`.

The `Map` library contains four additional functions, for removal, testing, and iterations: (4) `(remove m k)` returns a map containing the same binding as `m`, except for `k` which is unbound in the returned map, (5) `(present m k)` returns **`true`** if there is binding of `k` in `m`, or **`false`** otherwise, (6) `(iter m f)` applies `f` to all bindings in map `m`, discarding the results; `f` receives the key as first argument and the associated values as second argument; the order in which the bindings are passed to `f` is unspecified, (7) `(fold m f a)` computes `(f` k_n v_n `...` `(f` k_1 v_1 `a)` `...` `)`, where k_1 `...` k_n are the keys of all bindings in `m`, and v_1 `...` v_n are the associated data; the order in which the bindings are presented to `f` is not specified.

3.3 Related Languages

A large number of other programming languages which are based on some formal model have appeared over the years. They include purely functional languages based on lambda-calculus (e.g. Haskell), "impure" functional languages (e.g. Scheme and ML [Mil84, MTHM97]), and languages which combined the concurrency primitives of process calculi with some more traditional features for sequential programming (e.g. Amber [Car86], Concurrent ML [Rep93], and open source industry languages based on object or functional model, e.g. Obliq [Car95] and Erlang [Lab98]). A main motivation for using such languages (apart from making it easier to write distributed applications) is that they integrate different computational paradigms in a clean

and well understood programming model, which has the potential to automate some formal reasoning about the behaviour and properties of programs expressed in these languages (e.g. in proofs of program correctness).

Below, we describe Facile and the Join Language , two general-purpose programming languages which support code mobility. In both cases, process calculi have been successfully used in the specification of language semantics and on different stages of the language design and implementation. Since all the system and middleware services for code mobility and communication are tightly integrated within one single framework, Facile and the Join Language can also facilitate formal reasoning about programs.

3.3.1 Facile

Facile [GMP89, PGM90, TLG93] is a higher-level, strongly-typed, modular, distributed programming language. The industry-strength implementation of Facile at ECRC [TLP+93] provided concurrency and synchronous channel communication extensions to Standard ML of New Jersey [AM91]. As an extension of ML, Facile brings all the concepts and techniques from advanced language research to the fore. It offers a strong but flexible type system (with polymorphism and type inference), convenient primitives for manipulation of complex data structures, lexical scoping and higher-order programming (e.g. functions can be defined in any scope and passed to other functions or returned as results). In Facile, all these features are seamlessly integrated with a simple interface to distributed programming.

The computational model of Facile consists of a collection of *nodes*, possibly located on different host machines, each node running zero or more Facile *processes*. A node can be thought of as a virtual processor with a shared address space[1]. Processes execute by evaluating expressions (in a functional style), and can communicate values between each other by synchronising over typed abstract *channels*. New nodes, channels, and processes can be created dynamically, processes can be spawned on a given node and execute a given script. Communication on channels is reliable and synchronous (both the server and receiver processes are blocked until communication can be completed). Any data, including process scripts, channel names, and node identifiers, which are first-class values, can be sent over channels. Importantly, they can be sent to a remote site, while preserving type safety. A non-deterministic selection of ready communications over a dynamically specified list of channels can be made by using guarded *choice*. A timeout

[1]Facile processes running on the same node can be thought of as light-weight threads. They share an address space and execute concurrently under control of a preemptive scheduler of the Facile run-time system.

mechanism and exception handling support writing fault-tolerant applications. The *module system* of Standard ML has been extended with *supplying* and *demanding* constructs for dynamic connection of applications that were started independently. Applications may store modules by supplying them to servers on the network, and other applications may retrieve these modules by demanding them based on an interface specification (a signature). Description of the language constructs can be found in a Facile tutorial [TLK96].

A formal model of Facile is an extension of λ-calculus with primitives for concurrency, channel communication, and distribution. The concurrency model of Facile is based on CCS and its higher order and mobile extensions (CHOCS [Tho93] and the π-calculus). Furthermore, constructs for distributed computing are based on results from timed process algebra and true concurrency theory. The semantics of Facile has been studied extensively, either focusing on defining the (abstract) execution of programs in terms of transition systems, reduction systems or abstract machines, or being concerned with the development of program equivalences. Concurrent functional programming has been studied in a framework of a simply typed $\lambda_{||}$ calculus (Amadio, Leth, and Thomsen [ALT95], see also [Ama94]), which comprises three basic ingredients of Facile: (1) call-by-value λ-calculus extended with the parallel evaluation of expressions, (2) a notion of channels and primitives for synchronous input/output (communications are performed as side effects of expression evaluation), (3) the possibility of dynamic generation of new channels. The calculus includes CHOCS and a simply sorted part of the (synchronous) π-calculus as its sub-calculi. In addition, Amadio and Prasad [AP94] study issues related to physical distribution, namely locations and failures. In their π_l-calculus, basic resources are nodes (locations), and channels and processes *located* at these nodes. A node can *fail* (in a *fail-stop* fashion); the consequence of node failure is that all processes located at that node abruptly terminate and the communication channels allocated there become unavailable to other processes. At the semantic level, this obviously means that processes that could normally be considered equivalent in a calculus without locations, may turn out to exhibit different behaviours when complex allocations and failures are allowed. The work on modelling mobility, initiated in Facile, has been continued by Amadio [Ama97, Ama].

In [ALT95, AP94], the authors gave simple translations of both calculi into the π-calculus, thus proving (at least in theory) that reasoning about Facile distributed applications can be carried out in the familiar interleaving semantics of the π-calculus (e.g. by using a barbed bisimulation). We sketch the translations in turn. The basic idea of translating the $\lambda_{||}$ calculus into the π-calculus is to represent a function by means of a replicated input on a channel and to transmit channel names (pointers) instead of functions (a

similar translation has been used to desugar functions in concurrent but non-distributed Pict). However, in the real distributed implementation, process scripts (possibly containing functions) are sent between nodes together with their local execution environments. In case of the π_l-calculus translation, each node is represented by a special process, which interacts by means of a simple protocol with any process of the original program wanting to access resources depending on that location.

Discussion The Facile project has proven that formal reasoning can be successfully employed in designing a complex system. Also, people working on theory can benefit from a closer insight into practical problems encountered in system development. In Facile, the assumptions about system functionality have been exemplified by defining a formal model based on process calculi. It also became clear that the initial design choices in the model have a far going impact on system implementation, including efficiency and scalability. Therefore, it is important to precisely define a range of application targets and the future environment, while designing the system model. We have been much aware of this fact when designing Nomadic Pict, whose model has been guided by concrete assumptions about applications which we would like to support.

Facile uses synchronous channels for communication between concurrent processes (possibly running on networked machines). A function `send` takes channel and value as arguments, and returns unit value after communication is completed. A hand-shake protocol is used for delivering messages between networked machines. If the remote site is not reachable, a timeout exception can be handled by the sender process (which is blocked on `send`). The synchronous model better matches the concept of functions (a function returns after the computation is finished), thus leading to a natural integration of function invocations and primitives for channel communication. While some process A is blocked on `send`, other concurrent processes of the same application can continue execution unless they have pending communications with A. This communication dependency between processes may create a problem if processes are distributed in wide area networks, where links can be slow or broken. Thus, programmers may prefer to spawn new Facile processes each time there is some value to send across the network. A more asynchronous model is also required for applications executing in a mobile environment, where mobile devices can often work in a disconnected mode. Nomadic Pict supports asynchronous channels on top of standard network protocols. Such a simple communication model allows us to experiment with different concurrency control policies and levels of synchrony (the algorithms can be expressed as translations and used in applications).

From a conceptual point of view, a channel is a synchronisation point allowing many concurrent processes to communicate. In Facile, the concept has been extended to a distributed world. In the implementation, a channel is represented as a data structure. Each process attempting to communicate over a channel leaves a request on it. A decentralised approach for channel management has been adopted, in which a programmer has got control over the node on which channels are created. A channel manager (one per channel) takes care of matching senders and receivers using a particular channel. The manager can communicate with Facile processes using links (e.g. sockets). Channel implementations reside in the nodes where they have been created. If a node fails, channels created on this node disappear (the system can still communicate over channels created on other nodes, though). The channel access mechanism utilizes location hints in (globally unique) channel identifiers and as a fall-back mechanism, a hash table that maps each channel identifier to its current implementation.

One of the initial goals of Facile was to seamlessly extend the functional and concurrent primitives to distributed computing. This has been done while showing much care to preserve the semantic difference between distributed and concurrent (but within the same shared address space) computation. The issues of *locating* processes and being aware of failures and distribution is in sharp contrast with the popular object-oriented approaches which attempt to deliver full transparency. Nevertheless, implementing one single policy of distributed channels can be controversial. Synchronisation between the sender and receiver processes may be expensive in terms of additional control messages, especially if the two processes, located on different nodes, have to share a channel created on some third node, and thus they end up talking to this third node. Also, some further work can be needed by the channel managers to coordinate communications that stem from choices. Efficiency problems caused by using distributed channels in some applications have been noticed by Facile designers and future work was planned, e.g. to establish fast audio/video connections.

Currently, a programmer can choose the nodes where channels will be created, thus leading to improved efficiency in particular cases. Some applications, however, may require even more flexibility, which cannot easily be achieved with a static implementation of channels. An example is mobile communicating agents, which may migrate often and while migrating want to preserve any pending communications with other agents on the same channels. In those agent applications, which can span nodes distributed on the whole Internet, it would be desirable to optimise message communication (i.e. reduce the number of forwarding pointers or aliasing) and, in particular, avoid (if possible) any long distance queries to some third parties (here

channel structures). However, it has been demonstrated that Facile can be successfully used to build *Mobile Service Agents* (MSAs) [TKLC95], which can be understood in the usual way, i.e. as self-contained pieces of software which can serve as *local* representatives for remote services, provide interactive access to data they accompany, and carry out tasks for a mobile user temporarily disconnected from the network.

A follow-up of work on Facile and the Join Language (described below) is the π_1-calculus, an asynchronous calculus with *uniqueness* of the receptor [Ama]. It assumes that every channel name is associated with a *unique* process which receives messages addressed to that name (communication becomes point-to-point). Such asynchronous point-to-point communication does not require synchronisation between possibly distant processes and therefore it makes minimal assumptions on the capabilities of the distributed system. As a consequence of receptor uniqueness, any received names will have a send only capability.

In contrast, our best effort approach allows for unrestricted communication on π-channels in all those cases which do not constrain efficiency of the implementation, so they can be freely used, e.g., for a local computation within the shared address space of an agent. However, in the case of communication between migrating agents, the use of channels is restricted (see §2.2.1). At the same time, our approach is liberal, i.e. although we admit that there is no implementation of distributed channels that is efficient for all possible applications, we do not try to evade the problem. Instead, we offer a programmer the possibility of encoding distributed channels for particular applications, as translations on top of agents.

3.3.2 The Join Language

The Join Language [FGL+96] is a distributed programming language based on the join-calculus, a calculus of concurrent processes that communicate through named, one-directional channels [FG96]. Concerns about mobility and distribution resulted in abandoning the π-calculus channel communication and integrating the input action and receiving process within one language primitive called a *join-pattern*; the join-pattern additionally allows many inputs (from many sending processes) to be synchronised in one action. In many important respects, the join-calculus resembles the π-calculus, e.g. a channel name can be communicated through channels, possibly outside the lexical scope of its definition. However, only the process that creates a channel can receive messages on the channel (so here channel names can better mimic, e.g. Amoeba port identifiers).

Below is an example process term of the join-calculus showing the main syntactic categories of processes, definitions, and join-patterns.

$$\textbf{def}\ (x\langle y\rangle|t\langle\rangle \rhd\ P) \wedge (x\langle z\rangle|t\langle\rangle \rhd\ Q)\ \textbf{in}\ x\langle a\rangle|t\langle\rangle|x\langle b\rangle$$

The main process $x\langle a\rangle|t\langle\rangle|x\langle b\rangle$ is a parallel composition of three outputs on channels x and t. The definition $(x\langle y\rangle|t\langle\rangle \rhd\ P) \wedge (x\langle z\rangle|t\langle\rangle \rhd\ Q)$ can receive some value (a or b in our example) on the channel named x and a null value on channel t, and continue with either some guarded process P or Q ($\wedge$ denotes a non-deterministic choice). The guarded process can be executed only if there are inputs available on *both* channels x and t in the corresponding join-pattern $x\langle\_\rangle|t\langle\rangle$. Since we have two outputs on x but only one output on t, the execution of P and Q is mutually exclusive. Below, we explain the syntactic forms in more detail.

A join-pattern J is a non-empty list of message patterns, denoted $J_1|J_2|..|J_n$, where each message pattern J_i is of the form $x_i\langle y_1, ..., y_{m_i}\rangle$ $(1 \leq i \leq n)$. A join-pattern J is much like a guard for a definition $J \rhd P$. Let us consider first a simple case where J is a single message input $x\langle y_1, ..., y_m\rangle$. In this case, process P can only execute if there is some message matching join-pattern J (i.e. some value v has been sent on x, and v has arity m). The names $y_1, ..., y_m$ are bound in P, and should all be distinct. The name x is also bound in P — intuitively, it is the name of a polyadic channel that is being defined. The novelty of the join-calculus stems from the fact that a number of synchronous inputs can be grouped together and represented by a join-pattern of the form $J_1|J_2|..|J_n$, where each J_i $(1 \leq i \leq n)$ is a single input. In this case, process P in definition $J \rhd P$ will block until the whole join-pattern can be triggered, i.e. there are values matched by *all* inputs $J_1..J_n$. If there are a number of messages matching J then accordingly a number of instances of P may execute concurrently.

In addition to definitions D of the form $J \rhd P$, the language allows definitions of the form $D \wedge D'$. In particular, a conjunction such as $x\langle y\rangle \rhd P \wedge x\langle z\rangle \rhd Q$, where the same name x appears in more than one definition D, is also legal, and if there is a message on x, then either P or Q will be nondeterministically chosen for execution.

A process P is the asynchronous output (such as $x\langle y\rangle$), definition of port names **def** D **in** P, or parallel composition of processes $P_1|..|P_n$. Definitions obey lexical scoping rules. In particular, given a process **def** D **in** P, a channel name defined in D is recursively bound in the whole process, including D.

In order to express distributed programs and mobile agents, the Join Language uses explicit locations and primitives for mobility defined in the distributed join-calculus [FGL+96]. Intuitively, a location resides on a physical site and contains a group of processes. Locations form a tree of locations with a meta-location as a root. Children of the root location abstract away physical machines. To introduce new locations, the syntax of definitions is extended with a new location constructor $a[D : P]$, which creates a sublocation of the current location containing the unique definition D and the unique running process P (the initialisation of location a). There are some assumptions about the uniqueness of location and channel names, such as a channel name defined in a definition can only appear in the join-patterns of one location. We can atomically move locations between sites: the migration primitive $go\langle b, \kappa\rangle$ invoked inside location a will cause a and its subtree to move in the tree of locations so that a will become a sublocation of b. On arrival, the continuation $\kappa\langle\rangle$ can trigger other computations.

The failure semantics of the join-calculus assumes the *fail-stop* model of locations. There are two primitives `halt` and `fail` in the calculus to model failures. The former primitive executed at location a will make the location permanently inert, while `fail`$\langle a, \kappa\rangle$ triggers $\kappa\langle\rangle$ after it detects that a has failed, i.e. that a or one of its parent locations has halted.

Discussion Join-patterns may simplify writing of distributed protocols which have to synchronise receipt of messages from many sources. The non-trivial synchronisation can be expressed concisely by using only one language primitive (definition), which has a well-defined semantics. Since join-patterns are meant to be used heavily in the Join Language programs, as the only synchronisation primitive available, their compilation requires much care. The deterministic finite-state automata modelling the synchronisation of message receipts tend to be rather complicated (however some optimisations are possible, as described in [ML98]). In Facile and Nomadic Pict, the composite synchronisation has to be expressed using more elementary language primitives which have a more straightforward interpretation.

A definition $x\langle y\rangle \triangleright P$, which contains a join-pattern with a single message input, resembles a process abstraction **`def`** `x y = P` in Pict, which denotes a replicated input on polyadic channel x with a guarantee (forced by a type system) that channel x is not used for input in any other place; x and y are bound in process P (see also §3.2.4).

Formal encoding of the non-distributed choice-free join-calculus into the π-calculus (described, e.g. in [FG96]) is rather simple, because the join-calculus is somehow the π-calculus with restrictions on communication pat-

terns. However, some care is needed in the formal translation to prevent the context of the (typeless) π-calculus from reading on channels they receive from the translation.

The reverse translation (i.e. from the π-calculus to join-calculus) is less straightforward; it assumes that we associate with each channel x of the π-calculus two names x_o and x_i, for an output and input, and an enclosing definition that matches output and input. The sender simply sends values on x_o; the receiver defines a name for its continuation, and sends it as a reception offer on x_i (see [FG96] for details).

In a translation from the join-calculus to Nomadic π-calculus, agents could possibly be used to build "firewalls" to prevent the context from interfering with the translation, and also to model locations. Intuitively, a process **def** $x\langle p\rangle|y\langle z\rangle \rhd P$ **in** $x\langle v\rangle|y\langle w\rangle$ could be encoded using a replicated input abstraction inside an agent, and location-independent communication, e.g. as **new** x, y **in agent** $a = x?{*}p \rightarrow y?z \rightarrow [\![P]\!]$ **in** $(\langle a@?\rangle x!v \mid \langle a@?\rangle y!w)$. The conjunction of definitions might be translated into a parallel composition of (replicated) input abstractions (however, the order of inputs in each branch of a parallel composition is then important and some care is needed to avoid deadlock). Locations could be encoded using agents (which explicitly maintain the corresponding tree structure when migrating). Intuitively, a process **def** $a[x\langle p\rangle \ \rhd P \ : Q]$ **in** R could be encoded as **new** c **in agent** $a = (\textbf{new}\ x\ \textbf{in}\ (x?{*}p \rightarrow [\![P]\!]|[\![Q]\!]))$ **in** $[\![R]\!]$, where process Q (after performing some initialisation of location a) must explicitly extrude name x to R on continuation channel c. A fully abstract translation would have to be more subtle though.

The most important difference between the join-calculus and Nomadic π-calculus is that, in the former calculus, migrating agents can only communicate by location-independent messages. In any distributed implementation of the join-calculus, messages must be transparently and reliably routed to migrating agents (or "locations" in the join-calculus terminology). Thus, the Join Language corresponds to a first approximation to High Level Nomadic Pict, together with a suitable infrastructure encoded in Low Level Nomadic Pict.

In order to fully implement the semantics of the join-calculus, in particular location-independent communication and migration of nested locations, a sophisticated infrastructure is required (to deliver messages transparently, atomically migrate the whole subtree of locations, and prevent race conditions from arising between migrating locations). Such an infrastructure can be difficult to implement efficiently, especially in wide-area networks.

The "strong asynchronous" model of the join-calculus failure semantics assumes that we can observe location failure and no messages will be deliv-

ered from a location which is detected as failed. A basic impossibility result in distributed systems states that we cannot achieve distributed consensus (such as agreeing on which sites have failed) in a system which consists of a collection of asynchronous processes [FLP85]. Thus, complete accuracy of failure detection by non-faulty processes in an asynchronous system where failures may happen is impossible. In practice, however, a good approximation is provided by some degree of synchrony and failure detection, but the algorithms required are costly. Fortunately, the semantics of the join-calculus does not require that all locations have to maintain a consistent view on which locations have failed (i.e. some locations can detect failure, other locations do not). Thus, simpler algorithms can be used. For example, the "weak asynchronous" model [FGL+96], implemented in the Join Language as testing-equivalent of the join-calculus "strong asynchronous" model, assumes only that location a suspected as failed cannot respond to messages; this can be enforced by blocking output to a from all locations which detected a as failed (or have received messages which are causally depended on this failure detection). We argue, however, that the fail-stop model of location failure may not be convenient in the practice of distributed programming, especially for applications in networks where partitions can happen (e.g. caused by disconnected operation). A location which has been partitioned from other locations, and then suspected as failed, is supposed to terminate silently. Therefore, after the network heals (or after reconnection), the "failed" location may have to recreate its state under a new location name and join the application explicitly, so that all locations will get to know the new location name. This may not be practical in computation which involves many locations.

To summarise, the join-calculus offers a set of interesting primitives for synchronised patterns of communication, expressing locations, and modelling failures. The ability to transparently and atomically move a whole collection of locations from one place to another may also be convenient in some applications (e.g. when a server is about to shut down). Criticism of the approach is concerned with the difficulty of providing any efficient and scalable implementations of the model semantics in wide-area networks. In contrary, the primitives offered by Nomadic Pict have been explicitly designed for writing applications in wide-area networks with mobility. Such applications require primitives which have simple failure semantics, and efficient, straightforward implementations, immediately above present day computer networks. The language architecture should also allow the system to be split into layers, one of which deals with infrastructure algorithms (including a layer which deals with fault detection and failure semantics), and reason about the algorithms implementing each layer formally.

To sum up, Facile and the Join Language offer a lot of expressiveness through a small set of higher-level primitives, whereas our approach is aimed at a small number of low-level primitives that can be implemented efficiently, together with an easy way of encoding more expressive primitives on top. The implementations of Facile and the Join Language have evolved from toy systems to more industrial-strength distributions (like Facile Antigua Release [TLP+93] and JoCaml [CF99]), which support extensive libraries, optimised runtime systems, and have been used to write various applications. The Nomadic Pict language has currently only one implementation, which is described as part of this thesis.

Chapter 4

Infrastructure for Location-Independent Communication

In chapter 1, we introduced the problem of distributed infrastructure, required for location-independent message delivery to migrating agents. In this chapter, we describe the space of algorithms which might be useful for building such infrastructures. These are simple, generic versions of the algorithms which are used in real distributed systems with object mobility and in mobile networks. In §4.1, we use natural language to describe the algorithms. However, natural language descriptions are often ambiguous (as demonstrated in the last section, §4.3, of this chapter). Therefore, in §4.2, we present two algorithms in detail, expressed as translations in Nomadic Pict.

4.1 Algorithms

Let us define the space of algorithms for location-independent message delivery to migrating agents. The algorithms should support two operations: "migrate", facilitating the move of an agent to a new site, and "deliver", locating a specified agent and delivering a message. The tasks of minimizing the communication overhead of these two operations appear to be in conflict.

Awerbuch and Peleg [AP95] (see also Mullender and Vitányi [MV88]) stated the analogous problem of keeping track of mobile users in a distributed network (they consider two operations: "move", facilitating the move of a user to a new destination, and "find", enabling one to contact a specified user at its current address). They first examined two extreme strategies.

The *full-information* strategy requires every site in the network to maintain complete up-to-date information on the whereabouts of every user. This makes the "find" operation cheap. On the other hand, "move" operations are very expensive, since it is necessary to update information at every site. In contrast, the *no-information* approach does not assume any updates while migrating, thus the "move" operation has got a null cost. On the other hand, the "find" operation is very expensive because it requires global searching over the whole network. However, if a network is small and migrations frequent, the strategy can be useful. In contrary, the *full-information* strategy is appropriate for a near-static setting, where agents migrate relatively rarely, but frequently communicate with each other. Between these two extreme cases, there is space for designing intermediate strategies, that will perform well for any or some specific communication to migration pattern, making the costs of both "find" and "move" operations relatively cheap.

Awerbuch and Peleg [AP95] introduced the graph-theoretic concept of *regional matching*, and demonstrated how finding a regional matching with certain parameters enables efficient tracking of mobile users in a distributed network. The communication overhead of maintaining a distributed directory server used to keep track of mobile users is within a polylogarithmic factor of the lower bound. This result is important in the case of mobile computing, where the infrastructure should perform well, considering all mobile users and their potential communication to migration patterns. These patterns can vary, depending on people.

The choice of infrastructure algorithm(s) for a given application with mobile agents will depend strongly on many characteristics of the application and target network, especially on the expected statistical properties of communication and migration. In many mobile agent applications, however, we know the communication to migration pattern of mobile agents precisely. This enables the design of algorithms which are optimal for these special cases and simpler than the directory server mentioned above. In contrast, the design of an adaptive customized infrastructure for mobile computing depends on probabilistic assumptions about the estimated behaviour of mobile users. The infrastructure should therefore support all migration and communication scenarios, and optimise those scenarios which are likely to happen more often.

The task of deciding on a mobile agent infrastructure may involve many criteria. In our examination, we expand the space for interesting algorithms to many dimensions, considering not only the communication cost but also other factors, such as scalability, interoperability, and fault-tolerance. In wide area applications, sophisticated distributed algorithms will be required, allowing for dynamic system reconfigurations such as adding new sites to

the system, migrating parts of the distributed computation before shutting down some machines, tracing locations of different kinds of agents, and implementing tolerance of partial failures. The space of feasible algorithms and the trade-offs involved require detailed investigation. We are not giving a quantitative theoretical or empirical view of the algorithms, however, because it would be too hard to take under consideration all these factors. In chapter 6, we take an example application and define assumptions about the target network and behaviour of all agents involved. Then, we design an infrastructure which behaves well for our application (although other infrastructures would also be plausible). Any change of assumptions (e.g. about failures in the system) will require the infrastructure to be extended accordingly.

Different infrastructures can be characterised by a number of properties such as scalability, and tolerance of failures. An infrastructure is *scalable* if adding new sites or agents, or expanding the system to wide-area networks does not severely degrade overall system performance (in these terms, we consider two different kinds of scalability which explore either the *numerical* or *geographical* dimensions). Fault-tolerance is costly; the level of fault-tolerance and methods which can be used will depend on the target network and application demands. Below, we describe the algorithms and give some hints about the infrastructure scalability and fault-tolerance, where possible.

4.1.1 Central Server

Central Forwarding Server The server records the current site of every agent. Before migration an agent A informs the server and waits for ACK (containing the number of messages sent from the server to A). It then waits for all the messages due to arrive. After migration it tells the server it has finished moving.

If B wants to send a message to A, B sends the message to the server, which forwards it. During migrations (i.e. after sending the ACK) the server suspends forwarding.

Central Query Server The server records the current site of every agent. Migration support is the same as in the case of a Central Forwarding Server (except that ACK from the server to A would have to contain globally unique IDs of all messages which were included in the queries about A's location since A's last migration). The difference is in how locations communicate. The application messages will be delivered directly to destinations without any forwarding.

If B wants to send a message to A, B sends a query (containing the message ID) to the server asking for the current site of A, gets the current site s of A and sends the message to s. During A's migration (i.e. after sending the ACK to A) the server postpones replying to all queries about A's location until it gets confirmation from A that A has migrated. Before sending the ACK, the query server can send a current record about A's location (say s). The name s can be used again for direct communication with A. If a message arrives at a site that does not have the recipient then a message is returned saying 'you have to ask the name server again'.

Home Server Each site s has a server (one of the above) that records the current site of some agents — usually the agents which were created on s. Agent names contain an address of the server which maintains their locations.

On every migration agent A synchronises with the server whose name is part of A's name. If B wants to send a message to A, B resolves A's name and contacts A's server. Other details are as above.

Discussion Central Forwarding and Query Servers do not scale. If the number of agents is growing and communication and migration are frequent, the server can be a bottleneck. Home servers can improve the situation. The infrastructure can work fine for small-to-medium systems, where the number of agents is small.

If migrations are rare and also in the case of stream communication or large messages, the Query Server seems the better choice.

The algorithms do not support locality of agent migration and communication, i.e. migration and communication involve the cost of contacting the server which can be far away. If agents are close to the server, the cost of migration, search, and update is relatively low.

The server is a single-point of failure. In this and other algorithms we can use some of the classical techniques of fault-tolerance, e.g. state checkpointing, message logging and recovery [JZ88]. We can also replicate the server on different sites to enhance system availability and fault-tolerance. Group communication can provide adequate multicast primitives for implementing either *primary-backup* or *active* replication [GS96].

Mechanisms similar to Home Servers have been used in many systems which support process migration, e.g. in Sprite [DO91]. Caching has been used, e.g. in LOCUS [PW85], and V [Che88], allowing operations to be sent directly to a remote process without passing through another site. If the

cached address is wrong a home site of the process is contacted (LOCUS) or multicasting is performed (V).

In §4.2.1, we describe the Central Server algorithm in detail, as a translation in Nomadic Pict. In chapter 6, we begin with a centralised algorithm with caching, which evolves to match application demands of scalability (execution on wide-area networks) and support for disconnected operation (on laptop computers). In the translations, we shall use explicit acknowledgements instead of piggybacking. For example, the server simply does not acknowledge A's migration until it has received confirmation that all messages to A have been delivered. We obtain an algorithm which is a bit less asynchronous and optimised, but easier to understand.

