Nomadic Pict: Language and Infrastructure Design for Mobile Agents

Paweł T. Wojciechowski **Peter Sewell**

Computer Laboratory, University of Cambridge.

{Pawel.Wojciechowski,Peter.Sewell}@cl.cam.ac.uk

Abstract

We study the distributed infrastructures required for location-independent communication between migrating agents. These infrastructures are problematic: different applications may have very different patterns of migration and communication, and require different performance and robustness properties; algorithms must be designed with these in mind. To study this problem we introduce an agent programming language – *Nomadic Pict*. 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. The levels are based on rigorously-defined process calculi, they provide sharp levels of abstraction. In this paper we describe the language and use it to develop an infrastructure for an example application. The language and examples have been implemented; we conclude with a description of the compiler and runtime.

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, KR98, VE97]. 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.

In the *Nomadic Pict* project [SWP98, SWP99] we are addressing these issues in the context of the eponymous mobile agent programming language. 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.

In our earlier work we focussed on the design of the Nomadic π-calculus, in [SWP99] giving its operational semantics and expressing two simple infrastructure algorithms as translations from a high-level to a low-level calculus. In this paper we introduce the programming language in more detail (§2). We discuss a small example application and the design of an infrastructure suited to it (§3,4), and describe the language implementation (§5). 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 [SWP99], would not be.

A number of other 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 [FGL+96] provides location-independent messages using a built-in infrastructure, based on forwarding pointer chains that are collapsed when possible. Voyager [Obj97] 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) [MLC98] 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).

2 The Nomadic Pict Language

We have designed and implemented Nomadic Pict as a vehicle for exploring distributed infrastructure. It builds on the Pict language of Pierce and Turner [PT97, 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 standardized low-level runtime that is common to many machines, with divergent high-level facilities chosen and installed at run time. 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 in [SWP99]. They have since been extended to large fragments of the language, for use in correctness proofs, but are not described here.

We have focussed on the simplest language that allows us to study the core problem of §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 and distributed algorithms, in the absence of mobility, has been demonstrated in the FOX and Ensemble projects [HLP98, 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 infrastructure following. We begin with an example. Below is a program in the low-level language 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.

```
getApplet ?* [a s] =
   agent b =
      migrate to s   ( <a@s'>ack!b  |  B )
   in
      ()
```

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 be on site `s'`) containing its name. In parallel, the body `B` of the applet commences execution.

The example illustrates the main entities of the language: sites, agents and channels. *Sites* should be thought of as physical machines or, more accurately, as instantiations of the Nomadic Pict runtime system on machines; each site has a unique name. This paper does not explicitly address questions of network failure and reconfiguration, or of security. Sites are therefore unstructured; neither network topology nor administrative domains are represented in the language. *Agents* are units of executing code; an agent has a unique name and a body consisting of some Nomadic Pict process; 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 language is built above asynchronous messaging, both within and between sites; in the implementation inter-site messages are sent on TCP connections, created on demand.

The inter-agent message `<a@s>ack!b` is characteristic of the low-level language. 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 the implementation at most one inter-site message is sent.

Names As in the π-calculus, names play a key rôle. New names of agents and channels can be created dynamically. These names are *pure*, in the sense of Needham [Nee89]; no information about their creation is visible within the language (in our current implementation they do contain site IDs, but could equally well be implemented by choosing large random numbers).

Types The language inherits a rich type system from Pict, including higher-order polymorphism, simple recursive types and subtyping. It has a partial type inference algorithm. It adds new base types `Site` and `Agent` of site and agent names, and a type `Dynamic` (to date only partially implemented) for implementing traders. In this paper we make most use of `Site`, `Agent`, the base type `String` of strings, the type `^T` of channel names that can carry values of type `T`, tuples `[T1 .. Tn]`, 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 variants and a type operator `Map` from the libraries, taking two types and giving the type of maps, or lookup tables, from one to the other.

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 communication of processes (except for the migration of whole agents) or of higher-order functions. **Patterns** `p` are of the same shapes as values.

Low-Level Language The main syntactic category is that of *processes* (we confuse processes and declarations for brevity). We will introduce the main low-level primitives

in groups.

agent a=P **in** Q	agent creation
migrate to s 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 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 | Q	parallel composition
()	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:T P	new channel name creation
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

To express computation within an agent, while keeping a lightweight implementation and semantics, we include π-calculus-style interaction primitives. Execution of new c:^T P creates a new unique channel name for carrying values of type T; c is binding in P. An output c!v (of value v on channel c) 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. A replicated input c?*p=P behaves similarly except that it persists after the synchronisation, and so may receive another value. In both c?p=P and c?*p=P the names in p are binding in P.

