A Fault-Tolerant Directory Service for Mobile Agents based on Forwarding Pointers

Luc Moreau
L.Moreau@ecs.soton.ac.uk
Department of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ UK

Keywords
mobile agents, distributed directory service, fault tolerance

ABSTRACT

A reliable communication layer is an essential component of a mobile agent system. We present a new fault-tolerant directory service for mobile agents, which can be used to route messages to them. The directory service, based on a technique of forwarding pointers, introduces some redundancy in order to ensure resilience to stopping failures of nodes containing forwarding pointers; in addition, it avoids cyclic routing of messages, and it supports a technique to collapse chains of pointers that allows direct communications between agents. We have formalised the algorithm and derived a *fully mechanical proof* of its correctness using the proof assistant Coq; we report on our experience of designing the algorithm and deriving its proof of correctness. The complete source code of the proof is made available from the WWW.

1. INTRODUCTION

Mobile agents have emerged as a major programming paradigm for structuring distributed applications [3, 5]. For instance, the MAGNITUDE project [13] investigates the use of mobile agents as intermediary entities capable of *negotiating* access to information resources on behalf of mobile users. Several important issues remain to be addressed before mobile agents become a mainstream technology for such applications: among them, a *communication system* and a *security infrastructure* are needed respectively for facilitating communications between mobile agents and for protecting agents and their hosts.

Here, we focus solely on the problem of communications, for which we have adopted a peer-to-peer communication model using a performative-based agent communication language [11], as prescribed by KQML and FIPA. Various authors have previously investigated a communication layer for mobile agents based on *forwarding pointers* [16, 10]. In such an approach, when mobile agents migrate, they leave forwarding pointers that are used to route messages. A point of concern is to avoid cyclic routing when agents migrate to previously visited sites; additionally, lazy updates and piggy-backing of information on messages can be used to collapse chains of pointers [12]. For structuring and clarity purposes, a communication layer is usually defined in terms of a message router and a directory service; the latter tracks mobile agents' locations, whereas the former forwards messages using the information provided by the latter.

Directory services based on forwarding pointers are currently *not tolerant* to failures: the failure of a node containing a forwarding pointer may prevent finding agents' positions. The purpose of this paper is to present a directory service, fully distributed and resilient to failures exhibited by intermediary nodes, possibly containing forwarding pointers. This algorithm may be used by a fault-tolerant message router (which itself will be the object of another publication).

We consider stopping failures according to which processes are allowed to stop during the course or their execution [7]. The essence of our fault-tolerant distributed directory service is to introduce redundancy of forwarding pointers, typically by making N copies of agents' location information. This type of redundancy ensures the resilience of the algorithm to a maximum of $N-1$ failures of intermediary nodes. We will show that the complexity of the algorithm remains linear in N. Our specific contributions are:

1. A *new directory service* based on forwarding pointers, fault-tolerant, preventing cyclic routing, and not involving any static location;
2. A *full mechanical proof* of its correctness, using the proof assistant Coq [1]; the complete source code of the proof (involving some 25000 tactic invocations) may be downloaded from the following URL [9].

We begin this paper by a survey of background work (Section 2) and follow by a summary of a routing algorithm based on forwarding pointers (Section 3). We present our new directory service and its formalisation as an abstract

SAC 2002, Madrid, Spain
@2002 ACM 1-58113-445-2/02/03 ... $5.00

machine (Section 4). The purpose of Section 5 is to summarise the correctness properties of the algorithm: its safety states that the distributed directory service correctly and uniquely identifies agents' positions, whereas the liveness property shows that the algorithm reaches a stable state after a finite number of transitions, once agents stop migrating. Then, in Section 6, we report on our experience of designing the algorithm and deriving its proof of correctness, and we suggest possible variants or extensions.

2. BACKGROUND

The topic of mobile agent tracking and communication has been researched extensively by the mobile agent community. Very early on, location-aware communications were proposed: they consist of sending messages to locations where agents are believed to be, but typically result in failure when the receiver agent has migrated [15, 19].

For a number of applications, such a service is not satisfactory because the key property is to get messages *reliably* delivered to a recipient, wherever its location and whatever the route adopted (for instance, when two mobile agents undertake a negotiation on how to solve a specific problem). Location-transparent communication services were introduced as a means to route and deliver messages automatically to mobile agents, independently of their migration. (Such services have been shown to be implementable on top of a location-aware communication layer [19].)

In the category of location-transparent communication layers, there are essentially two approaches, respectively based on *home agents* and *forwarding pointers*. In systems based on home agents, such as Aglets [5], each mobile agent is associated with a non-mobile home agent. In order to communicate with a mobile agent, a message has to be sent to its associated home agent, which forwards it to the mobile one; when a mobile agent migrates, it informs its home agent of its new position. Alternatively, in mobile agent systems such as Voyager [16], agents that migrate leave trails of forwarding pointers, which are used to route messages.

