Scalable Message Routing for Mobile Software Assistants

Paweł T. Wojciechowski
Poznań University of Technology
60-965 Poznań, Poland
ptw@cs.put.poznan.pl

May 16, 2006

Technical Report RA-010/06

Abstract

In this paper we define an algorithm for location-independent communication of mobile software Personal Assistants (PAs). The algorithm extends the Query Server with Caching algorithm that we proposed earlier, with support of message routing in the wide-area networks. Our algorithm is suitable for two kinds of PAs collaboration: (1) within a local 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 via the global network and move to other groups. 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 agents, distributed computing, protocols, middleware, P2P

Contents

List of Figures

1 Introduction

The ongoing growth of wide-area networks has brought up considerable interest in *mobile agents* [CHK97, MDW99, KGR02, Car99]; mobile agents are units of executing code that can migrate between machines and perform tasks locally. It has been widely argued [Car99, KGR02, TAK01] that mobile computation provides a useful enabling technology for wide-area applications, such as web services, scientific computation, and collaborative work.

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. Sewell, Wojciechowski, and Pierce [SWP99] 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.

In [WS00], we described the *Personal Assistant (PA)* – an application that uses mobile agents to support collaborative work of mobile individuals. The PA application uses the *Query Server with Caching (QSC)* infrastructure algorithm for location-independent communication. We have prototyped our application using *Nomadic Pict* [SWP99, WS00] – a statically-typed, distributed programming language, which is based on the π-calculus [MPW92] extended with distribution and agent mobility. The QSC algorithm however does not scale to wide-area networks, nor to many groups of PA agents, which would be required in the full-scale, practical implementation of the PA application.

In this paper, we therefore extend the QSC algorithm to support wide-area collaboration. We have done so with the following model of collaboration in mind: (1) mobile individuals within a local working group can communicate frequently using different computers connected to a local-area network (possibly via a wireless medium), and (2) some individuals may also communicate via the global network and move to other groups. This 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 occasionally contact a distant expert or manager, or migrate to other working group.

We propose a *Federated Query Server with Caching (FQSC)* algorithm that fits well into the above model of collaboration. The algorithm uses a federation of servers. Each federated server is responsible for managing communication within a local group of PAs, and maintaining a dynamic forwarding pointers chain used for communication between groups. The FQSC algorithm behaves as well as the optimal-within-LAN QSC algorithm proposed in [WS00]. 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.

Our paper is aimed at developers of mobile agent applications, and researchers interested in distributed (or peer-to-peer) algorithms. To avoid any ambiguities in the description of our algorithm, we present it formally using Nomadic Pict. The formal specification is concise but gives enough details to be directly translated by application programmers using their language of choice.

The paper is organized as follows. Section 2 presents the PA application. Section 3 describes FQSC informally. Section 4 shows the Nomadic Pict notation and presents a formal specification of the FQSC algorithm as a Nomadic Pict encoding. Section 5 proposes several extensions. Section 6 discusses related work, and Section 7 concludes.

2 The Personal Assistant Application

We consider the support of collaborations within (say) a large computer science department, spread over several buildings. Most individuals will be involved in a few collaborations, each of 2–10 people. Individuals move frequently between offices, labs and public spaces; impromptu working meetings may develop anywhere. Individuals would therefore like to be able to *summon* their working state (which may be complex, consisting of editors, file browsers, tests-in-progress etc.) to any machine. These summonings should preserve any communications that they are engaged in, for example audio/video links with other members of the project. To achieve this, the user's working state can be encapsulated in a mobile agent, an electronic *personal assistant (PA)*, that can migrate on demand.

We also consider the support of remote collaborations. Individuals can either visit other institutions and summon their personal assistants there, or the PAs can be temporarily *delegated* to other groups. For example, a personal assistant agent encapsulating a buggy program (which may include source files, makefiles, and test data) can be delegated to language experts, who can analyse the program while interacting remotely with the program developer who launched the PA agent, then modify code, check the modified code using the original test data, and finally send the PA (with corrected program) back to the developer.

