# Communication Infrastructure Design for Wide-Area Mobile Computation: Specification in Nomadic Pict

Paweł T. Wojciechowski
Ecole Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
Pawel.Wojciechowski@epfl.ch

August 4, 2005

Technical Report LSR-2005-001

**Abstract**

We review an example of wide-area mobile agent applications: video-on-demand, long-lived scientific computation, and collaborative work, and the design of a distributed infrastructure required in each of these applications for location-independent communication. For the latter application, we propose an infrastructure algorithm that assumes two kinds of collaboration: (1) within a group of "mobile" individuals, who can communicate frequently using different computers connected to a local-area network (possibly via a wireless medium), and (2) some individuals may also communicate outside their groups using the global network. The algorithm has been specified formally, as an executable encoding in Nomadic Pict. The formal specification is concise but gives enough details to be directly translated by application programmers using their language of choice.

**Key-words:** mobile computation, algorithms, languages, concurrency

# Contents



# List of Figures

# 1 Introduction

The ongoing growth of wide-area networks has brought up considerable interest in *software mobility*, in several forms. *Mobile agents* [8, 16, 12] are units of executing code that can migrate between machines, and perform tasks locally. The simplest form of mobile agents — *mobile code* — is now commonly used in web applications, e.g. as Java [11] applets. *Mobile computation* [7] goes further and allows a running computation (code, data, threads of control) to migrate. It has been widely argued [7, 12, 30, 33] that software mobility provides a useful enabling technology for wide-area applications, such as web services, scientific computation, and collaborative work. With the advent of *mobile systems*, computers and people can move dynamically. This paves way for novel wide-area applications. Software mobility can greatly simplify development of such applications. For instance, architecture-independence and dynamic rebinding, which are characteristics of software mobility, allow migratory applications to be easily adapted to changes in physical locations and underlying environment.

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 having to explicitly track each other's movements. To provide these above standard network technologies (which directly support only location-dependent communication) requires some distributed infrastructure. Sewell, Wojciechowski, and Pierce [27] argued that the choice or design of an infrastructure must be somewhat application-specific — any given *infrastructure algorithm* will only have satisfactory performance for some range of migration and communication behaviour; the algorithms must be matched to the expected properties of applications and the communication network. Some applications also demand disconnected operation (on laptops) and a higher-level of fault-tolerance.

## 1.1 Communication Infrastructures for Wide-Area Mobile Computation

Continuing the above work, we propose some example infrastructures for three wide-area applications of mobile computation: the video-on-demand web service, long-lived scientific computation, and collaborative work. We justify our choices, based on a specific migration and communication pattern of these applications. In the first two applications the pattern is actually quite limited, thereby a very simple infrastructure is sufficient. The most interesting case appeared to be collaborative work. In the paper, we design an infrastructure algorithm that assumes a model with two kinds of collaboration: (1) within a group of "mobile" individuals, who can communicate frequently using different computers connected to a local-area network (including mobile devices with wireless connections), and (2) some individuals may also communicate over the global network or move between groups. This *two-tier* model of collaboration covers many real-world scenarios, e.g. think of people working closely on the same project or task within a local (indoor or outdoor) area, who may occa-

sionally contact a distant expert or manager or other group.

In our earlier work with Sewell [35, 36], we discussed the *Personal Assistant (PA)* — a small application for *local* collaborative work and the design of an infrastructure suited to it. We focused on demonstrating the benefits of a multi-level architecture based on clearly defined levels of abstraction, that correspond to the location-dependent and location-independent communication between mobile agents. A simple distributed algorithm for location-independent communication that we have described in [35, 36] assumes a local-area network (LAN) rather than the wide-area case that we are most interested in here.

In this paper, we explore the full space of infrastructure algorithms for the *wide-area* PA application, and propose a novel infrastructure algorithm that matches the model of two-tier collaboration. The algorithm uses a federation of servers. Each federated server is responsible for managing communication within a group of PAs, with a simple discovery protocol for establishing communication between groups. Such architecture fits naturally into our collaboration model, optimizing inter and intra-group communications and migrations. A nice property of the algorithm is that it behaves as well as the optimal-within-LAN query server algorithm proposed in [35, 36]. However, it avoids a single point of failure. Furthermore, cache information and compaction techniques are used so that also the communication between LANs requires only one network message in the common case.

## 1.2 Formal Specification of Infrastructure Algorithms in Nomadic Pict

Many similar propositions of infrastructure algorithms appeared in various implementations of mobile agents (see e.g. [10, 32, 4, 1, 18, 19, 6] among others) and mobile systems [22, 37, 25]; we characterize this work in Section 6. However, the details of these algorithms remain the matter of implementation (with few exceptions, such as a formally verified algorithm in [18]). The existing informal or textual descriptions do not unambiguously identify a single algorithm — they often lack important details such as:

- what data are actually locked and for how long,
- how many control messages are needed for application message delivery,
- are the migrations and communications synchronized?

A formal specification can make such details explicit, and so would greatly help one to make a solid judgment about efficiency, scalability and robustness of these algorithms. In this paper, we therefore specify our algorithm formally, thereby making all the details of concurrency and synchronization precise. For this, we have used *Nomadic Pict* [27, 36, 33, 29] — a statically-typed, distributed programming language with agent mobility and message-passing communication. The language has been proposed especially for designing and reasoning about communication infrastructure algorithms for mobile agents.