4.1.2 Forwarding Pointers

Algorithm There is a forwarding daemon on each site. The daemon on site s maintains a current guess about the site of agents which migrated from s. Every agent knows the initial home site of every agent (it is part of an agent's name). If A wants to migrate from s_1 to s_2 it leaves a forwarding pointer at the local daemon. Communications follow all the forwarding pointers. If there is no pointer to agent A, A's home site is contacted. Forwarding pointers are left around forever.

Discussion There is no synchronisation between migration and communication as there was in centralised algorithms. A message may follow an agent which frequently migrates, leading to a race condition. A chain of forwarding pointers is sometimes used in combination with other infrastructures, e.g. hierarchical directory [AP95], where it is ensured (using locking in the directory) that the search will eventually be able to reach the migrating agent, even if the agent repeatedly moves.

The Forwarding Pointers algorithm is not practical for a large number of migrations to distinct sites (a chain of pointers is growing, increasing the cost of search). Some "compaction" methods can be used to collapse the chain, e.g. movement-based and search-based. In the former case, an agent would send backward a location update after performing a number of migrations (i.e. "move" operations); in the latter case, after receiving a number of messages (i.e. after a fixed number of "find" operations occurred).

Some heuristics can be further used such as search-update. A plausible algorithm can be as follows. On each site there is a daemon which maintains

forwarding addresses for all agents which ever visited this site. A *forwarding address* is a tuple $(timestamp, site)$ in which the site is the last known location of the agent and timestamp specifies the age of the forwarding address. Every message sent from agent B to A along the chain of forwarding pointers contains the latest available forwarding address of A. The receiving site may then update its forwarding address for the referenced agent, if required. Given conflicting guesses for the same agent, it is simple to determine which one is most recent using timestamps. When the message is eventually delivered to the current site of the agent, the daemon on this site will send an ACK to the daemon on the sender site, containing the current forwarding address. A similar algorithm has been used in Emerald [JLHB88], where the new forwarding address is piggybacked onto the reply message in the object invocation. Fowler [Fow85] has shown that it is sufficient to maintain the timestamp as a counter, incremented every time the object moves.

A single site fail-stop in a chain of forwarding pointers breaks the chain. A solution is to replicate the location information in the chain on k consecutive sites, so that the algorithm is tolerant of a failure of up to $k - 1$ adjoint sites. Stale pointers should be eventually removed, either after waiting a sufficiently long time, or purged as a result of a distributed garbage collection. Distributed garbage collection would require detecting global termination of all agents that might ever use the pointer, therefore the technique may not always be practically useful. Alternatively, some weaker assumptions could be made and the agents decide arbitrarily about termination, purging the pointers beforehand.

An analytical comparison of many pointer-based, centralised and distributed location management schemas for mobile computing can be found in [Kri96].

4.1.3 Broadcast

Data Broadcast Sites know about the agents that are currently present. An agent notifies a site on leaving and a forwarding pointer is left over until agent migration is finished.

If agent B wants to send a message to A, B sends the message to all sites in a network (domain). A site s discards or forwards the message if A is not at s (we omit details).

Query Broadcast Sites know about agents that are currently present. An agent notifies a site on leaving and a forwarding pointer is left over until agent migration is finished.

If agent B wants to send a message to A, B sends a query to all sites in a network (domain) asking for the current location of A. If site s receives the query and A is present at site s, then s suspends any migration of A until A receives the message from B. A site s discards or forwards the query if A is not at s.

Notification Broadcast Every site in a network (domain) maintains a current guess about agent locations. After migration A distributes in the network (domain) information about its new location.

If site s receives a message whose recipient is not at s (because it has already migrated or the initial guess was wrong), it waits for information about the agent's new location. Then s forwards the message. Location information is time-stamped. Messages with stale location information are discarded.

Discussion Broadcasts are not scalable to a large network (broadcasting in a large region is impractical), however some algorithms may be scalable to many agents on a small network.

The cost of communication in Query and Data Broadcasts is high (packets are broadcast in the network) but the cost of migration is low. Query Broadcast saves bandwidth if messages are large or in the case of stream communication.

Notification Broadcast has a high cost of migration (the location message is broadcast to all sites) but the communication cost is low and similar to forwarding pointers with pointer chain compaction.

In Data and Notification Broadcasts, migration can be fast because there is no synchronisation involved (in Query Broadcast migration is synchronised with communication); the drawback is a potential for race conditions if migrations are frequent. Site failures do not disturb the algorithms.

Although we usually assume that the number of sites is too large to broadcast anything, we may allow occasional broadcasts within, e.g. a local Internet domain, or local Ethernet. Broadcasts can be accomplished efficiently in bus-based multiprocessor systems. They are also used in radio networks. A realistic variant is to broadcast within a group of sites which belong to the itinerary of mobile agents known in advance.

Broadcast has also been used in Emerald [JLHB88] to find an object, if a node specified by a forwarding pointer is unreachable or has stale data. To reduce message traffic, only a site which has the specified object responds to the broadcast. If the searching daemon receives no response within a time limit, it sends a second broadcast requesting a positive or negative reply from

all other sites. All sites not responding within a short time are sent a reliable, point-to-point message with the location request.

The Jini lookup and connection infrastructure [AWO+99] uses multicast in the discovery protocol. A client wishing to find a Lookup Service sends out a known packet via multicast. Any Lookup Service receiving this packet will reply (to an address contained in the packet) with an implementation of the interface to the Lookup Service itself.

4.1.4 Group Communication

Algorithm The agents forming a group maintain a current record about the site of every agent in the group. Agent names form a totally ordered set. We assume communication which takes place within a group only.

Before migration an agent A informs the other agents in the group about its intention and waits for ACKs (containing the number of messages sent to A). It then waits for all the messages due to arrive and migrates. After migration it tells the agents it has finished moving. Multicast messages to each agent within a group are delivered in the order sent (using a first-in-first-out multicast).

If B wants to send a message to A, B sends the message to site s which is A's current location. During A's migrations (i.e. after sending the ACK to A) B suspends sending any messages to A.

If two (or more) agents want to migrate at the same time there is a conflict which can be resolved as follows. Suppose A and C want to migrate. If B receives migration requests from A and C, it sends ACKs to both of them and suspends sending any messages to agents A and C (in particular any migration requests). If A receives a migration request from C after it has sent its own migration request it can either grant ACK to C (and C can migrate) or postpone the ACK until it has finished moving to a new site. The choice is made possible by ordering agent names.

Discussion The advantage of this algorithm is that sites can be stateless (the location data are part of agent state).

However, in a system with failures the algorithm is more complicated than above. Agents are organised into *groups*, corresponding to multicast delivery lists, that cooperate to perform a *reliable multicast* (i.e. if one agent on the delivery list receives a reliable multicast message, every agent on the delivery list receives the message). A precise meaning to the notion of delivery list can be given by using *virtual synchrony* defined for non-movable groups [BJ87].

The current list of agents to receive a multicast is called the *group view*. The group view is consistent among all agents in the group. Processes are added to and deleted from the group via *view changes*. If agent A is removed from the view, the agents remaining in the view would assume that A has failed. Virtual synchrony guarantees that no messages from A will be delivered in the future (if A has not failed it must rejoin the group explicitly, using a *membership protocol*).

A problem is how agents can dynamically join the group, which can change sites. One solution is to leave forwarding pointers, such that agents which want to join (or rejoin) the group can follow them and "catch up" with at least one group member. Another solution is to have one agent within a group (a coordinator or manager) which never migrates. The algorithm for inter-group communication could then use the pointers or coordination agent for delivering messages that cross group boundaries.

The algorithm is suitable for frequent messages (or stream communication) between mobile agents and when migrations are rare. Agent failures and network partitions will not disturb agents which are alive; however, there are detailed subtleties which depend on the semantics of the algorithm implementing virtual synchrony. The group service algorithms for non-movable processes which have been originally proposed, e.g. in ISIS, are costly in terms of control messages and hard to use in networks larger than a LAN. However, they are also examples of scalable group membership and communication services implementing the virtual synchrony semantics, designed for wide-area networks [KSMD99].

4.1.5 Hierarchical Directory

Algorithm A tree-like hierarchy of servers forms a location directory (similar to DNS). Each server in the directory maintains a current guess about the site of some agents. Sites belong to regions, each region corresponds to a sub-tree in the directory (in the extreme cases the sub-tree is simply a leaf-server for the smallest region, or the whole tree for the entire network). The algorithm maintains an invariant that for each agent there is a unique path of forwarding pointers which forms a single branch in the directory; the branch starts from the root and finishes at the server which knows the actual site of the agent (we call this server the "nearest").

Before migration an agent A informs the "nearest" server X_1 and waits for ACK. After migration it registers at a new "nearest" server X_2, tells X_1 it has finished moving and waits for ACK. When it gets the ACK there is

already a new path installed in the tree (this may require installing new and purging old pointers within the smallest sub-tree which contains X_1 and X_2).

Messages to agents are forwarded along the tree branches. If B wants to send a message to A, B sends the message to the B's "nearest" server, which forwards it in the directory. If there is no pointer the server will send the message to its parent.

Discussion The algorithm above has a translation similar to the Forwarding Pointers translation (see 4.2.2), in which daemons would be replaced by directory servers. Certain optimisations are plausible, e.g. if an agent migrates very often within some sub-tree, only the root of the sub-tree would contain the current location of the agent (the cost of a "move" operation would be cheaper).

Different variants of the directory infrastructure can be proposed. A hierarchical directory was first proposed by Awerbuch and Peleg [AP89, AP95], for online tracking of mobile users. A proposition of Globe [vSHT99] uses a hierarchical location service for worldwide distributed objects [vSHBT98]. In [Mor99], Moreau describes an algorithm for routing messages to migrating agents which is also based on distributed directory service.

The Hierarchical Directory scales better than Forwarding Pointers and Central Servers. Also, some kinds of fault can be handled more easily (see [AP95], and there is also a lightweight crash recovery in the Globe system [BvST99]).

4.1.6 Arrow Directory

Some algorithms can be devised for a particular migration pattern. Below we describe the Arrow Distributed Directory protocol proposed by Demmer and Herlihy [DH98] for distributed shared memory systems. The protocol assumes that the whole object is always sent to the object requester.

Algorithm The arrow directory is given by a minimum spanning tree for a network, where the network is modelled as a connected graph. Each vertex models a node (site), and each edge a reliable communication link. A node can send messages directly to its neighbours, and indirectly to non-neighbours along a path. The directory tree is initialised so that following arrows (pointers) from any node leads to the node where the object resides.

When a node wants to acquire exclusive access to the object, it sends a message *find* which is forwarded via arrows and sets its own arrow to itself. When the other node receives the message, it immediately "flips" the arrow

to point back to the immediate neighbour who forwarded the message. If the node does not hold the object, it forwards the message. Otherwise, it buffers the message *find* until it is ready to release the object to the object requester. The node releases the object by sending it directly to the requester, without further interaction with the directory.

If two *find* messages are issued at about the same time, one will eventually cross the other's path and be "diverted" away from the object, following arrows towards the node (say v) where the other *find* message was issued. Then, the message will be blocked at v until the object reaches v, is accessed and eventually released.

Discussion The directory imposes an optimal distributed queue of object requests, with no point of bottleneck.

The arrow directory protocol was motivated by emerging *active network* technology [LWG98], in which programmable network switches are used to implement customized protocols, such as application-specific packet routing.

Naimi, Trehel, and Arnold [NTA96] describe a distributed mutual exclusion algorithm which is also based on a dynamically changing distributed directed graph. In the algorithm, when a node receives a message, it flips its edge to point to the node from which the request originated, instead of to the immediate source of the message as above.

4.2 Example Translations in Nomadic Pict

Due to limited space, we present only two translations in Nomadic Pict of the infrastructure algorithms described above. They will be the Central Forwarding Server and Forwarding Pointers infrastructure. Assumptions about the system are described in §2.2. The translations are taken almost verbatim from a program which can be executed by the Nomadic Pict system.

The first is one of the simplest algorithms possible, highly sequential and with a centralized server daemon; the second is one step more sophisticated, with multiple daemons maintaining forwarding-pointer chains. The algorithms have been chosen to illustrate the model, and the use of the language — algorithms that are widely applicable to actual mobile agent systems would have to be yet more delicate, both for efficiency and for robustness under partial failure.

Even the simplest of our algorithms, however, requires delicate synchronization that is easy to get wrong; expressing them as translations between well-defined low- and high-level languages provides a solid basis for discussion and algorithm design, subject to stated assumptions about the behaviour of

externally-provided network services. In the end of this section we discuss alternative descriptions.

A formal correctness proof of these two algorithms within our Nomadic π-calculus framework will appear in the complementary work by Unyapoth [Uny], who extended the Nomadic π-calculus with a labelled transition system and suitable techniques based on bisimulation.

4.2.1 A Central-Server Infrastructure Translation

The Central-Forwarding-Server algorithm presented in this section involves a central daemon that keeps track of the current sites of all agents and forwards any location-independent messages to them. The daemon is itself implemented as an agent which never migrates; the translation of a program then consists roughly of the daemon agent in parallel with a compositional translation of the program. For simplicity we consider only programs that are initiated as single agents, rather than many agents initiated separately on different sites. (Programs may, of course, begin by creating other agents that immediately migrate). The precise definition is given in Figures 4.1 and 4.2. Figure 4.2 defines a top-level translation $[\![]\!]^{Top}$. For each term `P` of the high-level language, considered as the body of an agent named `a` and initiated at site `s`, the result $[\![\texttt{P}]\!]^{Top}_{[\texttt{a s}]}$ of the translation is a term of the low-level language. The definition of $[\![]\!]^{Top}$ involves the body *Daemon* of the daemon agent (given in Figure 4.2) and an auxiliary compositional translation $[\![\texttt{P}]\!]_{[\texttt{a D SD}]}$, defined phrase-by-phrase, of `P` considered as part of the body of agent `a`, where the daemon agent `D` is assumed to be at site `SD`. The compositional translation is given in Figure 4.1.

Let us look first at the daemon. It contains three replicated inputs, on the `register`, `migrating`, and `message` channels, for receiving messages from the encodings of agents. The daemon is essentially single-threaded — the channel `lock` is used to enforce mutual exclusion between the bodies of the replicated inputs, and the code preserves the invariant that at any time there is at most one output on `lock`. The `lock` channel is also used to maintain the site map — a finite map from agent names to site names, recording the current site of every agent. The body of each replicated input begins with an input on `lock`, thereby acquiring both the lock and the site map.

Turning to the compositional translation $[\![\_]\!]_{[\texttt{a D SD}]}$, only three clauses are not trivial — for the location-independent output, agent creation, and agent migration primitives. We discuss each, together with their interactions with the daemon, in turn.

```
⟦ c@b!v ⟧[a D SD] def= <D@SD>message![b c v]

⟦agent b=P in Q⟧[a D SD] def= currentloc?s=
 (agent b =
    ( deliver?*[#X c:^X v:X] = ( <D@SD>dack![] | c!v )
    | <D@SD>register![b s]
    | ack?_= iflocal <a>ack![] then
               ( currentloc!s
               | ⟦P⟧[b D SD] )
             else ())
  in
    ack?_= ( currentloc!s
           | ⟦Q⟧[a D SD] ))

⟦migrate to s P⟧[a D SD] def= currentloc?_=
  ( <D@SD>migrating!a
  | ack?_ = ( migrate to s
              ( <D@SD>migrated!s
              | ack?_ = ( currentloc!s
                        | ⟦P⟧[a D SD])
              )))
```

Figure 4.1: A Central-Server Translation: The Compositional Translation

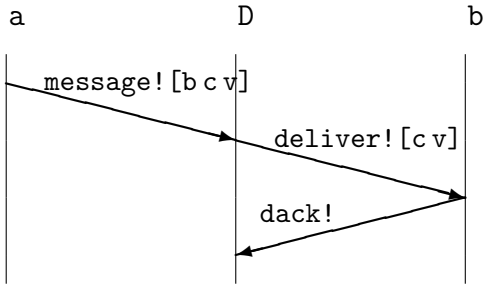
Location-independent output A location-independent output in an agent `a` is implemented simply by using a location-dependent output to send a request to the daemon `D`, at its site `SD`, on its channel `message`:

$$[\![\ \texttt{c@b!v}\ ]\!]_{[\texttt{a D SD}]} \stackrel{def}{=} \texttt{<D@SD>message![b c v]}$$

The corresponding replicated input on channel `message` in the daemon

```
| message?*[#X a:Agent c:^X v:X]= lock?m=
            switch (map.lookup m a) of
              {Found> s:Site} ->
                ( <a@s>deliver![c v]
                | dack?_ = lock!m)
              {NotFound> _} -> ()
```

first acquires the lock and current site map `m`, then looks up the target agent's site in the map and sends a location-dependent message to the `deliver` channel of that agent. It then waits to receive an acknowledgement (on the `dack` channel) from the agent before relinquishing the lock. This prevents the agent migrating before the `deliver` message arrives. Note that the `NotFound` branch of the lookup will never be taken, as the algorithm ensures that all agents register before messages can be sent to them. The inter-agent communications involved in delivery of a single location-independent output are illustrated below.



Creation In order for the daemon's site map to be kept up to date, agents must register with the daemon, telling it their site, both when they are created and after they migrate. Each agent records its current site internally as an output on its `currentloc` channel. This channel is also used as a lock, to enforce mutual exclusion between the encodings of all agent creation and migration commands within the body of the agent.

The encoding of an agent creation in an agent `a`

```
[[agent b=P in Q]]_[a D SD]  =def=
     currentloc?s=
       (agent b =
          ( deliver?*[#X c:^X v:X] = ( <D@SD>dack![] | c!v )
          | <D@SD>register![b s]
          | ack?_= iflocal <a>ack![] then
                     ( currentloc!s
                     | [[P]]_[b D SD] )
                   else ())
        in
          ack?_= ( currentloc!s
                 | [[Q]]_[a D SD] ))
```

first acquires the lock and current site `s` of `a`, and then creates the new agent `b`. The body of `b` sends a `register` message to the daemon and waits for an acknowledgement. It then sends an acknowledgement to `a`, initializes the lock for `b` and allows the encoding of the body `P` of `b` to proceed. Meanwhile, in `a` the lock is kept until the acknowledgement from `b` is received. The body of `b` is put in parallel with the replicated input

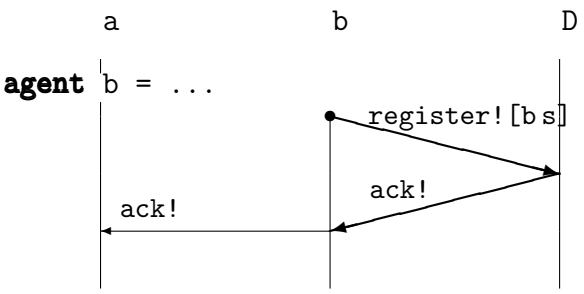
```
| deliver?*[#X c:^X v:X] = ( <D@SD>dack![] | c!v )
```

which will receive forwarded messages for channels in `b` from the daemon, send an acknowledgement back, and deliver the value locally to the appropriate channel.

The replicated input on `register` in the daemon

```
| register?*[a s]= lock?m=
                   ( lock!(map.add m a s)
                   | <a@s>ack![])
```

first acquires the lock and current site map, replaces the site map with an updated map, thereby relinquishing the lock, and sends an acknowledgement to the registering agent. The inter-agent communications involved in a single agent creation are illustrated below.



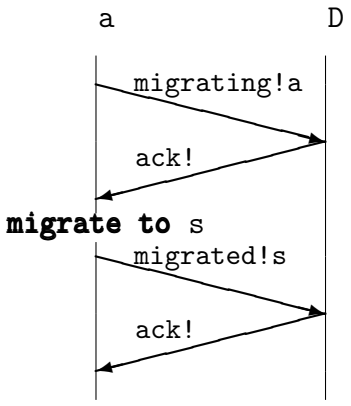
Migration The encoding of a **migrate** in agent a

```
⟦migrate to s P⟧[a D SD] =def currentloc?_=
  ( <D@SD>migrating!a
  | ack?_ = ( migrate to s
              ( <D@SD>migrated!s
              | ack?_ = ( currentloc!s
                        | ⟦P⟧[a D SD])
              )))
```

first acquires the lock for a (discarding the current site data). It then sends a migrating message to the daemon, waits for an ack, migrates to its new site s, sends a migrated message to the daemon, waits again for an ack, and releases the lock (with the new site s). The replicated input on migrating in the daemon

```
| migrating?*a= lock?m= switch (map.lookup m a) of
    ({Found> s:Site} ->
       ( <a@s>ack![]
       | migrated?s' =
          ( lock!(map.add m a s')
          | <a@s'>ack![]))
     {NotFound> _} -> ())
```

first acquires the lock and current site map, looks up the current site of a[1] and sends an ack to a at that site. It then waits to receive the new site, replaces the site map with an updated map, thereby relinquishing the lock, and sends an acknowledgement to a at its new site. The inter-agent communications involved in a single migration are shown below.



The top level Putting the daemon and the compositional encoding together, the top level translation, defined in Figure 4.2, creates the daemon

[1] Alternatively, the current site of a could be sent in the migrating message.

```
new register   : ^[Agent Site]
new migrating  : ^Agent
new migrated   : ^Site
new message    : ^[#X Agent ^X X]

new dack       : ^[]
new deliver    : ^[#X ^X X]
new ack        : ^[]
new currentloc : ^Site
```

$[\![\mathtt{P}]\!]^{Top}_{[\mathtt{a\ s}]} \stackrel{def}{=}$

```
(agent D = Daemon in
 val SD = s
  ( deliver?*[#X c:^X v:X] = ( <D@SD>dack![] | c!v )
  | <D@SD>register![a s]
  | ack?_ = (currentloc!s | [[P]]_[a D SD])))
```

$Daemon \stackrel{def}{=}$

```
(new lock : ^(Map Agent Site)
 (lock!(map.make ==)
 | register?*[a s]= lock?m=
     ( lock!(map.add m a s)
     | <a@s>ack![])
 | migrating?*a= lock?m=
     switch (map.lookup m a) of (
       {Found> s:Site} ->
         ( <a@s>ack![]
         | migrated?s' =
             ( lock!(map.add m a s')
             | <a@s'>ack![]))
       {NotFound> _} -> ())
 | message?*[#X a:Agent c:^X v:X]= lock?m=
     switch (map.lookup m a) of (
       {Found> s:Site} ->
         ( <a@s>deliver![c v]
         | dack?_ = lock!m)
       {NotFound> _} -> ())
))
```

where the **new**-bound names, SD, and D, do not occur in P.

Figure 4.2: A Central-Server Translation: The Top Level and the Daemon

agent, installs the replicated input on `deliver` for `a`, registers agent `a` to be at site `s`, initializes the lock for `a`, and starts the encoding of the body $[\![P]\!]_{[a\ D\ SD]}$.

4.2.2 A Forwarding-Pointers Infrastructure Translation

In this section we give a more distributed Forwarding-Pointers algorithm, in which daemons on each site maintain chains of forwarding pointers for agents that have migrated. It removes the single bottleneck of the centralised-server solution in the preceding section; it is thus a step closer to algorithms that may be of wide practical use. The algorithm is more delicate; expressing it as a translation provides a more rigorous test of the framework.

As before, the translation consists of a compositional encoding of the bodies of agents, given in Figure 4.3, daemons, defined in Figure 4.4, and a top-level translation putting them together, given in Figure 4.5. The top-level translation of a program, again initially a single agent, creates a daemon on each site mentioned by the agent. These will each maintain a collection of forwarding pointers for all agents that have migrated away from their site. To keep the pointers current, agents synchronize with their local daemons on creation and migration. Location independent communications are implemented via the daemons, using the forwarding pointers where possible. If a daemon has no pointer for the destination agent of a message then it will forward the message to the daemon on the site where the destination agent was created; to make this possible an agent name is encoded by a triple of an agent name and the site and daemon of its creation. Similarly, a site name is encoded by a pair of a site name and the daemon name for that site. There is a translation of types with clauses

$$[\![\texttt{Agent}]\!] \stackrel{def}{=} \texttt{[Agent Site Agent]}$$
$$[\![\texttt{Site}]\!] \stackrel{def}{=} \texttt{[Site Agent]}$$

We generally use lower case letters for site and agent names occurring in the source program and upper case letters for sites and agents introduced by its encoding.

Looking first at the compositional encoding, in Figure 4.3, each agent uses a `currentloc` channel as a lock, as before. It is now also used to store both the site where the agent is and the name of the daemon on that site. The three interesting clauses of the encoding, for location-independent output, creation, and migration, each begin with an input on `currentloc`. They are broadly similar to those of the simple Central-Server translation.

```
⟦ c@b!v ⟧A  def=  currentloc?[S DS]=
      iflocal <DS>message![b c v] then
        currentloc![S DS]
      else currentloc![S DS]

⟦ agent b=P in Q ⟧A  def=  currentloc?[S DS]=
      agent B =
       (val b = [B S DS]
        ( currentloc![S DS]
        | <DS>register!B
        | ack?_= iflocal <A>ack![] then ⟦P⟧B else () ))
      in
        val b = [B S DS]
        ack?_= (currentloc![S DS]
               | ⟦Q⟧A)

⟦ migrate to u P ⟧A  def=  currentloc?[S DS]=
     (val [U DU] = u
      if (&& (== #Agent DS DU) (== #Site S U)) then
        ( currentloc![U DU]
        | ⟦ P ⟧ A )
      else
        ( <DS>migrating!A
        | ack?_=
            (migrate to U
             ( <DU>register!A
             | ack?_=
                ( <DS@S>migrated![A [U DU]]
                | ack?_=
                    ( currentloc![U DU]
                    | ⟦ P ⟧ A ))))))

⟦ iflocal <b>c!v then P else Q ⟧A  def=
     (val [B _ _] = b
      iflocal <B>c!v then ⟦P⟧A else ⟦Q⟧A)
```

Figure 4.3: A Forwarding-Pointers Translation: The Compositional Translation

$Daemon_{[\texttt{S DS}]} \stackrel{def}{=}$

```
(new lock : ^(Map Agent ^[Site Agent])
 (lock!(map.make ==)
 | register?*B:Agent=
     lock?m= switch (map.lookup m B) of (
       {Found> Bstate:^[Site Agent]} ->
          Bstate?_=
          ( Bstate![S DS]
          | lock!m
          | <B>ack![])
       {NotFound> _} ->
          (new Bstate : ^[Site Agent]
           ( Bstate![S DS]
           | lock!(map.add m B Bstate)
           | <B>ack![])))
 | migrating?*B:Agent=
     lock?m= switch (map.lookup m B) of (
       {Found> Bstate:^[Site Agent]} ->
          Bstate?_=
          ( lock!m
          | <B>ack![])
       {NotFound> _} -> ())
 | migrated?*[B:Agent [U:Site DU:Agent]]=
     lock?m= switch (map.lookup m B) of (
       {Found> Bstate:^[Site Agent]} ->
          ( lock!m
          | Bstate![U DU]
          | <B@U>ack![])
       {NotFound> _} -> ())
 | message?*[#X [B:Agent U:Site DU:Agent] c:^X v:X]=
     lock?m= switch (map.lookup m B) of (
       {Found> Bstate:^[Site Agent]} ->
          ( lock!m
          | Bstate?[R DR]=
              iflocal <B>c!v then
                Bstate![R DR]
              else (<DR@R>message![#X [B U DU] c v]
                   | Bstate![R DR]))
       {NotFound> _} ->
          ( lock!m
          | <DU@U>message![#X [B U DU] c v])))))
```

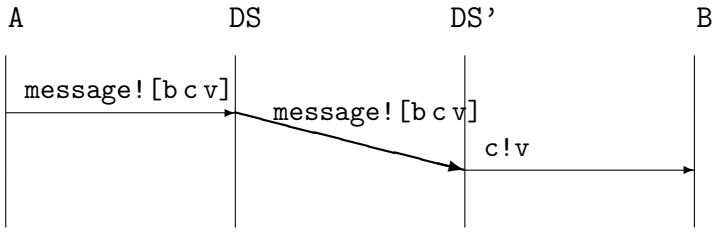
Figure 4.4: A Forwarding-Pointers Translation: The Daemon

Turning to the body of a daemon, defined in Figure 4.4, it is parametric in a pair `s` of the name of the site `S` where it is and the daemon's own name `DS`. It has four replicated inputs, on its `register`, `migrating`, `migrated`, and `message` channels. Some partial mutual exclusion between the bodies of these inputs is enforced by using the `lock` channel. The data stored on the `lock` channel now maps the name of each agent that has ever been on this site to a lock channel (e.g. `Bstate`) for that agent. These agent locks prevent the daemon from attempting to forward messages to agents that may be migrating. Each stores the site and daemon (of that site) where the agent was last seen by this daemon — i.e. either this site/daemon, or the site/daemon to which it migrated from here. The use of agent locks makes this algorithm rather more concurrent than the previous one — rather than simply sequentialising the entire daemon, it allows daemons to process inputs while agents are migrating, so many agents can be migrating away from the same site, concurrently with each other and with delivery of messages to other agents at the site.