In situations such as the pervasive computing environment, the mechanism of a home agent may defeat the purpose of using mobile agents by re-introducing centralisation: the home agent approach puts a burden on the infrastructure, which may hamper its scalability, in particular, in massively distributed systems. A typical illustration is two mobile agents with respective home bases in the US and Europe having to communicate at a host in Australia. In such a scenario, routing via home agents is not desirable, and may not be possible when the host is temporarily disconnected from the network. If we introduce a mechanism by which home agents change location dynamically according to the task at hand, we face the problem of how to communicate reliably with a home agent, which is itself mobile. Alternatively, we could only use the home agent to bootstrap communication, and then shortcut the route, but this approach becomes unreliable once agents migrate. Finally, the home agent also appears as a *single point of failure*: when it exhibits a failure, it becomes impossible to track the mobile agent or to route messages to it.

A naive forwarding pointer implementation causes communications to become more expensive as agents migrate, because chains of pointers increase. Chains of pointers need to be collapsed promptly so that mobile agents become independent of the hosts they previously visited. Once the chain has collapsed direct communications become possible and avoid the awkward scenario discussed above. As far as tolerance to failures is concerned, the crash of an intermediary node with a forwarding pointer prevents upstream nodes to forward messages. Collapsing chains of pointers also has the benefit of reducing the system's exposure to failures.

Coordination models offer a more asynchronous form of communication, typically involving a tuple space [4]. As coordination spaces are non-mobile, they may suffer from the same problem as the home agent; solutions such as distributed spaces may be introduced for that purpose but maintaining consistency is a non-trivial problem. An inconvenient of the coordination approach is that it requires coordinated processes to poll tuple spaces, which may be inefficient in terms of both communication and computation. As a result, tuple spaces generally provide a mechanism by which registered clients can be notified of the arrival of a new tuple: when clients are mobile, we are back to the problem of how to deliver such notifications reliably. If the tuple space itself is mobile [17], the problem is then to deliver messages to the tuple space.

This discussion shows that reliable delivery of messages to mobile agents without using static locations to route messages is essential, even if peer-to-peer communications are not adopted as the high-level interaction paradigm between agents. Previous work has focused on formalisation [10] and implementation [16] of forwarding pointers, but solutions were not fault-tolerant. We summarise such an approach in Section 3 before extending it with support for failures in Section 4.

3. SUMMARY OF DIRECTORY SERVICE

In this section, we summarise the principles of a communication layer based on forwarding pointers [10] without any fault-tolerance. The algorithm comprises two components: a distributed directory service and a message router, which we describe below.

Distributed Directory Service. Each mobile agent is associated with a timestamp that is increased every time the agent migrates. When an agent has autonomously decided to migrate to a new location, it requests the communication layer to transport it to its new destination. When the agent arrives at a new location, an acknowledgement message containing both its new position and its newly-incremented timestamp is sent to its previous location. As a result, for each site, one of the following three cases is valid for each agent A: *(i)* the agent A is local, *(ii)* the agent A is in transit but has not acknowledged its new position yet, or *(iii)* the agent A is known to have been at a remote location with a given timestamp. Timestamps are essential to avoid race conditions between acknowledgement messages: by using timestamps, a site can decide which position information is the most recent, and therefore can avoid creating cycles in the graph of forwarding pointers. In order to avoid an increasing cost of communication when the agent migrates, a mechanism was specified to propagate information

about agent's position, which in turn reduces the length of chains of pointers [10].

Message Router. Sites rely on the information about agents' positions in order to route messages. For any incoming message aimed at an agent A, the message will be delivered to A if A is known to be local. If A is in transit, the message will be enqueued, until A's location becomes known; otherwise, the message is forwarded to A's known location.

Absence of Fault Tolerance. There is no redundancy in the information concerning an agent's location. Indeed, sites only remember the most recent location of an agent, and only the previous agent's location is informed of the new agent's position after a migration. As a result, a site (transitively) pointing at a site exhibiting a failure has lost its route to the agent.

4. FAULT-TOLERANT ALGORITHM

The intuition of our solution to the problem of failures is to introduce some *redundancy* in the information about agents' positions. Two essential elements are used for this purpose. First, agents remember N previous different sites that they have visited; once an agent arrives at a new location, it informs its N previous locations of its new position. Second, sites remember up to N different positions for an agent, and their associated timestamps. We shall establish that the algorithm is able to determine the agent's position correctly, provided that the number of stopping failures remains smaller or equal to $N-1$.

Remark *We aim to design an algorithm which is resilient to failures of intermediary nodes. We are not concerned with reliability of agents themselves. Systems replicating agents and using failure detectors such as [8] may be used for that purpose; they are complementary to our approach.*

We adopt an existing framework [10] to model the distributed directory service as an abstract machine, whose state space is summarised in Figure 1. For the sake of clarity, we consider a single mobile agent; the formalisation can easily be extended to multiple agents by introducing names by which agents are being referred to. An abstract machine is composed of a set of sites taking part in a computation. Agent timestamps, which we call *mobility counters*, are defined as natural numbers. A *memory* is defined as an association list, associating locations with mobility counters; we represent an empty memory by $\emptyset$. The value N is a parameter of the algorithm. We will show that the agent's memory has a size N and that the algorithm tolerates at most $N-1$ failures.