Specifying our infrastructure algorithm in Nomadic Pict has several advantages. The algorithm's specification is concise (it fits into this paper) but provides all relevant implementation details. This means that application programmers are able to translate the specification almost automatically into a low-level code using their language of choice.

All infrastructure algorithms that we are aware of, use some common design patterns, such as forwarding pointers, location servers, broadcast and publish/subscribe interactions [34]. The Nomadic Pict specification makes design patterns easy to identify, aiding code reuse and clarity.

Message-passing communication in Nomadic Pict is expressed using *typed channels*, with a type system that is able to verify communication safety. This means that common errors, such as packet data format mismatch or dangling communication, can be found and corrected early in the design process, thus reducing the probability of such errors later when implementing a full-size infrastructure in a target language. Moreover, specifications in Nomadic Pict can be also *tested* on distributed machines since the language has been implemented; the source code is available [28].

The paper is organized as follows. Section 2 describes some example applications of mobile computation with a simple communication infrastructure. Section 3 presents the PA application which requires a complex infrastructure. Section 4 summarizes briefly the Nomadic Pict notation that we use later. Section 5 presents a formal specification of the PA infrastructure algorithm as a Nomadic Pict encoding. Section 6 discusses related work. Section 7 concludes.

# 2 Mobile Agent Applications with Simple Infrastructure

In some distributed applications, software mobility has potential advantages over traditional approaches. Firstly, we describe a video-on-demand web service, in which by bringing the computation closer to data, it is often possible to decrease latency and increase throughput of network transmission. Then, we describe long-lived scientific computation, in which mobility can help to efficiently use computer resources. As we will see, both applications require a rather simple communication infrastructure.

## 2.1 Video-On-Demand Web Service

The use of mobile computation makes a lot of sense in case of applications that are launched from mobile devices. Suppose we want to download a large file from a video-on-demand web service to a mobile computer; the file can be fragmented, and different fragments are available on different busy servers. Since mobile devices are only intermittently connected to a network, our application can develop an agent request, possibly while *disconnected*, and begin downloading the file once it is *connected*. Note that the communication can often be via low-bandwidth, high-latency, high-cost connections (wireless or dial-up

links). The agent can therefore react to a drop in network bandwidth (or battery level) and clone itself on a well-connected site, e.g. the user's home PC. The network-intensive transfer is continued at this remote site. In the meantime, the application on the mobile computer can resume transfer if the connection is reestablished. Progress on both sites, if any, is reported during a subsequent connect session, so that the agents can adjust their downloading criteria and download only those fragments of a file that are needed to complete the transfer. During the periods of time when the connection is stable, the missing fragments are downloaded directly from the user's home PC. In a more advanced case, we can have two or more agents that migrate to distant Internet locations to transfer data in parallel.

**Migration and Communication Pattern** The application uses a *one-hop* migration in a wide-area network. The parts of a running application may be moved dynamically (together with a local state and possibly also a thread of control) to well-connected locations, or *sites*, where they continue downloading the file. The migrated state describes the current state of the downloading protocol, e.g. names of files, directories, file fragments, and web addresses. Different file fragments downloaded at remote sites are communicated directly to the mobile computer, where they are merged by the application that initiated the downloading process.

**Design of Appropriate Infrastructure** The application components at different sites create a star or a tree-like infrastructure, where a root of the tree is the main component that has initiated the process of downloading files. The locations of downloading agents do not change. However, the root component may be disconnected and connected again, possibly at a different location. Therefore, it needs to synchronize with the downloading agents on each reconnection, so that they can update the current location of the root, and switch its status to "connected". No global service for tracking migrations is however required.

## 2.2 Long-Lived Scientific Computation

The combinatorial optimization problems in a number of areas, including telecommunications, VLSI design, financial analysis, and biomedical analysis, may require considerable computations. In such cases, parallel implementations of metaheuristics that are applied to these problems, such as *genetic* and *tabu search* algorithms, become necessary to reach high-quality solutions in reasonable times. One approach to parallelization assumes that multiple instances of the algorithm (called *tasks*) are executed in parallel on different machines. Different tasks can start with the same or different initial solution. Each task performs a given number of iterations and then broadcasts the best solution. The best of all solutions becomes the initial solution for the next iteration of the algorithm.

Roughly, a larger number of parallel tasks leads to higher performance. Mobility can help to efficiently use all the machines and provide a quality of service

for concurrent users. For instance, tasks can be occasionally moved between machines as part of system maintenance, or for load balancing. The main advantage of mobile computation here is the possibility of moving tasks between locations *transparently*, e.g. as part of a system administration procedure before switching off or rebooting a machine, or when the system is overloaded. The idea of using migration for load balancing is not new [16] — it spawned a lot of interest in *process migration* in the 80s. The recent technology advances, including virtual machines, enable however more useful implementations, which remove some restrictions of the early approaches.

**Migration and Communication Pattern** The migration is infrequent, e.g. twice every 24 hours. During day-time, an application is executed only on a few dedicated machines, while during the night, it can possibly involve tens or several hundred computers connected to a university campus network, or over the Internet.