We require a clear relationship between the semantics of the low-level language 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*. Intuitively, there is a distinct π-calculus-style channel for each channel name in every agent.

iflocal <a>c!v **then** P **else** Q	test-and-send to agent a on this site
<a>c!v	send to agent a on this site
<a@s>c!v	send to agent a on site s

Finally, the low-level language includes primitives for interaction between agents. The execution of **iflocal** <a>c!v **then** 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 is still implementable locally, by the runtime system on each site. Two other useful constructs can be expressed in the language introduced so far: `<a>c!v` and `<a@s>c!v` attempt to deliver `c!v` to agent `a`, on the current site and on `s` respectively. They fail silently if `a` is not where expected and so are usually used only where `a` is predicatable.

Note that the language primitives are almost entirely asynchronous — only **migrate** and `<a@s>c!v` can involve network communication; they require at most one message to be sent between machines.

```
try c?p=P timeout n -> 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 language 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.

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 agent a
```

The intended semantics of an output `c@a!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. The low-level communication primitives are also available, for interacting with application agents whose locations are predictable.

Expressing Encodings The language for expressing encodings allows the translation of each interesting phrase (all those involving agents or communication) to be specified; the translation of a whole program can be expressed using this compositional translation. A translation of types can also be specified, and parameters can be passed through the translation. We omit the concrete syntax; the example infrastructure in §4 should give the idea.

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.

3 Example Application

In this section we discuss a small application that makes use of mobility and location-independent communication. Our primary goal is to present an example of the choice of a communication infrastructure based on a specific migration and communication pattern, together with the use of our two-level architecture. In Section 4 we give the key parts of the infrastructure encoding, providing an executable description of the algorithm. The application and infrastructure have been prototyped in Nomadic Pict. The example algorithm design assumes a large essentially-reliable LAN, rather than the wide-area unreliable case that we are most interested in, but it should give the feel of this style of working.

The PA 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 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 summonings should preserve any communications that they are engaged in, for example audio/video links with other members of the project.

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

High-Level Architecture We implement the PA application 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 a single *name server* agent, also static, which maintains 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..."))
```

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 nacks 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 ! []
                   end | 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 PAname = sys.read
   ( summonPA @ NameServer![PAname Summoner s]
   | notFound?_= print!(+$ PAname " 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.

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, we assume that the application is running over a large LAN, in which all machines are at roughly the same communication cost distance from each other. Machines are 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.

For the latter, 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 — eg audio/video messages or streams, and other data (that must be delivered reliably).

These assumptions are not wholly accurate — the application also demands disconnected operation (on laptops) and a higher level of fault-tolerance. We discuss infrastructure design addressing these briefly, at the end of §4, but for the sake of a clear example infrastructure we neglect them for now.

Design of Appropriate Infrastructure We develop our infrastructure in several steps, beginning with the two extremely simple algorithms described precisely in [SWP99]. 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. 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 migrations. 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 Section 4.

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 paper. 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 synchronisations between the source and all sinks of a stream on any migration.

4 Example Infrastructure

In this section we describe the Query Server with Caching algorithm as a Nomadic Pict encoding, thereby making all the details of concurrency and synchronisation precise. At first sight the code fragments may seem impenetrable, as space for a full exposition is

lacking, but we believe they repay study — almost the entire encoding can be given in 1.5 pages, rather concise for a non-trivial executable distributed infrastructure. 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. To give a feeling for such design the code fragments are taken almost verbatim from the executable source, with some minor sugar.

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 agents: the query server 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, are given in Figure 1; the interesting clauses of the compositional translation are in the text below.

The messages sent between agents fall into three groups, implementing the high-level agent creation, agent migration, and location-independent messages. Typical executions are illustrated in Figures 2 and 3 in the Appendix. Correspondingly, only these cases of the compositional translation are non-trivial.

Each class of agents 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 `trymessage` to `DS`, replacing the lock after the message is sent. 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 are 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 messages on `migrating`, `migrated`, and `ack` (see Figure 2).

$[\![\textbf{migrate to u P}]\!]_{\texttt{[a Q SQ]}} \stackrel{def}{=}$

```
currentloc?[S DS]=
    val [U DU] = u
```

```
  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> _:[]} -> ()
           end
    | message?*[#X DU U a:Agent c:^X v:X]=   (*  deal with a lost message  *)
          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> _:[]} -> ()
          end  )
```

```
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 )
            end
       | 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 1: Parts of the Top Level – the Query Server and Daemon Daemon

```
( <Q @ SQ>migrating!a
| ack?_ =  migrate to U
           ( <Q @ SQ>migrated![U DU]
           | ack?_ = ( currentloc![U DU]
                     | [[P]]_[a Q SQ]) ))
```

The query server's lock is kept during the migration. The agent's own record of it's current site and daemon must also be updated with 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

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

Similarly, a high-level agent `a` must synchronise with the query server while creating a new agent `b`, with messages on `register` and `ack` (see Figure 2).