The set of messages is inductively defined by two constructors. These constructors are used to construct messages, which respectively represent an agent in transit and an arrival acknowledgement. The message representing an agent in transit, typically of the form $\mathsf{agent}(s, l, \vec{M})$, contains the site s that the agent is leaving, the value l of the mobility counter it had on that site, and the agent's memory $\vec{M}$, i.e. the N previous sites it visited and associated mobility counters. The message representing an arrival acknowledgement, $\mathsf{ack}(s, l)$, contains the site s (and associated mobility counter l) where the agent is.

We assume that the network is fully connected, that communications are reliable, and that the order of messages in transit between pairs of sites is preserved. These communication hypotheses are formalised in the abstract machine by point-to-point communication links, which we define as queues using the following notations, where the expression $q_1 \S q_2$ denotes the concatenation of two queues q_1, q_2, and $first(q)$ the head of a queue q.

Each site maintains some information, which we abstract as "tables" in the abstract machine. The *location table* maps each site to a memory; for a site s, the location table indicates the sites where s believes the agent has migrated to (with their associated mobility counter). The *present table* is meant to be empty for all sites, except for the site where the agent is currently located, when the agent is not in transit; there, the present table contains the sites previously visited by the agent. The *mobility counter table* associates each site with the mobility counter the agent had when it last visited the site; the value is zero if the agent has never visited the site.

After the agent has reached a new destination, acknowledgement messages have to be sent to the N previous sites it visited. We decouple the agent's arrival from acknowledgement sending, so that transitions that deal with incoming messages are different from those that generate new messages. Consequently, we introduce a further table, the *acknowledgement table*, indicating which acknowledgements still have to be sent.

In our formalisation, we use a variable to indicate whether a machine is up and running. A site's *failure state* is allowed to change from false to true, which indicates that the site is exhibiting a failure. We are modelling *stopping failures* [7] since no transition allows a failure state to change from true to false.

A complete configuration of the abstract machine is defined as the Cartesian product of all tables and message queues. Our formalisation can be regarded as an *asynchronous* distributed system [7]. In a real implementation, tables are not shared resources, but their contents can be distributed at each site.

The behaviour of the algorithm is represented by transitions, which specify how the state of the abstract machine evolves. Figure 2 contain all the transitions of the distributed directory service. Transitions are assumed to be executed atomically. For convenience, we use some notations such as *post*, *receive* or table updates, which give an imperative look to the algorithm; their definitions is as follows. Given a configuration $\langle loc\_T, present\_T, mob\_T, ack\_T, fail\_T, k\rangle$, $mob\_T(s) := V$ denotes $\langle loc\_T, present\_T, mob\_T', ack\_T, fail\_T, k\rangle$, such that $mob\_T'(s) = V$ and $mob\_T'(s') = mob\_T(s')$, $\forall\ s' \neq s$. A similar notation is used for other tables. Given a configuration, $post(s_1, s_2, m)$ denotes $\langle loc\_T, present\_T, mob\_T, ack\_T, fail\_T, k'\rangle$, with $k'(s_1, s_2) = k(s_1, s_2)\S\{m\}$, and $k'(s_i, s_j) = k(s_i, s_j)$, $\forall (s_i, s_j) \neq (s_1, s_2)$. A similar notation is used for *receive*.

$\mathcal{S}$	=	$\{s_0, s_1, \ldots, s_{n_s}\}$	(Set of Sites)	*Characteristic variables* :
$\mathcal{L}$	=	$\mathbb{N}$	(Mobility Counters)	$s \in \mathcal{S}$
Ψ	=	$list(\mathcal{S} \times \mathcal{L})$	(Memory)	$m \in \mathcal{M}$
N	$\in$	$\mathbb{N}$	(Algorithm Parameter)	$k \in \mathcal{K}$
$\mathcal{M}$	:	$\mathsf{agent} : \mathcal{S} \times \mathcal{L} \times \Psi \to \mathcal{M} \mid \mathsf{ack} : \mathcal{S} \times \mathcal{L} \to \mathcal{M}$	(Messages)	$c \in \mathcal{C}$
$\mathcal{K}$	=	$\mathcal{S} \times \mathcal{S} \to Queue(\mathcal{M})$	(Message Queues)	$\vec{M} \in \Psi$
$\mathcal{LT}$	=	$\mathcal{S} \to \Psi$	(Location Tables)	$loc\_T \in \mathcal{LT}$
$\mathcal{PT}$	=	$\mathcal{S} \to \Psi$	(Present Tables)	$present\_T \in \mathcal{PT}$
$\mathcal{MT}$	=	$\mathcal{S} \to \mathcal{L}$	(Mobility Counter Tables)	$mob\_T \in \mathcal{MT}$
$\mathcal{AT}$	=	$\mathcal{S} \to \Psi$	(Acknowledgement Tables)	$ack\_T \in \mathcal{AT}$
$\mathcal{FT}$	=	$\mathcal{S} \to Bool$	(Failure State)	$fail\_T \in \mathcal{FT}$
$\mathcal{C}$	=	$\mathcal{LT} \times \mathcal{PT} \times \mathcal{MT} \times \mathcal{AT} \times \mathcal{FT} \times \mathcal{K}$	(Configurations)	$q \in Queue(\mathcal{M})$

Figure 1: State Space

In each rule of Figure 2, the conditions that appear to the left-hand side of an arrow are guards that must be satisfied in order to be able to fire a transition. For instance, the first four rules contain a proposition of the form $\neg fail\_T(s)$, which indicates that the rule has to occur for a site s that is up and running. The right-hand side of a rule denotes the configuration that is reached after transition. We assume that guard evaluation and new configuration construction are performed atomically. In order to illustrate our rules, we present graphical representations of configurations; the first part of Figure 3 illustrates an agent that has successively visited sites s_0, s_1, s_2, s_3 with respective timestamps $t-1, t, t+1, t+2$. In this example, we assume that the value of N is 3. (Note that s_0 is not represented in the figure.)