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

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 occasionally migrate between LANs. The migrations of different PAs are essentially uncorrelated in time. It is com-

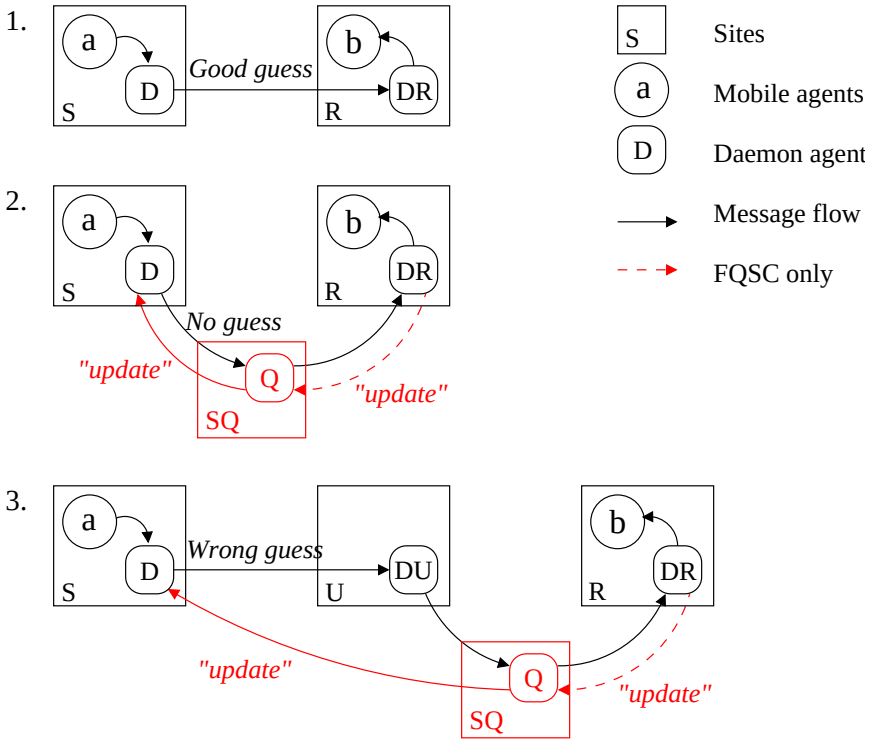


Figure 1: The QSC and FQSC algorithms – the delivery of a message from agent a to b

mon 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).

3 Design of Appropriate Infrastructure

We develop our infrastructure in several steps, beginning with two simple, centralized algorithms. Then we present our distributed algorithm.

3.1 The Central Server and Query Server with Caching

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. The location-independent, application messages are sent via the server. The central server is however a bottleneck for all inter-PA communication. Further-

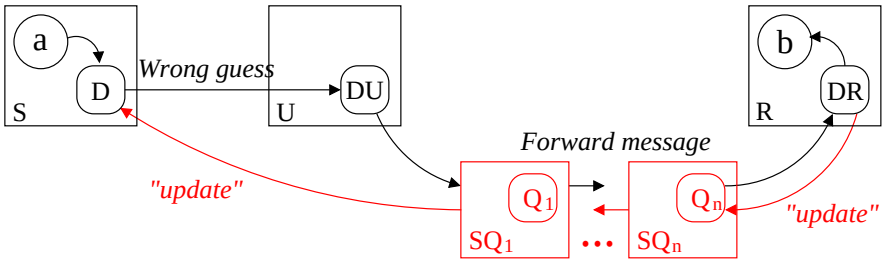


Figure 2: The FQSC algorithm – the delivery of a message in the wrong-guess case

more, all application messages must make two hops (and these messages make up the main source of network load).

Adapting the Central Server so as to reduce the number of application-message hops required, we have the *Query Server with Caching* algorithm, described in [WS00]. As before, it has a server Q that records the current site of every agent, and agents synchronize with it on migrations. In addition, each site has a daemon that maintains a cache of location data.

Consider the delivery of an application message from agent a to agent b. The message (see Fig. 1-1) is first sent to a daemon D on the current site S, which then forwards the message to the daemon DR on the target site R, which delivers the message to b. When a daemon D does not know the agent b (see Fig. 1-2), or when a daemon DU receives a mis-delivered message, for an agent b that has left its site U (see Fig. 1-3), the message is forwarded to server Q. The server both forwards the message on to the agent's current site R and sends a cache-update message to the originating daemon. In the common case application messages will here take only one hop (the "good guess" case in Fig. 1-1).