**Design of Appropriate Infrastructure** The tasks of a collaborative group synchronize among themselves on migrations, and multicast partial solutions within the group. Some tasks (with their state) can be lost due to process crashes. This is however not a problem since no task holds critical data. Thus new tasks can be easily added on-the-fly to replace those that crashed. If the algorithm runs long enough, it will eventually produce a solution with acceptable quality. As strong reliability of message delivery is not required, tasks can use best-effort or gossip-based multicasts [3], which scale well to a large number of tasks.

# 3 The Personal Assistant Application with Complex Infrastructure

We now discuss an application that requires a complex communication infrastructure.

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 (PA)*, that can migrate on demand.

We consider the support of remote collaborations, too. Personal assistants can be temporarily *delegated* to another institution. For instance, the programming state (which may include source files, makefiles, and test data) can be delegated to expert(s) who can analyse the files while interacting with the pro-

gram's developer over the Internet, modify code, run the modified code using the original test data, and finally send the corrected program back to the developer. Also, individuals can occasionally visit other institutions and summon their PA agents.

A closely related application for multimedia CSCW is described in [2], 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 synchronizations between the source and all sinks of a stream on any migration.

**Migration and Communication Pattern** To facilitate experimentation, we have implemented a minimal functionality of the PA application, i.e. an instant messaging service. The application has 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 (the application launches summoners dynamically, using the standard migration primitive, onto the list of active sites); and *name server* agents, also static, which maintain a lookup table from the textual keys of PAs to their internal agent names. For simplicity the implementation uses location-independent communication throughout, despite the fact that the name server and summoners are static.

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 collection of large LANs, which are connected to a wide-area network, or intranet. In each LAN reliable messaging can be provided by lower-level protocols 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.

For the latter, we suppose that the number of PA agents is of the same order as the number of people in the labs. Each PA will migrate infrequently, with minutes or hours between migrations. The path of migrations is unpredictable — it may range over the whole LAN, some PAs may also migrate between LANs. 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).

**Design of Appropriate Infrastructure** We develop our infrastructure in several steps, beginning with the simple algorithms described in [27] and [36]. Then, we discuss an algorithm that is more suitable for wide-area collaboration.

The *Central Server* algorithm has a single server that records the current site of every agent; agents synchronize 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 synchronize 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*, described precisely in [36], 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. However, the algorithm does not scale to a large number of agents and the global network. The obvious defect is the need to send control messages between the daemons and the server over the Internet, even if migrations and communications are local within a LAN. Furthermore the algorithm has single point of failure. The usual solution is to have many servers, each dealing with agents of a single user or a collaborative group. The problem is however how to manage the servers so that the number of messages between servers and daemons is optimized, considering the migration and communication pattern of our two-tier model of collaboration.

In this paper we propose the *Federated Query Server with Caching (FQSC)* as an example solution. It has a collection of query servers, each maintaining locations of agents that are present in a local *domain*, where a domain can range from a single mobile computer to a LAN. For each agent name there is at least one server (the *local* server) that records the current site of the agent; agents synchronize with the local server before and after migrations. In case of migrations to a new domain, agents must also register at a new query server, which now becomes their local server. The location-independent messages are sent directly to remote destinations according to the cache information, or — if there is no good cache data — via servers, following pointer chains that are collapsed when possible. In the common case application messages are 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 5.

# 4 The Nomadic Pict Language

Nomadic Pict has been designed and implemented as a vehicle for exploring distributed infrastructure. It builds on the Pict language of Pierce and Turner [24], a concurrent (but not distributed) language based on the asynchronous $\pi$-calculus [14]. 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.

In this section we introduce enough of the Nomadic Pict language for the example infrastructure following. The language constructs have been described informally; an operational semantics can be found in [27, 33].