The first transition of Figure 2 models the actions to be performed, when an agent decides to migrate from s_1 to s_2. In the guard, we see that the present table at s_1 must be non-empty, which indicates that the agent is present at s_1. After transition, the present table at s_1 is cleared, and an agent message is posted between s_1 and s_2; the message contains the agent's origin s_1, its mobility counter $mob\_T(s_1)$, and the previous content of the present table at s_1. Note that s_2, the destination of the agent, is only used to specify which communication channel the agent message must be enqueued into. The site s_1 does not need to be communicated this information, nor does it have to remember that site. In a real implementation, the agent message would also contain the complete agent state to be restarted by the receiver. The second part of Figure 3 illustrates changes in the system, when an agent has initiated its migration.

The second transition is concerned with s_2 handling a message[1] $\mathsf{agent}(s_3, l, \vec{M})$ coming from s_1. Tables are updated to reflect that s_2 is becoming the new agent's location, with $l+1$ its new mobility counter. Our algorithm prescribes the agent to remember N *different* sites it has visited. As s_2 may have been visited recently, we remove s_2 from $\vec{M}$, before adding the site s_3 where it was located before migration. The call $add(N, s, l, \vec{M})$ adds an association (s, l) to the memory $\vec{M}$, keeping at most N different entries with the highest timestamps. (Appendix A contains the complete definition of *add*.) In addition, the acknowledgement table of s_2 is updated, since acknowledgements have to be sent back to those previously visited sites. At this point, a proper implementation would reinstate the agent state and resume its execution. The third part of Figure 3 illustrates the system as an agent arrives at a new location.

According to the third transition, if the acknowledgement table on s_1 contains a pair (s_2, l_2), then an acknowledgement message $\mathsf{ack}(s_1, (mob\_T(s_1)))$ has to be sent from s_1 to s_2; the acknowledgement message indicates that the agent is on s_1 with a mobility counter $mob\_T(s_1)$.

If a site s_2 receives an acknowledgement message about site s_3 and mobility counter l, its location table has to be updated accordingly. Let us note two properties of this rule. First, we do not require the emitter s_1 of the acknowledgement message to be equal to s_3; this property allows us to use the same message for propagating more information about the agent's location. Second, we make sure that updating the location table *(i)* maintains information about *different* locations, *(ii)* does not overwrite existing location information with older one. This functionality is implemented by the function *add*, whose specification may be found in appendix A.

According to rule inform of Figure 2, any site s_1 believing that the agent is located at site s_3, with a mobility counter l, may elect to communicate its belief to another site s_2. Such a belief is also communicated by an ack message. It is important to distinguish the roles of the send_ack and inform transitions. The former is mandatory to ensure the correct behaviour of the algorithm, whereas the latter is optional. The purpose of inform is to propagate information about the agent's location in the system, so that the agent may be found in less steps. As opposed to previous rules, the inform rule is non-deterministic in the destination and location information in an acknowledgement message. At this level, our goal is to define a correct *specification* of an algorithm: any implementation strategy will be an instance of this specification; some of them are discussed in Section 6. The first part of Figure 4 illustrates the states of the system after sending acknowledgement messages, whereas the second one shows the effect of such messages.

[1]Note that s_3 is not required to be equal to s_1. Indeed, we want the algorithm to be able to support sites that forward incoming agents to other sites.

For a configuration $\langle loc\_T, present\_T, mob\_T, ack\_T, fail\_T, k\rangle$, legal transitions are:

migrate_agent(s_1, s_2) :

$s_1 \neq s_2 \ \wedge \ loc\_T(s_1) = \emptyset \ \wedge \ present\_T(s_1) \neq \emptyset$
$\wedge \ ack\_T(s_1) = \emptyset \ \wedge \ \neg fail\_T(s_1)$
$\rightarrow \ \{ \quad let \ \vec{M} = present\_T(s_1)$
$\quad in \quad present\_T(s_1) := \emptyset$
$\quad post(s_1, s_2, \mathsf{agent}(s_1, mob\_T(s_1), \vec{M})) \ \}$

receive_agent$(s_1, s_2, s_3, l, \vec{M})$:

$first(k(s_1, s_2)) = \mathsf{agent}(s_3, l, \vec{M}) \ \wedge \ \neg fail\_T(s_2)$
$\rightarrow \ \{ \quad receive(s_1, s_2)$
$\quad let \quad S' = add(N, s_3, l, remove(s_2, \vec{M}))$
$\quad in \quad loc\_T(s_2) := \emptyset$
$\quad present\_T(s_2) := S'$
$\quad mob\_T(s_2) := l + 1$
$\quad ack\_T(s_2) := S' \ \}$

send_ack$(s_1, s_2, \vec{M}, l_2)$:

$ack\_T(s_1) = (s_2, l_2) \ \S \ \vec{M} \ \wedge \ \neg fail\_T(s_1)$
$\rightarrow \ \{ \quad ack\_T(s_1) := \vec{M}$
$\quad post(s_1, s_2, \mathsf{ack}(s_1, mob\_T(s_1))) \ \}$

receive_ack(s_1, s_2, s_3, l) :

$first(k(s_1, s_2)) = \mathsf{ack}(s_3, l) \ \wedge \ \neg fail\_T(s_2)$
$\rightarrow \ \{ \quad receive(s_1, s_2)$
$\quad loc\_T(s_2) := add(N, s_3, l, loc\_T(s_2)) \ \}$

inform(s_1, s_2, s_3, l) :

$(s_3, l) \in loc\_T(s_1) \ \wedge \ \neg fail\_T(s_1)$
$\rightarrow \ \{ \quad post(s_1, s_2, \mathsf{ack}(s_3, l)) \ \}$

stop_failure(s) :

$fail\_T(s) = false$
$\rightarrow \ \{ \quad fail\_T(s) = true \ \}$

msg_failure(s_1, s_2, m) :

$first(k(s_1, s_2)) = m \ \wedge \ fail\_T(s_2)$
$\rightarrow \ \{ \quad receive(s_1, s_2) \ \}$

Figure 2: Fault-Tolerant Directory Service

Failure. The first five rules of Figure 2 require the site s where the transition takes place to be up and running, i.e. $\neg fail\_T(s)$. Our algorithm is designed to be tolerant to *stopping failure*, according to which processes are allowed to stop somewhere in the middle of their execution [7]. We model a stopping failure by the transition stop_failure, changing the failure state of the site that exhibits the failure. Consequently, a site that has stopped will be prevented from performing any of the first five transitions of Figure 2.

As far as distributed system modelling is concerned, it is unrealistic to consider that messages that are in transit on

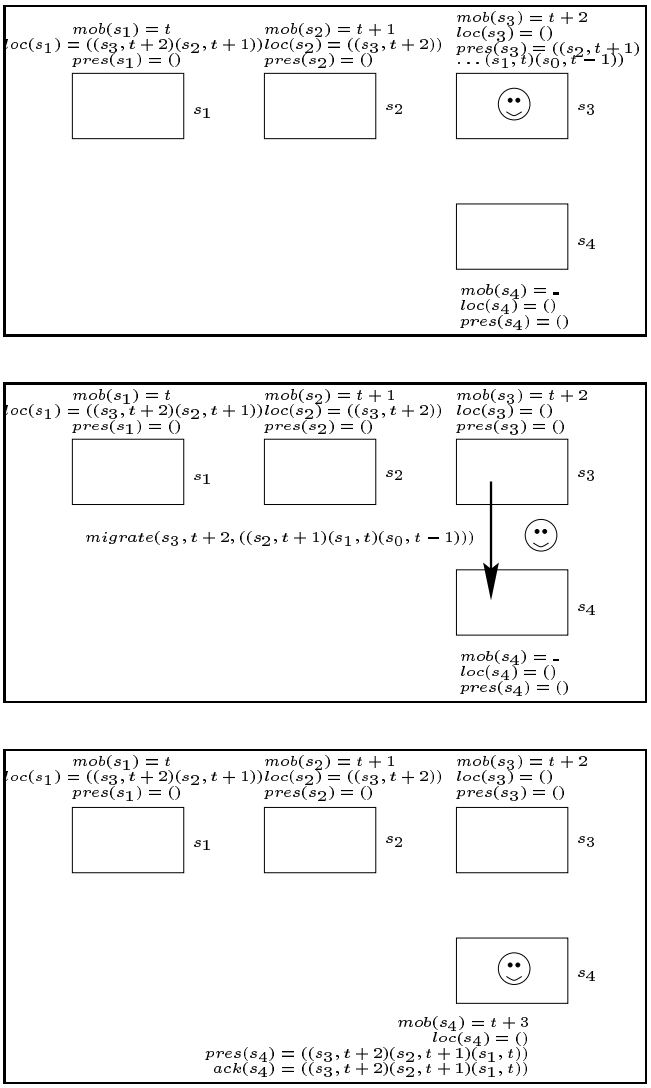
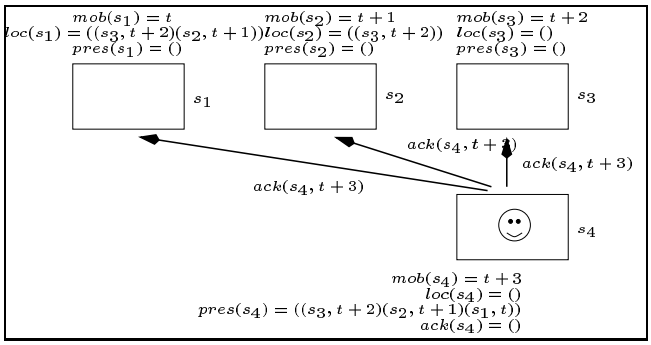


Figure 3: Agent Migration (part 1)

a communication link remain present if the destination of the communication link exhibits a failure. Rule msg_failure shows how messages in transit to a stopped site may be lost. A similar argument may also hold for messages that were posted (but not sent yet) at a site that stops. We could add an extra rule handling such a case, but we did not do so in order to keep the number of rules limited. As a result, our communication model can be seen as using buffered inputs and unbuffered outputs.