3.2 The Federated Query Server with Caching

The QSC algorithm however does not scale to a large number of PAs and a global network. Consider PA migration to a remote working group. The obvious defect in this case is the need to send control messages between the daemon and the server over the Internet, even if migrations and communications of the PA would be local within a LAN of the new group. Furthermore, the QSC algorithm has single point of failure. To overcome these drawbacks, we may have many servers, each one dealing with agents of a single user or collaborative group. Mobile computation introduces however an interesting problem: how to synchronize migrations and communications on these servers, so that the number of messages between servers and daemons is optimized ?

In this paper, we therefore propose the *Federated Query Server with Caching (FQSC)* algorithm, which removes the bottleneck. It employs a collection of

query servers Q_1, ..., Q_n for the specific migration and communication pattern of PA agents. Each server maintains locations of agents that are present in a local domain, where a *domain* can range from a single computer to a LAN.

For each agent there is at least one server, which records the current site of the agent; a *local server* in agent's current domain is an example of such a server. Agents synchronize with the local server before and after migrations. When an agent migrates away to a new domain, it must register at the query server in this new domain, which now becomes the agent's new local server.

As before, each site has a daemon that maintains a cache of location data. Application messages are sent via the daemons, much like in the QSC algorithm (see Fig. 1, 1-3). When a message cannot be delivered using cache information, the message is forwarded by query servers, using forwarding pointer chains that are collapsed when possible upon receipt of an "update" message (see Fig. 2). By compaction of the pointer chains, in the common case application messages are delivered in only one hop, as in the case of QSC.

If a server has no pointer for the destination agent b, then it will forward the message to b's home server, which has the pointer. An agent's *home server* is the query server on which the agent was originally registered upon its creation. The address of this server is recorded as part of the agent's high-level ID.

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. For example, it is difficult to explain in prose the following details:

- How are the migrations and communications synchronized (if at all) ?
- What data about agents are actually locked and for how long ?
- How to collapse the pointer chain on servers without loosing messages ?

To explain the details of the FQSC algorithm and to develop reasonable confidence in its correctness, a more precise description is required, ideally in an executable form. Below we give such a description using Nomadic Pict.

4 Formal Specification

4.1 The Nomadic Pict Language

In this section we introduce enough of the Nomadic Pict language for the example infrastructure following. We begin with an example. Below is a program showing how an applet server can be expressed. It can receive (on the channel named `getApplet`) requests for an applet; the requests contain a pair (bound to `a` and `s`) consisting of the name of the requesting agent and the name of its site.

```
getApplet ?* [a s] =
   agent b =
      migrate to s  ( <a@s'>ack!b | 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 π-calculus [MPW92], 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*; 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, static type system from the Pict language [PT00], on which it has been built. Nomadic Pict adds new base types `Site` and `Agent` of site and agent names, and a type `Dynamic` for implementing name service. 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 Channels allow the communication of first-order values, such as: names, 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). **Patterns** `p` are of the same shapes as values.

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; see Figure 3 for the whole set.

agent a=P **in** Q	agent creation
migrate to s P	agent migration
P | Q	parallel composition
()	nil
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
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
<a@?>c!v	location-independent output to agent a

Figure 3: The Nomadic Pict main primitives

Agent creation 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).

Agent migration Agents can migrate to 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. 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.

Pi-calculus To express computation within an agent, while keeping a lightweight implementation and semantics, the language includes 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 asynchronous 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.

Agent interaction 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. 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.

The construct for expressing encodings $\{P\}$=Q allows the infrastructure algorithm to be expressed as the translation from the high-level program P to the low-level program Q (that does not use the `<a@?>c!v` primitive).

Locks, methods and objects The language inherits a common idiom for expressing concurrent objects from Pict [PT94]. 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.

4.2 The FQSC Algorithm

We now describe the FQSC algorithm formally as a Nomadic Pict encoding, thereby making all the details of concurrency and synchronization precise.

The encoding involves three main classes of agents: the query servers `Q` (distributed on sites, so that there is at least one server in each LAN), the daemons `D` (one on each site), and the translations of high-level application agents (which may migrate). The query server code is given in Figure 4, and the code which launches daemons is in Figure 5; the interesting clauses of the compositional translation $\{P\}$ of high-level constructs P (all those involving agents or communication), are in Figure 6. The top-level translation (omitted here) launches all the query servers and daemons before executing the application program.

Each class of agents maintains some explicit state as an output on a lock channel, named `lock` or `currentlock`. For each mobile agent name there is at least one server that has the site and daemon where the agent *is* currently

