Algorithms for Location-Independent Communication between Mobile Agents

Paweł T. Wojciechowski
Swiss Federal Institute of Technology (EPFL)
Operating Systems Laboratory; CH-1015 Lausanne
Pawel.Wojciechowski@epfl.ch

Abstract

We study the distributed infrastructures required for location-independent communication between mobile agents. These infrastructures are problematic: different applications may have very different patterns of migration and communication, and require different performance and robustness properties. Some applications also demand disconnected operation (on laptop computers). Algorithms must be designed with these mind. In this paper we describe simple algorithms and techniques such as a central server, forwarding pointers, broadcast, group communication, and hierarchical location directory, and use Nomadic Pict to develop and implement an example infrastructure. The infrastructure can tolerate site disconnection; a user can disconnect the computer from the network, work in a disconnected mode for extended periods, and later reconnect. All messages that cannot be delivered to a laptop or sent out from the laptop due to disconnection will be transparently delivered upon reconnection.

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 (Chess et al., 1997). 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. In (Wojciechowski and Sewell, 2000; Wojciechowski, 2000a), we argue that 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 and the communication medium. Some applications also demand disconnected operation (on laptops) and a higher level of fault-tolerance.

The goal of this paper is to describe the space of algorithms which might be useful for building such infrastructures. These are simple, generic versions of the algorithms which are used in real distributed systems with object mobility and in mobile networks. In our earlier work (Wojciechowski and Sewell, 2000), we discussed a small application, the *Personal Mobile Assistant*, and the design of an infrastructure suited to it. We focused on demonstrating the Nomadic Pict distributed programming language (Sewell et al., 1999; Wojciechowski, 2000b) and the benefits of a multi-level architecture based on clearly defined levels of abstraction. In this paper we extend the *Query Server with Caching* algorithm in (Wojciechowski and Sewell, 2000) with support for disconnected operation on laptop computers, so that all messages to and from a temporarily disconnected site are transparently delivered to mobile agents irrespective of agent migration and temporal network unavailability. We describe the infrastructure as a Nomadic Pict encoding, thereby making all the details of concurrency, synchronisation, and distribution clear and precise.

2 The Design Space

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

Awerbuch and Peleg (Awerbuch and Peleg, 1995) (see also Mullender and Vitányi (Mullender and Vitányi, 1988)) stated the analogous problem of keeping track of mobile users in a distributed network (they consider two operations: "move", facilitating the move of a user to a new destination, and "find", enabling one to contact a specified user at its current address). They first examined two extreme strategies.

The *full-information* strategy requires every site in the

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

Awerbuch and Peleg (Awerbuch and Peleg, 1995) describe a distributed directory infrastructure for online tracking of mobile users. They introduced the graph-theoretic concept of *regional matching*, and demonstrated how finding a regional matching with certain parameters enables efficient tracking of mobile users in a distributed network. The communication overhead of maintaining the distributed directory is within a polylogarithmic factor of the lower bound. This result is important in the case of mobile telephony and infrastructures which support mobile devices, where the infrastructure should perform well, considering *all* mobile users and their potential communication to migration patterns. These patterns can vary, depending on people, and can only be estimated probabilistically. The infrastructure should therefore support all migration and communication scenarios, and optimise those scenarios which are likely to happen more often (preferably it should adapt to any changes in behaviour of mobile users dynamically). An analytical comparison of pointer-based, centralised and distributed simple location management schemas for mobile computing can be found in (Krishna, 1996). In mobile agent applications, however, the communication to migration pattern of mobile agents usually can be predicted precisely (Wojciechowski, 2000a). Therefore we can design algorithms which are optimal for these special cases and simpler than the directory server mentioned above.