Initial and Legal Configurations. In the initial configuration, noted c_i, we assume that the agent is at a given site *origin* with a mobility counter set to $N+1$. Obviously, at creation time, an agent cannot have visited N sites previously. Instead, the creation process elects a set $\mathcal{S}_i$ of different sites that act as "backup routers" for the agent in the initial configuration. Each site is associated with a different mobility counter in the interval $[1, N]$. Such N sites could be chosen non-deterministically by the system or could be



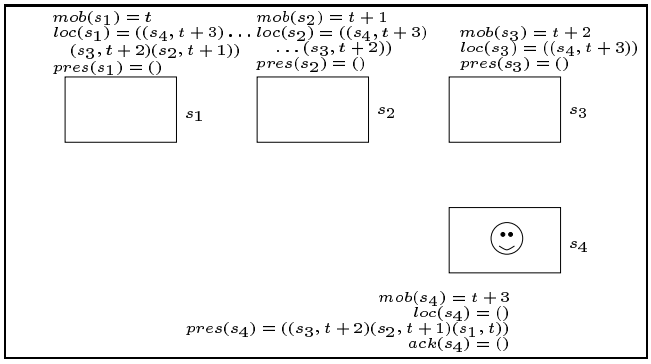


Figure 4: Agent Migration (part 2)

configured manually by the user. For each site in $\mathcal{S}_i$, the location table points to the origin and to sites of $\mathcal{S}_i$ with a higher mobility counter; the location table at all other sites contains the origin and the $N-1$ first sites of $\mathcal{S}_i$. The present table at *origin* contains the sites in $\mathcal{S}_i$. A detailed formalisation of the initial configuration is available from [9]. A configuration c is said to be *legal* if there is a sequence of transitions $t_1, t_2, \ldots, t_n$ such that c is reachable from the initial configuration: $c_i \mapsto^{t_1} c_1 \mapsto^{t_2} c_2 \ldots \mapsto^{t_n} c$. We define $\mapsto^*$ as the reflexive, transitive closure of $\mapsto$.

5. CORRECTNESS

The correctness of the distributed directory service is based on two properties: safety and liveness. The *safety* of the distributed directory service ensures that it correctly tracks the mobile agent's location, in particular in the presence of failures. The *liveness* guarantees that agent location information eventually gets propagated.

We *intuitively* explain the safety property proof as follows. An acknowledgement message results in the creation of a forwarding pointer that points towards the agent's location. Forwarding pointers may be modelled by a relationship *parent* that defines a directed acyclic graph leading to the agent's location.

In the presence of failures, we show that the relationship parent contains sufficient redundancy in order to guarantee the existence of a path leading to the agent, without involving any failed site: *(i)* Sites that belong to the agent's memory have the agent's location as a parent. *(ii)* Sites that do not belong to the agent's memory have at least N parents. Consequently, if the number of failures is strictly inferior to N, each site has always at least one parent that is closer to the agent's location; by repeating this argument, we can find the agent's location.

We summarise the liveness result similar to the one in [10]. A *finite* amount of transitions can be performed from any legal configuration (if we exclude migrate_agent and inform). Furthermore, we can prove that, if there is a message at the head of a communication channel, there exists a transition of the abstract machine that consumes that message. Consequently, if we assume that message delivery and machine transitions are fair, and if the mobile agent is stationary at a location, then location tables will eventually be updated, which proves the liveness of the algorithm.

All proofs were mechanically derived using the proof assistant Coq [1]. Coq is a theorem prover whose logical foundation is constructive logic. The crucial difference between constructive logic and classical logic is that $\neg\neg p \implies p$ does not hold in constructive logic. The consequence is that the formulation of proofs and properties must make use of constructive and decidable statements. Due to space restriction, we do not include the proofs but they can be downloaded from [9]. The notation adopted here are pretty-printed versions of the mechanically established ones.

6. ALGORITHM AND PROOF DISCUSSION

The constructive proof of the initial algorithm without fault-tolerance helped us understand the different invariants that needed to be preserved. In particular, the algorithm maintains a directed acyclic graph leading to the agent's position; interestingly, short-cutting chains of pointers by propagating acknowledgement messages ensures that the graph remains connected and acyclic. Using the same mechanism of timestamp in combination with replication preserves a similar invariant in the presence of failures.

The resulting algorithm turned out to be simpler because it uses less rules, and its correctness proof was easier to derive. When N is equal to 1, the algorithm has the same observable behaviour as [10]. From a practical point of view, generating the mechanical proof still remained a tedious process, though simpler, because it needed some 25000 tactic invocations, of which 5000 for the formalisation of the abstract machine were reused from our initial work.