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. We use $P$ and $Q$ (in the italic font) to denote other program fragments.

```
getApplet ?* [a s] =
   agent b =
      migrate to s  ( <a@s'>ack!b | P )
   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 acknowledgment to the requesting agent `a` (which is assumed to be on site `s'`) containing its name. In parallel, the body $P$ 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. *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 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.

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 $\pi$-calculus, names play a key rôle; sites, agents and channels are all named (they are distinguished by the type system). New names of agents and channels can be created dynamically. These names are *pure*, in the sense of Needham [21]; 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, which can statically detect some communication errors, e.g. due to mismatch between types of message senders and message receivers. It adds new base types `Site` and `Agent` of site and agent names, and a type `Dynamic` for implementing traders. In this paper we make most use of `Site`, `Agent`, the base type `Bool` of booleans, 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 and Patterns** Channels allow the communication of first-order values: names `a`,`b`,..., boolean values, 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, the language includes $\pi$-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 synchronize, 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 synchronization, and so may receive another value. In both `c?p=`$P$ and `c?*p=`$P$ the names in `p` are binding in $P$.

To express time-bound program exceptions, the language also includes a timed input construct with a timeout value, omitted here as we do not use it.

| | |
|---|---|
| **iflocal** `<a>c!v` **then** $P$ **else** $Q$ | test-and-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. Another useful construct can be expressed in the language introduced so far: `<a@s>c!v` attempts to deliver `c!v` to agent `a` on site `s`. It fails silently if `a` is not where expected and so is usually used only where `a` is predictable.

**High-Level Language** The high-level language is obtained by extending the low-level with location-independent communication primitives.

| | |
|---|---|
| `<a@?>c!v` | location-independent output to agent `a` |

The intended semantics of an output `<a@?>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. 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 Section 5 should give the idea. The concrete syntax has been described in the language documentation [28].

**Locks, methods and objects** The language inherits a common idiom for expressing concurrent objects from Pict [23]. 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.

# 5 Example Infrastructure for Personal Assistants

In Section 3, we introduced the Federated Query Server with Caching (FQSC) infrastructure for the PA application. In this section we describe the infrastructure algorithm as a Nomadic Pict encoding, thereby making all the details of concurrency and synchronization precise. At first sight the code fragments may seem impenetrable, but we believe they repay study – almost the entire encoding can be given in 2.5 pages, rather concise for a non-trivial executable distributed infrastructure.

## 5.1 Specification of the FQSC Algorithm in Nomadic Pict

An encoding consists of three parts, a top-level translation (applied to whole programs), an auxiliary compositional translation $\{P\}$ of high-level constructs $P$, and an encoding of types. The FQSC encoding involves three main classes of agents: the query servers `Q` (distributed on sites so that there is at least one

```
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 lock *)
      | register?*[a [S DS]] =
          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 1: Parts of the Top Level in the FQSC Algorithm – the Query Server

server in each LAN), the daemons D (one on each site), and the translations of high-level application agents (which may migrate). The top-level translation launches all the query servers and the daemons before executing the application program. The query server, and the code which launches daemons, are given in Figures 1, 2; the interesting clauses of the compositional translation are in the text below.

Each class of agents maintains some explicit state as an output on a lock channel. For each agent name there is at least one server – including some *local*

```
daemondaemon?*[S:Site [Q:Agent SQ:Site]] =

   (* launch a daemon D on site S *)
   (* Q is a local Query Server 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 2: Parts of the Top Level in the FQSC Algorithm – the Daemon Daemon

server in a LAN where the agent currently is – that has the site and daemon where the agent is currently located, stored in a map `m`. As we will see below, the encoding of each high-level agent records its current site and daemon, and the name and site of the local server. This is kept accurate when agents are created or migrate. To guarantee this property, the cross-domain migration involves a local server *hand-over*.

Each daemon maintains a map `m` 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. If a local server does not have correct location data, a message will be delivered using forwarding pointer chains that are collapsed when possible. If a query server has no pointer for the destination agent of a message then it will forward the message to its *home server*, i.e. the server on which the agent was originally registered, which has the pointer. To make this possible an agent name is encoded by a triple of an agent name and the names

of the agent's home server and the home server's site.

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 3, 4 and below. Correspondingly, only these cases of the compositional translation are non-trivial.

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 `<b@?>c!v`, 'send `v` to channel `c` in agent `b`', is below.

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

This first reads from the agent's lock channel `currentloc`: the name `S` of the current site, the name `DS` of the local daemon, the name `Q` of the local query server, and the name `SQ` of the server's site, then sends `[b c v]` on the channel `try_message` to `DS`, replacing the lock after the message is sent. The translation is parametric on the name `a` of the agent containing this phrase — for this phrase, `a` is however not used. We return later to the process of delivery of the message.

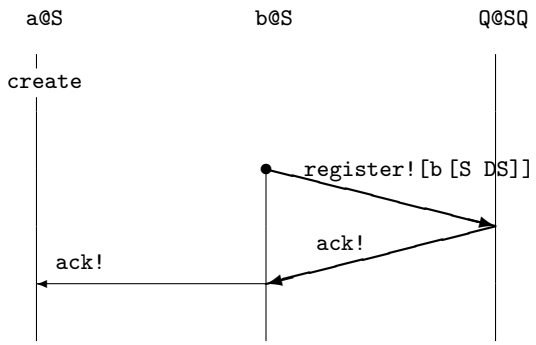
A high-level agent `a` synchronizes with the query server while creating a new agent `b`, with messages on `register` and `ack`.

```
{ 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 ))
```

The current site/daemon/server data for the new agent must be initialised to `[S DS Q SQ]`; 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`. Note that the name `b` of the new agent in the high-level program is actually encoded by a triple of an agent name `B` and the names of its home server `Q` and the home server's site `SQ`, i.e. `b = [B Q SQ]`; there is a translation of a type