The task of deciding on an infrastructure may involve many criteria. In our study, we expand the space for interesting algorithms to many dimensions, considering not only the communication cost but also other factors, such as scalability and the cost of fault-tolerance. An infrastructure is *scalable* if adding new sites or agents, or expanding the system to wide-area networks does not severely degrade overall system performance (in these terms, we consider two different kinds of scalability which explore either the *numerical* or *geographical* dimensions). The semantics of failure depends on the failure model, e.g. we may assume that if a site fails then all agents on this site are lost (Crash/no Recovery model) or they can be recovered upon site recovery (under the *same* name). Fault-tolerance is costly; the level of fault-tolerance and methods which can be used will depend on the target network and application demands.

3 Example Algorithms

Below, we describe example algorithms and give some hints about the infrastructure scalability and fault-tolerance, where possible. We are not giving a quantitative theoretical or empirical view of the algorithms, however, because it would be too hard to take under consideration all the factors which exist in real systems.

3.1 Central Server

Central Forwarding Server The server records the current site of every agent. Before migration an agent A informs the server and waits for ACK (containing the number of messages sent from the server to A). It then waits for all the messages due to arrive. After migration it tells the server it has finished moving. If B wants to send a message to A, B sends the message to the server, which forwards it. During migrations (after sending the ACK) the server suspends forwarding.

Central Query Server The server records the current site of every agent. If B wants to send a message to A, B sends a query (containing the message ID) to the server asking for the current site of A, gets the current site s of A and sends the message to s. The name s can be used again for direct communication with A. If a message arrives at a site that does not have the recipient then a message is returned saying 'you have to ask the name server again'. Migration support is similar as above.

Home Server Each site s has a server (one of the above) that records the current site of some agents — usually the agents which were created on s. Agent names contain an address of the server which maintains their locations. On every migration agent A synchronises with the server whose name is part of A's name. If B wants to send a message to A, B resolves A's name and contacts A's server. Other details are as above.

Discussion If migrations are rare and also in the case of stream communication or large messages, the Query Server seems the better choice. However, Central Forwarding and Query Servers do not scale. If the number of agents is growing and communication and migration are frequent, the server can be a bottleneck. Home Servers can improve the situation. The infrastructure can work fine for small-to-medium systems, where the number of agents is small.

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

The server is a single-point of failure. In this and other algorithms we can use some of the classical techniques of fault-tolerance, e.g. state checkpointing, message logging and recovery. We can also replicate the server on different sites to enhance system availability and fault-tolerance. Group communication can provide adequate multicast primitives for implementing either *primary-backup* or *active* replication (Guerraoui and Schiper, 1996).

Mechanisms similar to Home Servers have been used in many systems which support process migration, e.g. in Sprite (Douglis and Ousterhout, 1991). Caching has been used, e.g. in LOCUS (Popek and Walker, 1985), and V (Cheriton, 1988), allowing operations to be sent directly to a remote process without passing through another site. If the cached address is wrong a home site of the process is contacted (LOCUS) or multicasting is performed (V).

3.2 Forwarding Pointers

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

Discussion There is no synchronisation between migration and communication as there was in centralised algorithms. A message may follow an agent which frequently migrates, leading to a race condition. The Forwarding Pointers algorithm is not practical for a large number of migrations to distinct sites (a chain of pointers is growing, increasing the cost of search). Some "compaction" methods can be used to collapse the chain, e.g. movement-based and search-based. In the former case, an agent would send backward a location update after performing a number of migrations; in the latter case, after receiving a number of messages (i.e. after a fixed number of "find" operations occurred).

Some heuristics can be further used such as search-update. A plausible algorithm can be as follows. On each site there is a daemon which maintains forwarding addresses (additionally to forwarding pointers) for all agents which ever visited this site. A *forwarding address* is a tuple $(timestamp, site)$ in which the site is the last known location of the agent and timestamp specifies the age of the forwarding address. Every message sent from agent B to A along the chain of forwarding pointers contains the latest available forwarding address of A. The receiving site may then update its forwarding address (and/or forwarding pointer) for the referenced agent, if required. Given conflicting guesses for the same agent, it is simple to determine which one is most recent using timestamps. When the message is eventually delivered to the current site of the agent, the daemon on this site will send an ACK to the daemon on the sender site, containing the current forwarding address. The address received replaces any older forwarding address but *not* the forwarding pointer (to allow updating the chain of pointers during any subsequent communication). A similar algorithm has been used in Emerald (Jul et al., 1988), where the new forwarding address is piggybacked onto the reply message in the object invocation. It is sufficient to maintain the timestamp as a counter, incremented every time the object moves.