The complexity of the algorithm is linear in N as far as the number of messages (N acknowledgement messages per migration), message length (size of a memory is $O(N)$), space per site (size of a memory is $O(N)$), and time per migration are concerned. Our proof established the correctness in the worst-case scenario. Indeed, the algorithm may tolerate more than N failures provided that one parent, at least, remains up and running for each site.

For a given application, the designer will have to choose the value of N. If N is chosen to be equal to the number of nodes in the network, the system will be fully realiable but its complexity, even though linear, is too high on an Internet scale. Instead, an engineering decision should be made: in a practical network, from network statistics, one can derive the probability of obtaining $1, 2, \ldots, N$ simultaneous fail-

ures. For each application, and for the quality of service it requires, the designer selects the appropriate failure probability, which determines the number of simultaneous failures the system should be able to tolerate.

A remarkable property of the algorithm is that it does not impose any delay upon agents when they initiate a migration. Forwarding pointers are created temporarily until a stable situation is reached and they are removed. This has to be contrasted with the home agent approach, which requires the agent to notify its homebase, before and after each migration. Interestingly, our algorithm does not preclude us also from using other algorithms; we could envision a system where such algorithms are selected at runtime according to the network conditions and the quality of service requirements of the application.

Propagating agent location information with rule inform is critical in order to shorten chains of forwarding pointers, because shorter chains reduce the cost of finding an agent's location. The ideal strategy for sending these messages depends on the type of distributed system, and on the applications using the directory service. A range of solutions is possible and two extremes of the spectrum are easily identifiable. In an eager strategy, every time a mobile agent migrates, its new location is broadcasted to all other sites; such a solution is clearly not acceptable for networks such as the Internet. Alternatively, a lazy strategy could be adopted [12] but it requires cooperation with the message router. The recipient of a message may inform its emitter, when the recipient observes that that the emitter has out-of-date routing information. In such a strategy, tables are only updated when user messages are sent.

In Section 4, communication channels in the abstract machine are defined as queues. We have established that swapping any two messages in a given channel does not change the behaviour of the algorithm; in other words, messages do not need to be delivered in order.

Message Router. This paper studied a distributed directory service, and we can sketch two possible uses for message routing.

Simple Routing. The initial message router [10] can be adopted to the new distributed directory service. A site receiving a message for an agent that is not local forwards the message to the site appearing in its location table with the highest mobility counter; if the location table is empty, messages are accumulated until the table is updated. This simple algorithm does not use the redundancy provided by the directory service and is therefore not tolerant to failure.

Parallel Flooding. A site must endeavour to forward a message to N sites. If required, it has to keep copies of messages until N acknowledgements have been received. By making use of redundancy, this algorithm would guarantee the delivery of messages. We should note that the algorithm needs a mechanism to clear messages that have been delivered and are still held by intermediate nodes.

Further Related Work. Murphy and Picco [14] present a reliable communication mechanism for mobile agents. Their study is not concerned with nodes that exhibit failures, but with the problem of guaranteeing delivery in the presence of runaway agents. Whether their approach could be combined with ours remains an open question.

Lazar *et al.* [6] migrate mobile agents along a logical hierarchy of hosts, and also use that topology to propagate messages. As a result, they are able to give a logarithmic bound on the number of hops involved in communication. Their mechanism does not offer any redundancy: consequently, stopping failures cannot be handled, though they allow reconnections of temporarily disconnected nodes.

Baumann and Rothermel [2] introduce the concept of a shadow as a handle on a mobile agent that allows applications to terminate a mobile agent execution by notifying the termination to its associated shadow. Shadows are also allowed to be mobile. Forwarding pointers are used to route messages to mobile agents and mobile shadows. Some fault-tolerance is provided using a mechanism similar to Jini leases, requiring message to be propagated after some timeout. This differs from our approach that relies on information replication to allow messages to be routed through multiple routes.

Mobile computing devices share with mobile agents the problem of location tracking. Prakash and Singhal [18] propose a distributed location directory management scheme that can adapt to changes in geographical distribution of mobile hosts population in the network and to changes in mobile host location query rate. Location information about mobile hosts is replicated at $O(\sqrt{m})$ base stations, where m is the total number of base stations in the system. Mobile hosts that are queried more often than others have their location information stored at a greater number of base stations. The proposed algorithm uses replication to offer improved performance during lookups and updates, but not to provide any form of fault tolerance.

7. CONCLUSION

In this paper, we have presented a fault-tolerant distributed directory service for mobile agents. Combined with a message router, it provides a reliable communication layer for mobile agents. The correctness of the algorithm is stated in terms of its safety and liveness.

Our formalisation is encoded in the mechanical proof assistant Coq, also used for carrying out the proof of correctness. The constructive proof gives us a very good insight on the algorithm, which we want to use to specify a reliable message router. This work is part of an effort to define a mechanically proven correct mobile agent system. Besides message routing, we also intend to investigate and formalise security and authentication methods for mobile agents.