```
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 4: Parts of the Top Level in the FQSC algorithm – the Query Server

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

   (* launch a daemon D on site S *)
   (* Q is a local Query Server 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 5: Parts of the Top Level in the FQSC algorithm – the Daemon Daemon

located; servers store these data in a map `m`. Each daemon maintains its own map `m` from agent names to the site and daemon where they *guess* the agent is located. This is updated only when a message delivery fails. 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.

The messages sent between agents fall into three groups, implementing the location-independent messages, the high-level agent creation, and agent migration. Typical executions are illustrated in Figures 7, 8 and below. Correspondingly, only these cases of the compositional translation are non-trivial.

Location-independent communication 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 in Fig. 6-1. It first reads from the agent's lock `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 a message `[b`

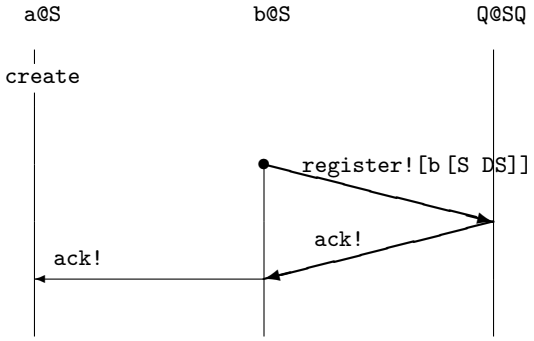
```
1.    {<b @ ?>c ! v}a  =
        currentloc?[S DS Q SQ]=
          iflocal <DS>try_message![b c v] then
            currentloc![S DS Q SQ]
          else ()

2.    { 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 ))

3.    { 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 )))))
```

Figure 6: Compositional translation

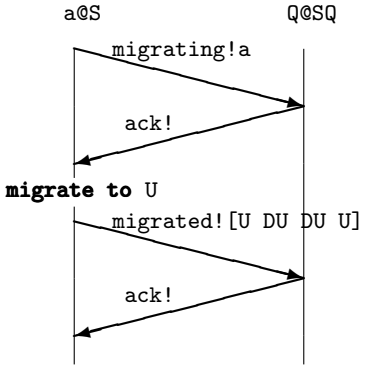
`c v]` on the channel `try_message` to `DS`, replacing the lock after the message is sent. The translation is parametric on the name `a` of the agent containing this phrase – for this phrase, `a` is however not used. We return later to the process of delivery of the message.

Agent creation A high-level agent `a` synchronizes with the query server while creating a new agent `b`, with messages on `register` and `ack` (see Fig. 6-2). 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.

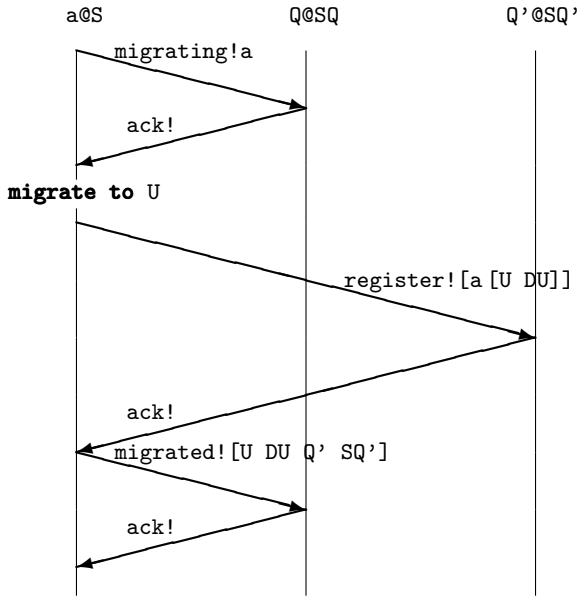


Agent migration To migrate while keeping the local query server's map accurate, the translation of a **`migrate`** in a high-level agent (see Fig. 6-3) must synchronize with the local query server before and after actually migrating, with `migrating`, `migrated`, and `ack` messages. A sample execution of a migration in a local domain is below.