Location-independent output A location-independent output `c@b!v` in agent `A` is implemented by requesting the local daemon to deliver it. (Note that `A` cannot migrate away before the request is sent to the daemon and a lock on `currentloc` is released.)

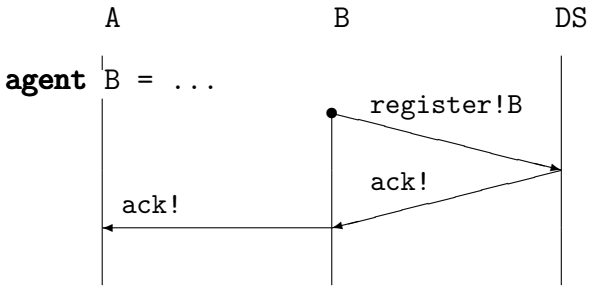
The `message` replicated input of the daemon gets the map `m` from agent names to agent lock channels. If the destination agent `B` is not found, the message is forwarded to the daemon `DU` on the site `U` where `B` was created. Otherwise, if `B` is found, the agent lock `Bstate` is grabbed, obtaining the forwarding pointer `[R DR]` for `B`. Using **`iflocal`**, the message is then either delivered to `B`, if it is here, or to the daemon `DR`, otherwise. Note that the `lock` is released before the agent lock is requested, so the daemon can process other inputs even if `B` is currently migrating.

A single location-independent output, forwarded once between daemons, involves inter-agent messages as below. (Communications that are guaranteed to be between agents on the same site are drawn with thin arrows.)



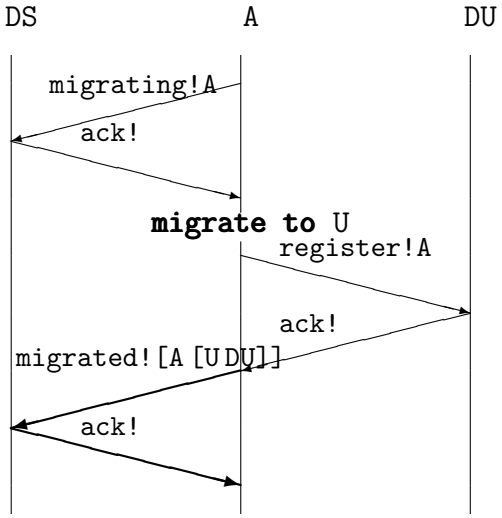
Creation The compositional encoding for **agent** is similar to that of the encoding in the previous section. It differs in two main ways. Firstly the source language name `b` of the new agent must be replaced by the actual agent name `B` tupled with the names `S` of this site and `DS` of the daemon on this site. Secondly, the internal forwarder, receiving on `deliver`, is no longer required: the final delivery of messages from daemons to agents is now always local to a site, and so can be done using **iflocal**. An explicit acknowledgement (on `dack` in the simple translation) is likewise unnecessary.

A single creation involves inter-agent messages as below.



Migration Degenerate migrations, of an agent to the site it is currently on, must now be identified and treated specially; otherwise the Daemon can deadlock. An agent `A` executing a non-degenerate migration now synchronises with the daemon `DS` on its starting site `S`, then migrates, registers with the daemon `DU` on its destination site `U`, then synchronises again with `DS`. In between the first and last synchronisations the agent lock for `A` in daemon `DS` is held, preventing `DS` from attempting to deliver messages to `A`.

A single migration involves inter-agent messages as below.



```
new register   : ^Agent
new migrating  : ^Agent
new migrated   : ^[Agent [Site Agent]]
new message    : ^[#X [Agent Site Agent] ^X X]
new ack        : ^[]
new currentloc : ^[Site Agent]
```

$[\![\mathtt{P}]\!]^{Top}_{[\mathtt{a\ s1\ \ldots\ sn}]} \stackrel{def}{=}$

```
(new daemondaemon : ^Site
 new nd : ^[Site Agent]
 agent A =
   (daemondaemon?*S:Site=
      (agent D =
       (migrate to S
         ( Daemon[S D]
         | <A@s1>nd![S D]))
       in ())
   | daemondaemon!s1 | nd?s1=
      ...
        (daemondaemon!sn | nd?sn=
           (val [S1 DS1] = s1
            val a = [A S1 DS1]
             (currentloc!s1
             | <DS1>register!A
             | ack?_= [[ P ]]A
            ))))
in ())
```

where P is initiated on site s1, the free site names in P are s1 .. sn, and the **new**-bound names, S1, DS1, and A do not occur in P.

Figure 4.5: A Forwarding-Pointers Translation: The Top Level

Local communication The translation of **iflocal** must now extract the real agent name B from the triple b, but is otherwise trivial.

The top level The top-level translation of a program P, given in Figure 4.5, dynamically creates a daemon on each site mentioned in P. Each site name si is re-bound to the pair [si DSi] of the site name together with the respective daemon name. A top-level agent A is created and initialised; the agent name a is re-bound to the triple [A S1 DS1] of the low-level agent name A together with the initial site and daemon names.

4.3 Alternative Descriptions

Distributed infrastructure algorithms can usefully be expressed as translations between the two-levels of the Nomadic Pict language. Almost the entire encoding of the Forwarding-Pointers protocol can be given in 2.5 pages, rather concise for a non-trivial executable distributed infrastructure. The code fragments are taken almost verbatim from the executable source, with some minor sugar. In our experience with designing such algorithms we have found that the language provides a good level of abstraction at which potential problems (such as deadlocks and lost messages) can be seen rather clearly. The uniform treatment of concurrency and asynchronous messages both within agents and between machines is a significant gain.

Below, we mention about other possible ways of describing distributed algorithms — we briefly consider diagrammatic, natural language, pseudocode, and automata based approaches.

The *diagrams* used in §4.2.1, 4.2.2 convey basic information about the algorithms — the messages involved in isolated transactions — but they are far from complete descriptions and can be misleading. The correctness of the algorithms depends on details of synchronisation and locking that are precisely defined by the translation but are hard to express visually. For example, a message flow diagram of the **agent** b=P **in** Q encoding would have to be parametrised on actual sites and conditional upon behaviour of the body P of b, thus leading to complicated multi-diagrams.

Natural language descriptions, as given in §4.1, are often ambiguous. For example, in the case of the Forwarding-Pointers translation (see Figure 4.3), if we replace in the compositional translation of **agent** a line

```
| ack?_= iflocal <A>ack![] then [[P]]_B else ()
```

by a following line

```
| ack?_= ( <A>ack![] | [[P]]_B )
```

then we get a following definition of agent creation which is erroneous

```
              ⟦ agent b=P in Q ⟧A  def=  currentloc?[S DS]=
                   (val A = A
                    agent B =
                      (val b = [B S DS]
                       ( currentloc![S DS]
line  6:               | <DS>register!B
line  7:               | ack?_= ( <A>ack![] | ⟦P⟧B ) ))
                    in
                      val b = [B S DS]
line 10:              ack?_= (currentloc![S DS]
                             | ⟦Q⟧A))
```

However, both definitions might well be described informally using exactly the same text, as follows:

> The body of `b` sends a `register` message to the daemon (see line 6) and waits for an acknowledgement (line 7). It then sends an acknowledgement to `a` and allows the encoding of the body `P` of `b` to proceed (line 7). Meanwhile, in `a` the lock is kept until the acknowledgement from `b` is received (line 10).

What is missing are possible interactions between the encoding of **agent** and user program `P`. In the original definition we have a *guarantee* that the encoding of the body `P` of `b` will not proceed *before* the acknowledgement to `a` is actually sent and delivered. In the definition above, the acknowledgement to `a` is sent in *parallel* with the execution of `P`. Therefore, it does not prevent `b` from migrating away to a new site (e.g. if `P` would proceed with **migrate**) and sending a local message `ack` on that new site. In that case, `a` would stay locked and wait for `ack` forever.

For a *pseudocode* description to provide a clear (if necessarily informal) description of an algorithm the constructs of the pseudocode must themselves have clear intuitive semantics. This may hold for pseudocodes based on widespread procedural languages, such as Pascal. Infrastructure algorithms, however, involve constructs for agent creation, migration and communication. These do not have a widespread, accepted, semantics — a number of rather different semantic choices are possible — so more rigorous descriptions are required for clear understanding.

Automata-based descriptions have been widely used for precise specification of distributed algorithms, for example in the text of Lynch [Lyn96]. Automata do not allow agent creation and migration to be represented directly, so for working with a mobile agent algorithm one would either have

to use a complex encoding or consider only an abstraction of the algorithm — a non-executable model, rather than an executable complete description.

The modelling approach has been followed by Amadio and Prasad in their work on IP mobility [AP98]. They consider idealizations of protocols from IPv6 proposals for mobile host support, expressed in a variant of CCS, and prove correctness results. There is a trade-off here: the idealizations can be expressed in a simpler formal framework, greatly simplifying correctness proofs, but they are further removed from implementation, inevitably increasing the likelihood that important details have been abstracted away.

Chapter 5

Infrastructure Design for Mobile Agents

In this chapter, we describe several kinds of application using mobile agents in order to improve locality of computation, support disconnected operation, avoid transferring large volumes of data, facilitate fault-tolerance, and adapt to changes in the network characteristics and in the user environment. For each kind we define a simple example application and describe an appropriate infrastructure for location-independent communication and migration in a wide-area network. In the previous chapter, we characterised the algorithms for delivering location-independent messages to migrating entities and assess their usefulness in a general case. Below, we apply this knowledge and use some of the algorithms and techniques in the design of infrastructures which are application-specific.

The goal of this chapter is to demonstrate a variety of simple infrastructures which are useful for specific applications. It can be observed that in many different applications, the pattern of agent migration is different and often limited, e.g. agents migrate only once or twice, migration is within a local-area network or between a few sites which are known in advance, agents can only migrate to or from a central site, and between a mobile computer and the network, etc. We do not attempt to specify all the infrastructures formally, so we use natural language descriptions. In the next chapter, however, we describe the design of a suitable infrastructure for an example application in more detail, presenting an almost complete executable description of the infrastructure in Nomadic Pict.

5.1 Resource Monitoring

Mobile agents (or mobile code) can be useful as *monitors*, i.e. programs that are dispatched to remote sites, where they monitor a set of sensors or data streams, and take some actions whenever certain prescribed conditions apply. For example, agents can be dispatched to a remote server and control a unique piece of external equipment (we assume here that the latency in network transmission is high compared to real-time constraints imposed by the external equipment, so RPC cannot be used). In other applications, agents can be sent to stock market sites, auction sites, vendors' e-commerce sites, or other data stores, and notify their clients that some event has happened (e.g. share prices dropped below some threshold or a new book of a favourite author has been published).

The advantage of mobile agents over the client-server and event systems is the higher level of customization possible: both prescribed *conditions* and triggered *actions* are defined by the client and dispatched as an autonomous mobile agent. The remote site provides access only to raw data which can be trusted and verified on demand (e.g. a stock-market server would publish a list of companies and current share prices).

Moreover, bringing computation to the source of data has two potential benefits. Firstly, mobile agents acting on behalf of a user can react immediately, irrespective of network delays and broken links (so, for example, agents in the ticket reservation system could book specified flights as soon as they become available). Secondly, the client can stay disconnected while the mobile agents perform their actions, and only reconnect to collect results.

5.1.1 Migration and Communication Pattern

At the application level, we would like to think of agent migration as "one-hop" between the client and resource sites. However, in many applications we may not have direct access to the resource sites (e.g. because they are hidden behind firewalls, or they are temporarily disconnected from the network). Instead, mobile agents migrate to some host machines where they are authenticated and stored in a queue. The system can periodically allow the agents from the queue to migrate from the host machine to the resource site. In some applications, agents may have to migrate forth and back between the host machine and resource sites (e.g. if the resource site needs to be temporarily disconnected, or the agent expired and the client has to pay for service continuation). In such situations, it is convenient to move both code and thread of control between machines; therefore we use mobile agents, not mobile code.

Clients may not be aware of the low-level migration between many machines; they use location-independent asynchronous messages for communicating with monitoring agents. The client may stay disconnected, and only connect to collect the data.

5.1.2 Example Infrastructure

The concept of *waiting room* forms a main building block of our infrastructure. The "waiting room" is a site where agents can be received, registered, and typechecked to prevent malicious behaviour, but not executed. The agents are queued and forwarded for execution to other sites upon request.

Such a two-tier mechanism of migration is useful when the destination sites may be: (1) temporarily disconnected from the network, (2) located inside firewalls, or (3) have unknown addresses. A similar mechanism in Aglets, called *Agent Boxes*, is described in [AO98]. The "waiting room" can provide a secure mechanism for accepting agents inside firewalls and security domains (since agents are first authenticated and then pulled out from the "waiting rooms", which are located outside the firewalls or domains; the agents in the "waiting room" queue are inactive and thus malicious agents cannot harm the server). The mechanism can also facilitate managing agents (e.g. agents whose lifetime expired might have to move back to the "waiting room" and wait there until the client collects results and/or pays for the service continuation). Eventually, the "waiting room" may coordinate sharing and distribution of agents (e.g. to improve load balancing).

Encodings In the infrastructure translation, the primitive **`migrate`** is encoded as low-level migrations between the client site, "waiting room", and resource sites. Communication with mobile agents is forwarded through a daemon, located in the "waiting room", which will know current agent locations. The daemon maintains the queue of agents sitting in the "waiting room" and manages the extraction of agents from the "waiting room".

The naming scheme is as follows. Before creating mobile monitors, we look up the name (say D) of the database or resource to be monitored. This name should allow resolving the address of the daemon dedicated for this particular database or resource (e.g. using a local trader mechanism of the agent system). Mobile agents are created dynamically and their names, used for communicating with the agents, would contain either D or simply the daemon address resolved from D.

5.2 Mobile Devices

The use of mobile agents makes a lot of sense in the case of applications launched from mobile devices, as described below. Mobile devices such as laptops, notebooks and personal communicators are only intermittently connected to a network, hence they have only intermittent access to a server. The connection with a server can often be via low-bandwidth, high-latency, high-cost connections (wireless or dial-up links). Applications running on mobile devices can react to a drop in network bandwidth by moving network-intensive computations to a well-connected proxy site. Example applications include remote access to data repositories. The client can develop an agent request — possibly while disconnected — launch the agent during a brief connection session, and then immediately disconnect. The response, if any, is collected during a subsequent connect session from a proxy client left at the edge of the network. The user could spawn mobile agents from thin clients like Personal Digital Assistants (PDAs) and cellular phones, and be notified about certain events through the phone or e-mail (e.g. in [JJ99], a weather alarm application is described, where the alarm conditions can be programmed on a mobile phone, using a simple scripting language, and sent to a remote server which monitors the weather; if the alarm is triggered, a notification is sent to the client).

5.2.1 Example Infrastructure

In [GKN+96], a *docking* system has been described, which allows an agent to transparently move between mobile computers, regardless of when the computers connect to the network. Below, we describe a similar infrastructure to support disconnected operation, however, with less transparency. The "waiting room" described in §5.1.2 will serve as a docking station for agents who want to migrate from a stable part of the network to a mobile computer. However, communication in the opposite direction remains under application control. We focus on the semantics of location-independent communication between clients and monitoring agents.

Communication from a stable part of the network to clients executing on mobile computers is as follows. The high-level location-independent output would be encoded as either sending a message (or the agent carrying the message) directly to the client if it is connected, or otherwise, sending the message (or agent) to the "waiting room" (and possibly saving their state on disk), so that it can be fetched by the client upon reconnection.

Communication initiated from mobile devices requires a bit more consideration. It has to be decided whether connection and reconnection should be made transparent to the application (and the user) or not. For example, the CODA distributed file system [Kis93] supports transparent access to remote files from mobile clients, and the client performing some operations (such as writing to a remote file) may not be aware of disconnection — the operation is serviced by a local server emulator which will complete the operation when the client machine reconnects. On the other hand, in some applications the client must be aware of any changes between the "connect" and "disconnect" states, so that it can react accordingly (e.g. informing the user who wants to read e-mail from a mobile phone that it is not possible at that time because of some problems with connecting the mobile to the network, etc.; the user may not wish to see the e-mail pop up later when the connection is resumed).

There is a range of choices for the infrastructure support of location-independent communication from mobile devices. It would be good if we have support for two kinds of disconnection "triggers": an *intentional* or *software* trigger (such as switching a mobile device to the "power save" mode), and *inadvertent* (such as connection failure, e.g. when a physical obstruction blocks the signal from a cellular modem). The software trigger can be delayed, but the latter is out of our control. We might try our best to deliver a message if the "trigger" can be delayed. In order to delay the software "trigger", the underlying operating system should support some mechanism for blocking disconnection (at least for some time), e.g. using semaphores. Then, the encoding of the location-independent message output might use this mechanism and block system-controlled disconnection until the message is actually sent off. Otherwise, if message sending fails (after a few attempts to deliver the message), some exception would be generated to be handled by the application. All messages which are not delivered would either be discarded by the application or queued to try again (depending on the application logic).

In the docking system of [GKN+96], a fully transparent solution has been adopted, where all agents which did not succeed in being sent off from the mobile device are silently stored on disk under the control of the system dock-master. The dock-master continually monitors the network status and will attempt to transfer all waiting agents, e.g. during a brief reconnection period. It is assumed that the mobile device will eventually reconnect. In our opinion, however, complete transparency may not be good for some applications (e.g. we might want to know if and when eventually the agents are sent off to the network, and be able to cancel all pending communications, e.g. if the application is about to terminate). Thus, there should be the possibility of checking and deleting agents from the queue maintained by the dock-master.

5.3 Information Retrieval

Mobile agents can be used in data-mining and retrieving context-based information from multiple resources (such as data repositories and the World-Wide Web). In [OPL94], Oates et al. discuss benefits of viewing information gathering as distributed problem solving (which subsumes distributed processing). The approach, called Cooperative Information Gathering (CIG), involves concurrent and asynchronous access and composition of associated information spread across a network of information servers by a group of intelligent agents. Top level queries drive the creation of partially elaborated information gathering plans, resulting in the employment of multiple cooperative agents for the purpose of achieving goals and subgoals within these plans. For example, intelligent agents of the Vacation Planner search multiple databases – weather, car rental, hotel and "places of interest" – to plan an appropriate vacation for the user. Each agent searches its assigned resource independently but uses partial results from the other agents to adjust its search criteria when needed. For example, the place agent might assume good weather initially, but then redo portions of its search when the weather agent tells it that bad weather is forecast for a particular area. Eventually the agents arrive at a consistent vacation plan.

The potential advantage of using mobile agents for information retrieval is the generic interface which they provide, and the possibility of saving network bandwidth (which is important if the application is launched from a mobile device). In the client-server approach, the client may sometimes have to download large volumes of data from many data repositories in order to perform some more customized filtering locally. Mobile agents can perform filtering at the repository which is optimal for the query. For example, let us suppose we want to apply a query x to data retrieved from databases A and B as a result of performing two queries, respectively q_1 at A and q_2 at B (neither interface to A nor to B supports query x). If one of the queries q_1 and q_2 (say q_1) returns a very large volume of data, we could save the network bandwidth as follows. First, we send two agents to A and B where they locally collect results of the queries q_1 and q_2. Then, they communicate and agree that the agent carrying the results of query q_2 (as lower in quantity) could migrate to database A, where the query x would be performed on both sets of retrieved data and the final result sent back to the client.

The example showed the advantage of using mobile agents over traditional SQL-based interfaces. In database systems, we can specify an arbitrary query, expressed as an SQL program, executed as a transaction which is distributed and highly optimised. However, we cannot easily access data that are not maintained by SQL-servers. Moreover, the agents can migrate or

clone themselves on new sites and repeat querying, without necessarily occupying resources on sites previously visited (they also do not require the client, who launched the query, to be connected). The flexibility offered by mobile agent interface seems a valuable feature on the Internet, where e-commerce applications may span many domains which are administered or managed separately. A typical e-commerce transaction may require access to different data repositories which almost certainly do not support the same interface (e.g. SQL). Mobile agents can retrieve various data (e.g. represented as Web documents, objects, or relational tuples, etc.) and filter them together in a common context. Plausible actions include: notifying the client about any progress in searching, user-customized on-the-fly generation of Web links, etc. Below, we describe a simple example of the net search application and propose a good infrastructure for it.

5.3.1 Migration and Communication Pattern

A client enters a query which is semantically reformulated, so as to identify the domain and goals of searching. The *yellow page* directory or net search engines are used to identify target servers for performing information gathering locally. We assume that the target servers advertise themselves somehow, specifying the interface and access control. A first group of mobile agents is dispatched to target servers, where the agents cooperatively gather information, asking the client for additional attributes of the search process, getting some feedback on the quality of partial data retrieved, and dynamically dispatching themselves to new target servers if necessary (in practice, probably not more than once or twice).

It is assumed that communication is frequent and is mostly between the client and agents, or between adjacent agents (e.g. a parent and child). The volume of data depends on who is the sender. The client may send short messages (e.g. queries) to mobile search agents, but the agents may send back large portions of data to the client.

5.3.2 Example Infrastructure

Agents clone themselves (i.e. create child agents with the same or similar functionality). The clones migrate to new sites carrying partial results if necessary, the spawning agents remain on their sites. So, the style of agent migration is *one-hop*. The agents form a tree with the client as a root of the tree. The tree-like pattern of agent creation and communication suggest the Hierarchical Directory infrastructure (such as in §4.1.5). However, since

there are few agents involved and migrations are rare, the Home Server infrastructure (as in §4.1.1) would be good enough. We can combine these two techniques, as below. Since agents migrate only once, server daemons are not necessary and the infrastructure can be more light-weight.

The agents spawned on new sites must bind to local resources. The search agents look up local resource names using a local name service (the name of the local name server, or trader, would be known on each runtime system, so that agents landing on a new site can easily invoke this server and ask, e.g. for the interface to the local database). The agents expire after the search is completed (a termination message may need to be sent by the client, or they just expire after some sufficiently long timeout).

Messages from the client (root) to the search agents might be forwarded along the branches of the tree and anonymously broadcast to all clones in the branch (if required). Or, the client might want to contact only the current leaves of the tree. Then, the message is distributed in the tree as before but ignored by clones which are not leaves. Thus, agents can serve as active forwarding proxies, forwarding messages to their clones if required. Finally, a search agent might be contacted directly by the client if the client knows the agent name. The client is assumed not to migrate, otherwise, "waiting rooms" (described before) or proxy clients might be used. The proxy client would forward to the client all messages received from search agents.

Encodings The infrastructure should support creation of two kinds of agents: stationary agents which never migrate, and *one-hop* agents which can migrate only once. For convenience, we may introduce a new high-level language construct for creating one-hop agents. The execution of `spawn a s = P in Q` would create a new agent `a` on the current site, with body `P` (using **`agent`**), and immediately spawn the agent on site `s` (using **`migrate`**). After migration, `Q` commences execution, in parallel with the rest of the body of the spawning agent. An agent name `a` is binding in `P` and `Q`. The agent name `a` would be encoded by a pair of the actual agent name and the address of `s`.

Agents in a user program communicate using location-independent messages. The encoding is as follows. An agent `a` which wants to send a message to `b` first tries to send the message locally, if this fails then the message is sent directly to the site recorded in `b`'s name. Since agents cannot migrate once they are spawned, the message will be delivered.

The advantage of such an infrastructure is good scalability and good interoperability (since the infrastructure is deployed dynamically and does not depend on any global service for tracking locations). Agents of the same application manage their locations by themselves, forwarding messages from

other agents if required (no daemons necessary). The disadvantage is that agents can only migrate once. Some higher order communication mechanisms can be built on top of forwarding agents like active tuplespace (such as proposed in [CLZ98]). Agents can record some context-based data on sites they visit and enable incoming agents (of the same application) to retrieve these data.

5.4 Fault-Tolerance

The semantics of application operations (e.g. time related), may have a great impact on the design of infrastructure and tolerance of system failures. For example, the information gathering process is idempotent — if something goes wrong, we can interrupt the process (losing partial results) and start it again. The result of a new search might be more accurate or not, but usually it does not matter (since the client cannot observe any difference). However, if the computation is long-running we may want to occasionally store partial results and agents to a non-volatile store and recover the state of computation after failure when the system is restarted. On the other hand, the monitoring process requires real-time, continuous access to the resources monitored. It may not be desirable to stop this long-running process in the middle (e.g. selling and buying shares would depend on the fluctuations of the stock market; the stock agents are expected to react at the right moment). The resource servers have to be replicated in order to provide high availability in spite of failures. Different levels of fault tolerance are therefore required to satisfy the different semantics.

A simple method to achieve fault-tolerance in a distributed system is through process checkpointing, message logging, and recovery after failures. Checkpointing means recording the process state from time to time on a stable store. After a failure occurs and the machine is restarted, we recover the process from this state (called the process *checkpoint*). Processes must restart computation in a *consistent state*. That is, the restarting state of one process should not causally depend on the restarting state of another process. The problem is to ensure that after distributed recovery, there will be no *orphan* messages observed (i.e. messages which are received without having been sent), and no messages that have been sent but not delivered. A way to handle these problems is to roll back the execution of the processes until a consistent global checkpoint is found.

A recovery algorithm based on repeatedly rolling back processes to earlier checkpoints may lead to a number of problems that have to be sorted out by the algorithm. *Cascading rollbacks* may lead to a total failure when the only

consistent state is the initial state. A computation that is made fault resilient by checkpointing and recovery might exhibit *stuttering*, i.e. producing the same output or asking for the same input twice. Messages in transit from failed or rolled-back processes cause a problem of *message duplication* (see, e.g. [CJ97] for references).

The method of distributed checkpointing and recovery requires that the processes executing on each site have access to stable storage on that site and if the site fails, it will be restarted. Below, we present a solution where we do not require all machines to have access to stable storage; host machines are chosen from a pool of available machines and can fail-stop (except those doing checkpoints, which must be recoverable).

5.4.1 Mobile Agent Support for Checkpointing

Our motivation is that mobile agents could reduce the dependency on some host machines and network connections, by moving computation around if necessary. For example, processes that execute on unreliable machines or computers connected ad-hoc by wireless connections, might want to leave these machines at some critical points of execution (e.g. a period of frequent communication), and move the distributed computation state to one machine (called a "meeting place"), where they could exchange messages locally, with no need for communication over the network. After the communication is finished, processes could migrate back to their host sites, and continue computation and interaction with the users. If the "meeting place" site is reliable and allows process checkpointing but the other sites are not reliable and do not allow access to a disk store, the "meeting place" may facilitate the building of fault-tolerant applications as described below.

We define a *communication transaction* in process A to be finite computation during which A will frequently output or input messages (e.g. the application code of some finite negotiation protocol might be defined as a communication transaction). Our transaction is local to the process where it has been defined — in particular, the communication transaction T defined in agent A is independent of any transactions defined in processes communicating with A.

Processes of the communication transaction that execute on unreliable host machines could migrate to the "meeting place", checkpoint their state, execute the communication transaction, checkpoint again and migrate back to their host machines (or some other machine if the host machine failed). If a host machine fails, we recover processes which have been executing on this machine from their last checkpoints stored at the "meeting place". Since this checkpoint is guaranteed to be consistent (processes on the failed machine

did not communicate over network), there is no need to roll back processes executing on other machines. Thus, we can avoid problems of distributed checkpointing and recovery because the most critical part of the algorithm is localised (i.e. the consistent global checkpoint is found locally). Moreover, traditional techniques of checkpointing and recovery would be of no use, since we assumed that some host machines do not have access to stable storage (therefore, with every failure we would have to restart computation from the beginning).

Below, we consider an example of a large-scale parallel computation where mobility allows for the dynamic deployment of the application. Then, we propose a suitable infrastructure for it, which tolerates machine and link failures. Another similar application of mobility is dynamic relocation of computation from a site which is about to shut down.