8. ACKNOWLEDGEMENTS

Thanks to Nick Gibbins, Dan Michaelides, Victor Tan and the anonymous referees for their comments. This research is funded in part by QinetiQ and EPSRC Magnitude project (reference GR/N35816).

9. REFERENCES

[1] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Mu noz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997.

[2] J. Baumann and K. Rothermel. The shadow approach: An orphan detection protocol for mobile agents. In *Second Int. Workshop on Mobile Agents MA'98*, Lecture Notes in Computer Science, pages 2–13. Springer-Verlag, 1998.

[3] Krishna Bharat and Luca Cardelli. Migratory Applications. In Christian Tschudin and Jan Vitek, editors, *Mobile Object Systems: Towards the Programmable Internet*, pages 131–149. Springer-Verlag, April 1997. Lecture Notes in Computer Science No. 1222.

[4] G. Cabri, L. Leonardi, and F. Zambonelli. Reactive Tuple Spaces for Mobile Agent Coordination . In *Proceedings of the 2nd International Workshop on Mobile Agents (MA'98)*, number 1477 in LNCS, 1998.

[5] Danny B. Lange and Mitsuru Ishima. *Program and Deploying Java Mobile Agents with Aglets.* Addison-Wesley, 1998.

[6] Sashi Lazar, Ishan Weerakoon, and Deepinder Sidhu. A Scalable Location Tracking and Message Delivery Scheme for Mobile Agents. Technical report, University of Maryland, 1998.

[7] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, December 1995.

[8] S. Mishra, X. Jiang, and B. Yang. Providing Fault Tolerance to Mobile Intelligent Agents. In *Proceedings of the ISCA 8th International Conference on Intelligent Systems*, Denver, June 1999.

[9] Luc Moreau. A Fault-Tolerant Distributed Directory Service for Mobile Agents: the Constructive Proof in Coq. Available from `http://www.ecs.soton.ac.uk/~lavm/coq/failure/`, September 2000.

[10] Luc Moreau. Distributed Directory Service and Message Router for Mobile Agents. *Science of Computer Programming*, 39(2–3):249–272, 2001.

[11] Luc Moreau, Nick Gibbins, David DeRoure, Samhaa El-Beltagy, Wendy Hall, Gareth Hughes, Dan Joyce, Sanghee Kim, Danius Michaelides, Dave Millard, Sigi Reich, Robert Tansley, and Mark Weal. SoFAR with DIM Agents: An Agent Framework for Distributed Information Management. In *The Fifth International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agents*, pages 369–388, Manchester, UK, April 2000.

[12] Luc Moreau and Daniel Ribbens. Mobile Objects in Java. *Scientific Programming*, 2002. Special issue of the International Workshop on Performance-oriented Application Development for Distributed Architectures (PADDA'2001).

[13] Luc Moreau, David De Roure, Wendy Hall, and Nick Jennings. MAGNITUDE: Mobile AGents Negotiating for ITinerant Users in the Distributed Enterprise. http://www.ecs.soton.ac.uk/ lavm/magnitude/, 2001.

[14] Amy L. Murphy and Gian Pietro Picco. Reliable Communication for Highly Mobile Agents. In *First International Symposium on Agent Systems and Applications/Third International Symposium on Mobile Agents (ASA/MA'99)*, October 1999.

[15] Saurab Nog, Sumit Chawla, and David Kotz. An RPC mechanism for transportable agents. Technical Report TR96-280, Department of Computer Science, Dartmouth College, Hanover, N.H., 1996.

[16] ObjectSpace. Voyager. `http://www.objectspace.com/`.

[17] Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. LIME: Linda meets mobility. In *International Conference on Software Engineering*, pages 368–377, 1999.

[18] R. Prakash and M. Singhal. A Dynamic Approach to Location Management in Mobile Computing Systems. In *The 8th International Conference on Software Engineering and Knowledge Engineering (SEKE'96)*, pages 488–495, Lake Tahoe, Nevada, June 1996.

[19] Pawel Wojciechowski and Peter Sewell. Nomadic Pict: Language and Infrastructure Design for Mobile Agents. In *First International Symposium on Agent Systems and Applications/Third International Symposium on Mobile Agents (ASA/MA'99)*, October 1999.

A. ADD FUNCTION

The function *add* adds a pair site–timestamp to an association list, making sure that no two entries have a same timestamp or site. A maximum of N entries is kept in the association list, and they are sorted by decreasing timestamp order.

A functional definition of *add* (close to its definition in Coq) appears below, and it uses auxiliary functions `remove` to remove an entry with a specific site from an association list and `firstN` which keeps the first N entries of an association list.

```
fun add (N:int;s1:site;n1:int;q:(Alist site int)) :=
   (firstN N (insert s1 n1 q))

fun insert (s:site;n:int;q:(Alist site int)) :=
  match q
   nil => (cons (s,l) q)
   (cons (s1,n1) q') =>
       if (s=s1)
       then
         if (n <=  n1)
         then q
         else (cons (s,n) q')
       else
         if (n<n1)
         then (cons (s1,n1) (insert s n q'))
         elif (n=n1)
         then q
         else (cons (s,n) (remove s q))
```