```
{Agent} = AgentTy = [Agent Agent Site]  .
```

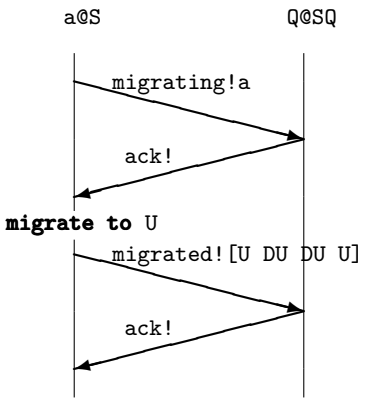
A sample execution is below.



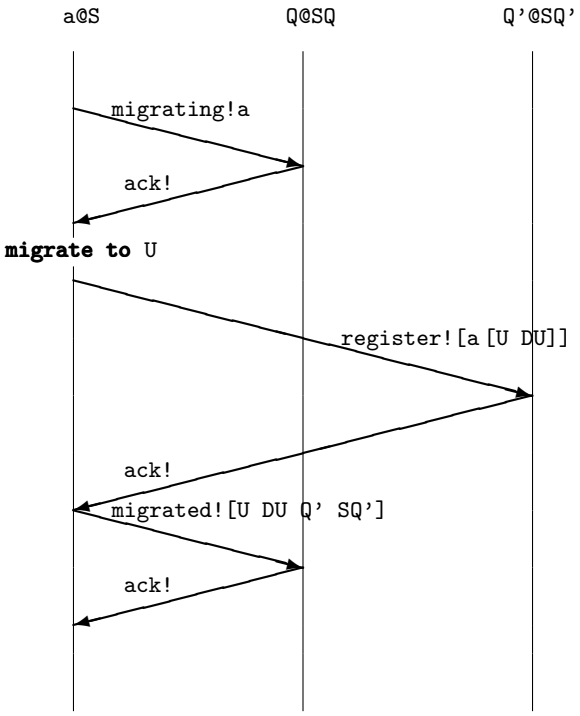

To migrate while keeping the local query server's map accurate, the translation of a **migrate** in a high-level agent must synchronize with the local query server before and after actually migrating, with `migrating`, `migrated`, and `ack` messages. If the target site `U` is in the domain managed by a different query server `Q'` (see an **else** clause below) then the agent registers at `Q'` (which is now the agent's new local server) and sends a `migrated` message to `Q` (which updates its cache with the new server's name/site).

```
{ migrate to u P }a =
   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
              (* migration within a domain *)
          ( <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 )))))
```

A sample execution of a migration in a domain is below.

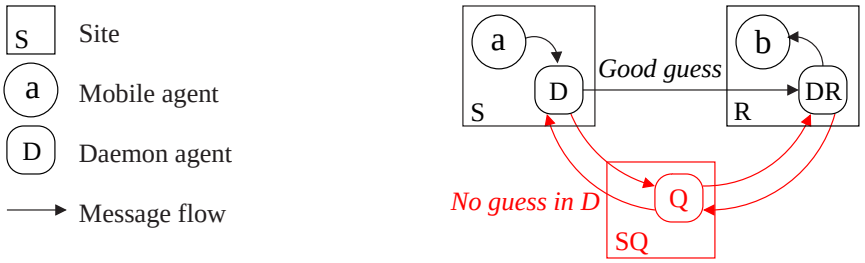



A sample execution of a cross-domain migration with registration at Q' is following.

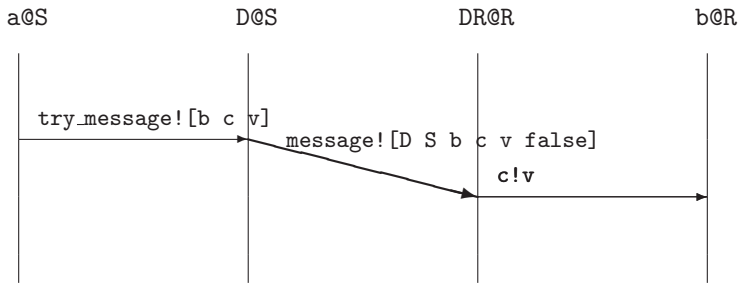