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

3.3 Broadcast

Data Broadcast Sites know about the agents that are currently present. An agent notifies a site on leaving and a forwarding pointer is left over until agent migration is finished. If agent B wants to send a message to A, B sends the message to all sites in a network. A site s discards or forwards the message if A is not at s (we omit details).

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

Notification Broadcast Every site in a network maintains a current guess about agent locations. After migration an agent distributes in the network information about its new location. Location information is time-stamped. Messages with stale location information are discarded. If site s receives a message whose recipient is not at s (because it has already migrated or the initial guess was wrong), it waits for information about the agent's new location. Then s forwards the message.

Discussion The cost of communication in Query and Data Broadcasts is high (packets are broadcast in the network) but the cost of migration is low. Query Broadcast saves bandwidth if messages are large or in the case of stream communication. Notification Broadcast has a high cost of migration (the location message is broadcast to all sites) but the communication cost is low and similar to forwarding pointers with pointer chain compaction. In Data and Notification Broadcasts, migration can be fast because there is no synchronisation involved (in Query Broadcast migration is synchronised with communication); the drawback is a potential for race conditions if migrations are frequent. Site failures do not disturb the algorithms.

Although we usually assume that the number of sites is too large to broadcast anything, we may allow occasional broadcasts within, e.g. a local Internet domain, or local Ethernet. Broadcasts can be accomplished efficiently in bus-based multiprocessor systems. They are also used in radio networks. A realistic variant is to broadcast within a group of sites which belong to the itinerary of mobile agents known in advance. Broadcast has also been used in Emerald to find an object, if a node specified by a forwarding pointer is unreachable or has stale data. To reduce message traffic, only a site which has the specified object responds to the broadcast. If the searching daemon receives no response within a time limit, it sends a second broadcast requesting a positive or negative reply from all other sites. All sites not responding within a short time are sent a reliable, point-to-point message with the request. The Jini lookup and connection infrastructure (Arnold et al., 1999) uses multicast in the discovery protocol. A client wishing to find a Lookup Service sends out a known packet via multicast. Any Lookup Service receiving this packet will reply (to an address contained in the packet) with an implementation of the interface to the Lookup Service itself.

3.4 Group Communication

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

Before migration an agent A informs the other agents in the group about its intention and waits for ACKs (containing the number of messages sent to A). It then waits for all the messages due to arrive and migrates. After migration it tells the agents it has finished moving. Multicast messages to each agent within a group are delivered in the order sent (using a first-in-first-out multicast). If B wants to send a message to A, B sends the message to site s which is A's current location. During A's migrations (i.e. after sending the ACK to A) B suspends sending any messages to A. If two (or more) agents want to migrate at the same time there is a conflict which can be resolved as follows. Suppose A and C want to migrate. If B receives migration requests from A and C, it sends ACKs to both of them and suspends sending any messages to agents A and C (in particular any migration requests). If A receives a migration request from C after it has sent its own migration request it can either grant ACK to C (and C can migrate) or postpone the ACK until it has finished moving to a new site. The choice is made possible by ordering agent names.