5.5 Large-Scale Parallel Computation

We have long-running, parallel computation spawned on a large number of host machines in a wide-area network. Parallel computation can utilize the CPU power of many single computers and thus improve the overall performance. Examples of such computations are large-scale scientific computations built on top of MPI (the Message-Passing Interface standard) or PVM (Parallel Virtual Machine).

5.5.1 Migration and Communication Pattern

Concurrent processes of the computation may occasionally want to communicate (e.g. exchange partial results). Communication is infrequent. The host computers are generally unreliable. They may fail, be switched off, or rebooted at any time. Processes executing on a machine which failed are lost. The computation is a repeatable process and it does not necessarily require specific resources (i.e. host machines can be chosen from a pool of free machines). However, it would be undesirable to have to repeat the computation from the beginning, each time some failure occurred. It may take a long time to complete the computation. Therefore, we should ensure that distributed computation will make progress in spite of machine failures and broken links. Since the host computers do not have access to stable storage, standard methods of checkpointing and recovery cannot be used. Due to a very large number of computers and processes, other techniques, such as process replication on distinct machines, may not be practically useful nor efficient in wide-area networks.

5.5.2 Example Infrastructure

Let the computation be deployed as a collection of mobile agents, each executing a number of concurrent processes. The agents are spawned from a machine which is assumed to be reliable and having access to a stable store (we call this site a "meeting place"). A star-like pattern of migration suggests a centralised infrastructure, such as the Central Server infrastructure described in §4.1.1, where the server would be a meeting place. Below we first describe the general idea of the infrastructure, then the encoding of agent creation and location-independent output. We assume there is a daemon agent in the "meeting place" which knows current agent locations and is responsible for agent checkpointing and recovery.

If agent A executing on a remote site wants to communicate with some agent B, it will first migrate to the "meeting place" and call for B. The agent B must suspend execution at some point and migrate to the "meeting place" (unless it is already there), and then agents A and B can perform the communication transaction locally. The application programmer should ensure that there would not be any frantic migrations forth and back. They can do that by grouping frequent input and output operations into a single programming block — the communication transaction. For example, a given process might execute the protocol for exchanging partial results as one communication transaction. Mobile agents involved in the transaction will be called up to the "meeting place" in a lazy way (i.e. when they are needed). After the transaction has completed, the agent would migrate to the remote site, which would be either the previous site, or a site chosen from a pool of available non-failed sites (e.g. if the previous site failed). If we were to adopt some scheduling policy of choosing sites, it would even be possible to balance the workload on host machines by scheduling agent migrations to sites which are free (or less loaded).

Encodings The application is initially spawned from the "meeting place" — a site which is assumed to be reliable. Agents are created dynamically whenever required. However, each time a new agent is created it has to be correctly checkpointed. Therefore, the encoding of **`agent`** would require the spawning agent to first migrate to the "meeting place" (if not there), where a new child agent is created, registered at the daemon, and checkpointed. Then, both agents can migrate back to the remote site.

The application programmer uses location-independent messages for communication, and two high-level language operations **`do`** `"start"` and **`do`** `"commit and migrate to" s` for expressing a communication transaction, encoded as follows. The execution of **`do`** `"start"` as part of an agent results

in the whole agent migrating to a "meeting place". The execution of **do** `"commit and migrate to" s` results in checkpointing the whole agent state on the current site and migration of the agent to site `s`, or some arbitrary site if none is specified. The location-independent output is encoded as follows. We first migrate to the "meeting place" (unless we were already there, e.g. performing **do** `"start"` earlier). Then, we send locally. If this fails, the "meeting place" daemon calls for the message recipient to migrate and then the output can be finished locally. After the message has been delivered, the sender and receiver either migrate to their previous sites, or if the output or input is part of some transaction, the agent executing it will continue at the "meeting place" until **do** `"commit and migrate to" s` is performed. On departure from the "meeting place" site, agents leaving the site are always checkpointed.

If a remote site or agent on that site has been suspected of failure (e.g. as a result of calling for the agent), all agents on the suspected site (or only the affected one) are recovered from the last checkpoint stored at the "meeting place" and executed locally or sent to a free site (or to the same site after it recovers). Agents are given incarnation numbers which are incremented each time the agent is recovered from the last checkpoint. Any agents arriving to the "meeting place" from a site which has been incorrectly suspected of failure, will be first examined by a daemon and discarded if some new incarnations of these agents (with higher numbers) have already been recovered from local checkpoints. The daemon will know all agent locations and incarnation numbers.

5.6 Event-Driven Mobility

In this section, we describe two applications of event-driven mobility: multimedia and collaborative work (CSCW) in the local-area network, and videoconferencing and chat applications in the wide-area network. In the first application, mobility results from physical movements of people, in the second one, mobility is a tool for adapting to variations in network characteristics. An event here is an asynchronous message or signal, which may trigger some actions.

In a teleporting system [RBM+94], a user application interface follows the user. The current physical location of the user is monitored by an active badge location system [HH94]. Each user wears a badge which periodically transmits a unique infra-red signal. A network of detectors allows the physical location of the user to be detected. When a user is near some machine and clicks a button on their active badge, their current session shows up on the

new machine. The teleporting is based on the X-windows system. Only the user interface follows a user; applications do not themselves migrate. Bacon, Bates and Halls [BBH97], extend the idea and describe using an event system for multimedia and CSCW applications, which are able to move around a network, remapping user interfaces and stream-based endpoints like cameras, microphones, and speakers to the user's current location. In general, applications could use some event infrastructure and register their interest in some events. The infrastructure would notify its clients of any events which match their interests. The clients may then trigger some actions, such as dispatching code on the machine which is near the user, or summoning mobile agents to a new site. The advantage of using mobility is a dynamic deployment of the application on new machines, possibly with automatic redirection of communications.

A different approach to using event-driven mobility is represented in [BPR98], where code mobility is used in a videoconference system as a means to achieve better system customization. It is achieved by: (1) enabling users to dynamically upload the conference server with code describing some customized processing (e.g. coding algorithms, QoS policy, etc.), (2) enabling the server to enhance its performance through server migration or cloning on a different site, triggered by changes in the network conditions. The videoconference server is an example of *network-aware* mobile programs, i.e. programs that can use mobility as a tool to adapt to variations in network characteristics. Mobility allows applications to recover from a poor initial placement of some data-structures (services, etc.) used by the application by repositioning it to a more suitable location. The advantage of a mobility-based strategy over replicating these structures at all suitable points in the network, is that it allows services used for a short time only to be set up without requiring extensive server placement.

Network-awareness inspired several applications, e.g. a chat server [RASS97], capable of enhancing its performance through server code migration. It allows multiple users to have an online conversation. To ensure that all participants see the same conversation and that new participants can dynamically join ongoing conversations, a central server is used to serialise and broadcast the contributions. The desired behaviour of the chat application is to provide a rapid response time to all participants so that a conversation can make quick progress. The response time for a particular participant depends on the latency between it and the central server. The application can take advantage of the mobility support to place the chat server so as to minimise the maximum response time seen by any participant. It is sensitive to changes in the latency of a link over a period of time, so that the server can change its location during conversations. Network-awareness also provided the rationale for exploiting *active networks* [TSS+97].

5.6.1 Mobile Agent Support for Events

The event system must provide support for event registration and notification. Before clients can receive event notification, they should first register with the *event manager* by sending the manager a list of types of all events which they want to be notified about, together with the location information about where the clients wish the notifications to be sent. The *event manager* filters each event it receives by notifying only those clients that have registered to receive events of that type. This mechanism can be extended to *group-oriented* events, i.e. events that are reliably sent to a group of clients. The clients forming the group have to register with the *event manager*.

One could imagine that mobile agents are clients who register at the event manager and are notified about events. For example, Concordia [WPWD97] supports events and event groups to enable mobile agents to collaborate. The system offers two kinds of event groups for collaboration: basic and persistent (in the latter, group membership survives site failures, and reliable, transparent recovery from failures via proxy objects is provided; the details are not given). In our model, we could build the event infrastructure just above the infrastructure supporting location-independent messages. Then, the event manager may use location-independent messages for sending event notifications to mobile agents. For event groups, the event manager should reliably broadcast events to all agents that form a group[1].

In some applications, mobile agents may only be interested in receiving certain events while executing on a given site. After moving to a new site, the events from previous locations are irrelevant, and the agents might want to register again at the *local* event manager. In that case we can think of infrastructure, where a mobile agent registering with the event manager specifies the location where the event notifications should be sent (usually it would be the agent's current site). The event manager can now use location-dependent communication for sending event notifications to the agent. If the agent migrated away, the event system would not make any attempt to deliver the event; the event would be discarded and the agent silently unregistered from the (local) event system.

[1]The *reliable broadcast* (see, e.g. [HT94]) guarantees three properties: (1) all non-faulty processes agree on the set of events they deliver, (2) all events broadcast by non-faulty processes are delivered, and (3) no spurious events are ever delivered. While these properties may suffice for many applications, sometimes the order in which events are delivered is important (e.g. first-in-first-out, causal, and total order broadcasts would be required).

5.6.2 Migration and Communication Pattern

Network-aware applications, such as chat and videoconference servers, could dynamically spawn monitoring agents to predefined sites, where the agents would register at the local resource monitors. These agents would be informed about variations in local network characteristics by receiving events from the local resource monitors. If performance conditions, e.g. network latency from the clients, were not satisfactory, the chat or videoconference servers might interrogate the monitoring agents and migrate to a different site, which received more than a threshold score. The server would use location-dependent communication to monitoring agents (they do not migrate a second time). The clients would use location-independent communication to communicate with the network-aware server as it migrated.

5.6.3 Example Infrastructure

The monitoring agents migrate to predefined sites where they register at the local resource managers. The address of the manager daemon would be obtained from a local trader, maintained by the agent system (all public names, such as local daemon names are assumed to be registered in the local trader when the daemon starts up). The event registration message would contain the agent current location (this site should not change — in the case of site failures, the application server would have to spawn the monitoring agent again). The monitoring agents collect events about the local network characteristics and compute a summary for the application server.

The infrastructure for location-independent communication between the server and monitoring agents can be such as described in §5.1. Also, it would be convenient to abstract away from the location information when expressing the application protocol of the communication between the server and clients (i.e. participants of the videoconference or Internet chat). Note, that some infrastructure algorithms, such as forwarding pointers, are clearly not appropriate here, because they would still depend on congested sites and links. Instead, the server might synchronise their moves with all the clients (as in the Group Communication infrastructure, described in §4.1.4). Since migration is infrequent, the infrastructure should perform well. Similarly, the client could synchronise with the server while migrating (it does not need to synchronise with other counterparts since we assume that the architecture is centralised and the whole communication flow between the conference or chat participants is forwarded through the server).

It also has to be decided how new users could register at the server and join the conference. The server should either leave forwarding pointers from some initial site (forwarding messages along a chain of sites is acceptable here since registration takes place only once), or in some other way make it possible for the new participants to know its current location.

Chapter 6

The PA Application and Infrastructure Design

In this chapter we discuss a small example application, the Personal Assistant (PA), and the design of an infrastructure suited to it in more detail (a preliminary discussion appeared in [WS99]). The focus is on demonstrating the benefits of a multi-level architecture based on clearly defined levels of abstraction; we have therefore chosen a somewhat idealised example application. The required infrastructure is still far from trivial, however. Expressing it as a Nomadic Pict translation allows us to include an almost complete executable description, making the details of concurrency, synchronisation and distribution clear and precise. By considering the migration and communication patterns of the application we can argue that this infrastructure algorithm is a practicable choice, whereas many others, including those in chapter 4, would not be.

We begin with a simple centralised algorithm in §6.2.1, which is further extended to provide support for disconnected operation (§6.2.2). Then, we extend the original algorithm of §6.2.1 to obtain a scalable infrastructure (§6.2.3). Finally, we discuss informally how to merge the algorithm of §6.2.2 with the scalable architecture described in §6.2.3, so as to enable mobile computing in wide-area and ad-hoc networks.

In order to give a feeling of "mobile agent world", we have implemented the Mobile-Chat-Room application, and asked people in different offices to try our demo. The Mobile-Chat-Room uses a somewhat simplified idea of the PA application, where we have only a single PA agent, which stores the whole state of people's conversation and can migrate on demand. The users can summon the PA agent to their site using a summoner agent. The static summoners (one per site) are dynamically spawned on all sites when the application starts. The migrating agent carries the history log of the messages

typed by users on different machines. We tested the program on different machines connected to the Computer Laboratory network; different architectures were involved (Alpha and ix86). The application communicates with the user by an X-windows interface, implemented using a simple graphical library of Nomadic Pict.

6.1 Application

We consider the support of collaborations within (say) a large computer science department, spread over several buildings. Most individuals will be involved in a few collaborations, each of 2–10 people. Individuals move frequently between offices, labs and public spaces; impromptu working meetings may develop anywhere. Individuals may also travel to other institutions, and continue collaboration from there over a wide-area network, as well as being involved in any new collaborations locally. They would therefore like to be able to summon their working state (which may be complex, consisting of editors, file browsers, tests-in-progress etc.) to any machine. These transfers should preserve any communications that they are engaged in, for example audio/video links with other members of the project. It should also be possible to summon the working state to a mobile computer, work in a disconnected mode, and later reconnect to the network (possibly in an other institution), and complete all pending communications.

To achieve this, the user's working state can be encapsulated in a mobile agent, an electronic *personal assistant* (PA), that can migrate on demand.

6.1.1 High-Level Architecture

We implement the PA application in High-Level Nomadic Pict with three classes of agents: the PAs themselves, which migrate from site to site; *summoner* agents, which are static (one per site) and are used to call the PAs; and *name server* agents, also static, which maintain a lookup table from the textual keys of PAs to their internal agent names. They interact using location-independent communication on channel names.

```
registPA : ^[ String Agent ]          moveOn   : ^Site
summonPA : ^[ String Agent Site ]     notFound : ^[]
mid      : ^String
```

A sample PA is below. It has 4 parallel components; a registration message, a message sent to another PA, a replicated input that receives data from other PAs and prints it, and a replicated input that receives migration commands and executes them.

```
agent PA1 =
  ( registPA @ NameServer ! ["pawelsPA" PA1]
  | mid @ PA2 ! "Outgoing data stream"
  | mid ?* d = print!(+$ "Incoming data:" d )
  | moveOn ?* s =
     ( migrate to s (print!"Hello Pawel! Your PA has arrived..."))
```

For simplicity, we assume a single name server. The name server below maintains a map from strings to agent names; it receives new mappings on `registPA`. The map is stored as an output on the internal channel `names`. Summon requests are received on `summonPA`, containing a textual key and the name/site of the summoner. If the key has been registered the name server sends a migration command to the corresponding PA agent, otherwise it sends an error message to the summoner.

```
agent NameServer =
  new names : ^(Map String Agent)
  ( names ! (Map.make ==)
  | registPA ?* [descr PA] = names ? m = names!(map.add m descr PA)
  | summonPA ?* [descr Su s] = names ? m =
                  (switch (map.lookup m descr) of (
                     {Found> PA : Agent} -> moveOn @ PA ! s
                     {NotFound> _} -> notFound @ Su ! [])
                  | names!m))
```

The summoner at site `s` is as below. It gets strings from the local console, sending them as requests to the name server.

```
agent Summoner =
   val PAkey = (sys.read_line [])
   ( summonPA @ NameServer![PAkey Summoner s]
   | notFound?_= print!(+$ PAkey " not found!"))
```

In the actual implementation the top-level encoding launches summoners dynamically, using the standard migration primitive, onto the list of active sites. For simplicity the implementation uses location-independent communication throughout, despite the fact that the name server and summoners are static.

6.1.2 Migration and Communication Pattern

A usable infrastructure for the PA application can only be designed in the context of detailed assumptions, both about the system properties and about the expected behaviour of the high-level agents.

For the former, in the first stage we assume that the application is running over a large LAN, in which reliable messaging can be provided by lower-level

protocols[1] and all machines are at roughly the same communication cost distance from each other. Machines are also basically reliable, although from time to time it is necessary to reboot or turn off. The LAN is under a single management, with no internal firewalls.

As for the assumptions about the expected behaviour of the high-level agents, we suppose that the number of PA agents is of the same order as the number of people in the lab. Each PA will migrate infrequently, with minutes or hours between migrations. The path of migrations is unpredictable — it may range over the whole LAN. The migrations of different PAs are essentially uncorrelated in time. It is common for people to work for extended periods at machines out of their offices. PAs communicate between each other frequently, with significant bandwidth — e.g. audio/video messages or streams, and other data (that must be delivered reliably).

These assumptions are not wholly appropriate — the application also demands disconnected operation (on laptops) and a higher level of fault-tolerance. Therefore, in the second stage we also assume that PAs can migrate to laptop computers. A user can disconnect the computer from the network, work in a disconnected mode for extended periods, and later reconnect in the same or other network domain. All messages that cannot be delivered to a laptop or sent out from the laptop due to disconnection will be transparently delivered upon reconnection. A migration from a disconnected computer fails, causing an exception in the high-level program. In the last stage, we assume migration and communication in a wide-area network, so the system should scale well. We discuss infrastructure design addressing the problems of disconnected operation and scalability in §6.2.2 and §6.2.3, but for the sake of a clear example infrastructure we neglect them for now.

6.2 Design of Appropriate Infrastructure

We develop our infrastructure in several steps, beginning with the two extremely simple algorithms described precisely in §4.2. The *Central Server* algorithm has a single server that records the current site of every agent; agents synchronise with the server before and after migrations; application (location-independent) messages are sent via the server. The *Forwarding Pointers* algorithm has a daemon on each site; when an agent migrates away it leaves a pointer to the site that it is going to (and the daemon there). Application messages are delivered by the daemons, following the pointers.

[1] We do not deal with the unreliable case here so as not to complicate the encoding too much, however a simple algorithm for disconnected operation, described in 6.2.2, can tolerate some message losses; it should give the feel of this style of working.

Neither of these algorithms suffice for the PA application. The central server is a bottleneck for all inter-PA communication; further, all application messages must make two hops (and these messages make up the main source of network load). The forwarding pointers algorithm removes the bottleneck, but there application messages may have to make many hops, even in the common case.

Adapting the Central Server so as to reduce the number of application-message hops required, we have the *Query Server* algorithm. As before, it has a server that records the current site of every agent, and agents synchronise with it on migration. In addition, each site has a daemon. An application message is sent to the daemon which then queries the server to discover the site of the target agent; the message is then sent to the daemon on the target site. If the agent has migrated away, the message is returned to the original daemon to try again. In the common case application messages will here take only one hop. The obvious defect is the large number of control messages between daemons and the server; to reduce these each site's daemon can maintain a cache of location data.

The *Query Server with Caching* does this. When a daemon receives a mis-delivered message, for an agent that has left its site, the message is forwarded to the server. The server both forwards the message on to the agent's current site and sends a cache-update message to the originating daemon. In the common case application messages are therefore delivered in only one hop.

This may seem well-suited to the PA application, but the textual description omits many critical points — it does not unambiguously identify a single algorithm. To do so, and to develop reasonable confidence in its correctness and performance, a more precise description is required, ideally in an executable form. We give such a description, as a Nomadic Pict encoding, in §6.2.1.

These algorithms clearly explore only a part of the design space — one can envisage e.g. splitting the servers into many parts (one dealing with agents created for each user), forwarding pointers in which long chains are collapsed, and server-less algorithms in which the agents of a collaborative group synchronise among themselves. An exhaustive discussion is beyond the scope of this dissertation. One can also analyse the application further — in fact, the migrations of each user's PA may usually be within a small group of machines, e.g. those of a research group. More sophisticated infrastructures might use some heuristics to take advantage of this. For a critical application a quantitative analysis may be required.

A closely related application for multimedia CSCW is described in [BHB97], implemented (with real video support) using the *Tube* Mobile

Agent System. A low-level multimedia stream library was used; streams were reconnected on movement at the application level. Moving this into the infrastructure would involve synchronisation between the source and all sinks of a stream on any migration.

6.2.1 Example Infrastructure: The QSC Algorithm

In this section we describe the Query Server with Caching (QSC) algorithm as a Nomadic Pict encoding, thereby making all the details of concurrency and synchronisation precise.

An encoding consists of three parts, a top-level translation (applied to whole programs), an auxiliary compositional translation $[\![\texttt{P}]\!]$ of subprograms `P`, defined phrase-by-phrase, and an encoding of types. The QSC encoding involves three main classes of agent: the query server `Q` itself (on a single site), the daemons (one on each site), and the translations of high-level application agents (which may migrate). The top-level translation of a program `P` launches the query server and all the daemons before executing $[\![\texttt{P}]\!]$. The query server, and the code which launches daemons (which is assumed to be part of agent `toplevel` on site `firstSite`), are given in Figure 6.1; the interesting clauses of the compositional translation are in the text below.

The messages sent between agents fall into three groups, implementing high-level agent creation, agent migration, and location-independent messages. Typical executions are illustrated in Figure 6.2 and below.

Each class of agent maintains some explicit state as an output on a lock channel. The query server maintains a map from each agent name to the site (and daemon) where the agent is currently located. This is kept accurate when agents are created or migrate. Each daemon maintains a map from some agent names to the site (and daemon) that they guess the agent is located at. This is updated only when a message delivery fails. The encoding of each high-level agent records its current site (and daemon).

To send a location-independent message the translation of a high-level agent simply asks the local daemon to send it. The compositional translation of `c@b!v`, 'send `v` to channel `c` in agent `b`', is below.

$$[\![\texttt{c @ b ! v}]\!]_{\texttt{[a Q SQ]}} \stackrel{def}{=}$$

```
currentloc?[S DS]=
   iflocal <DS>try_message![b c v] then
        currentloc![S DS]
   else ()
```

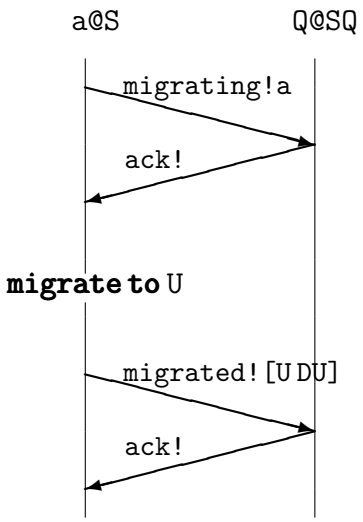
This first reads the name `S` of the current site and the name `DS` of the local daemon from the agent's lock channel `currentloc`, then sends `[b c v]` on the channel `try_message` to `DS`, replacing the lock after the message is sent.

Once the lock is released, the agent is free and may, for example, migrate away while the message is delivered. The translation is parametric on the triple `[a Q SQ]` of the name `a` of the agent containing this phrase, the name `Q` of the query server, and the site `SQ` of the query server — for this phrase, none of those is used. We return later to the process of delivery of the message.

To migrate while keeping the query server's map accurate, the translation of a **migrate** in a high-level agent synchronises with the query server before and after actually migrating, with `migrating`, `migrated`, and `ack` messages.

```
⟦migrate to u P⟧[a Q SQ] =def
   currentloc?[S DS]=
     val [U DU] = u
     ( <Q @ SQ>migrating!a
     | ack?_ = migrate to U
          ( <Q @ SQ>migrated![U DU]
          | ack?_ = ( currentloc![U DU]
                    | ⟦P⟧[a Q SQ])))
```

A sample execution is below.



The query server's lock is kept during the migration. The agent's own record of its current site and daemon must also be updated with the new data `[U DU]` when the agent's lock is released. Note that in the body of the encoding the name `DU` of the daemon on the target site must be available. This is achieved by encoding site names in the high-level program by pairs of a site name and the associated daemon name; there is a translation of types

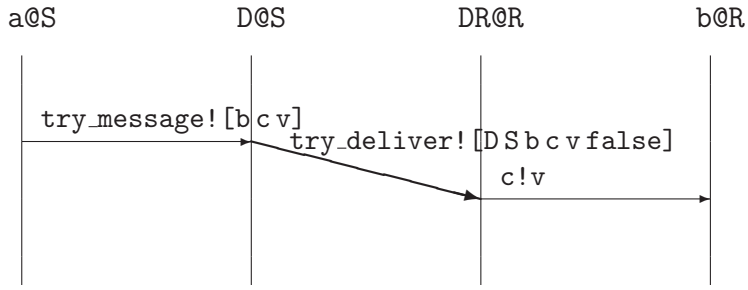
```
⟦Agent⟧ =def Agent
⟦Site⟧  =def [Site Agent]
```

```
agent Q =                          (*  the query server  *)
  new lock : ^(Map Agent [Site Agent])
  (lock!(map.make ==)              (*  initialise the lock  *)
  | register?*[a [S DS]]=          (*  register a new agent  *)
        lock?m=
          ( lock!(map.add m a [S DS])
          | <a@S>ack![])
  | migrating?*a=                  (*  lock during a migration  *)
        lock?m= switch (map.lookup m a) of (
          {Found> [ S:Site DS:Agent ]} ->
             ( <a@S>ack![]
             | migrated?[S' DS'] =
               ( lock!(map.add m a [S' DS'])
               | <a@S'>ack![]))
          {NotFound> _} -> ())
  | message?*[#X DU U a:Agent c:^X v:X]= (* deal with a lost msg *)
        lock?m= switch (map.lookup m a) of (
          {Found> [R : Site DR : Agent]} ->
             ( <DU @ U>update![a [R DR]]
             | <DR @ R>try_deliver![Q SQ a c v true]
             | dack?_ = lock!m)
          {NotFound> _} -> ()))
```

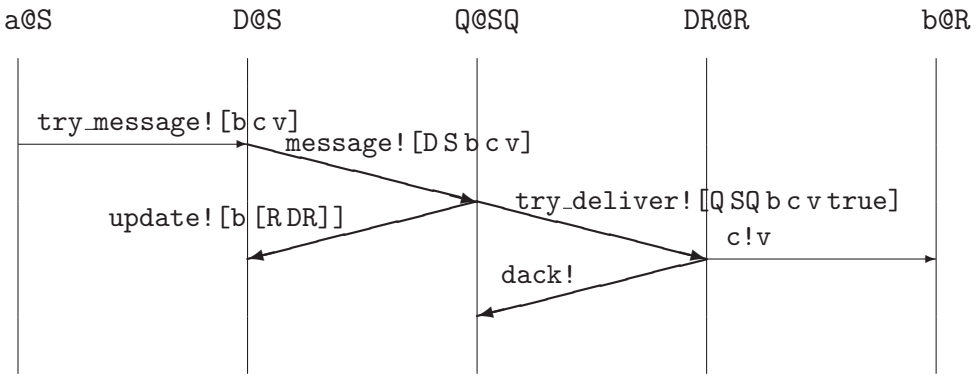
```
daemondaemon?*S:Site=            (* launch a daemon on site S *)
  agent D =
     migrate to S
       new lock : ^(Map Agent [Site Agent]) (*  the daemon body  *)
       ( <toplevel@firstSite>ndack![S D]
       | lock!(map.make ==)
       | try_message?*[#X a:Agent c:^X v:X]=
            lock?m= switch (map.lookup m a) of (
              {Found> [R : Site DR : Agent]} ->
                 ( <DR @ R>try_deliver![D S a c v false]
                 | lock!m )
              {NotFound> _} ->
                 ( <Q @ SQ>message![D S a c v]
                 | lock!m))
       | try_deliver?*[#X DU:Agent U:Site a:Agent c:^X v:X ackme:Bool] =
            iflocal <a>c!v then
              if ackme then <DU @ U>dack![] else ()
            else <Q @ SQ>message![DU U a c v]
       | update?*[a s] = lock?m= lock!(map.add m a s))
```

Figure 6.1: Parts of the Top Level in the QSC Algorithm – The Query Server and Daemon Daemon

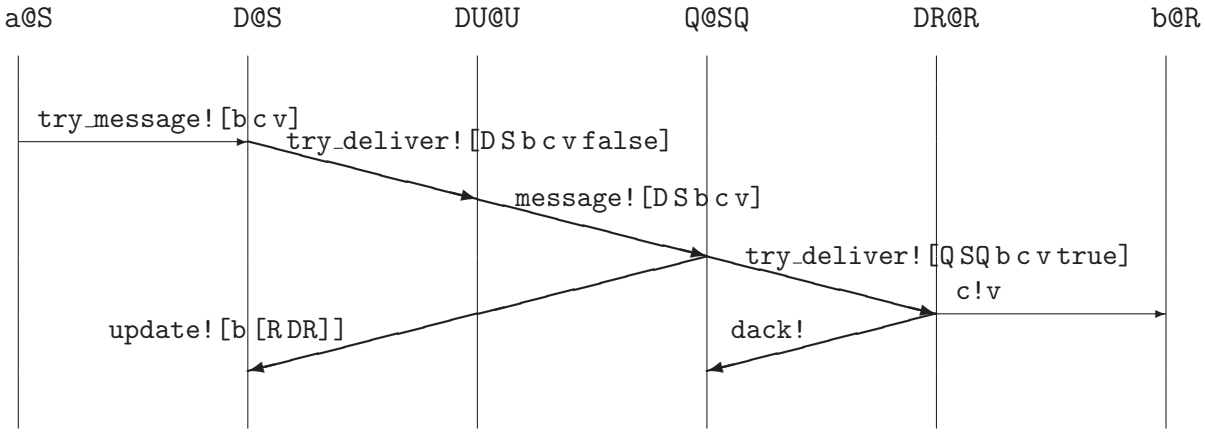
The best scenario: good guess in the D cache. This should be the common case.