If the target site `U` is in the domain managed by a different query server `Q'` (see an **`else`** clause in Fig. 6-3) 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). A sample execution of a cross-domain migration with registration at `Q'` is following.



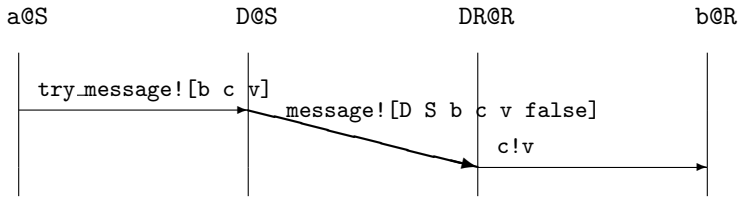
The query server's lock is kept during the migration. The agent's own record of its current site and daemon and its local server must also be updated with the new data `U DU Q' SQ'` when the agent's lock is released. Note that in the body of the encoding the name `DU` of the daemon on the target site 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 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]`.

Message delivery Returning to the process of message delivery, there are three 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.

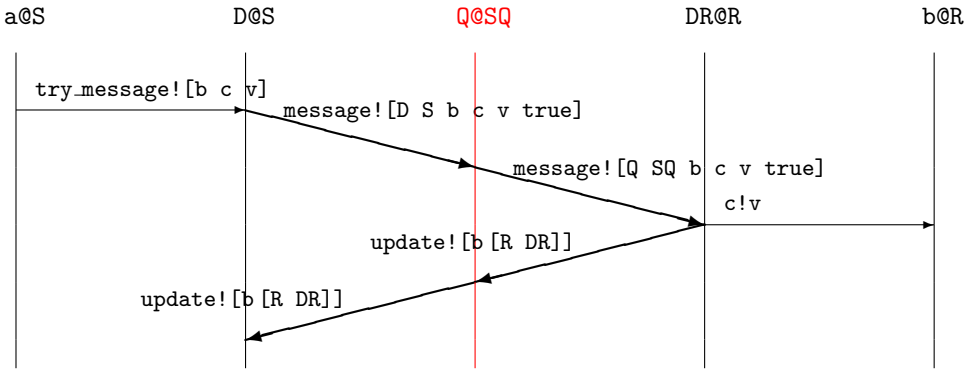
In the first case (see at the top of Fig. 7) `D` sends a `message` message to `DR` which delivers the message to `b` using **`iflocal`**. For the PA application this should be the common case, including the cross-domain communication; it requires only one network message.

In the cache-miss case (see at the bottom of Figure 7) 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

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



No guess in the D cache.



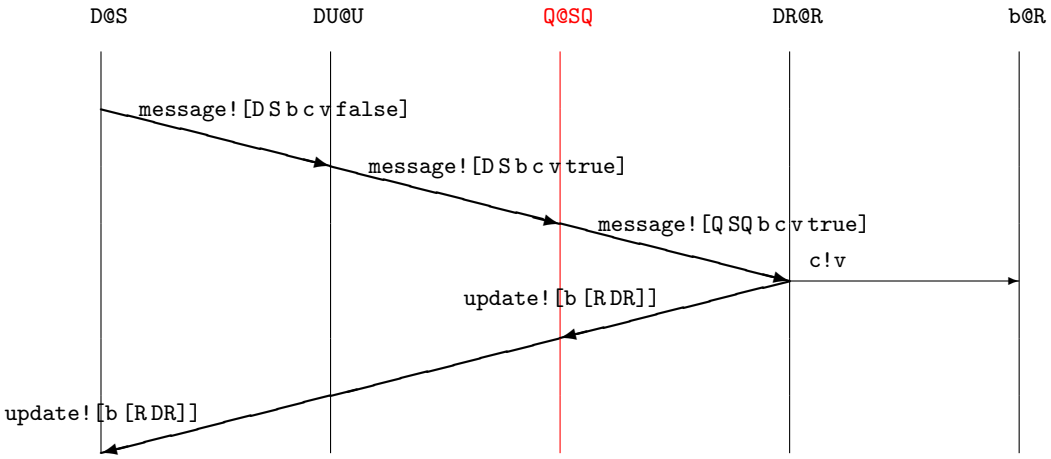
Horizontal arrows are synchronized communications within a single machine (using **iflocal**); slanted arrows are asynchronous messages.