Discussion The advantage of this algorithm is that sites can be stateless (the location data are part of agent state). However, in a system with failures the algorithm is more complicated than above. Agents are organised into groups, corresponding to *multicast delivery lists*, that cooperate to perform a *reliable multicast* (i.e. if one agent on the delivery list receives a reliable multicast message, every agent on the delivery list receives the message). A precise meaning to the notion of delivery list can be given by using *virtual synchrony* defined for non-movable groups (Birman and Joseph, 1987). The current list of agents to receive a multicast is called the *group view*. The group view is consistent among all agents in the group. Processes are added to and deleted from the group via *view changes*. If agent A is removed from the view, the agents remaining in the view would assume that A has failed. Virtual synchrony guarantees that no messages from A will be delivered in the future (if A has not failed it must rejoin the group explicitly under a new name).

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

The algorithm is suitable for frequent messages (or stream communication) between mobile agents and when migrations are rare. Agent failures and network partitions will not disturb agents which are alive; however, there are detailed subtleties which depend on the semantics of the algorithm implementing virtual synchrony. The group service algorithms for non-movable processes which have been originally proposed, e.g. in ISIS, are costly in terms of control messages and hard to use in networks larger than a LAN. However, they are also examples of scalable group membership and communication services implementing the virtual synchrony semantics, designed for wide-area networks (Keidar et al., 1999).

3.5 Hierarchical Location Directory

Algorithm A tree-like hierarchy of servers forms a location directory (similar to DNS). Each server in the directory maintains a current guess about the site of some

agents. Sites belong to regions, each region corresponds to a sub-tree in the directory (in the extreme cases the sub-tree is simply a leaf-server for the smallest region, or the whole tree for the entire network). The algorithm maintains an invariant that for each agent there is a unique path of forwarding pointers which forms a single branch in the directory; the branch starts from the root and finishes at the server which knows the actual site of the agent (we call this server the "nearest"). Before migration an agent A informs the "nearest" server X_1 and waits for ACK. After migration it registers at a new "nearest" server X_2, tells X_1 it has finished moving and waits for ACK. When it gets the ACK there is already a new path installed in the tree (this may require installing new and purging old pointers within the smallest sub-tree which contains X_1 and X_2). Messages to agents are forwarded along the tree branches. If B wants to send a message to A, B sends the message to the B's "nearest" server, which forwards it in the directory. If there is no pointer the server will send the message to its parent.

Discussion Certain optimisations are plausible, e.g. if an agent migrates very often within some sub-tree, only the root of the sub-tree would contain the current location of the agent (the cost of a "move" operation would be cheaper). In (Moreau, 1999), Moreau describes an algorithm for routing messages to migrating agents which is also based on distributed directory service. A proposition of Globe uses a hierarchical location service for world-wide distributed objects (van Steen et al., 1998). The Hierarchical Location Directory scales better than Forwarding Pointers and Central Servers. Also, some kinds of fault can be handled more easily (see (Awerbuch and Peleg, 1995), and there is also a lightweight crash recovery in the Globe system (Ballintijn et al., 1999)).

3.6 Arrow Directory

Some algorithms can be devised for a particular communication pattern. For example, if agents do not require instant messaging, a mail-box infrastructure can be used, where senders send messages to static mailboxes and all agents periodically check mailboxes for incoming messages. Demmer and Herlihy (Demmer and Herlihy, 1998) describe the Arrow Distributed Directory protocol for distributed shared object systems, which is devised for a particular object migration pattern; it assumes that the whole object is always sent to the object requester. The arrow directory imposes an optimal distributed queue of object requests, with no point of bottleneck.

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

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

When a node wants to acquire exclusive access to the object, it sends a message *find* which is forwarded via arrows and sets its own arrow to itself. When the other node receives the message, it immediately "flips" the arrow to point back to the immediate neighbour who forwarded the message. If the node does not hold the object, it forwards the message. Otherwise, it buffers the message *find* until it is ready to release the object to the object requester. The node releases the object by sending it directly to the requester, without further interaction with the directory.

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

4 The Nomadic Pict Language

In this section we introduce enough of the Nomadic Pict language for the example infrastructure following (see (Wojciechowski, 2000b) for details).