No guess in the D cache.



The worst scenario: wrong guess in the D cache.



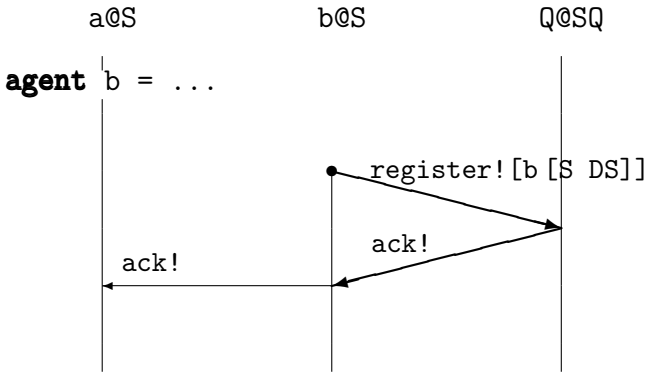
Horizontal arrows are synchronised communications within a single machine (using **iflocal**); slanted arrows are asynchronous messages.

Figure 6.2: The Delivery of Location-Independent Message c@b!v from a to b in the QSC Algorithm

Similarly, a high-level agent `a` must synchronise with the query server while creating a new agent `b`, with messages on `register` and `ack`.

```
[[agent b = P in P']][a Q SQ] =def
   currentloc?[S DS]=
      agent b =
        ( <Q @ SQ>register![b [S DS]]
        | ack?_= iflocal <a>ack![] then
                   ( currentloc![S DS]
                   | [[P]][b Q SQ])
                 else () )
      in
        ack?_= ( currentloc![S DS]
               | [[P']][a Q SQ])
```

The current site/daemon data for the new agent must be initialised to `[S DS]`; the creating agent is prevented from migrating away until the registration has taken place by keeping its `currentloc` lock until an `ack` is received from `b`. A sample execution is below.



Returning to the process of message delivery, there are three cases (see Figure 6.2). Consider the implementation of `c@b!v` in agent `a` on site `S`, where the daemon is `D`. Suppose `b` is on site `R`, where the daemon is `DR`. Either `D` has the correct site/daemon of `b` cached, or `D` has no cache data for `b`, or it has incorrect cache data. In the first case `D` sends a `try_deliver` message to `DR` which delivers the message to `b` using **`iflocal`**. For the PA application this should be the common case; it requires only one network message.

In the cache-miss case `D` sends a `message` message to the query server, which both sends a `try_deliver` message to `DR` (which then delivers successfully) and an `update` message back to `D` (which updates its cache). The query server's lock is kept until the message is delivered, thus preventing `b` from migrating until then.

Finally, the incorrect-cache-hit case. Suppose `D` has a mistaken pointer to `DU@U`. It will send a `try_deliver` message to `DU` which will be unable to deliver the message. `DU` will then send a `message` to the query server, much as before (except that the cache update message still goes to `D`, not to `DU`).

Refinements and Extensions The algorithm is very asynchronous; some additional optimisations are feasible (e.g. updating the daemon's cache more frequently, more asynchrony in QS, replacing explicit acknowledgement messages by piggybacking control data, e.g. a number of messages in transit). It should have good performance for the PA application, with most application-level messages delivered in a single hop and none taking more than three hops (though 5 messages). The query server is involved only between a migration and the time at which all relevant daemons receive a cache update; this should be a short interval.

The algorithm does, however, depend on reliable machines. The query server has critical state; the daemons do not, and so in principle could be re-installed after a site crash, but it is only possible to reboot a machine when no other daemons have pointers (that they will use) to it. In a refined version of the protocol daemons and the QS would use a store-and-forward protocol to deliver all messages reliably in spite of failures; the QS would be replicated. In order to extend collaboration between clusters of domains (e.g. over a wide-area network), a federated architecture of interconnected servers must be adopted. In order to minimise communication between domains, the agents should register and unregister with the local QS on changing domains. We present an example federated architecture translation in the end of this chapter.

6.2.2 Disconnected Operation: The QSCD Algorithm

In this section we describe the *Query Server with Caching and Disconnection* (QSCD) algorithm which tolerates temporal disconnection of sites. An agent can disconnect a current site from the network and later reconnect, so that all high-level messages to and from the site are transparently delivered irrespective of agent migration and site disconnection. No messages are ever lost. No duplicate messages are ever received by agents. However, agent migration is not transparent - a program exception is raised in a high-level agent if the agent tries to migrate out from a disconnected site; migration to a site which has been disconnected is blocked until the site is back in the network.

The algorithm implements disconnection-aware daemons and extends the high-level language with primitives `"disconnect" in P` and `"connect to" s:Site in P` to handle disconnection. The algorithm translations are similar

to the QSC translations in §6.2.1; the main difference is that the inputs used to receive acknowledgments are replaced by inputs with timeout in order to detect disconnection. If there is a timeout then an alternative action is performed (e.g. a message can be sent again, an operation can be blocked, etc.). The precise definition of the query server and daemon is given in Figures 6.3 and 6.4. The translations described below are mainly to illustrate the use of a rudimentary module system of the high-level language, an input with timeout, and replicated messages (with a property that exactly one message is eventually delivered). Therefore, in order not to complicate the algorithm, we made a few simplifications. Firstly, the algorithm specified below is not very practical since a site disconnection will block *all* agent migrations and *all* communications which need to be forwarded through the query server. Secondly, each time the operation **agent** or **migrate** fails due to a timeout, an exception is invoked in the application (in a more practical algorithm, the infrastructure should rather try to repeat the operation with a slightly longer timeout before finally signalling problems). Therefore, the algorithms that are applicable to actual systems with mobile computers would have to be yet more delicate and complex. We discuss some of these refinements and extensions informally in the end of §6.2.2 and §6.2.3 (e.g. an algorithm which allows ad-hoc connection of computers, i.e. with no connection to the stable part of the network).

The messages sent between agents fall into five groups, implementing high-level agent creation, agent migration, location-independent messages, and two operations for site disconnection and reconnection to the network. Below, we describe the compositional translation of these cases. We omit detailed description of the translation whenever it is very similar to the translation of the simple QSC algorithm, presented in §6.2.1.

Each class of agents maintains some explicit state as an output on a lock channel. The meaning of this state is as in the simple QSC algorithm.

To send a location-independent message the translation of a high-level agent first tries to send the message locally. If that fails, the message is forwarded to the local daemon as in the QSC algorithm. The composition translation of `c@b!v`, 'send `v` to channel `c` in agent `b`', is below.

```
[[c @ b ! v]]_ =def
   iflocal <b>c!v then ()
       else currentloc?[S DS]=
           iflocal <DS>try_message![b c v] then
               currentloc![S DS]
           else ()
```

The local output (in the 2nd line) allows adjacent agents (on the same site) to communicate even if the local daemon will be blocked in the case of site

disconnection. We return later to the process of delivery of the message which is sent to the local daemon.

To migrate while keeping the query server's map accurate, the translation of a **migrate** in a high-level agent `a` synchronises with the query server `[Q SQ]` before and after actually migrating, with `migrating`, `migrated`, and `mack` messages. We also deal with a case when the current site is disconnected. If the query server does not respond within a certain period of time `t` (i.e. the current site is disconnected or the communication link is slow), migration will be abandoned (with an exception message `err`). Alternatively, we could ask the local daemon for more accurate information (the daemon always knows about the connection/reconnection status) but due to the lack of space we omit details here.

```
⟦ migrate to u P ⟧[a Q SQ t err] =def
       currentloc?[S DS]=
          val [U DU] = u
          new mack : ^[]
          ( <Q @ SQ>migrating![a mack]
          | wait mack?_ = (migrate to U
                              ( <Q @ SQ>migrated![U DU]
                              | mack?_ = ( currentloc![U DU]
                                         | ⟦P⟧[a Q SQ t err])))
            timeout t ->
              ( currentloc![S DS]
              | mack?_ = (* connection is back, or timeout too short *)
                  <Q @ SQ>migrated![S DS]
              | err!"No connection." (* raise exception *)
              | ⟦P⟧[a Q SQ t]))
```

This first creates a fresh private channel `mack`, then sends `[a mack]` on the channel `migrating` to the query server, in parallel with a timed input on the channel `mack`. If the reply on `mack` is received within `t` seconds (approximately), the migration proceeds exactly as in the basic QSC algorithm. Otherwise, the timeout clause is triggered and the migration is abandoned. However, if in fact the connection to the server was made possible (e.g. a timeout was simply too short) then the message `migrating` would be delivered to the server and the server would send to the agent a reply message `mack`. Note, that the query server blocks any disconnection requests after receiving a message `migrating` and can only release the lock after receiving an acknowledgement that migration is finished. Therefore, although migration failed the agent may still have to send a message `migrated` in the timeout clause and release the lock in the query server; the message will then contain an address `[S DS]` of the current site.

```
agent Q =                (*  the query server  *)
 (migrate to SQ
  new lock : ^(Map Agent SiteTy)
  ( <toplevel@firstSite>nd![SQ Q]  (*  ack that Q is on SQ *)
  | lock!(map.make ==)     (*  initialise the lock  *)
  | register?*[a [S DS]]=  (*  register a new agent  *)
      lock?m=
        ( lock!(map.add m a [S DS])
        | <a@S>ack![])
  | migrating?*[a:Agent ack:^[]] = (*  lock during a migration  *)
      lock?m= switch (map.lookup m a) of (
        {Found> [S : Site  DS : Agent]} ->
           ( <a@S>ack![]
           | migrated?[S' DS'] =
             ( lock!(map.add m a [S' DS'])
             | <a@S'>ack![]))
        {NotFound> _} -> ())
  | message?*[#X DU U a:Agent c:^X v:X dack:^SiteTy]=
                          (*  deal with a lost message  *)
      lock?m= switch (map.lookup m a) of (
        {Found> [R : Site  DR : Agent]} ->
           ( <DU @ U>dack![R DR]
           | <DR @ R>message![Q SQ a c v dack]
           | dack?_ = lock!(map.add m a [R DR]))
        {NotFound> _} -> ())
  | block?*[a:Agent S:Site]=
      lock?m= ( <a@S>ack![]
              | buffer!m )
  | unblock?*[a:Agent S:Site]=
      buffer?m= ( lock!m
                | <a@S>ack![])
 ))
```

Figure 6.3: Parts of the Top Level in the QSCD Algorithm – the Query Server

```
daemondaemon?*S:Site=  (*  launch a daemon D on site S  *)
 (agent D =            (*  the daemon body  *)
   (migrate to S
    new lock   : ^(Map Agent SiteTy)

    def send_message [#X Q:Agent SQ:Site D:Agent S:Site
                         a:Agent c:^X v:X m:(Map Agent SiteTy)
                         dack:^SiteTy] =
        ( <Q @ SQ>message![D S a c v dack]
        | dack?s= lock!(map.add m a s ))

    ( <toplevel@firstSite>nd![S D]  (*  ack that D is on S  *)
    | lock!(map.make ==)
    | try_message?*[#X a:Agent c:^X v:X]=
        lock?m= switch (map.lookup m a) of (
          {Found> [R : Site  DR : Agent]} ->
            (new dack : ^SiteTy
             ( <DR @ R>message![D S a c v dack]
             | wait
                 dack?s= lock!(map.add m a s)
               timeout t ->
                 send_message![Q SQ D S a c v m dack]))
          {NotFound> _} -> send_message![Q SQ D S a c v m
                                         (new dack : ^SiteTy)])
    | message?*[#X DU:Agent U:Site a:Agent c:^X v:X dack:^SiteTy] =
        iflocal <a>msg![dack c v] then <DU @ U>dack![S D]
        else lock?m= ( <Q @ SQ>message![D S a c v dack]
                     | dack?s= ( lock!(map.add m a s)
                               | <DU @ U>dack!s ))
    | disconnect?*a =  lock?m= ( buffer!m | <Q @ SQ>block![a S])
    | connect?*[a _ _] =   (*  connect and unblock msgs  *)
                       buffer?m= ( <Q @ SQ>unblock![a S]
                                 | lock!m) ))
 ())
```

Figure 6.4: Parts of the Top Level in the QSCD Algorithm – the Daemon Daemon

The query server's lock is kept during migration. This lock will protect the current and target sites from being disconnected by other agents while migration is in progress. The agent's own record of its current site and daemon must also be updated with the new data `[U DU]` (or restored from the old data if the migration failed) when the agent's lock is released. Site names in the high-level program are encoded as before, i.e. by pairs of a site name and the associated daemon name; there is a translation of types

$$[\![\texttt{Agent}]\!] \stackrel{def}{=} \texttt{Agent}$$
$$[\![\texttt{Site}]\!] \stackrel{def}{=} \texttt{[Site Agent] = SiteTy}$$

Similarly, a high-level agent `a` must synchronise with the query server while creating a new agent `b`, with messages on `register` and `ack`. If the query server is not accessible, the creation fails.

$[\![$ **agent** b = P **in** P' $]\!]_{[\texttt{a Q SQ t err}]} \stackrel{def}{=}$

```
  currentloc?[S DS]=
    agent b =
      (new msglog : ^(Map Id [])
       ( <Q @ SQ>register![b [S DS]]
       | wait ack?_=  iflocal <a>ack![] then
                         ( currentloc![S DS]
                         | [[P]][b Q SQ t err])
                      else ()
         timeout t -> ( <a>ack![]
                      | err!"No connection.")  (* raise exception *)
       | msglog!(map.make ==)
       | msg?*[#X id c v]= msglog?m= switch (map.lookup m id) of (
              {NotFound> _} -> (c!v | msglog!(map.add m id []))
              {Found> _} -> msglog!m))) (* ignore duplicate *)
    in
      ack?_= ( currentloc![S DS]
             | [[P']][a Q SQ t err])
```

As in the original algorithm, the current site/daemon data for the new agent must be initialised to `[S DS]`; the creating agent is prevented from migrating away until registration has taken place by keeping its `currentloc` lock until an `ack` is received from `b`. The connection with the query server is tested by a timeout mechanism. If connection is suspected of being broken, the `ack` is sent immediately to the creating agent. The last two clauses of the body of `b` are responsible for ignoring duplicate messages received by the agent. A message log `msglog` is created to store unique identifiers of all messages received on the channel `msg`. Messages whose identifiers are not found in the

log are registered with the log and sent to proper local channels, or discarded as duplicates otherwise.

Returning to the process of message delivery, there are three basic cases (see Figure 6.6) as in the simple QSC algorithm. Consider the implementation of `c@b!v` in agent `a` on site `S`, where the daemon is `D`. Suppose `b` is on site `R`, where the daemon is `DR`. Either `D` has the correct site/daemon of `b` cached, or `D` has no cache data for `b`, or it has incorrect cache data. In the first case `D` sends a `message` message to `DR` which delivers the message to `b` using **`iflocal`** and sends an acknowledge message `dack`. For the PA application this should be the common case; it requires only two network messages. If `dack` is not received within a certain time (which means that either site `R` is disconnected or the communication link to site `R` is slow), `D` sends a `message` message to the query server which delivers it correctly as in the cache-miss case, described below. Each message is augmented with a unique name `dack` of a freshly created acknowledge channel. This name is later used by agent `b` to look up the message log and discard the message if it has already been delivered (when the timeout was caused by a slow link between `S` and `R`). Agents `DR` and `Q` use `dack` to sent back acknowledgments and location updates, which are delivered unambiguously.

In the cache-miss case `D` sends a `message` message to the query server, which both sends a `message` message to `DR` (which then delivers successfully) and a `dack` message back to `D` (which updates its cache). The query server's lock is kept until the message is delivered, thus preventing `b` from migrating until then.

Finally, the incorrect-cache-hit case. Suppose `D` has a mistaken pointer to `DU@U`. It will send a `message` message to `DU` which will be unable to deliver the message. `DU` will then send a `message` to the query server, much as before (the cache update messages are sent first to `DU` which then forwards it to `D`). If `D` has not received the cache update acknowledgement for a long enough time, it suspects that something went wrong, and sends a `message` (with a `dack`) to the query server, as in the cache-miss case.

To disconnect a site while not missing messages sent between the site and a stable part of the network, the translation of a `"disconnect"` macro in a high-level agent `a` synchronises with the local daemon and the query server. Messages sent from the stable network to the disconnected site will be blocked in the query server until the site reconnects. In the opposite direction, cross-network messages sent by agents on the disconnected site will be blocked in the local daemon. No messages are ever lost.

```
⟦ "disconnect" foo in P ⟧[a Q SQ t err]  def=
     currentloc?[S DS]=
       iflocal <DS>disconnect!a then
          ack?_= ( currentloc![S DS]
                 | print!"Now you can safely disconnect
                                                 your computer."
                 | ⟦P⟧[a Q SQ t err])
       else ()
```

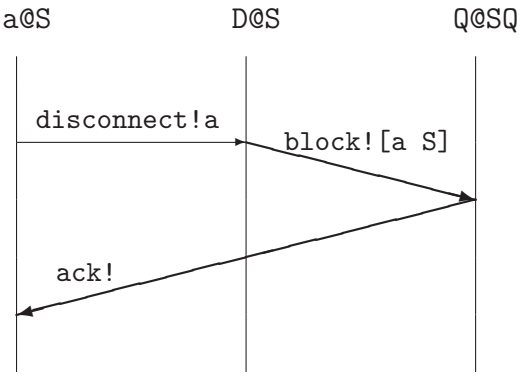
Similarly, a high-level agent `a` must reconnect to the network by invoking a `"connect"` macro with a parameter `s`, 'connect to a query server which is on site `s`'. Here, the parameter `s` is actually not used by the encoding since the algorithm assumes only one query server. Later, we describe a scalable version of the algorithm which uses many query servers, and the parameter can then be useful.

```
⟦ "connect to" s in P ⟧[a Q SQ t err]  def=
       currentloc?[S DS]= val [SQ:Site Q:Agent] = s
         iflocal <DS>connect![a SQ Q] then
            ack?_= ( currentloc![S DS]
                   | ⟦P⟧[a Q SQ t err])
         else ()
```

Note that the server's site in the high-level program (of type `Site`) is encoded by a pair of a site name and the associated daemon (query server) name. Typical executions are illustrated in Figure 6.5.

Refinements and Extensions The algorithm should have good performance for the PA application if the timeout mechanism is set up correctly (e.g. using some stabilising failure detector), with most application-level messages delivered in a single hop and none taking more than three hops (though 6 messages). The query server is involved only between migration and the time at which all relevant daemons receive a cache update; this should be a short interval. Messages to a disconnected site cannot be delivered and so they are buffered in the query server which will deliver them upon site reconnection. However, the algorithm described above is not very practical, since the query server uses a global lock during disconnected operation, i.e. the QS blocks high-level messages to *all* sites if at least one site is disconnected. Also, an operation "create a new agent" fails with a program exception raised in a spawning agent, each time the operation is invoked from a disconnected site. A refined version of this algorithm which is free from these problems is described below.

Disconnect a site from the network.



Reconnect a site to the network.

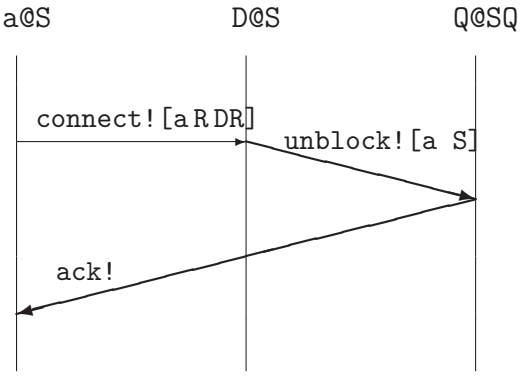
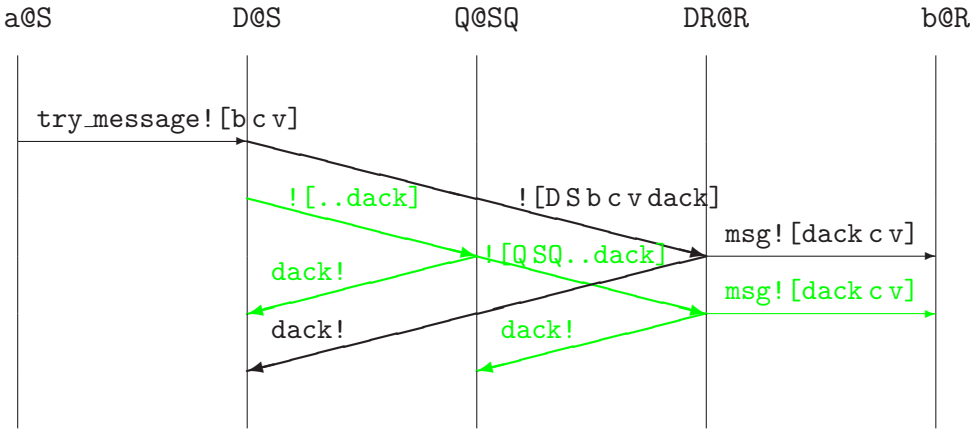
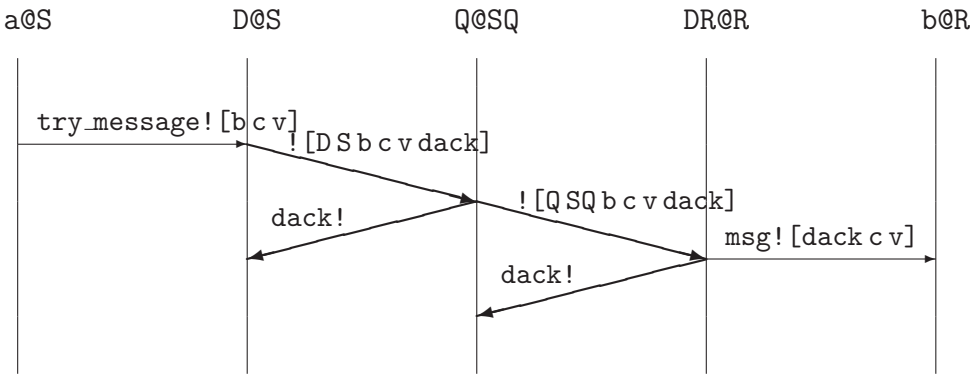


Figure 6.5: The Disconnection and Reconnection Requests in the QSCD Algorithm

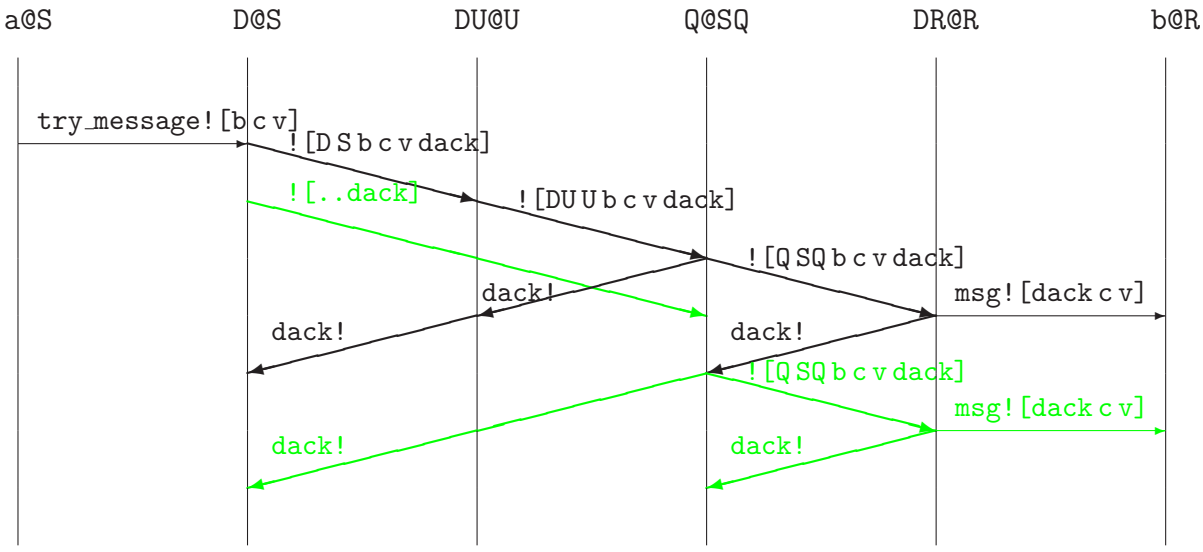
The best scenario: good guess in the D cache. This should be the common case.



No guess in the D cache.



The worst scenario: wrong guess in the D cache.



The communication in grey colour is executed only if there is a timeout. Abbreviations: `![..]` for `message![..]`, and `dack!` for `dack![R DR]`

Many sites should be able to disconnect and reconnect at the same time, and the query server should block communication and migration only to a site which is currently disconnected. This requires that a query server maintains a separate map from sites to status information ("connected" or "disconnected"). A map of agents must contain little locks (each per agent entry) so that only messages to agents in disconnected sites are buffered. A local daemon has exact knowledge whether there is connection to the query server or not, so we can improve the algorithm by synchronising agent migrations with the local daemon (taking care to avoid distributed deadlock with disconnection requests). Also, only minor refinements are required to be able to re-install daemons after a site crash (making a query server fault-tolerant is much more difficult). In the protocol presented here, it is only possible to reboot a machine when a query server does not have an active communication link to it.

In the next section, we discuss extensions necessary for ad-hoc connection of mobile computers. These extensions also allow for non-blocking agent creation, since the registration messages from a laptop computer may be sent to a local QS, installed on the laptop, and thus do not depend on the network availability. Further improvements of the disconnected mode are plausible, e.g. operations `connect` and `disconnect` might be implicit if the operating system could provide a flag or an interrupt every time the local network connection goes up or down (though it might still be useful to have the operation "connect" in a high-level language).

6.2.3 Wide-Area Architecture: The FQSC Algorithm

In this section we describe the *Federated Query Server with Caching* (FQSC) algorithm. We extend the original QSC algorithm which we described in §6.2.1, so as to allow many query servers, one per local network (domain). We then discuss refinements and extensions which are required to support transferring mobile computers between local networks, and establishing ad-hoc connections between mobile computers.

The algorithm has a collection of query servers. For each agent there is at least one server (the current local server) that records the current site of the agent; agents synchronise with the local server before and after migrations (and register at a new query server if moving to a new domain); application (location-independent) messages are sent directly to destinations according to the cache information, or — if there is no good cache data — via the servers, which work as forwarding pointer chains that are collapsed when possible.

The protocol is almost the same as the original QSC algorithm, except that a query server can forward a message to another query server which eventually delivers the message. If a query server has no pointer for the destination agent of a message then it will forward the message to the QS in the network (domain) where the destination agent was created; to make this possible an agent name is encoded by a triple of an agent name and the names of the site and query server on which the agent was originally registered. Similarly, a site name is encoded by a record of a site name and the names of the daemon, query server, and server's site for that site. The precise definition of the query server and daemon is given in Figures 6.7 and 6.8. A translation of types is following:

$$[\![\texttt{Agent}]\!] \stackrel{def}{=} \texttt{[Agent Agent Site]} = \texttt{AgentTy}$$
$$[\![\texttt{Site}]\!] \stackrel{def}{=} \texttt{[Site Agent Agent Site]} = \texttt{SiteTy}$$

To send a location-independent message the translation of a high-level agent simply asks the local daemon to send it, exactly as in the original QSC algorithm. The compositional translation of `c@b!v`, 'send `v` to channel `c` in agent `b`', is below.

$$[\![\texttt{c @ b ! v}]\!]_{\texttt{a}} \stackrel{def}{=}$$

```
currentloc?[S DS Q SQ]=
   iflocal <DS>try_message![b c v] then
        currentloc![S DS Q SQ]
   else ()
```

This time the translation is parametric only on the name `a` of the agent containing this phrase but the agent's lock channel `currentloc` stores four values: the name `S` of the current site, the name `DS` of the local daemon, and the names `Q` and `SQ` of the current query server and the server's site.