$[\![$**agent** b = P **in** P'$]\!]_{[a\ Q\ SQ]} \stackrel{def}{=}$

```
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`.

Returning to the process of message delivery, there are three cases (see Figure 3). 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 `trydeliver` 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 `trydeliver` 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 `trydeliver` 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`).

The algorithm is very asynchronous; some additional optimisations are feasible (e.g. updating the daemon's cache more frequently). 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 QSCs must be adopted. In order to avoid long hops, the agents should register and unregister with the local QS on changing domains.

5 Language Implementation

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 runtime. The typechecker performs partial type inference. Typechecking is performed twice, before and after an encoding is applied. In the last phases, any separately compiled modules are joined and the compiler incrementally optimises the resulting intermediate code. The intermediate code is architecture-independent; its constructs correspond approximately to those of the Low Level Nomadic π-calculus (extended with value types and system function calls).

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

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 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.

The runtime 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. The purely local implementability of the runtime is good for this.

References

[AO98] Y. Aridor and M. Oshima. Infrastructure for Mobile Agents: Requirements and Design. In *Proc. 2nd Int. Workshop on Mobile Agents, LNCS 1477*, volume 1477, pages 38–49, 1998.

[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), LNCS 1477*, volume 1477, pages 136–147, September 1998.

[Bou92] Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sofia-Antipolis, May 1992.

[CHK97] D. Chess, C. Harrison, and A. Kershenbaum. Mobile agents: Are they a good idea? In *Mobile Object Systems – Towards the Programmable Internet. LNCS 1222*, pages 25–48, 1997.

[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.

[Hay98] Mark Hayden. The ensemble system. Technical Report TR98-1662, Cornell University, Computer Science, January 8, 1998.

[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, 1998.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *Proceedings of ECOOP '91, LNCS 512*, pages 133–147, July 1991.