We have designed and implemented Nomadic Pict as a vehicle for exploring distributed infrastructure for mobile computation (Sewell et al., 1999; Wojciechowski, 2000a). It builds on the Pict language of Pierce and Turner (Pierce and Turner, 2000), a concurrent (but not distributed) language based on the asynchronous π-calculus (Milner et al., 1992; Honda and Tokoro, 1991). Low-Level Nomadic Pict supports agent creation, the migration of agents between sites, fine-grain concurrency and the communication of 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.

We begin with an example. Below is a program in the high-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  ( ack@a!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` 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. *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. The messages can be received out of order. Channels also have unique names. The language is built above asynchronous messaging, both within and between sites; in our implementation inter-site messages are sent on TCP connections, created on demand, but they could use instead some leyer which provides reliable communication on top of UDP. Our algorithms do not depend on the message ordering that could be provided by TCP.

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` 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.

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 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
<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 predictable.

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.

wait c?p=P **timeout** n -> Q	input with timeout

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.

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 below should give the idea.

5 Example Infrastructure

In this section we describe the QSCD (*Query Server with Caching and Disconnected Operation*) infrastructure, expressed in Nomadic Pict. We give the key parts of the infrastructure encoding, providing an executable description of the algorithm. The infrastructure has been prototyped in Nomadic Pict. The algorithm implements *disconnection-aware daemons* (which are spawned on each site) and defines two high-level agent operations `"disconnect" in P` and `"connect to" s:Site in P` to handle disconnection. All messages to and from a site are transparently delivered irrespective of agent migration and site disconnection. No messages are ever lost. No duplicate messages are ever received by agents. However, agent migration is not transparent – a program exception is raised in a high-level agent if the agent tries to migrate out from a disconnected site; migration to a site which has been disconnected is blocked until the site is back in the network.

The algorithm is, however, still somewhat idealised – due to lack of space we made a few simplifications. Firstly, a site disconnection will block *all* agent migrations and *all* communications which need to be forwarded through the query server. Secondly, each time the operation **`agent`** or **`migrate`** fails due to a timeout, an exception is invoked in the application (in a more practical algorithm, the infrastructure should rather try to repeat the operation with a slightly longer timeout before finally signalling problems). Therefore, the algorithms that are applicable to actual systems with mobile computers would have to be yet more delicate and complex. In the end of the section, we discuss some of these refinements and extensions informally.

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 as below, and an encoding of types. The QSCD encoding involves three main classes of agents: the query server `Q` itself (on a single site), the daemons (one on each site), and the translations of high-level application agents (which may migrate). The top-level (not given here) of program `P` launches the query server and all the daemons before executing $[\![\texttt{P}]\!]$. The precise definition of the query server and daemon in Nomadic Pict is given in Figures 1 and 2; the interesting clauses of the compositional translation are in the text below.

Each class of agent maintains some explicit state as an output on a `lock` channel. The query server maintains a map from each agent name to the site (and daemon) where the agent is currently located (`SiteTy = [Site Agent]`). 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) as an output on `currentloc`.

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

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

The local output (in the 2nd line) allows adjacent agents (on the same site) to communicate even if the local daemon will be blocked in the case of site disconnection. We return later to the process of delivery of the message which is sent to the local daemon.

To migrate while keeping the query server's map accurate, the translation of a **`migrate`** in a high-level agent `a` synchronises with the query server `[Q SQ]` before and after actually migrating, with `migrating`, `migrated`, and `mack` messages. We have used explicit acknowledgements instead of piggybacking control information (as in 3) in order to improve readability (the resulting algorithm is, however, less asynchronous). For example, the server simply does not acknowledge A's migration until it has received confirmation that all messages to A have been delivered. We also deal with a case when the current site is disconnected. If the query server does not respond within a certain period of time `t` (i.e. the current site is disconnected or the communication link is slow), migration will be abandoned (with an exception message `err`). Alternatively, we could ask the local daemon for more

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

Figure 1: Parts of the Top Level – the Query Server

accurate information (the daemon always knows about the connection/reconnection status) but due to the lack of space we omit details here.