Figure 7: The delivery of location-independent message <b@?>c!v from a to b in the no-guess and good-guess cases

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 8). 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 8). 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).

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.

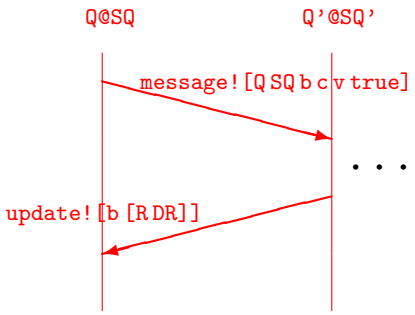


Figure 8: The delivery of location-independent message <b@?>c!v from a to b in the worst scenarios

5 Further Extensions

The FQSC algorithm avoids sending too many cache updates over the Internet. 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: communications are more frequent than migrations, and the inter-domain migrations, which correspond to delegation or a physical movement of individuals, are less frequent than migrations within a local domain. If PA behaviour would be different, it may be worth to collapse the forwarding pointer chains more often. For example, upon each cross-domain migration, the cache of *several* daemons and servers could be updated, not just those last visited.

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.

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 (e.g. using the *group communication* middleware [MSW03]).

6 Related Work

Many authors present strategies for *locating* mobile objects and devices (see, e.g., surveys [WL00, PS01]). Similar to locating objects are mechanisms for resource discovery, e.g. Dimakopoulos and Pitoura [DP03] 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 the 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. Some mobile agent infrastructure algorithms are for locating agents only, which – as we explained above – provides weaker guarantees.

For instance, Mobile Objects and Agents (MOA) [MLC98] supports four schemes for locating agents; these are used as required to deliver location-independent messages. Stream communication between agents is also described, with communicating channel managers informing each other on migration.

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

The infrastructure work of Aridor and Oshima [AO98] provides three main forms of message delivery: location-independent using either forwarding pointers or location servers, and location dependent (they also provide other mechanisms for *locating* an agent).

Roth and Peters [RP01] propose a scalable global service for locating mobile agents, with encryption and decryption capabilities to prevent security attacks through agent impersonating.

The Join Language [FGL+96] provides location-independent messages using a built-in infrastructure, based on forwarding pointer chains that are collapsed when possible.

The Mobile Object Workbench [BHDH98] provides location independent interaction, using a hierarchical directory service for locating clusters of objects that have moved. There is a single infrastructure, although it is stated that the architecture is flexible enough to allow others.

Moreau [Mor01] 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 [Mor02], he describes the directory extended with pointer redundancy to tolerate node crashes; the algorithm has been verified using the proof assistant Coq.

Our model assumes direct message routing, while other approaches are also possible, e.g. Murphy and Picco [MP99] 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. Cao *et al.* [CZYD04, CZFD03] 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 [CZYD04], and path compression for better performance [CZFD03].

The use of home servers in our FQSC algorithm resembles the *Internet Mobile Host Protocol (IMHP)* proposed by Perkins *et al.* [PMJ94] 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 specific migration and communication pattern of PA agents. 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).

7 Conclusion

In this paper we have proposed a distributed algorithm for scalable location-independent message delivery to mobile agents, that is suitable for the Personal Assistants application. The algorithm reflects the expected behaviour of the Personal Assistant agents: communications are more frequent than migrations, and the inter-domain migrations, which correspond to delegation or a physical movement of individuals, are less frequent than migrations within a local domain.

Our algorithm has been presented formally, as an executable specification in the Nomadic Pict 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.

Acknowledgments. We would like to thank Peter Sewell and Asis Unyapoth for many useful discussions.

References

[AO98] Yariv Aridor and Mitsuru Oshima. Infrastructure for mobile agents: Requirements and design. In *Proc. 2nd Int. Workshop on Mobile Agents*, LNCS 1477, September 1998.

[BHDH98] Michael Bursell, Richard Hayton, Douglas Donaldson, and Andrew Herbert. A Mobile Object Workbench. In *Proc. the 2nd Int. Workshop on Mobile Agents*, LNCS 1477, September 1998.