To migrate while keeping the query server's map accurate, the translation of a **`migrate`** in a high-level agent synchronises with the local query server before and after actually migrating, with `migrating`, `migrated`, and `ack` messages.

```
serverserver?*SQ:Site=  (*  launch a query server Q on site SQ  *)
  (agent Q =
   (migrate to SQ
    new lock : ^(Map AgentTy SiteTy)
    ( <toplevel@firstSite>nq![Q SQ]
    | lock!(map.make ==)     (*  initialise the lock  *)
    | register?*[a [S DS]]=  (*  register a new agent  *)
         lock?m= ( lock!(map.add m a [S DS])
                 | (val [A _ _] = a  <A@S>ack![]))
    | migrating?*a =  (*  lock during a migration  *)
         lock?m= switch (map.lookup m a) of (
           {Found> [S : Site  DS : Agent]} ->
             (val [A _ _] = a
              ( <A@S>ack![]
              | migrated?[S' DS' DR' R'] =
                ( lock!(map.add m a [R' DR'])
                | <A@S'>ack![])))
           {NotFound> _} -> ())
    | message?*[#X DU U a:AgentTy c:^X v:X _]=
                                 (*  deal with a lost message  *)
         lock?m= switch (map.lookup m a) of (
           {Found> [R : Site  DR : Agent]} ->
              ( <DR @ R>message![Q SQ a c v true]
              | update?[_ [S' DS']] =
                   ( <DU @ U>update![a [S' DS']]
                   | lock!(map.add m a [S' DS'])))
           {NotFound> _} ->
              (val [A Q' SQ'] = a
               ( <Q' @ SQ'>message![Q SQ a c v true]
               | update?[_ [S' DS']] =
                   ( <DU @ U>update![a [S' DS']]
                   | lock!(map.add m a [S' DS'])))))
   ))
   ())
```

Figure 6.7: Parts of the Top Level in the FQSC Algorithm – the Query Server

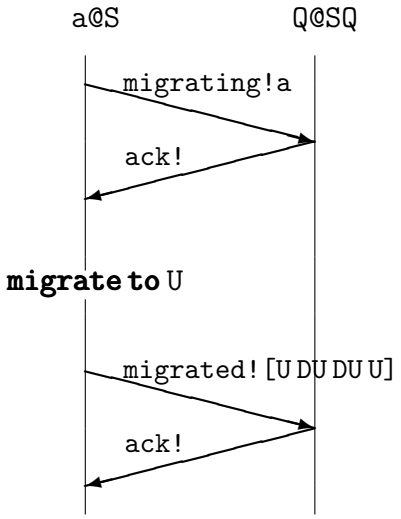
```
daemondaemon?*[S:Site [Q:Agent SQ:Site]]=
               (*  launch a daemon D on site S  *)
               (*  Q is a local Query Server located at site SQ  *)
  (agent D =  (*  the daemon body  *)
   (migrate to S
    new lock : ^(Map AgentTy SiteTy)
    ( <toplevel@firstSite>nd![S D Q SQ]
    | lock!(map.make ==)
    | try_message?*[#X a:AgentTy c:^X v:X]=
        lock?m= switch (map.lookup m a) of (
          {Found> [R : Site  DR : Agent]} ->
             ( <DR @ R>message![D S a c v false]
             | lock!m )
          {NotFound> _} ->
             ( <Q @ SQ>message![D S a c v true]
             | lock!m ))
    | message?*[#X DU:Agent U:Site a:AgentTy c:^X v:X ackme:Bool] =
       (val [A _ _] = a
        iflocal <A>c!v then
          if ackme then <DU @ U>update![a [S D]] else ()
        else <Q @ SQ>message![DU U a c v true])
    | update?*[a s] = lock?m= lock!(map.add m a s) ))
 ())
```

Figure 6.8: Parts of the Top Level in the FQSC Algorithm – the Daemon Daemon

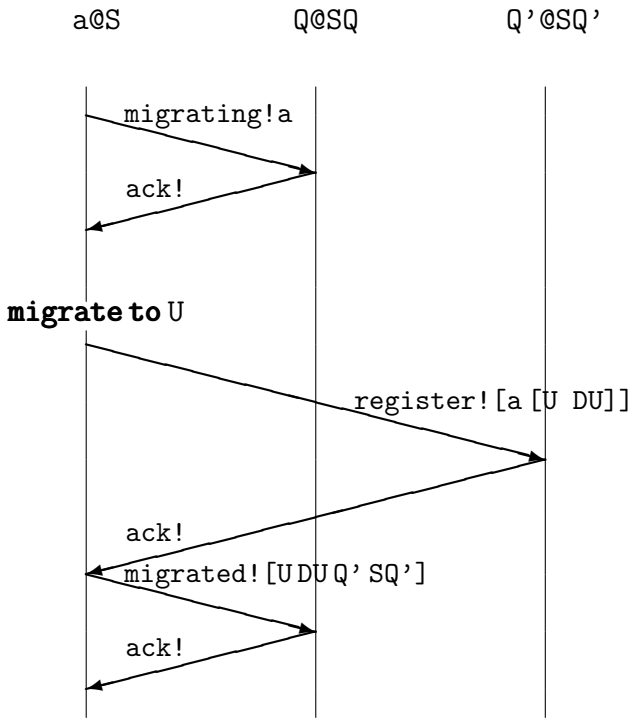
```
⟦ migrate to u P ⟧a  def=
  currentloc?[S DS Q SQ]=
    val [A _ _] = a
    val [U DU Q' SQ'] = u
    ( <Q @ SQ>migrating!a
    | ack?_= (migrate to U
              if (== [Q' SQ'] [Q SQ]) then (* an easy case *)
                 ( <Q @ SQ>migrated![U DU DU U]
                 | ack?_ = (currentloc![U DU Q SQ]
                           | ⟦P⟧a))
              else                          (* a cross-domain hop! *)
                 ( <Q' @ SQ'>register![a [U DU]]
                 | ack?_= ( <Q @ SQ>migrated![U DU Q' SQ']
                          | ack?_ = ( currentloc![U DU Q' SQ']
                                    | ⟦P⟧a)))))
```

After migration we check whether the destination site is in the same domain. If so (see a **then** clause), we proceed as in the original QSC algorithm. In the case of cross-domain migration (see an **else** clause), this registers an agent `a` at the new query server `Q'` with a `register` message, and then sends `[U DU Q' SQ']` on the channel `migrated` to the old query server `Q`, releasing the lock with a new value after the message is sent. A first message for the destination agent `a` sent to the old query server `Q` will be forwarded to `Q'`, which will forward it to a daemon `DU` that delivers the message as usual.

A sample execution of a local (within domain) migration is below.



A sample execution of a cross-domain migration with registration at Q'.

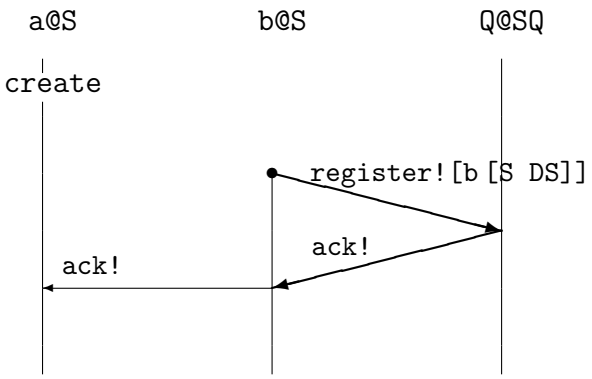


Similarly, a high-level agent `a` must synchronise with the query server while creating a new agent `b`, with messages on `register` and `ack`. The encoding is the same as in the original QSC algorithm except for the parameters.

```
⟦ agent b = P in P' ⟧a =
  currentloc?[S DS Q SQ]=
    (val [A _ _] = a
     agent B =
        val b = [B Q SQ]
        ( <Q @ SQ>register![b [S DS]]
        | ack?_= iflocal <A>ack![] then
                    ( currentloc![S DS Q SQ]
                    | ⟦P⟧b)
                  else ())
     in
        val b = [B Q SQ]
        ack?_= ( currentloc![S DS Q SQ]
               | ⟦P'⟧a))
```

In the record `[S DS Q SQ]`, `S` is a name of a current site, `DS`, a name of a current daemon, `Q`, a name of a local query server, and `SQ`, a name of `Q`'s site.

A sample execution is below.

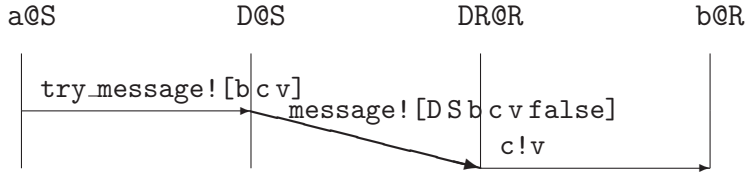


Returning to the process of message delivery, there are three basic cases as in the original QSC algorithm, and a few variations (see Figure 6.9). Consider the implementation of `c@b!v` in agent `a` on site `S`, where the daemon is `D`. Suppose `b` is on site `R`, where the daemon is `DR`. Either `D` has the correct site/daemon of `b` cached, or `D` has no cache data for `b`, or it has incorrect cache data. In the first case `D` sends a `message` message to `DR` which delivers the message to `b` using **`iflocal`**. For the PA application this should be the common case (also for the cross-domain communication); it requires only one network message.

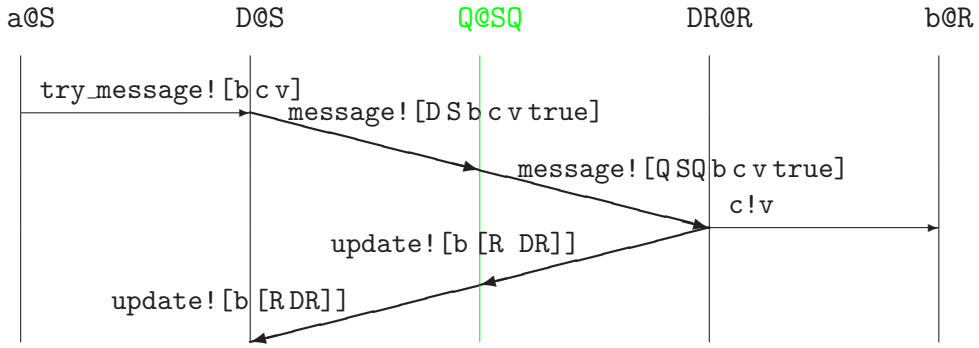
In the cache-miss case `D` sends a `message` message to the query server `Q` which forwards the message. After receiving an `update` message the query server `Q` forwards the update back to `D` (which updates its cache). In Figure 6.9 the `message` message is forwarded directly to `DR` (which then delivers successfully). However, two other situations are possible. If the forwarding pointer for the agent `b` is not found, `Q` sends the message to the query server in the domain where `b` was created (names of this query server and its site are encoded as part of the name `b`). Similarly, if `b` has migrated between domains but there has been no communication to `b` since then (and so no cache updates), `Q` will contain a pointer to the query server in the domain visited by `b`. In this case, the `message` message is forwarded between query servers until it eventually reaches `DR`. Both situations, i.e. a server's cache miss and cross-domain forwarding, are illustrated in Figure 6.9 using a grey colour. The forwarding pointer chain is collapsed by sending the `update` messages which update caches with `b`'s current location.

Finally, the incorrect-cache-hit case. Suppose `D` has a mistaken pointer to `DU@U`. It will send a `message` message to `DU` which will be unable to deliver the message. `DU` will then send a `message` to the query server, much as before (except that the cache update message still goes to `D`, not to `DU`). Note, that the daemon `D` can also be a query server (if this were a cross-domain communication).

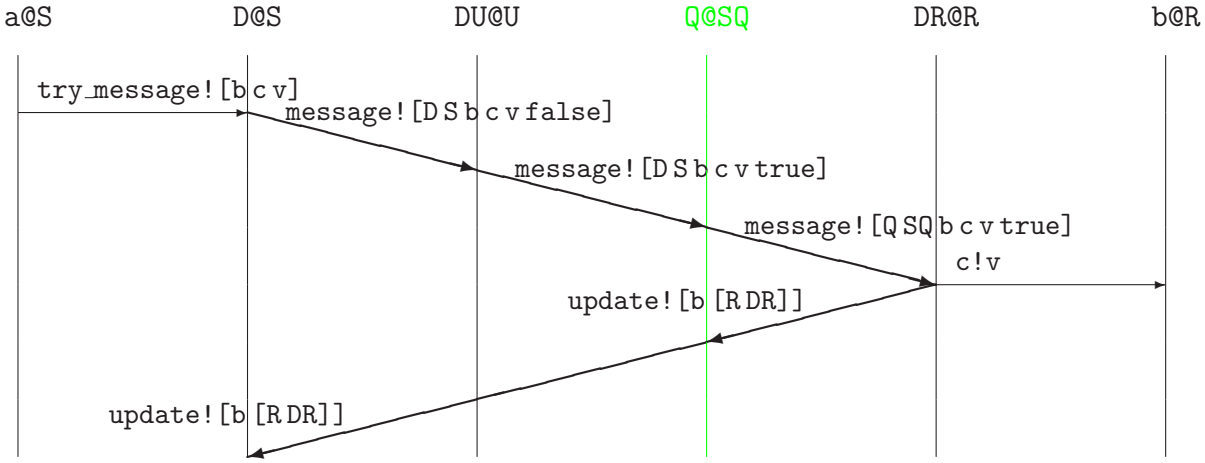
The best scenario: good guess in the D cache. This should be the common case.



No guess in the D cache.



The 1st worst scenario: wrong guess in the D cache.



The 2nd worst scenario: not-updated (or no) guess in the query server's cache.

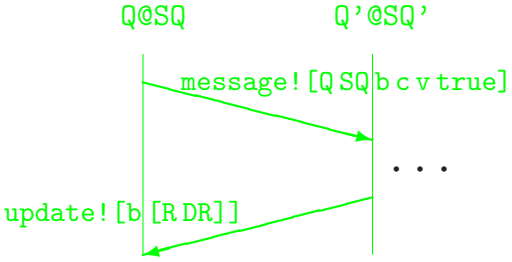


Figure 6.9: The Delivery of Location-Independent Message c@b!v from a to b in the FQSC Algorithm

Refinements and Extensions Some additional optimisations are feasible (e.g. updating the daemon's and server's cache more frequently). In the algorithm presented above, the forwarding pointer chain may grow if there is no communication to migrating agents. However, in the PA application migrations between domains are rare and they usually take place between no more than a few domains only (since agent cross-domain migrations correspond to travels of individuals working on projects). The cost of forwarding a message to an agent in an other domain is paid only for the first message (then the forwarding pointer chain is collapsed and any subsequent messages for this agent are sent directly, if the agent does not migrate, or indirectly through the local domain QS if the agent moves locally). Local migrations *within* a domain can be more frequent than inter-domain moves, and thus sending too many cache updates must be avoided. For example, if agent migrations are local, the algorithm sends a cache-update message to other query servers only in the case of incorrect-cache-hits from these servers (this is similar to the "wrong guess in the `D` cache" case of the original QSC algorithm). The algorithm could be made more asynchronous and use time-stamped asynchronous messages instead of explicit locking (e.g. in the case of the hand-over operation when agents and messages travel between domains), e.g. as in the protocol for location directory maintenance in mobile networks described in [RRD95], which allows for consistent location information to be maintained about mobile hosts that switch off and arbitrarily reappear in some other part of the network; the algorithm tolerates base station failures and the corruption of a logical time stamp.

The algorithms described in §6.2.2 and §6.2.3 can be used to build a complete generic infrastructure for the PA application that uses a federated architecture, augmented with support for disconnected operation. To disconnect a site (e.g. a mobile computer) from a current local area network and reconnect it in some other network, the translation of a site daemon acts in a similar way to the translation of agent migration between domains, i.e. it has to synchronise with the query servers of these two domains. The algorithm allows mobile computers to connect to each other and establish ad-hoc communications, assuming that at least one has a local query server installed (so that this computer can be a "domain" to which the other connects).

A fault-tolerant version of the algorithm may require an additional level of infrastructure to detect failures. The QSCD algorithm uses a timeout mechanism to detect disconnection of laptop computers from a local QS server. If the laptop is connected to the server via a LAN (as assumed in 6.2.2), then we should be able to set the timeout properly. However, detecting faults by setting timeouts on remote agents in a wide-area network is bound to be inaccurate since message latency in these networks tends to be large and highly

unpredictable. We can still attempt to detect failures using timeouts in the case of local communication (i.e. between agents and QS server in a LAN), but for the inter-domain communication the algorithm may use some additional infrastructure, such as a network event notification service, or some other similar service, e.g. the gossip-based scalable failure detector [RMH98]. The event notification service would have daemons that use timeouts only on neighbouring servers and local clients, and notify the service subscribers about failures. In addition to timeouts, the infrastructure may use other techniques to detect failures where appropriate (see, e.g. [Vog96]), for example failure notifications generated by the operating system which recovered after a failure.

Chapter 7

Nomadic Pict Implementation

Nomadic Pict implementation has a two-level architecture, illustrated in Figure 7.1, following that of the language. The low-level extends the Pict language [PT97a] by providing direct support for agent creation, migration and location-dependent communication. The high level supports location-independent communication by applying translations – the compiler takes as input a program in the high-level language together with an encoding of each high-level primitive into the low-level language (expressed in a simple meta-language). It type-checks and applies the encoding; the resulting low-level intermediate code is executed on a relatively straightforward distributed run-time system. The source code of the compiler doubled the size of the Pict compiler, and is around 15000 lines of Objective Caml. The runtime system is only around 1700 lines of Objective Caml; this, however, does not include distributed infrastructures and standard libraries, which are written in Nomadic Pict. Below, we describe the compiler and runtime system in more detail.

7.1 Architecture of the Compiler

Programs in Nomadic Pict are compiled in the same way as they are formally specified, by translating the high-level program into the low-level language, which in turn is compiled to the intermediate code executed by the run-time system. The compilation of a Nomadic Pict program has the following phases:

- parsing the high-level program and infrastructure encoding
- importing separately compiled units (e.g. standard libs)

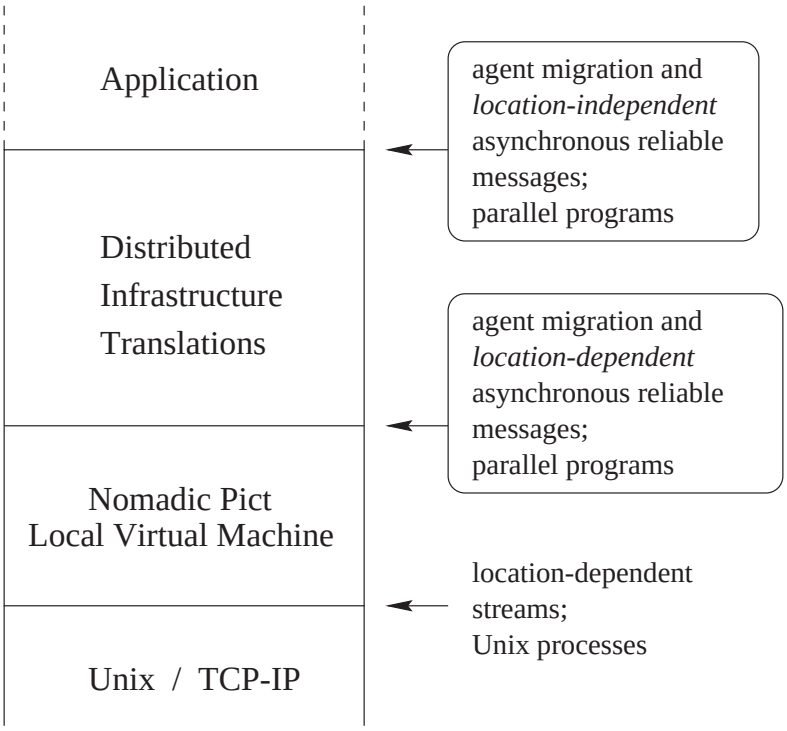


Figure 7.1: The Nomadic Pict Two-Levels of Abstraction

- scope resolution and typechecking the high-level program and meta-definitions of the encoding
- applying the encoding and generating the low-level code
- scope resolution and typechecking the low-level code
- continuation-passing translation of the low-level code to the intermediate code
- joining imported code (if there are any bindings exported from a unit)
- incremental optimisation of the intermediate code

7.1.1 Compilation Phases

Below, we describe briefly the more interesting compilation phases. The generation of the core language from the low-level language is based on modules of Pierce and Turner's Pict compiler, extended with rules for the Nomadic Pict language. See the Pict definition [PT97b] for a formal description of this translation for Pict constructs.

Parsing The compiler uses standard lexing and parsing tools to generate an abstract syntax tree in which issues of precedence and parenthesization have been resolved. Some very straightforward derived forms are desugared during parsing, e.g. the parallel composition of three or more processes is transformed to a nested sequence of binary parallel compositions.

Importing A program consists of a collection of named *compilation units*, each comprising a sequence of `import` statements followed by a sequence of declarations. Individual units can be compiled separately. Compilation begins with the unit that has been designated as the *main unit*. A program defined in the main unit can use the high-level language. In such a case, a top-level clause of the infrastructure encoding and compositional translation of high-level primitives must be included. The program begins execution from the top-level clause which contains all the necessary daemons and initial values of the encoding parameters.

Scope Resolution The process of resolving variable scopes yields an alpha-renamed copy of the original term. The alpha-renamed term has the property that every bound variable is unique, so that a simplified implementation of substitution and inlining can be used.

Typechecking Some languages, such as ML and Haskell, which are based on the Hindley-Milner type system, can automatically infer all necessary type annotations. Pict's type system, however, is significantly more powerful then the Hindley-Milner type system (e.g. it allows higher-order polymorphism and subtyping) and a simple partial type inference algorithm has been used (the algorithm is partial, in the sense that it may sometimes have to ask the user to add more explicit type information rather than determine the types itself). The algorithm is formalised in [PT97b]. It exploits the situations where the type assigned to a bound variable can be completely determined by the surrounding program context. The inference is local, in the sense that it only uses the immediately surrounding context to try to fill in a missing type annotation. For example, the variable `x` in the input expression `c?x=e` has type `Int` if the channel `c` is known to have type `^Int`.

In Nomadic Pict, typechecking is performed twice, before and after an encoding is applied. This allows more precise type error reporting. Types are erased before execution and so there is no way that type annotations in the program could affect its behaviour (an exception is type `Dyn`, to date only partially implemented, which allows data that are created dynamically to be used safely).

Applying Encodings Each high-level construct in a program is replaced by its meta-definition, in such a way that free occurrences of variables in the meta-definition are substituted by current variables from the program. Also certain types, such as `Agent` and `Site` defined in the program are replaced by their encodings.

Continuation Passing Style The compiler uses some binding-time improvements, like conversion of a program to continuation passing style[1], in order to remove the overhead of interpreting the source program and the overhead of environment lookups. In particular, the continuation-passing transformations are used to simplify complex expressions of the low-level language so that they fall within the core language. The complex expressions are complex values, value declarations `(`**`val`**` x = v P)`, application `(v v1 ... vn)`, and abstractions such as a "function definition" **`def`**` f (a1 a2) = v`.

[1]In functional languages, a program is in *continuation passing style* (CPS) if every function takes a continuation as a parameter, and whenever a function would normally return a result r to its caller, it instead returns the result of applying the continuation to r. A continuation is a kind of abstract return address, and represents the whole of the rest of the computation after the function call.

The "continuation-passing" conversion in Pict is similar to those used in some compilers for functional languages (e.g. [App92]). In essence, it transforms a complex value expression into a process that performs whatever computation is necessary and sends the final value along a designated *continuation channel*.

A complex value is always evaluated "strictly" to yield a simple value, which is substituted for the complex expression. For example, when we write `c![13 (v v1 v2)]`, we do not mean to send the *expression* `[13 (v v1 v2)]` along `c` but to send a simple value evaluated from this complex value. Thus, the expression must be interpreted as a core language expression that evaluates first the 'function' value `v`, followed by the argument values `v1` and `v2`, then calls the function instructed to return its result along the application expression's continuation channel, and finally packages the result received along the continuation channel into a simple tuple along with the integer 13 and sends the tuple along `c`.

Optimisations In the last phase, all separately compiled units are joined with the main unit, and the compiler incrementally optimises the resulting intermediate program. It does a static analysis and partial evaluation of a program, reducing π-computations whenever possible and removing inaccessible fragments of code. The remaining computations make up the generated or "residual" program executed by the runtime system. The Pict optimiser also checks the program's consistency — the following conditions must hold: no unbound variables (every variable mentioned in the program must be in scope), all bound variables must be unique, static variables (i.e. ones whose value is known to be a compile-time constant) are represented as global variables in the generated code. In the current implementation of Nomadic Pict, global variables are dynamically copied to a local agent environment upon agent creation; other solutions are plausible in a more optimised version of the compiler and runtime system.

7.1.2 Architecture-Independent Core Language

The compiler generates the intermediate code of the core language which is executed by the Nomadic Pict runtime system. The intermediate code is architecture-independent; its constructs, forming a *core language*, correspond approximately to those of the Low Level Nomadic π-calculus (extended with value types and system function calls). *Process* terms are output atoms, input and migrate prefixes, parallel compositions, processes prefixed by declarations, terminate, test-and-send, and conditional processes. There is no separate primitive for cross-network communication — these are all encoded

by terms of agent migration and test-and-send. *Declarations* introduce new channels and agents. Finally, *Values* (i.e. entities that can be communicated on channels) include variables, agent and channel names, records of values, and constants (such as `String`, `Char`, `Int`, and `Bool`). Record values generalise tuple values (since the labels in a record are optional).

A program that uses only the Pict language is compiled to a subclass of the core language, and an original Pict backend can be chosen to translate it to a C program which is then compiled and executed on a single machine. See [Tur96] for a detailed description of generating C code from Pict core language.

7.2 Architecture of the Runtime System

Because much of the system functionality, including all distributed infrastructure, is written in Nomadic Pict, the runtime system has a very simple architecture. It consists of two layers, illustrated in Figure 7.2: the Virtual Machine and I/O server, above TCP. It is written in Objective Caml (O'Caml) [Ler95]. The implementation of the virtual machine builds on the simple abstract machine designed for Pict [Tur96].

7.2.1 Virtual Machine and Execution Fairness

The virtual machine maintains a state consisting of an *agent store* of agent closures; the agent names are partitioned into an *agent queue*, of agents waiting to be scheduled, and a *waiting room*, of agents whose process terms are all blocked. An agent closure consists of a *run queue*, of Nomadic π process/environment pairs waiting to be scheduled (round-robin), *channel queues* of terms that are blocked on internal or inter-agent communication, and an environment. Environments record bindings of variables to channels and basic values. The virtual machine executes in steps, in each of which the closure of the agent at the front of the agent queue is executed for a fixed number of interactions. This ensures fair execution of the fine-grain parallelism in the language. Agents with an empty run queue wait in the waiting room. They stay suspended until some other agent sends an output term to them. The only operations that remove agent closures from the agent store are `terminate` and `migrate`. A `migrate` moves an agent to a remote site. On the remote site, the agent is placed at the end of the agent queue.