The query server's lock is kept during the migration. The agent's own record of its current site and daemon and its local server must also be updated with the new data `U DU Q' SQ'` when the agent's lock is released. Note that in the body of the encoding the name `DU` of the daemon on the target site and the names `Q'` and `SQ'` of the server and its site of the target domain must be available. This is achieved by encoding site names in the high-level program by quadruples of

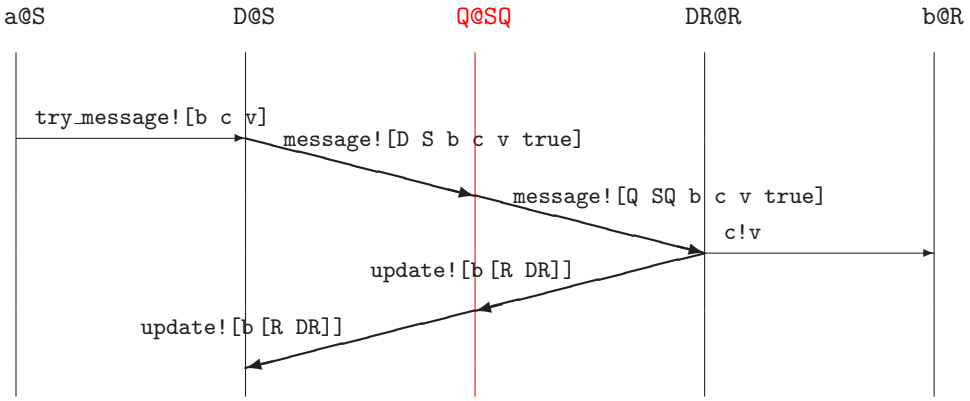
Message flow in good-guess and no-guess scenarios.




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



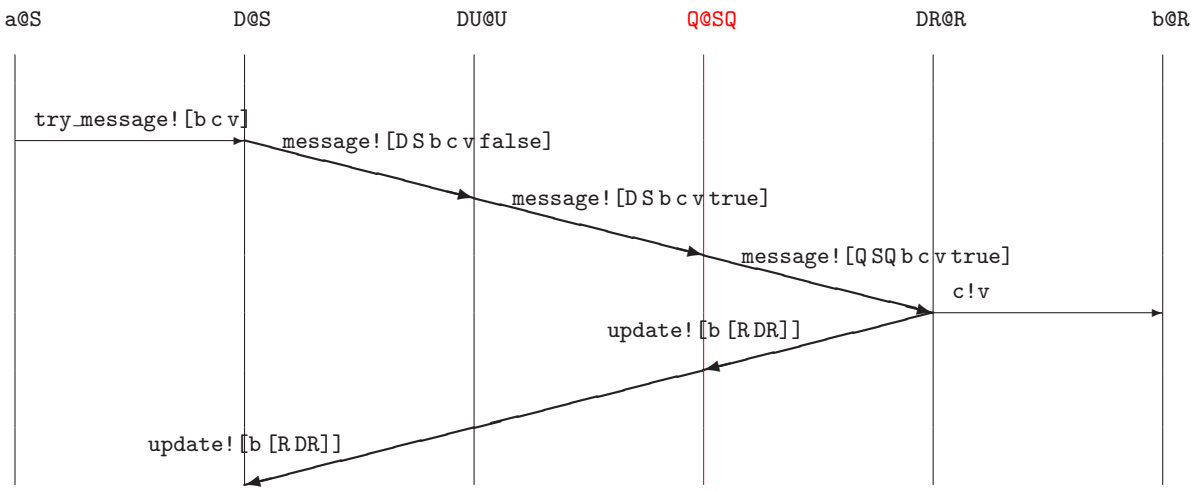

No guess in the `D` cache.




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

Figure 3: The Delivery of Location-Independent Message `<b@?>c!v` from `a` to `b`

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.

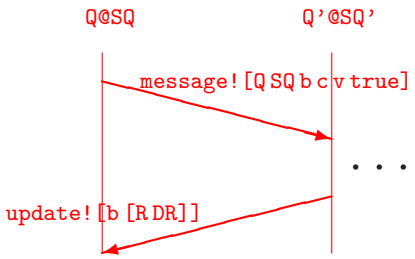



Figure 4: The Delivery of Location-Independent Message <b@?>c!v from a to b

a site name and the associated daemon name and a query server name/site for that site; there is a translation of a type