[Car99] Luca Cardelli. Abstractions for mobile computation. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, LNCS 1603, pages 51–94. Springer, 1999.

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

[CZFD03] Jiannong Cao, Liang Zhang, Xinyu Feng, and Sajal K. Das. Path compression in forwarding-based reliable mobile agent communications. In *Proc. ICPP '03: the 32nd Int. Conference on Parallel Processing*, October 2003.

[CZYD04] Jiannong Cao, Liang Zhang, Jin Yang, and Sajal K. Das. A reliable mobile agent communication protocol. In *Proc. ICDCS '04: the 24th Int. Conference on Distributed Computing Systems*, March 2004.

[DP03] Vassilios V. Dimakopoulos and Evaggelia Pitoura. A peer-to-peer approach to resource discovery in multi-agent systems. In *Proc. CIA '03: the 7th Workshop on Cooperative Information Agents*, LNCS 2782, August 2003.

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proc. CONCUR '96: the 7th Int. Conference on Concurrency Theory*, LNCS 1119, August 1996.

[KGR02] David Kotz, Robert Gray, and Daniela Rus. Future directions for mobile agent research. *IEEE Distributed Systems Online*, 3(8), August 2002.

[MBB+98] D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran, and J. White. MASIF: The OMG Mobile Agent System Interoperability Facility. In *Proc. 2nd Int. Workshop on Mobile Agents*, LNCS 1477, September 1998.

[MDW99] Dejan Milojičić, Frederick Douglis, and Richard Wheeler, editors. *Mobility: Processes, Computers, and Agents*. Addison-Wesley, 1999.

[MLC98] Dejan S. Milojičić, William LaForge, and Deepika Chauhan. Mobile Objects and Agents (MOA). In *Proc. COOTS '98: the 4th USENIX Conference on Object-Oriented Technologies and Systems*, April 1998.

[Mor01] Luc Moreau. Distributed directory service and message router for mobile agents. *Science of Computer Programming*, 39(2–3):249–272, 2001.

[Mor02] Luc Moreau. A fault-tolerant directory service for mobile agents based on forwarding pointers. In *Proc. SAC '02: the 17th Symp. on Applied Computing: Track on Agents, Interactions, Mobility and Systems*, March 2002.

[MP99] Amy L. Murphy and Gian Pietro Picco. Reliable communication for highly mobile agents. In *Proc. ASA/MA '99: 1st Int. Symposium on Agent Systems and Applications / 3rd Int. Symposium on Mobile Agents*, October 1999.

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

[MSW03] Sergio Mena, André Schiper, and Paweł T. Wojciechowski. A step towards a new generation of group communication systems. In *Proc. Middleware '03*, LNCS 2672, June 2003.

[PMJ94] Charles Perkins, Andrew Myles, and David B. Johnson. IMHP: a mobile host protocol for the Internet. *Computer Networks and ISDN Systems*, 27(3):479–491, December 1994.

[PS01] Evaggelia Pitoura and George Samaras. Locating objects in mobile computing. *IEEE Transactions on Knowledge and Data Engineering*, 13(4):571 – 592, July/August 2001.

[PT94] Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In *Proc. TPPP '94: Int. Workshop on Theory and Practice of Parallel Programming*, LNCS 907, November 1994.

[PT00] 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.

[RP01] Volker Roth and Jan Peters. A scalable and secure global tracking service for mobile agents. In *Proc. of the 5th Int. Workshop on Mobile Agents*, LNCS 2240, pages 169–181, December 2001.

[SWP99] Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location-independent communication for mobile agents: A two-level architecture. In *Internet Programming Languages*, LNCS 1686, pages 1–31. Springer, 1999.

[TAK01] Anand Tripathi, Tanvir Ahmed, and Neeran M. Karnik. Experiences and future challenges in mobile agent programming. *Microprocessors and Microsystems*, 25(2):121–129, April 2001.

[WL00] Vincent Wong and Victor Leung. Location management for next generation personal communication networks. *IEEE Network*, 14(5):8–14, Sept./Oct. 2000.

[WS00] Paweł T. Wojciechowski and Peter Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. *IEEE Concurrency*, 8(2):42–52, April-June 2000.