The agent scheduler provides *fair* execution, guaranteeing that runnable concurrent processes of all non-terminating agents will eventually be executed, and that processes waiting to communicate on a channel will eventu-

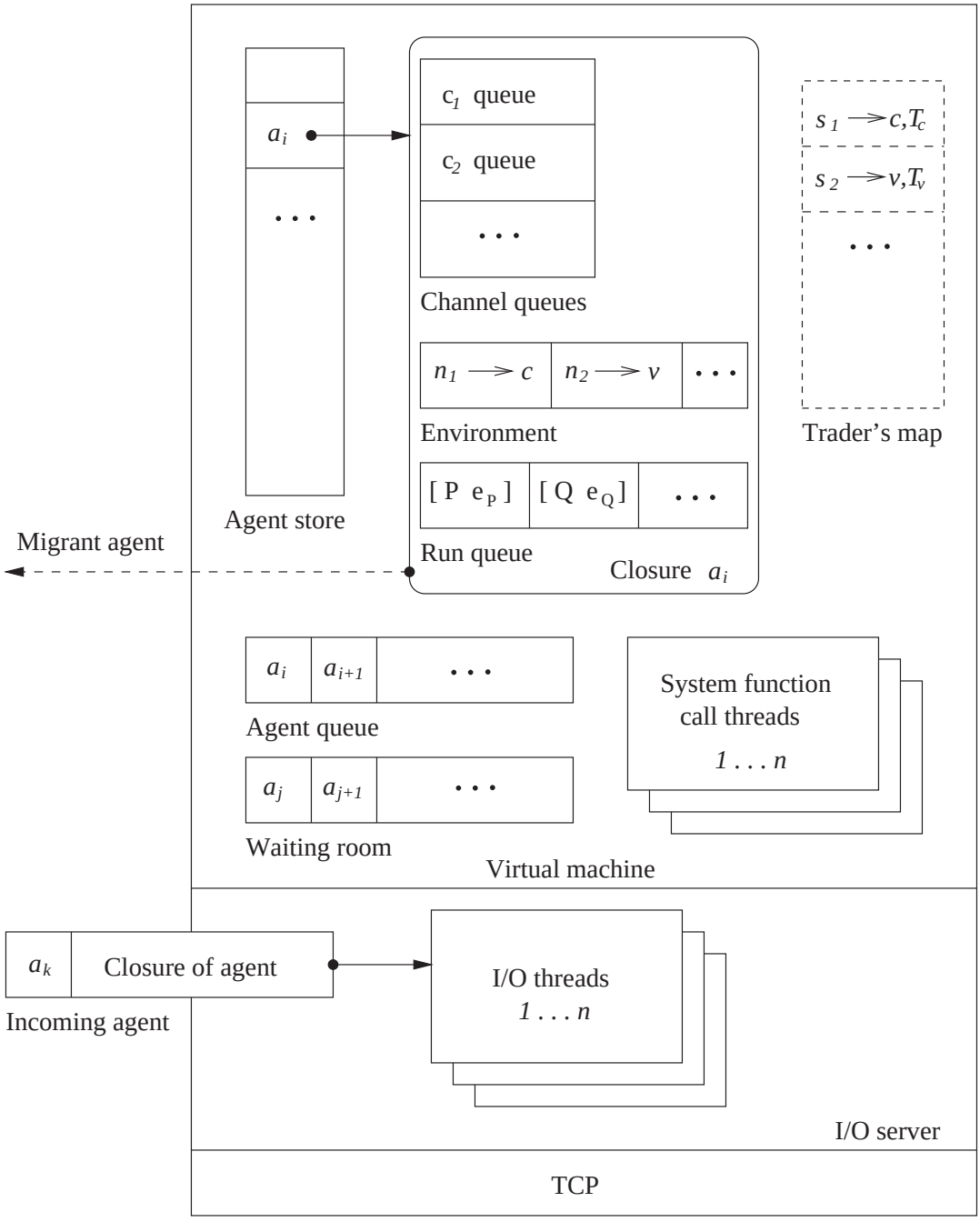


Figure 7.2: Architecture of the Nomadic Pict Runtime System. Abbreviations: a_i, agent IDs; c_i, channel IDs; n_i, names; v, values; P or Q, processes; e_i, local environments; s_i, strings

ally succeed (of course, if sufficient communication partners become available on a local or remote site). The implementation is deterministic and the language parallel operations are interleaved fairly. Non-deterministic behaviour will naturally arise because of time-dependent interactions between the abstract machine, the I/O server, and the system function calls to the operating system.

7.2.2 Interaction with an Operating System and User

For many library functions execution consists of one or more calls to corresponding Unix I/O routines. For example, processing `print!"foo"` involves an invocation of the O'Caml library call `output_string`. All interaction between the abstract behaviour of a Nomadic Pict library function and its environment (the operating system and user) occurs via invocations of *system function* calls. When a system function call reaches the front of the run queue some special processing takes place. The interpreter invokes the system function, passing all the function parameters and a result channel. The functions which can block for some time or can potentially never return (such as input from a user) will be executed within a separate execution thread, so that they do not block parallel computation. The agent operations `migrate` and `terminate` are special cases — they have to wait until all threads that execute system functions invoked inside the agent have terminated. If the system function returns any value, the Nomadic Pict program will receive it along the result channel.

7.2.3 I/O Server and Trader Service

The multithreaded I/O server receives incoming agents, consisting of an agent name and an agent closure; they are unmarshalled and placed in the agent store. Note that an agent closure contains the entire state of an agent, allowing agent execution to be resumed from the point where it was suspended. Agent communication uses standard network protocols (TCP in our first implementation). The runtime system does not support any reliable protocols that are tailored for agents, such as the Agent Transfer Protocol of [LA97]. Such protocols must be encoded explicitly in an infrastructure encoding – the key point in our experiments is to understand the dependencies between machines (both in the infrastructure and in application programs); we want to understand exactly how the system behaves under failure, not simply to make things that behave well under very partial failure. This is assisted by the purely local nature of the runtime system implementation.

The trader service offers two library functions `publish` and `subscribe` that can be used in programs which are executed separately in order to exchange names (such as channel and agent names), basic values (e.g. strings), and any complex values which can be sent along channels. The function `publish` takes as arguments a value to be published (which must be converted to a type `Dyn`) and a string keyword to identify the value. A program which wants to receive the value invokes a function `subscribe`, passing as arguments a string keyword and the current agent and site names. The function blocks until the value is available. The function `subscribe` returns a dynamic value which can be typechecked against expected types using **`typecase`**. If the dynamic typechecking succeeds basic values extracted from the dynamic value can be used in the program (e.g. for communication with the other program). When the runtime system starts up, the user has to specify an address for the runtime system selected to maintain the trader's map from strings to published names and values. The library functions `publish` and `subscribe`, written in Nomadic Pict, implement the whole distributed protocol which is necessary to contact the trading runtime system (so, the implementation of the runtime system remains purely local).

Chapter 8

Conclusions and Future Work

In this dissertation we have looked at mobile agents from the perspective of programming languages. We have shown that the π-calculus extended with primitives for agent creation, migration and location-aware communication can form a basis for distributed programming language design. Our experience suggests that high-level concurrent programming languages, which have a powerful type system and type inference, have a significant advantage over conventional imperative languages. This is simply because writing distributed applications using these languages is easier and less error-prone, and so can reduce costs of the product development cycle. While the technical arguments are convincing, they are not sufficient for most industrial companies, and several non-technical hurdles must be addressed along the way. The gap between the best that research has to offer and the current industrial standard is often too large, which results in some misconceptions (such as "recursive style of programming is hard to learn", etc.). There are, however, examples in which emerging application areas have allowed the gap to be bridged and old technology to be displaced. For example, Java (which shares some ideas with early predecessors, such as ML) has become popular with the emergence of the World-Wide Web and applets, and has managed to displace C++ in many areas. The Ericsson example shows that functional languages are already being chosen instead of C++, Java, or C in the development of some large industrial applications [Arm96].

Work Done within This Thesis Emerging Internet applications require new infrastructures (such as Mobile Agents, Mobile IP, Jini, TSpaces), above standard network protocols. The infrastructure algorithms (especially with mobility) are complex; they need languages that have clean semantics. We have focused on one problem, the design of communication primitives for agents to interact. Location-independent primitives can potentially simplify

application development but they need delicate distributed infrastructure which must be somewhat application-oriented. To facilitate experimentation, we have implemented Nomadic Pict, a prototype mobile agent programming language corresponding to our high- and low-level process calculi. The high-level language, with particular infrastructures for location-independent communication, is obtained by applying user-supplied translations into the low-level language. The full language available to the user remains very close to the process calculus presentation, and can be given rigorous semantics in a similar style. The operational semantics of the Nomadic calculi provides a precise and clear understanding of the algorithms' behaviour, aiding design, and ultimately, one may hope, supporting proofs of correctness and robustness (see below).

We have used our language to investigate the behaviour of many infrastructure algorithms in practice, and to assess the usefulness of our two-level architecture in applications. For example, we developed a disconnection-aware and scalable communication infrastructure, designed for the PA agent application (described in chapter 6). Our infrastructure allows disconnected operation of PA agents. The PA agent uses location-independent primitives to communicate with a name server and other PAs. In the low-level encoding of the infrastructure, partition from the network is made explicit. Upon reconnection, any pending communication is reconciled. A federated architecture of name servers allows agents to maintain efficient communication on changing between local networks. The algorithms comprise strategies such as caching and simple adaptive searching; they are highly concurrent.

In our experience with designing such algorithms we have found that the language provides a good level of abstraction at which potential problems (such as deadlocks and lost messages) can be seen rather clearly. The uniform treatment of concurrency and asynchronous messages both within agents and between machines is a significant gain.

All infrastructures whose translations are included in this thesis have been prototyped in Nomadic Pict. For testing purposes, we have also written many short example distributed programs that use message communication, e.g. *Dining Philosophers*. Efficiency of the program execution appeared satisfactory (including the distributed algorithms which communicate frequently), which is encouraging, considering that the runtime system implementation of Nomadic Pict has not been optimised.

Future Work Proposal An obvious area of future work includes the design of different infrastructure algorithms for different applications. The design of infrastructures for wide-area networks should explicitly address the

problem of administrative boundaries and firewalls (encoded as part of the Nomadic Pict translations). A simple infrastructure for the PA application should also be further refined, in order to allow tolerance of arbitrary kinds of fault in the system. There are also many interesting problems to solve in the area of the language design and theory. Below, we sketch some of these problems.

There is work underway to develop proof techniques from the theory of process calculi (such as observational equivalence) that could be used within the Nomadic π-calculus framework [Uny]. In particular, Unyapoth is proving formally the correctness of algorithms proposed in §4.2. An analogous work should be conducted on formal proofs of more complex algorithms expressed in Nomadic Pict, such as those described in chapter 6 (this will require to extend the proof techniques so as to support an input with timeout).

Our low-level language extends the compiler and abstract machine of Pict, a concurrent but not distributed language based on the π-calculus, to support our primitives for agent creation, migration, and location-dependent communication. Analogous extensions could be given for other concurrent uniprocessor programming languages, such as Amber [Car86], Concurrent ML [Rep93], and Concurrent Haskell [PGF96].

Our experience shows that the type system designed for Pict is able to catch a significant number of the most common errors in Nomadic Pict programs. Of course, that is not to say that there are no useful refinements one can make to the type system, and indeed we did not attempt, for example, to refine a subtyping system for locality enforcement of channel types. Another important area for further work is the development of an appropriate module system for Nomadic Pict (e.g. following work on Standard ML). A simple type-safe trader, which has been currently implemented for dynamic connection of executing programs, could then be extended so that agents could publish and subscribe whole modules (as in Facile). The mechanism should scale well to support a large number of sites, services, and agents.

Turning to semantics, some better notion of time than used in 2.2.3 must be introduced into the low-level calculus, to allow timeouts to be expressed, yet the semantics must be kept tractable, to allow robustness properties to be stated and proved. Failure semantics will require further investigation, especially in the context of observational equivalence. An important general question is about the sense in which the semantics of Nomadic Pict relates to the behaviour of the actual implementation. An operational model by Sewell [Sew97b] of the interactions between a Pict implementation of Pierce and Turner (considered as the abstract behaviour of a C program) and its environment (modelling an operating system and user) is one example of such an analysis, but there are many further potential refinements needed. The im-

plementation of Nomadic Pict is significantly more complex than Pict (there are interactions with the network communication protocols such as TCP, communication and site failures may happen, the system function calls can be executed as separate threads of control, etc.). Further work is therefore required to prove that our abstract machine is indeed correct.

In our language, we have used single messages for communication between agents. One might also consider other high-level communication primitives, such as location-independent multicast, events, and agent primitives, such as tree-structured agents. More speculatively, the two levels of abstraction that we have identified may be a useful basis for work on security properties of mobile agent infrastructures. However, we have neglected it so far as not being immediately related to the area of our investigation. To consider whether a distributed infrastructure for mobile agents is secure one must first be able to define it precisely, and have a clear understanding of how it is distributed on actual machines. Recent years have seen a lot of research in the area of security for mobile agents; some results apply directly to the π-calculus style of communication (e.g. [AFG99]).

Conclusion In conclusion, we believe that the Nomadic π-calculus presented here can be used as a simple theoretic foundation for agents which need to communicate while migrating. Moreover, the low-level primitives are directly implementable above standard network protocols, and the Nomadic Pict experiment proves that they can be efficiently incorporated into a real programming language design.

Appendix A

Syntax

This chapter describes the syntax of Nomadic Pict programs (for description of lexical rules and Pict primitives we use extracts from [PT97b], by courtesy of Benjamin Pierce).

A.1 Lexical Rules

Whitespace characters are space, newline, tab, and formfeed (control-L). Comments are bracketed by `{-` and `-}` and may be nested. A comment is equivalent to whitespace.

Integers are sequences of digits (negative integers start with a `-` character). Strings can be any sequence of characters and escape sequences enclosed in double-quotes. Sites can be any sequence of characters and escape sequences enclosed in double single-quote characters (''), to denote the Internet address, followed by a colon and integer, to denote a port number. The escape sequences `\"`, `\n`, and `\\` stand for the characters double-quote, newline, and backslash. The escape sequence `\ddd` (where `d` denotes a decimal digit) denotes the character with code `ddd` (codes outside the range `0..255` are illegal). Character constants consist of a single quote character (`'`), a character or escape sequence, and another single quote.

Alphanumeric identifiers begin with a symbol from the following set:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Subsequent symbols may contain the following characters in addition to those mentioned above:

```
0 1 2 3 4 5 6 7 8 9 '
```

Symbolic identifiers are non-empty sequences of symbols drawn from the following set:

```
~ * %  + - < > = & | @ $ , `
```

A.2 Reserved Words

The following symbols are reserved words:

`Agent`	`agent`	`and`	`Bool`	`ccode`	`Char`	`DEC`	`def`	`dynamic`	`else`
`ENCODE`	`false`	`if`	`iflocal`	`import`	`inline`	`Int`	`in`	`migrate`	`new`
`now`	`of`	`PROC`	`rec`	`run`	`Site`	`String`	`terminate`	`then`	`timeout`
`to`	`Top`	`true`	`TYPE`	`Type`	`type`	`typecase`	`val`	`switch`	`wait`
`where`	`with`	`@`	`^`	`\`	`/`	`.`	`;`	`:`	`=`
`|`	`!`	`#`	`?`	`?*`	`_`	`<`	`>`	`->`	`{`
`(`	`[`	`}`	`)`	`]`					

A.3 Concrete Syntax

For each syntactic form, we note whether it is part of the core language (C), the language for expressing encodings (T), a derived form (D), an optional type annotation that is filled in during type reconstruction if omitted by the programmer (R), or an extra-linguistic feature (E). Syntactic forms characteristic for the Nomadic Pict language are marked by n.

Compilation units

TopLevel	=	*Import* ... *Import Dec* ... *Dec*	*E*	Compilation unit
		Import ... *Import TopDec* ... *TopDec*	*En*	Compilation unit
Import	=	`import` *String*	*E*	Import statement

Top Declarations

TopDec	=	*Dec*		Declaration
		`ENCODE TYPE AGENT =` *Type*	*Tn*	Agent type
		`ENCODE TYPE SITE =` *Type*	*Tn*	Site type
		`ENCODE TYPE PROGRAM =` *Type*	*Tn*	Type of program parameters
		`ENCODE TYPE TOPLEVEL =` *Type*	*Tn*	Type of toplevel parameters
		`ENCODE PROC PROGRAM` *Id* `=` *Proc*	*Tn*	Program declaration
		`ENCODE DEC TOPLEVEL` *Id Id* `=` *Dec*	*Tn*	Toplevel declaration
		`ENCODE DEF` *Id Abs*	*Tn*	`def` *Id Abs*
		`{ agent` *Id* `=` *Id* `in` *Id* `}` *Id* `=` *Dec*	*Tn*	Agent creation
		`{ migrate to` *Id Id* `}` *Id* `=` *Dec*	*Tn*	Agent migration
		`{ <` *Id* `@` *Id* `>` *Id* `!` *Id* `}` *Id* `=` *Proc*	*Tn*	Output to agent on site
		`{ <` *Id* `>` *Id* `!` *Id* `}` *Id* `=` *Proc*	*Tn*	Output to adjacent agent
		`{ iflocal <` *Id* `>` *Id* `!` *Id* `then` *Proc* `else` *Proc* `}` *Id* `=` *Proc*	*Tn*	Test-and-send to agent
		`{` *Id* `@` *Id* `!` *Id* `}` *Id* `=` *Proc*	*Tn*	Location-independent output
		`{` *String Id* `in` *P* `}` *Id* `=` *Proc*	*Tn*	Macro-definition

Declarations

Dec	=	`new` *Id* `:` *Type*	*C*	Channel creation
		`val` *Pat* `=` *Val*	*D*	Value binding
		`run` *Proc*	*D*	Parallel process
		Val `;`	*D*	Sequential execution
		`inline def` *Id Abs*	*D*	Inlinable definition
		`def` Id_1 Abs_1 `and ... and` Id_n Abs_n	*C*	Recursive definition ($n \geq 1$)
		`type` *Id* `=` *Type*	*D*	Type abbreviation
		`type (` *Id* $KindedId_1 \ldots KindedId_n$ `) =` *Type*	*D*	Type operator abbrev ($n \geq 1$)
		`now (` *Id Flag* ... *Flag* `)`	*E*	Compiler directive
		`agent` Id_1 `=` $Proc_1$ `and ... and` Id_n `=` $Proc_n$	*Cn*	Agent creation ($n \geq 1$)
		`agent` Id_1 `=` $Proc_1$ `and ... and` Id_n `=` $Proc_n$ `in`	*Cn*	Agent creation ($n \geq 1$)
		`migrate to` *Val*	*Cn*	Migrate to site
		`do` *String Val*	*Tn*	Plug in macro-definition
		`do` *String Val* `in`	*Tn*	Plug in macro-definition
		`{` *Id Id* `}`	*Tn*	Plug in declaration
Flag	=	*Id*	*E*	Ordinary flag
		Int	*E*	Numeric flag
		String	*E*	String flag

Abstractions

Abs	=	*Pat* `=` *Proc*	*C*	Process abstraction
		`(` *Label FieldPat* ... *Label FieldPat* `)` *RType* `=` *Val*	*D*	Value abstraction

Patterns

Pat	=	*Id RType*	*C*	Variable pattern
		`[` *Label FieldPat* ... *Label FieldPat* `]`	*C*	Record pattern
		`(` `rec` *RType Pat* `)`	*C*	Rectype pattern
		`_` *RType*	*C*	Wildcard pattern
		Id RType `@` *Pat*	*C*	Layered pattern
		`{` *Id* `}`	*Tn*	Plug in pattern
FieldPat	=	*Pat*	*C*	Value field
		`#` *Id Constr*	*C*	Type field

Type constraints

Constr	=	⟨*empty*⟩	*D*	No constraint
		`<` *Type*	*C*	Subtype constraint
		`=` *Type*	*C*	Equality constraint

Processes

Proc	=	*Val* `!` *Val*	*C*	Output atom
		Val `?` *Abs*	*C*	Input prefix
		Val `?*` *Abs*	*Cn*	Replicated input
		`wait` *Val* `?` *Abs* `timeout` *Val* `->` *Proc*	*Cn*	Input with timeout
		`<` *Val* `@` *Val* `>` *Val* `!` *Val*	*Dn*	Output to agent on site
		`<` *Val* `>` *Val* `!` *Val*	*Dn*	Output to adjacent agent
		`iflocal <` *Val* `>` *Val* `!` *Val* `then` *Proc* `else` *Proc*	*Cn*	Test-and-send to agent
		Val `@` *Val* `!` *Val*	*Dn*	Location-independend output
		`(` `)`	*C*	Null process
		`(` $Proc_1$ `|` `...` `|` $Proc_n$ `)`	*C*	Parallel composition ($n \geq 2$)
		`(` $Dec_1 \ldots Dec_n$ *Proc* `)`	*C*	Local declarations ($n \geq 1$)
		`if` *Val* `then` *Proc* `else` *Proc*	*C*	Conditional
		`terminate`	*Cn*	Terminate agent
		`typecase` *Val* `of` Pat_1 `->` $Proc_1$ `...` Pat_n `->` $Proc_n$ `else` $Proc_{n+1}$	*Dn*	Type matching ($n \geq 1$)
		`switch` *RType Val* `of` `(` `{` Id_1 `>` Pat_1 `}` `->` $Proc_1$ `...` `{` Id_n `>` Pat_n `}` `->` $Proc_n$ `)`	*Dn*	Variant matching ($n \geq 1$)
		`{` *Id Id* `}`	*Tn*	Plug in process

Values

Val	=	*Const*	*C*	Constant
		Path	*C*	Path
		`\` *Abs*	*D*	Process abstraction
		`[` *Label FieldVal* ... *Label FieldVal* `]`	*C*	Record
		`if` *RType Val* `then` *Val* `else` *Val*	*D*	Conditional
		`(` *Val RType* `with` *Label FieldVal* ... *Label FieldVal* `)`	*D*	Field extension
		`(` *Val RType* `where` *Label FieldVal* ... *Label FieldVal* `)`	*D*	Field override
		`(` *RType Val Label FieldVal* ... *Label FieldVal* `)`	*D*	Application
		`(` *Val* `>` $Val_1 \ldots Val_n$ `)`	*D*	Right-assoc application ($n \geq 2$)
		`(` *Val* `<` $Val_1 \ldots Val_n$ `)`	*D*	Left-assoc application ($n \geq 2$)
		`(` `rec` *RType Val* `)`	*C*	Rectype value
		`(` $Dec_1 \ldots Dec_n$ *Val* `)`	*D*	Local declarations ($n \geq 1$)
		`(` `ccode` *Int Id String FieldVal* ... *FieldVal* `)`	*E*	Inline C code (Pict)
		`(` `ccode` *Int Id String FieldVal* ... *FieldVal* `)`	*En*	Call system function
		`(` `dynamic` *Val RType* `)`	*Dn*	Typed value
		`{` *Id* `>` *Val* `}`	*Dn*	Variant
		`typecase` *RType Val* `of` Pat_1 `->` Val_1 `...` Pat_n `->` Val_n `else` Val_{n+1}	*Dn*	Type matching ($n \geq 1$)
		`switch` *RType Val* `of` `(` `{` Id_1 `>` Pat_1 `}` `->` Val_1 `...` `{` Id_n `>` Pat_n `}` `->` Val_n `)`	*Dn*	Variant matching ($n \geq 1$)
		`{` *Id* `}`	*Tn*	Plug in value

Path	=	*Id*	*C*	Variable
		Path `.` *Id*	*C*	Record field projection
FieldVal	=	*Val*	*C*	Value field
		`#` *Type*	*C*	Type field
Const	=	*String*	*C*	String constant
		Char	*C*	Character constant
		Int	*C*	Integer constant
		`true`	*C*	Boolean constant
		`false`	*C*	Boolean constant

Types

Type	=	`Top`	*C*	Top type
		Id	*C*	Type identifier
		`^` *Type*	*C*	Input/output channel
		`!` *Type*	*C*	Output channel
		`/` *Type*	*C*	Responsive output channel
		`?` *Type*	*C*	Input channel
		`Int`	*C*	Integer type
		`Char`	*C*	Character type
		`Bool`	*C*	Boolean type
		`String`	*C*	String type
		`[` *Label FieldType* ... *Label FieldType* `]`	*C*	Record type
		`(` *Type* `with` *Label FieldType* ... *Label FieldType* `)`	*D*	Record extension
		`(` *Type* `where` *Label FieldType* ... *Label FieldType* `)`	*D*	Record field override
		`\` $KindedId_1 \ldots KindedId_n$ `=` *Type*	*C*	Type operator ($n \geq 1$)
		`(` *Type* $Type_1 \ldots Type_n$ `)`	*C*	Type application ($n \geq 1$)
		`(` `rec` *KindedId* `=` *Type* `)`	*C*	Recursive type
		`Agent`	*Cn*	Agent type
		`Site`	*Dn*	Site type
		`Dyn`	*Dn*	Dynamic type
		`{` Id_1 `>` $Type_1$ `...` Id_n `>` $Type_n$ `}`	*Dn*	Variant type
		`{` *Id* `}`	*Tn*	Plug in type
FieldType	=	*Type*	*C*	Value field
		`#` *Id Constr*	*C*	Type field
RType	=	$\langle empty \rangle$	*R*	Omitted type annotation
		`:` *Type*	*C*	Explicit type annotation

Kinds

Kind	=	`(` $Kind_1 \ldots Kind_n$ `->` *Kind* `)`	*C*	Operator kind ($n \geq 1$)
		`Type`	*C*	Type kind
KindedId	=	*Id* `:` *Kind*	*C*	Explicitly-kinded identifier
		Id	*D*	Implicitly-kinded identifier

Labels

Label	=	⟨*empty*⟩	*C*	Anonymous label
		Id =	*C*	Explicit label

Bibliography

[ABB+86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings USENIX 1986 Summer Technical Conference and Exhibition, Atlanta, USA*, pages 93–112, June 1986.

[ACF87] Yeshayahu Artsy, Hung-Yang Chang, and Raphael Finkel. Interprocess communication in Charlotte. *IEEE Software*, 4(1):22–28, January 1987.

[ACS98] R. M. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous pi-calculus. *Journal of Theoretical Computer Science*, 195(2):291–324, March 1998. Also appeared in *Proceedings of CONCUR '96: Concurrency Theory (7th International Conference, Pisa, Italy, 26-29 August, 1996)*, Springer LNCS 1119.

[AD99] Thomas Arts and Mads Dam. Verifying a distributed database lookup manager written in Erlang. In *Proceedings of the World Congress on Formal Methods, Toulouse, France*, September 1999.

[AFG99] Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure communications processing for distributed languages. In *IEEE Symposium on Security and Privacy*, pages 74–88, May 1999.

[ALT95] Roberto M. Amadio, Lone Leth, and Bent Thomsen. From a concurrent λ-calculus to the π-calculus. In Horst Reichel, editor, *Fundamentals of Computation Theory (FCT '95, 10th International Conference, Dresden, Germany)*, volume 965 of *Lecture Notes in Computer Science*, pages 106–115. Springer Verlag, 1995. Full version as Technical Report ECRC-95-18, European Computer-Industry Research Center, Munich, Germany, 1995.

[AM91] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In J. Małuszyński and M. Wirsing, editors, *Proceedings of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming, PLILP '91, Passau, Germany*, volume 528 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, August 1991.

[Ama] Roberto M. Amadio. On modelling mobility. To appear in the *Journal of Theoretical Computer Science*.

[Ama94] Roberto M. Amadio. Translating core Facile. Technical Report ECRC-TR-3-94, European Computer-Industry Research Center, Munich, Germany, 1994. Also available as a technical report from CRIN(CNRS)-Inria (Nancy).

[Ama97] Roberto M. Amadio. An asynchronous model of locality, failure, and process mobility. In D. Garlan and D. Le Metayer, editors, *Coordination Languages and Models (Proceedings of COORDINATION '97, Berlin, Germany)*, volume 1282 of *Lecture Notes in Computer Science*. Springer Verlag, 1997. Extended version as Rapport de Recherche RR-3109, INRIA Sophia-Antipolis, 1997.

[And92] F. Andersen. *A Theorem Prover for UNITY in Higher Order Logic*. PhD thesis, Technical University of Denmark, March 1992.

[AO98] Y. Aridor and M. Oshima. Infrastructure for mobile agents: Requirements and design. In K. Rothermel and F. Hohl, editors, *Proc. 2nd Int. Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 38–49. Springer-Verlag, 1998.

[AP89] B. Awerbuch and D. Peleg. Online tracking of mobile users. Technical Memo MIT/LCS/TM-410, Massachusetts Institute of Technology, Laboratory for Computer Science, July 1989.

[AP94] Roberto M. Amadio and Sanjiva Prasad. Localities and failures. In Pazhamaneri S. Thiagarajan, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 880 of *Lecture Notes in Computer Science*, pages 205–216. Springer Verlag, 1994. Full version as Technical Report ECRC-94-18, European Computer-Industry Research Center, Munich, Germany, 1994.

[AP95] Baruch Awerbuch and David Peleg. Online tracking of mobile users. *Journal of the ACM*, 42(5):1021–1058, September 1995. A shorter version appeared as Technical Memo MIT/LCS/TM-410, Massachusetts Institute of Technology, Laboratory for Computer Science, July 1989.