```
{Site} = SiteTy = [Site Agent Agent Site]  .
```

Returning to the process of message delivery, there are three cases (see Figure 3 for the first two cases). Consider the implementation of `<b@?>c!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, including the cross-domain communication; it requires only one network message.

In the cache-miss case (see at the bottom of Figure 3) daemon `D` sends a `message` message to the local query server `Q`, which forwards the message to a daemon `DR` at site `R`, which then delivers successfully and sends an `update` message back to `D` via `Q` (both `D` and `Q` update their cache). The query server's lock is kept until the message is delivered, thus preventing `b` from migrating until then. Two other variants are possible. If the forwarding pointer for the agent `b` is not found, `Q` forwards the message to `b`'s home server (the server's name/site are encoded as part of the name `b`). Similarly, if `b` has moved between domains and 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` (see the chain of forwarding servers at the bottom of Figure 4). Note that 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 (see Figure 4). 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`).

## 5.2 Design Choices and Possible Extensions

The FQSC algorithm avoids sending too many cache updates over the Internet, e.g. as long as agent migrations are local, a cache-update message to other query servers is sent only in the case of incorrect-cache-hits from these servers. Consequently, the cost of forwarding a message to agents in other domains is paid only for the first message. Then, the forwarding pointer chain is collapsed and any subsequent messages (from the same location) are sent directly.

The above design choice reflects the expected behaviour of the PA agents, i.e. (1) the inter-domain migrations, which correspond to the PA delegation or a physical movement of individuals, are less frequent than migrations within a domain, and (2) the communication is more frequent than migration. Otherwise, it may be worth to collapse the forwarding pointer chain more often, e.g. the cache of *several* daemons and servers could be updated upon each cross-domain migration (not only the last-visited ones).

One can also analyse the application further. In fact, migrations of the PA agents may usually be within a small group of machines, e.g. those of a project group. More sophisticated infrastructures might use some heuristics to take advantage of this. For a critical application a quantitative analysis may be required. An exhaustive discussion is beyond the scope of this paper.

The PA application may also demand disconnected operation (on laptop computers). For this, we can easily extend the FQSC algorithm with the *Query Server with Caching and Disconnection (QSCD)* infrastructure, described in [34]. The QSCD infrastructure allows a program to disconnect a site (a laptop computer) from the network, and later reconnect, so that all application messages to and from the site are transparently delivered irrespective of agent migration and site disconnection. The FQSC augmented with QSCD infrastructure allows mobile computers to change domains. They can also connect to each other and establish *ad-hoc* communications, assuming that at least one computer has a query server, so that this computer can be a "domain" to which the other connects.

This paper does not explicitly address questions of security, fault-tolerance, or administrative domains. These should be addressed in the full-size implementation of the PA infrastructure. In order to tolerate machine crashes, the (logical) query servers can be replicated on several machines using a *group communication* middleware [13]. Reliable communication channels can be implemented using reliable point-to-point communication above UDP, e.g. the reliable transport protocol provided by the group communication middleware.

# 6 Related Work

Many authors present strategies for *locating* mobile objects and devices (see, e.g., survey papers [37, 25]). Similar to locating objects are mechanisms for resource discovery. For instance, Dimakopoulos and Pitoura [9] describe cached-based distributed flooding approaches to locate a peer that provides a particular resource, with cache updates propagated either upon resource lookup or change.

Our work builds on the above, but is focused on location-independent message delivery, which provides stronger properties than a pair of unsynchronized agent lookup and message sending actions. For instance, the FQSC algorithm guarantees that messages are not lost irrespective of agent migrations, and the upper bound on the number of hops required to deliver a message in case of local (within domain) migrations is known.

A number of 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 [10] provides location-independent messages using a built-in infrastructure, based on forwarding pointer chains that are collapsed when possible.

Voyager [32] 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 [4] 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 [1] 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) [17] 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.

Roth and Peters [26] propose a scalable global service for locating mobile agents, with encryption and decryption capabilities to prevent security attacks through agent impersonating.

The MASIF proposal [15] 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).

Murphy and Picco [20] present a distributed-snapshot-based algorithm — it attempts to deliver a message to every agent in the system using broadcast, and only the agents whose IDs match the message target actually accept the message.

Moreau [18] describes formally an algorithm for routing messages to migrating agents, which is based on distributed location directory service, with forwarding pointer chains that are collapsed when possible. In [19], he describes the directory extended with pointer redundancy to tolerate node crashes; the algorithm has been verified using the proof assistant Coq.

Cao *et al.* [6, 5] propose to separate agents and movable *mailboxes*, i.e. receivers of location-independent messages, with push and pull techniques that can be used by agents to obtain messages from their mailbox; they also discuss schemes to make the communication tolerant to mailbox crashes [6], and path compression for better performance [5].

The use of home servers in our FQSC algorithm resembles the *Internet Mobile Host Protocol (IMHP)* proposed by Perkins *et al.* [22] for transparent routing of IP packets to mobile hosts. By enabling sites to also cache bindings for mobile hosts (or mobile agents in FQSC) both protocols provide mechanisms for better routing which bypasses the default reliance on routes through the home server, and so they eliminate the likelihood that the home server would be a bottleneck. However, cache updates are performed differently, with FQSC optimizing the migration and communication pattern of Personal Assistants. The FQSC protocol normally delivers messages to mobile agents in one-hop,

while IMHP must route messages to mobile hosts via *care-of address* (which corresponds to the current local server of the target mobile agent in FQSC).

See also [34, 33] for discussion of many different infrastructure strategies for location independence, including some comparison and discussion of scalability and reliability issues.

# 7 Conclusion

The contribution of this paper is twofold:

- We discuss the communication and migration pattern of example wide-area mobile agent applications: video-on-demand and scientific computation, which require a simple communication infrastructure, and collaborative work that demands a more sophisticated infrastructure;
- For the latter, we propose the FQSC algorithm that is matched to the two-tier model of wide-area collaboration of mobile users; it optimizes message complexity, assuming frequent migrations and communications within groups of users and occasional inter-group migrations.

The FQSC algorithm has been presented formally as an executable specification in the Nomadic Pict language. The advantage of using Nomadic Pict to design communication protocols for mobile computation is that errors can be found and corrected early in the design process. The specifications can be verified using a type system, tested on distributed machines, and finally recoded in Java or other language.

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.

Last but not least, the Nomadic Pict language supports formal reasoning about software mobility. Unyapoth and Sewell [31] have developed language-based proof techniques which are based on Nomadic Pict, and used them to formally prove the correctness of an example infrastructure algorithm (a Central Server algorithm).

**Acknowledgments.** We would like to thank Peter Sewell and Asis Unyapoth for many useful discussions throughout the Nomadic Pict project.

# References

[1] Y. Aridor and M. Oshima. Infrastructure for mobile agents: Requirements and design. In *Proc. 2nd Int. Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 38–49, Sept. 1998.

[2] J. Bates, D. Halls, and J. Bacon. Middleware support for mobile multimedia applications. *ICL Systems Journal*, 12(2):289–314, Nov. 1997.

[3] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Bidiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999.

[4] M. Bursell, R. Hayton, D. Donaldson, and A. Herbert. A Mobile Object Workbench. In *Proc. of the 2nd Int. Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 136–147, Sept. 1998.

[5] J. Cao, L. Zhang, X. Feng, and S. K. Das. Path compression in forwarding-based reliable mobile agent communications. In *Proc. ICPP '03 (Int. Conference on Parallel Processing)*, Oct. 2003.

[6] J. Cao, L. Zhang, J. Yang, and S. K. Das. A reliable mobile agent communication protocol. In *Proc. ICDCS '04 (24th Int. Conference on Distributed Computing Systems)*, Mar. 2004.

[7] L. Cardelli. Abstractions for mobile computation. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 51–94. Springer, 1999.

[8] 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. Springer, 1997.

[9] V. V. Dimakopoulos and E. Pitoura. A peer-to-peer approach to resource discovery in multi-agent systems. In *Proc. CIA '03 (7th Int. Workshop on Cooperative Information Agents)*, volume 2782 of *Lecture Notes in Computer Science*, Aug. 2003.

[10] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proc. CONCUR '96 (7th Int. Conference on Concurrency Theory)*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421, Aug. 1996.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison Wesley, 2000.

[12] D. Kotz, R. Gray, and D. Rus. Future directions for mobile agent research. *IEEE Distributed Systems Online*, 3(8), Aug. 2002. 2002.

[13] S. Mena, A. Schiper, and P. T. Wojciechowski. A step towards a new generation of group communication systems. In *Proc. Middleware '03*, volume 2672 of *Lecture Notes in Computer Science*, pages 414–432, June 2003.

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

[15] 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. of the 2nd Int. Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 50–67, Sept. 1998.

[16] D. Milojičić, F. Douglis, and R. Wheeler, editors. *Mobility: Processes, Computers, and Agents*. Addison-Wesley, 1999.

[17] D. S. Milojičić, W. LaForge, and D. Chauhan. Mobile Objects and Agents (MOA). In *Proc. COOTS '98 (4th USENIX Conference on Object-Oriented Technologies and Systems)*, pages 179–194, Apr. 1998.

[18] L. Moreau. Distributed directory service and message router for mobile agents. *Science of Computer Programming*, 39(2–3):249–272, 2001.

[19] L. Moreau. A fault-tolerant directory service for mobile agents based on forwarding pointers. In *Proc. SAC '02 Track on Agents, Interactions, Mobility and Systems (17th Symposium on Applied Computing)*, pages 93–100, Mar. 2002.

[20] A. L. Murphy and G. P. Picco. Reliable communication for highly mobile agents. In *Proc. ASA/MA '99 (First Int. Symposium on Agent Systems and Applications / Third Int. Symposium on Mobile Agents)*, Oct. 1999.

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

[22] C. Perkins, A. Myles, and D. B. Johnson. IMHP: A mobile host protocol for the Internet. In *Proc. INET '94 (Conference of the Internet Society, in conjunction with the 5th Joint European Networking Conference)*, pages 479–491, 1994.

[23] B. C. Pierce and D. N. Turner. Concurrent objects in a process calculus. In T. Ito and A. Yonezawa, editors, *Theory and Practice of Parallel Programming, International Workshop TPPP '94, Sendai, Japan, November 7-9, 1994, Proceedings*, volume 907 of *Lecture Notes in Computer Science*. Springer, 1995.

[24] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[25] E. Pitoura and G. Samaras. Locating objects in mobile computing. *IEEE Transactions on Knowledge and Data Engineering*, 13(4), July/August 2001.

[26] V. Roth and J. Peters. A scalable and secure global tracking service for mobile agents. In *Proc. of the 5th Int. Workshop on Mobile Agents*, volume 2240 of *Lecture Notes in Computer Science*, pages 169–181, Dec. 2001.

[27] P. Sewell, P. T. Wojciechowski, and B. C. Pierce. Location-independent communication for mobile agents: A two-level architecture. In *Internet Programming Languages*, volume 1686 of *Lecture Notes in Computer Science*, pages 1–31. Springer, 1999.

[28] The Nomadic Pict Language. *Distribution Files and Documentation, http://lsrwww.epfl.ch/~pawel/nomadicpict.html.*

[29] The Nomadic Pict Project. *http://www.cl.cam.ac.uk/~pes20/nomadicpict.html.*

[30] A. Tripathi, T. Ahmed, and N. M. Karnik. Experiences and future challenges in mobile agent programming. *Microprocessors and Microsystems*, 25(2):121–129, Apr. 2001.

[31] A. Unyapoth and P. Sewell. Nomadic Pict: Correct communication infrastructure for mobile computation. In *Proc. POPL '01 (28th Symposium on Principles of Programming Languages)*, pages 116–127, Jan. 2001.

[32] Voyager Java Development Platform. *http://www.recursionsw.com.*

[33] P. T. Wojciechowski. Nomadic Pict: Language and infrastructure design for mobile computation. Technical Report 492, Computer Laboratory, University of Cambridge, June 2000.

[34] P. T. Wojciechowski. Algorithms for location-independent communication between mobile agents. In *Proc. AISB '01 Symposium on Software Mobility and Adaptive Behaviour*, pages 10–19, March 2001.

[35] P. T. Wojciechowski and P. Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. In *Proc. ASA/MA '99 (First Int. Symposium on Agent Systems and Applications / Third Int. Symposium on Mobile Agents)*, pages 2–12, Oct. 1999.

[36] P. T. Wojciechowski and P. Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. *IEEE Concurrency*, 8(2):42–52, April-June 2000.

[37] V. Wong and V. Leung. Location management for next generation personal communication networks. *IEEE Network*, 14(5):8–14, Sept./Oct. 2000.