$[\![$ **migrate to** u P $]\!]_{[a\ Q\ SQ\ t\ err]} \stackrel{def}{=}$

```
  currentloc?[S DS]= val [U:Site DU:Agent] = u
   new mack : ^[]
   ( <Q @ SQ>migrating![a mack]
   | wait mack?_ = (migrate to U
       ( <Q @ SQ>migrated![U DU]
       | mack?_ = ( currentloc![U DU]
                  | [[P]][a Q SQ t err])))
     timeout t ->
       ( currentloc![S DS]
       | mack?_ = <Q @ SQ>migrated![S DS]
       | err!"No connection."
       | [[P]][a Q SQ t]))
```

This first creates a fresh private channel `mack`, then sends `[a mack]` on the channel `migrating` to the query server, in parallel with a timed input on the channel `mack`. If the reply on `mack` is received within `t` seconds (approximately), the migration can proceed. Otherwise, the timeout clause is triggered and the migration is abandoned. However, if in fact the connection to the server was made possible (e.g. a timeout was simply too short) then the message `migrating` would be delivered to the server and the server would send to the agent a reply message `mack`. The query server blocks any disconnection requests after receiving a message `migrating` and can only release the lock after receiving an acknowledgement that migration is finished. Therefore, although migration failed the agent may still have to send a message `migrated` in the timeout clause and release the lock in the query server; the message will then contain an address `[S DS]` of the current site. The agent's own record of its current site and daemon must also be updated with the new data `[U DU]` (or restored from the old data if the migration failed) when the agent's lock is released. The query server's lock is kept during migration. This lock will protect the current and target sites from being disconnected by other agents while migration is in progress.

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

Figure 2: Parts of the Top Level – the Daemon Daemon

Site names in the high-level program are encoded by pairs of a site name and the associated daemon name; there is a translation of types

$[\![$`Agent`$]\!] \stackrel{def}{=}$ `Agent`

$[\![$`Site`$]\!] \stackrel{def}{=}$ `[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. If the query server is not accessible, the creation fails.

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

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

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

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

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

To disconnect a site while not missing messages sent between the site and a stable part of the network, a high-level agent a can use an operation "disconnect"

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

This synchronises with the local daemon and the query server, so that messages sent from the stable network to the disconnected site will be blocked in the query server until the site reconnects. In the opposite direction, cross-network messages sent by agents on the disconnected site will be blocked in the local daemon. Similarly, the composition translation of "connect" to s in P, 'connect to a query server which is on site s', is below.

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

Here, the parameter s is actually not used by the encoding since the algorithm assumes only one query server (it is useful in the scalable algorithm which assumes many query servers).

Refinements and Extensions If the timeout mechanism is set up correctly (e.g. using some stabilising failure detector) then the algorithm should behave well in a local-area network, with most application-level messages delivered in a single hop and none taking more than three hops (though 6 messages). The query server is involved only between migration and the time at which all relevant daemons receive a cache update; this should be a short interval. Messages to a disconnected site cannot be delivered and so they are buffered in the query server which will deliver them upon site reconnection. However, the

algorithm described above is not very practical, since the query server uses a global lock during disconnected operation, i.e. the `QS` blocks high-level messages to *all* sites if at least one site is disconnected. Also, an operation "create a new agent" fails with a program exception raised in a spawning agent, each time the operation is invoked from a disconnected site.