[KR98] Fritz Hohl (Eds.) Kurt Rothermel. *Mobile Agents, Proceedings of the Second International Workshop, MA'98*. Springer-Verlag, Germany, 1998.

[LA97] Danny B. Lange and Yariv Aridor. *Agent Transfer Protocol – ATP/0.1*. IBM Tokyo Research Laboratory, March 1997.

[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.

[MBB+98] D. Milojicic, 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 *Proc. 2nd Int. Workshop on Mobile Agents, LNCS 1477*, pages 50–67, 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, April 27–30 1998.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I + II. *Information and Computation*, 100(1):1–77, 1992.

[Nee89] R. M. Needham. Names. In S. Mullender, editor, *Distributed Systems*, pages 89–101. Addison-Wesley, 1989.

[Obj97] ObjectSpace. Voyager core package technology overview. Available from `http://www.objectspace.com/`, 1997.

[PT95] Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Theory and Practice of Parallel Programming (TPPP, Sendai, Japan, 1994), LNCS 907*, pages 187–215, 1995.

[PT97] 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*, MIT Press.

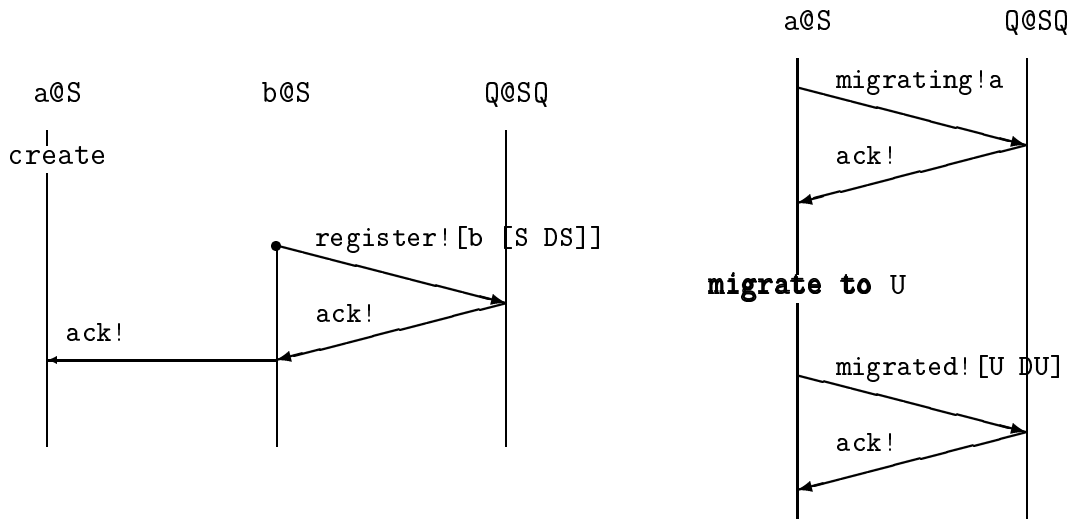
[SWP98] Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location independence for mobile agents. In *Workshop on Internet Programming Languages, Chicago*, May 1998.

[SWP99] Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location-independent communication for mobile agents: a two-level architecture. Technical Report 462, Computer Laboratory, University of Cambridge, 1999. A version of this is to appear in a WIPL'98 LNCS volume. Available from `http://www.cl.cam.ac.uk/users/pes20/`.

[Tur96] David N. Turner. *The Polymorphic Pi-calculus: Theory and Implementation.* PhD thesis, University of Edinburgh, 1996.

[VE97] Jan Vitek and Christian Tschudin (Eds.). *Towards the Programmable Internet, Proceedings of the Second International Workshop, MOS'96.* Springer-Verlag, 1997.

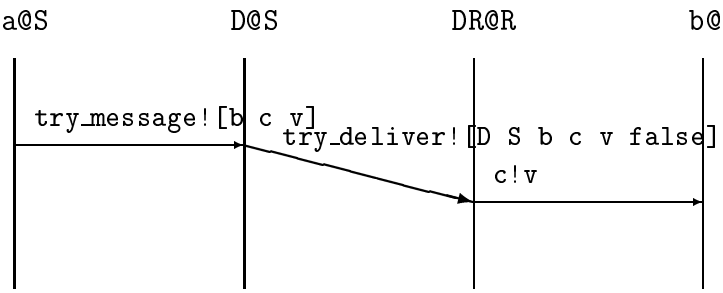
Appendix



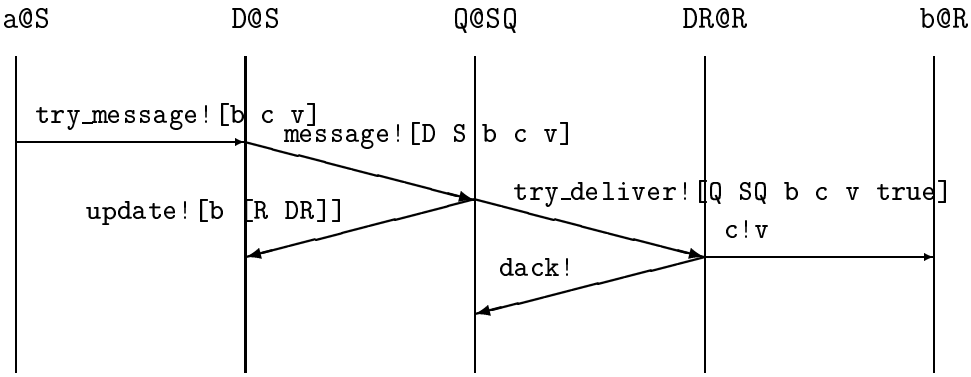
Horizontal arrows are synchronised communications within a single machine (using **iflocal**); slanted arrows are asynchronous messages.

Figure 2: Updating the Query Server on Creation and Migration

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.

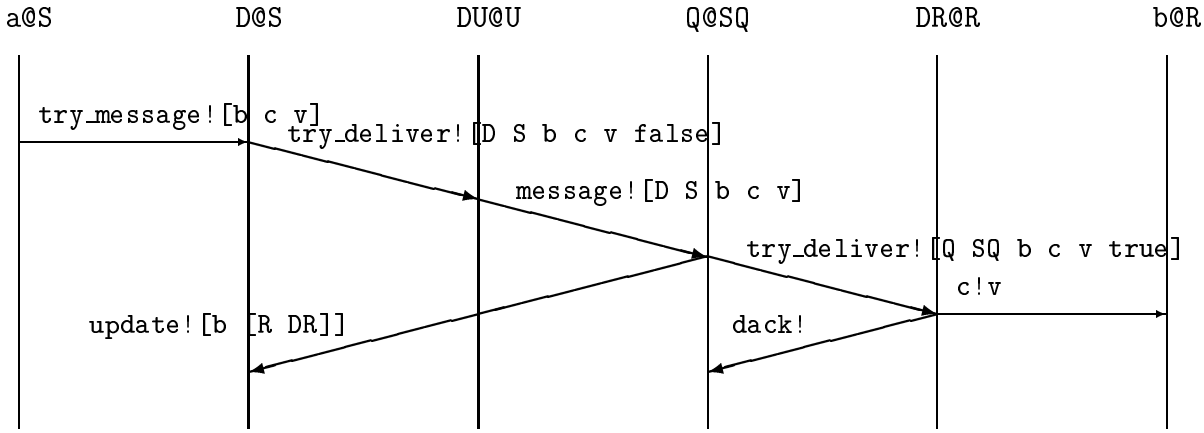


Figure 3: The Delivery of Location-Independent Message c@b!v from a to b.