[AP98] Roberto M. Amadio and Sanjiva Prasad. Modelling IP mobility. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR '98: Concurrency Theory (9th International Conference, Nice, France)*, volume 1466 of *Lecture Notes in Computer Science*, pages 301–316. Springer Verlag, September 1998.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[Arm96] Joe Armstrong. Erlang - a survey of the language and its industrial applications. In *Proceedings of the 9th Exhibitions and Symposium on Industrial Applications of Prolog, Hino, Tokyo, Japan*, 1996.

[AWO+99] Ken Arnold, Ann Wollrath, Bryan O'Sullivan, Robert Scheifler, and Jim Waldo. *The Jini specification*. Addison-Wesley, Reading, MA, USA, 1999.

[AWWV96] J. L. Armstrong, M. C. Williams, C. Wikström, and S. R. Virding. *Concurrent Programming in Erlang*. Prentice Hall, 2nd edition, 1996.

[BB92] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[BBH97] J. Bacon, J. Bates, and D. Halls. Location-oriented multimedia. *IEEE Personal Computer*, 4(5):48–57, October 1997.

[BGW93] Amnon Barak, Shai Guday, and Richard G. Wheeler. *The MOSIX distributed operating system: Load balancing for UNIX*, volume 672 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[BHB97] John Bates, David Halls, and Jean Bacon. Middleware support for mobile multimedia applications. *ICL Systems Journal*, 12(2):289–314, November 1997.

[BHDH98] Michael Bursell, Richard Hayton, Douglas Donaldson, and Andrew Herbert. A Mobile Object Workbench. In Kurt Rothermel and Fritz Hohl, editors, *Proceedings of the 2nd International Workshop on Mobile Agents (MA '98)*, volume 1477 of *Lecture Notes in Computer Science*, pages 136–147. Springer-Verlag, September 1998.

[BJ87] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. *Proc 11th ACM Symposium on OS Principles, Austin, TX, USA*, pages 123–138, November 1987.

[Bla99] Andrew P. Black. Object-oriented programming: Regaining the excitement. In R. Guerraoui, editor, *Proceedings ECOOP '99 (Lisbon, Portugal, June 1999)*, volume 1628 of *LNCS*, pages 519–528. Springer-Verlag, 1999.

[BNOW95] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. *Software–Practice and Experience*, 25(S4):87–130, December 1995. Also available as Digital Systems Research Center Research Report 115.

[Bou92] Gérard Boudol. Asynchrony and the pi-calculus. Technical Report RR-1702, Inria, Institut National de Recherche en Informatique et en Automatique, 1992.

[BPR98] M. Baldi, G. P. Picco, and F. Risso. Designing a videoconference system for active networks. In K. Rothermel and F. Hohl, editors, *Proc. 2nd Int. Workshop on Mobile Agents (MA98), Stuttgart, Germany, October 1998*, volume 1477 of *Lecture Notes in Computer Science*, pages 273–284. Springer-Verlag, 1998.

[BvST99] G. Ballintijn, M. van Steen, and A.S. Tanenbaum. Lightweight crash recovery in a wide-area location service. In *Proceedings of the 12th International Conference on Parallel and Distributed Computing Systems, Fort Lauderdale, Florida*, August 1999.

[Car86] L. Cardelli. Amber. In G. Cousineau, P.-L. Curien, and B. Robinet, editors, *Combinators and Functional Programming Languages*, volume 242 of *Lecture Notes in Computer Science*, pages 21–47. Springer-Verlag, 1986.

[Car95] Luca Cardelli. A language with distributed scope. In ACM, editor, *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: San Francisco, California, January 22–25, 1995*, pages 286–297, New York, NY, USA, 1995. ACM Press.

[CDK94] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, Reading, MA, USA, second edition, 1994.

[CF99] Sylvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications/Third International Symposium on Mobile Agents (ASA/MA '99)*, Palm Springs, CA, USA, October 1999.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS '98)*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, 1998.

[Che88] David Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[Che98] B. Chetali. Formal verification of concurrent programs using the Larch Prover. *IEEE Transactions on Software Engineering*, 24(1):46–62, 1998.

[CHK97] D. Chess, C. Harrison, and A. Kershenbaum. Mobile agents: Are they a good idea? In *Mobile Object Systems – Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 25–48, 1997.

[Chu32] A. Church. A set of postulates for the foundations of logic. *Ann. of Math.*, 33:346–366, 1932.

[Chu41] A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[CJ97] R. Chow and T. Johnson. *Distributed Operating Systems and Algorithms*. Addison-Wesley, 1997.

[CLZ98] G. Cabri, L. Leonardi, and F. Zambonelli. How to coordinate Internet applications based on mobile agents. In *Proc. 7th IEEE Workshops on Enablings Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 104–109, Stanford, CA, USA, June 1998. IEEE Computer Society Press.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, New York, NY, 1988.

[CS87] Berardo Costa and Colin Stirling. Weak and strong fairness in CCS. *Information and Computation*, 73(3):207–244, June 1987.

[DH98] M. J. Demmer and M. P. Herlihy. The arrow distributed directory protocol. In *Proceedings of 12th International Symposium on Distributed Computing (DISC '98), Andros, Greece, September 1998*, volume 1499 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[DO91] Fred Douglis and John Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software – Practice and Experience*, 21(8):757–785, August 1991.

[EE98] Guy Eddon and Henry Eddon. *Inside Distributed COM*. Microsoft Programming Series. Microsoft Press, Redmond, WA, 1998.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In ACM, editor, *Conference record of POPL '96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: St. Petersburg Beach, Florida, 21–24 January 1996*, pages 372–385, New York, USA, 1996. ACM Press.

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proceedings of CONCUR '96. LNCS 1119*, pages 406–421. Springer-Verlag, August 1996.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, 1985.

[Fow85] Robert J. Fowler. Decentralized object finding using forwarding addresses. Technical Report 85-12-1, Department of Computer Science, University of Washington, Seattle, WA, USA, December 1985.

[GG91] Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Technical Report 82, Digital Equipment Corporation, Systems Research Centre, December 1991.

[GJS97] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1997.

[GKN+96] Robert Gray, David Kotz, Saurab Nog, Daniela Rus, and George Cybenko. Mobile agents for mobile computing. Technical Report PCS-TR96-285, Dept. of Computer Science, Dartmouth College, May 1996.

[GKN+97] Robert Gray, David Kotz, Saurab Nog, Daniela Rus, and George Cybenko. Mobile agents: The next generation in distributed computing. In *Proceedings of the Second Aizu International Symposium on Parallel Algorithms/Architectures Synthesis (pAs '97)*, pages 8–24, Fukushima, Japan, March 1997. IEEE Computer Society Press.

[GL] Stephen J. Garland and Nancy A. Lynch. Using I/O automata for developing distributed systems. To appear in *Foundations of Component Based Systems*. Cambridge University Press.

[GL98] Stephen J. Garland and Nancy A. Lynch. The IOA language and toolset: Support for designing, analyzing, and building distributed systems. Technical Report MIT/LCS/TR-762, Laboratory for Computer Science, Massachusetts Institute of Technology, August 1998. (Original version: September 25, 1997).

[Gla98] G. Glass. ObjectSpace Voyager — the agent ORB for Java. *Lecture Notes in Computer Science*, 1368:38–48, 1998.

[GMP89] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. FACILE: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, April 1989. Also in *TAPSOFT '89*, ed. J. Diaz and F. Orejas, LNCS 352, pages 184-209. Springer-Verlag, 1989.

[Gra95] Robert S. Gray. Agent Tcl: A Transportable Agent System. In Tim Finin and James Mayfield, editors, *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM '95), Baltimore, MD, USA*, 1995.

[GS96] R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. In *Reliable Software Technologies - Ada-Europe '96*, LNCS 1088, pages 38–57. Springer-Verlag, June 1996.

[Hay98] Mark Hayden. The Ensemble system. Technical Report TR98-1662, Cornell University, Computer Science, January 8, 1998.

[HH94] Andy Harter and Andy Hopper. A distributed location system for the active office. *IEEE Network*, 8(1):62–70, January 1994. See also AT & T Lab. Cambridge Technical Report 94.1.

[HLP98] Robert Harper, Peter Lee, and Frank Pfenning. The Fox project: Advanced language technology for extensible systems. Technical Report CMU-CS-98-107, Carnegie Mellon University, January 1998.

[Hoa78a] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978. See corrigendum [Hoa78b].

[Hoa78b] C. A. R. Hoare. Corrigendum: "Communicating sequential processes". *Communications of the ACM*, 21(11):958–958, November 1978. See [Hoa78a].

[HR98a] Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. In *Workshop on High-Level Concurrent Languages*, 1998. Full version as University of Sussex technical report CSTR 98/02.

[HR98b] Matthew Hennessy and James Riely. Type-safe execution of mobile agents in anonymous networks. In *Proceedings of ECOOP '98 Workshop on Mobile Object Systems*, 1998. Full version as University of Sussex technical report CSTR 98/03.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '91)*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, 1991.

[HT94] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Computer Science Department, May 1994.

[IDJ91] John Ioannidis, Dan Duchamp, and Gerald Q. Maguire Jr. IP-based protocols for mobile internetworking. In *Proceedings of the 1991 ACM SIGCOMM Symposium*, pages 235–245, September 1991.

[IJ93] John Ioannidis and Gerald Q. Maguire Jr. The design and implementation of mobile internetworking architecture. In *USENIX Winter 1993 Conference*, pages 491–502, San Diego, CA, USA, January 1993.

[JJ99] Kjetil Jacobsen and Dag Johansen. Ubiquitous Devices United: Enabling Distributed Computing Through Mobile Code. In *Proceedings of the Symposium on Applied Computing (ACM SAC '99)*, February 1999.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[JvRS95] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. Operating system support for mobile agents. In *Proceedings of the Fifth Workshop Hot Topics in Operating Systems (HotOS '95)*, pages 42–45, Washington, USA, May 1995.

[JZ88] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using asynchronous message logging and checkpointing. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 171–181, Toronto, Ontario, Canada, August 1988.

[Kal94] Markus Kaltenbach. Model checking for UNITY. Technical Report CS-TR-94-31, University of Texas, Austin, December 1994.

[KG99] David Kotz and Robert S. Gray. Mobile agents and the future of the Internet. *ACM Operating Systems Review*, 33(3):7–13, August 1999.

[Kir99] Dilsun Kirli. A static type system for detecting potentially transmissible functions. In *Proceedings of the 5th ECOOP Workshop on Mobile Object Systems (MOS '99)*, Lisbon, Portugal, June 1999.

[Kis93] James J. Kistler. *Disconnected Operation in a Distributed File System*. PhD thesis, Carnegie Mellon University, May 1993.

[Kna95] Frederick Knabe. *Language Support for Mobile Agents*. PhD thesis, Carnegie Mellon University, December 1995.

[Kri96] P. Krishna. *Performance Issues in Mobile Wireless Networks*. PhD thesis, Texas A&M University, August 1996.

[KSMD99] Idit Keidar, Jeremy Sussman, Keith Marzullo, and Danny Dolev. A client-server oriented algorithm for virtually synchronous group membership in WANs. Technical Report CS1999-0623, University of California, San Diego, Computer Science and Engineering, July 1999.

[LA97] Danny B. Lange and Yariv Aridor. *Agent Transfer Protocol – ATP/0.1*. IBM Tokyo Research Laboratory, March 1997.

[Lab98] Ericsson Computer Science Laboratory. The birthplace of Erlang. Open-source Erlang available from `http://www.erlang.org/`, 1998.

[Ler95] Xavier Leroy. Le système Caml Special Light: modules et compilation efficace en Caml. Technical Report RR-2721, Inria, Institut National de Recherche en Informatique et en Automatique, 1995.

[LOKK97] Danny B. Lange, Mitsuru Oshima, Gunter Karjoth, and Kosaka Kazuya. Aglets: Programming mobile agents in Java. In T. Masuda, Y. Masunaga, and M. Tsukamoto, editors, *Proceedings of Worldwide Computing and Its Applications (WWCA '97)*, volume 1274 of *Lecture Notes in Computer Science*, pages 253–266, Tsukuba, Japan, March 1997.

[LS92] Michael Litzkow and Marvin Solomon. Supporting checkpointing and process migration outside the UNIX kernel. In USENIX Association, editor, *Proceedings of the Winter 1992 USENIX Conference: January 20 — January 24, 1992, San Francisco, California*, pages 283–290, 1992.

[LSN96] Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Using metaobjects to model concurrent objects with PICT. In *Proceedings of Langages et Modèles à Objects*, pages 1–12, Leysin, October 1996.

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In Fred B. Schneider, editor, *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, Vancouver, BC, Canada, August 1987. ACM Press.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands. Also appeared as Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology, November 1988.

[LV94] N. Lynch and F. Vaandrager. Forward and backward simulations part I: Untimed systems (replaces TM-486). Technical Memo MIT/LCS/TM-486b, Massachusetts Institute of Technology, Laboratory for Computer Science, October 1994.

[LWG98] Ulana Legedza, David J. Wetherall, and John Guttag. Improving the performance of distributed applications using active networks. In *Proceedings of the IEEE INFOCOM '98, San Francisco, CA, USA*, April 1998.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, USA, 1997.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, 1996.

[MBB+98] D. Milojičić, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran, and J. White. MASIF: The OMG Mobile Agent System Interoperability Facility. In K. Rothermel and F. Hohl, editors, *Proc. 2nd Int. Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 50–67, Stuttgart, Germany, 1998. Springer-Verlag, Berlin.

[MCG+99] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. Talx86: A realistic type assembly language. In *ACM SIGPLAN Workshop on Compiler Support for System Software, Atlanta, GA, USA*, May 1999.

[MDW99] Dejan Milojičić, Frederick Douglis, and Richard Wheeler, editors. *Mobility: Processes, Computers, and Agents*. Addison-Wesley, Reading, MA, USA, 1999.

[Mil84] Robin Milner. A proposal for Standard ML. *Polymorphism: The ML/LCF/Hope Newsletter*, 1(3), January 1984. Also appeared in *Proceedings of the ACM Symposium on Lisp and Functional Programming, Austin, Texas, USA, August 1984*, pages 184-197, and as Technical Report CSR-157-83, University of Edinburgh, 1983.

[Mil89] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[Mil91] Robin Milner. The polyadic π-calculus: A tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, October 1991. Appeared in *Proceedings of the International Summer School on Logic and Algebra of*

Specification, Marktoberdorf, August 1991. Reprinted in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer, and H. Schwichtenberg, Springer-Verlag, 1993.

[Mil92] Robin Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992. Previous version as Rapport de Recherche 1154, INRIA Sophia-Antipolis, 1990, and in *Proceedings of ICALP '91*, LNCS 443.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, May 1999.

[ML98] Luc Maranget and Fabrice Le Fessant. Compiling join-patterns. In Uwe Nestmann and Benjamin C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages (Nice, France, September 12, 1998)*, volume 16.3 of *Electronic Notes in Theoretical Computer Science (ENTCS)*. Elsevier Science Publishers, 1998.

[MLC98] Dejan S. Milojičić, William LaForge, and Deepika Chauhan. Mobile Objects and Agents (MOA). In *Proceedings of the 4th Conference on Object-Oriented Technologies and Systems (COOTS '98)*, pages 179–194. USENIX Association, April 1998.

[Mor99] Luc Moreau. Distributed directory service and message router for mobile agents. Technical Report ECSTR M99/3, University of Southampton, 1999.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100(1):1–77, 1992.

[MR97] P. J. McCann and G.-C. Roman. Mobile UNITY coordination constructs applied to packet forwarding for mobile hosts. In *Coordination Languages and Models, Berlin, 1997*, volume 1282 of *Lecture Notes in Computer Science*, 1997.

[MR98] P. J. McCann and G.-C. Roman. Compositional Programming Abstractions for Mobile Computing. *IEEE Transactions on Software Engineering*, 24(2):97–110, 1998.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[MV88] Sape J. Mullender and Paul M. B. Vitányi. Distributed match-making. *Algorithmica*, 3:367–391, 1988.

[MvR92] S. J. Mullender and G. van Rossum. Amoeba: A distributed operating system for the 1990s. In Akkihebbal L. Ananda and Balasubramaniam Srinivasan, editors, *Distributed Computing Systems: Concepts and Structures*, pages 201–212. IEEE Computer Society Press, Los Alamos, CA, 1992.

[MZDG93] Dejan Milojičić, Wolfgang Zint, Andreas Dangel, and Peter Giese. Task migration on the top of the Mach microkernel. In USENIX, editor, *Proceedings of the USENIX Mach III Symposium, April 19–21, 1993, Santa Fe, New Mexico, USA*, pages 273–289, April 1993.

[Nee89] R. M. Needham. Names. In S. Mullender, editor, *Distributed Systems*, pages 89–101. Addison-Wesley, 1989.

[NTA96] Naimi, Trehel, and Arnold. A log(N) distributed mutual exclusion algorithm based on path reversal. *JPDC: Journal of Parallel and Distributed Computing*, 34, 1996.

[Obj97] ObjectSpace. Voyager core package technology overview. Available from `http://www.objectspace.com/`, 1997.

[OCD+87] John Ousterhout, Andrew Cherenson, Fred Douglis, Michael Nelson, and Brent Welch. The Sprite Network Operating System. Technical Report UCB/CSD/ 87/359, Computer Science Division (EECS), University of California, Berkeley, June 1987.

[OMG91] Object Management Group, Inc. *The Common Object Request Broker: Architecture and Specification*, OMG document number 91.12.1, 1.1 edition, December 1991.

[OPL94] Tim Oates, M. V. Nagendra Prasad, and Victor R. Lesser. Cooperative information gathering: A distributed problem solving approach. Technical Report 94-66, Department of Computer Science, University of Massachusetts, September 1994.

[Pau94] L. C. Paulson. Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*, 828, 1994. Springer-Verlag.

[PGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In ACM, editor, *Conference Record of POPL '96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: St. Petersburg Beach, Florida, 21–24 January 1996*, pages 295–308, New York, NY, USA, 1996. ACM Press.

[PGM90] Sanjiva Prasad, Alessandro Giacalone, and Prateek Mishra. Operational and algebraic semantics for Facile: A symmetric integration of concurrent and functional programming. In Michael S. Paterson, editor, *Automata, Languages and Programming, 17th International Colloquium*, volume 443 of *Lecture Notes in Computer Science*, pages 765–780. Springer-Verlag, 1990.

[Pie97] Benjamin C. Pierce. Foundational calculi for programming languages. In *Allen B. Tucker, Jr. (Editor-in-Chief), The Computer Science and Engineering Handbook*. CRC Press, in cooperation with ACM, 1997.

[PMR99] Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. LIME: Linda meets mobility. In *Proceedings of the 21st International Conference on Software Engineering*, pages 368–377. ACM Press, May 1999.

[PRM97] Gian Pietro Picco, Gruia-Catalin Roman, and Peter J. McCann. Expressing code mobility in mobile UNITY. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 500–518. LNCS 1013, Springer–Verlag, September 1997.

[PT95] Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Proceedings of the Theory and Practice of Parallel Programming (TPPP, Sendai, Japan, 1994)*, volume 907 of *Lecture Notes in Computer Science*, pages 187–215. Springer Verlag, 1995.

[PT97a] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, MIT Press.

[PT97b] Benjamin C. Pierce and David N. Turner. *Pict Language Definition*, 1997. Available electronically as part of the Pict distribution.

[PW85] G. J. Popek and B. J. Walker. *The Locus Distributed System Architecture*. MIT Press, Cambridge, Mass., 1985.

[RASS97] Mudumbai Ranganathan, Anurag Acharya, Shamik Sharma, and Joel Saltz. Network-aware mobile programs. In *Proceedings of the USENIX 1997 Annual Technical Conference*, Anaheim, CA, USA, January 1997.

[RBM+94] Tristan Richardson, Frazer Bennett, Glenford Mapp, Andy Harter, and Andy Hopper. Teleporting – making applications mobile. In *Proceedings of ACM CSCW '94 Conference on Computer-Supported Cooperative Work*, Formal Video Program: Prototypes and Enabling Technologies, pages 9–10, 1994.

[Rep93] J. H. Reppy. Concurrent ML: Design, application and semantics. In P. E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, pages 165–198. Springer-Verlag, 1993.

[RH97] James Riely and Matthew Hennessy. Distributed processes and location failures. In *Proceedings of ICALP '97. LNCS 1256*, pages 471–481. Springer Verlag, July 1997.

[RH98] James Riely and Matthew Hennessy. A typed language for distributed mobile processes. In *Proceedings of the 25th POPL*, January 1998.

[RMH98] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. Technical Report TR98-1687, Cornell University, Computer Science, May 1998.

[RMP97] G.-C. Roman, P. McCann, and J. Plun. Mobile UNITY: Reasoning and Specification in Mobile Computing. *ACM Transactions on Software Engineering and Methodology*, 6(3):250–282, June 1997.

[RRD95] S. Rangarajan, K. Ratnam, and A. T. Dahbura. A fault-tolerant protocol for location directory maintenance in mobile networks. In *The Twenty-Fifth International Symposium on Fault-Tolerant Computing (FTCS '95)*, pages 164–173, Los Alamitos, June 1995. IEEE Computer Society Press.

[San] Davide Sangiorgi. Asynchronous process calculi: The first-order and higher-order paradigms (Tutorial). To appear in *Theoretical Computer Science*.

[San96] Davide Sangiorgi. A theory of bisimulation for the π-calculus. *Acta Informatica*, 33:69–97, 1996. Earlier version published as Report ECS-LFCS-93-270, University of Edinburgh. An extended abstract appeared in the *Proceedings of CONCUR '93*, LNCS 715.

[San98] Davide Sangiorgi. An interpretation of typed objects into typed pi-calculus. *Information and Computation*, 143(1), 1998.

[San99] Davide Sangiorgi. Interpreting functions as pi-calculus processes: A tutorial. Also appeared as INRIA Technical Report RR-3470, August 1998; revised February 1999.

[SBH96] M. Strasser, J. Baumann, and F. Hohl. Mole – A Java based mobile agent system. In *Proceedings of the 2nd ECOOP Workshop on Mobile Object Systems*, University of Linz, Austria, July 1996.

[Sew] Peter Sewell. A brief introduction to applied Pi. Lecture notes for the *MATH-FIT Instructional Meeting on Recent Advances in Semantics and Types for Concurrency: Theory and Practice, Imperial College, July 1998.*

[Sew97a] Peter Sewell. Global/local subtyping for a distributed π-calculus. Technical Report 435, University of Cambridge, August 1997.

[Sew97b] Peter Sewell. On implementations and semantics of a concurrent programming language. In Antoni Mazurkiewicz and Józef Winkowski, editors, *Proceedings of CONCUR '97. LNCS 1243*, pages 391–405. Springer Verlag, 1997.

[Sew98] Peter Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Proceedings of ICALP '98, LNCS 1443*, pages 695–706, 1998.

[SJ95] Bjarne Steensgaard and Eric Jul. Object and native code thread mobility among heterogeneous computers. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 68–78, December 1995.

[SWP98] Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location independence for mobile agents. In *Proceedings of ICCL '98 Workshop on Internet Programming Languages, Chicago, USA*, May 1998. It is largely superseded by [SWP99].

[SWP99] Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location-independent communication for mobile agents: A two-level architecture. In Henri E. Bal, Boumediene Belkhouche, and Luca Cardelli, editors, *Internet Programming Languages (ICCL '98 Workshop, Chicago, USA, May 1998)*, volume 1686 of *Lecture Notes in Computer Science*, pages 1–31. Springer, 1999. Also appeared as Technical Report 462, Computer Laboratory, University of Cambridge, April 1999.

[SY97] Tatsurou Sekiguchi and Akinori Yonezawa. A calculus with code mobility. In Howard Bowman and John Derrick, editors, *Formal Methods for Open Object-based Distributed Systems (Proceedings of FMOODS '97)*, pages 21–36. IFIP, Chapman and Hall, July 1997.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, New Jersey, 1992.

[Tho93] Bent Thomsen. Plain CHOCS. A second generation calculus for higher order processes. *Acta Informatica*, 30(1):1–59, 1993.

[TKLC95] Bent Thomsen, Frederick Knabe, Lone Leth, and Pierre-Yves Chevalier. Mobile agents set to work. *Communications International*, July 1995.

[TLG93] B. Thomsen, L. Leth, and A Giacalone. Some issues in the semantics of Facile distributed programming. Technical Report ECRC-92-32, European Computer-Industry Research Centre, 1993.

[TLK96] Bent Thomsen, Lone Leth, and Tsung-Min Kuo. A Facile tutorial. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference, Pisa, Italy, 26–29 August, 1996*, volume 1119 of *Lecture Notes in Computer Science*, pages 278–298. Springer-Verlag, 1996.

[TLP+93] Bent Thomsen, Lone Leth, Sanjiva Prasad, Tsung-Min Kuo, Andre Kramer, Fritz C. Knabe, and Alessandro Giacalone. Facile Antigua Release – Programming guide. Technical Report ECRC-93-20, European Computer Industry Research Centre, Munich, Germany, December 1993.

[TSS+97] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[Tur96] David N. Turner. *The Polymorphic Pi-calculus: Theory and Implementation.* PhD thesis, University of Edinburgh, 1996.

[TvRvS+90] Andrew S. Tanenbaum, Robert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experience with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.

[Uny] Asis Unyapoth. *Nomadic π-Calculus: Its Theories and Applications.* A forthcoming PhD thesis, University of Cambridge, Computer Laboratory.

[VC98] Jan Vitek and Guiseppe Castagna. Towards a calculus of mobile computations. In *Proceedings of ICCL '98 Workshop on Internet Programming Languages, Chicago, USA*, May 1998.

[VC99] Jan Vitek and Guiseppe Castagna. Seal: A framework for secure mobile computations. In Henri E. Bal, Boumediene Belkhouche, and Luca Cardelli, editors, *Internet Programming Languages (ICCL '98 Workshop, Chicago, USA, May 1998)*, volume 1686 of *Lecture Notes in Computer Science*, pages 47–77. Springer, 1999.

[Vic94] Björn Victor. *A Verification Tool for the Polyadic π-Calculus.* Licentiate thesis, Department of Computer Systems, Uppsala University, Sweden, May 1994. Available as report DoCS 94/50.

[Vog96] Werner Vogels. World wide failures. In *Proceedings of the ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.

[vSHBT98] Maarten van Steen, Franz J. Hauck, Gerco Ballintijn, and Andrew S. Tanenbaum. Algorithmic design of the Globe wide-area location service. *The Computer Journal*, 41(5):297–310, 1998.

[vSHT99] Maarten van Steen, Philip Homburg, and Andrew S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, 7(1):70–78, January 1999.

[VT97] Jan Vitek and Christian Tschudin, editors. *Towards the Programmable Internet, Proceedings of the Second International Workshop on Mobile Object Systems, MOS '96*. Springer-Verlag, 1997.

[Wal99] Jim Waldo. Object-oriented programming on the network. In R. Guerraoui, editor, *Proceedings ECOOP '99, Lisbon, Portugal, June 1999*, volume 1628 of *Lecture Notes in Computer Science*, pages 441–448. Springer-Verlag, 1999.

[Whi96] J. E. White. Telescript technology: Mobile agents. General Magic white paper. In J. Bradshaw, editor, *Software Agents*. AAAI/MIT Press, 1996.

[Whi98] James E. White. Telescript retrospective. Afterword to mobile agents. In Dejan Milojičić, Frederick Douglis, and Richard Wheeler, editors, *Mobility: Processes, Computers, and Agents*. Addison-Wesley, 1998.

[WPW98] T. Walsh, N. Paciorek, and D. Wong. Security and reliability in Concordia. In *31st Hawaii International Conference on System Sciences*, volume VII, Software Technology Track, 1998.

[WPWD97] D. Wong, N. Paciorek, T. Walsh, and J. DiCelie. Concordia: An infrastructure for collaborating mobile agents. In K. Rothermel and R. Popescu-Zeletin, editors, *Proceedings of the First International Workshop on Mobile Agents*, volume 1219 of *Lecture Notes in Computer Science*, pages 86–97. Springer-Verlag, 1997.

[WRW96] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java system. *Computing Systems*, 9(4):265–290, 1996.

[WS99] Paweł T. Wojciechowski and Peter Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. In *Proceedings of ASA/MA '99 (First International Symposium on Agent Systems and Applications/Third International Symposium on Mobile Agents), Palm Springs, CA, USA*, October 1999.

[WWWK97] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. In *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 49–64. Springer-Verlag, April 1997. A reprint of the Sun Microsystems Lab. Technical Report TR-94-29, 1994.