A refined version of this algorithm which is free from the problems stated above may be designed as follows. Many sites should be able to disconnect and reconnect at the same time, and the query server should block communication and migration only to a site which is currently disconnected. This requires that a query server maintains a separate map from sites to status information ("connected" or "disconnected"). A map of agents must contain little locks (each per agent entry) so that only messages to agents in disconnected sites are buffered. A local daemon has exact knowledge whether there is connection to the query server or not, so we can improve the algorithm by synchronising agent migrations with the local daemon. Also, only minor refinements are required to be able to re-install daemons after a site crash (making a query server fault-tolerant is much more difficult). In the protocol presented here, it is only possible to reboot a machine when a query server does not have an active communication link to it. Actually, each computer which can be disconnected should have installed its own query server to allow for non-blocking agent creation (since the agent registration messages would be sent to the local server, and thus do not depend on the network availability). Further improvements of the disconnected mode are plausible, e.g. operations `connect` and `disconnect` might be implicit if the operating system could provide a flag or an interrupt every time the local network connection goes up or down (though it might still be useful to have the operation "connect" in a high-level language).

In (Wojciechowski, 2000a) we also describe a scalable version of the algorithm which assumes many LANs interconnected by a wide-area network, and extensions required for ad-hoc networks.

Acknowledgements We would like to thank Peter Sewell, Ken Moody, and André Schiper for discussion and comments.

References

Ken Arnold, Ann Wollrath, Bryan O'Sullivan, Robert Scheifler, and Jim Waldo. *The Jini specification*. Addison-Wesley, 1999.

Baruch Awerbuch and David Peleg. Online tracking of mobile users. *Journal of the ACM*, 42(5):1021–1058, September 1995.

G. Ballintijn, M. van Steen, and A.S. Tanenbaum. Lightweight crash recovery in a wide-area location service. In *Proc 12th Conference on Parallel and Distributed Computing Systems*, August 1999.

K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proc 11th ACM Symposium on OS Principles*. ACM, November 1987.

David Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

D. Chess, C. Harrison, and A. Kershenbaum. Mobile agents: Are they a good idea? In J. Vitek and C. Tschudin, editors, *Mobile Object Systems – Towards the Programmable Internet*, volume LNCS 1222, 1997.

M. J. Demmer and M. P. Herlihy. The arrow distributed directory protocol. In *Proc 12th Symposium on Distributed Computing*, volume LNCS 1499, 1998.

Fred Douglis and John Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software – Practice and Experience*, 21(8):757–785, August 1991.

R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. In *Reliable Software Technologies - Ada-Europe '96*, LNCS 1088, June 1996.

Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *Proc of ECOOP'91*, LNCS 512, 1991.

Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

Idit Keidar, Jeremy Sussman, Keith Marzullo, and Danny Dolev. A client-server oriented algorithm for virtually synchronous group membership in WANs. Technical Report CS1999-0623, University of California, San Diego, 1999.

P. Krishna. *Performance Issues in Mobile Wireless Networks*. PhD thesis, Texas A&M University, 1996.

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

Luc Moreau. Distributed directory service and message router for mobile agents. Technical Report ECSTR M99/3, University of Southampton, 1999.

Sape J. Mullender and Paul M. B. Vitányi. Distributed match-making. *Algorithmica*, 3:367–391, 1988.

Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

G. J. Popek and B. J. Walker. *The Locus Distributed System Architecture*. MIT Press, 1985.

Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location-independent communication for mobile agents: A two-level architecture. In Henri E. Bal, Boumediene Belkhouche, and Luca Cardelli, editors, *Internet Programming Languages*, LNCS 1686, 1999. Also appeared as Technical Report 462, Computer Laboratory, University of Cambridge, April 1999.

Maarten van Steen, Franz J. Hauck, Gerco Ballintijn, and Andrew S. Tanenbaum. Algorithmic design of the Globe wide-area location service. *The Computer Journal*, 41(5):297–310, 1998.

Paweł T. Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation*. PhD thesis, University of Cambridge, 2000a. Also appeared as Technical Report 492, Computer Laboratory, University of Cambridge, June 2000.

Paweł T. Wojciechowski. *The Nomadic Pict System*, 2000b. Available electronically as part of the Nomadic Pict distribution. `http://lsewww.epfl.ch/~pawel/nomadicpict.html`.

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