A Fault-Tolerant Distributed Directory Service for Mobile Agents

Luc Moreau
L.Moreau@ecs.soton.ac.uk
Department of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ UK
Technical Report M2000/X3

Abstract

Two major approaches have been proposed to route messages transparently to mobile agents. Either messages may be sent to a fixed “home agent”, which forwards them to mobile agents, or messages may follow forwarding pointers left by mobile agents. A home agent appears as a single point of failure: when it exhibits a failure, it becomes impossible to track mobile agents or to route messages to them. Similarly, the failure of a node containing a forwarding pointer may prevent the delivery of messages routed through that node; however, the distributed nature of the forwarding pointer approach offers opportunities to introduce redundancy in order to tolerate failures.

In the context of a communication layer for mobile agents, we define a directory service as the component that tracks mobile agents’ positions; the information it provides may be used to route messages to mobile agents. We have designed a fault tolerant distributed directory service for mobile agents, which extends the forwarding pointer approach, by introducing redundancy in the information about agents’ locations. The correctness of the algorithm is established by proving two properties. The safety property ensures that the location returned by the directory service is the one occupied by the agent; the liveness property guarantees that location information gets updated as agents migrate. We have formalised the algorithm and derived a fully mechanical proof of its correctness using the proof assistant Coq; the complete source code of the proof is made available from the WWW.

1 Introduction

Over the last few years, mobile agents have emerged as a major programming paradigm for structuring distributed applications [4, 6]. For the purpose of this discussion, we adopt a generic view of mobile agents, which we define as some entity capable of migrating to
new locations, typically in order to take advantage of better resources (such as processor, communication link, . . . ). However, several issues remain to be addressed before mobile agents become a mainstream technology: a robust and scalable communication system is needed to facilitate communications between mobile agents [11, 13, 7, 16]; a security infrastructure is required for managing authorisations and for protecting agents and their hosts [1, 14].

This paper focuses on the specific problem of communications between mobile agents. Two major approaches exist, respectively based on home agents and forwarding pointers. In systems based on home agents, such as Aglets [6] or the latest version of the Internet Protocol [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. In some situations, the mechanism of a home agent may defeat the purpose of using mobile agents by reintroducing centralisation: the home agent approach puts a burden on the infrastructure, which may hamper its scalability, in particular, in massively distributed systems. Furthermore, 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.

In mobile agents systems such as Voyager [15], migrating agents leave trails of forwarding pointers, which are used to route messages. In order to avoid an increase of the communication cost when agents migrate, a mechanism is required to collapse chains of forwarding pointers. In a previous paper [11], we formalised a communication layer for mobile agents based on forwarding pointers; we particularly focused on avoiding cyclic routing when agents migrate to previously visited sites. For structuring and clarity purposes, our communication layer is composed of a directory service and a message router; the latter tracks mobile agents’ locations, whereas the former forwards messages using the information provided by the latter. Our directory service is distributed as it does not rely on any central or fixed location, but instead maintains knowledge about agents’ locations in a distributed manner. Our approach appears flexible, because it does not depend on a fixed home agent to route messages; still, it is not tolerant to failures because the failure of a node containing a forwarding pointer may prevent finding the agent’s position. However, the distributed nature of the forwarding pointer approach offers opportunities to tolerate failures by introducing redundancy.

In this paper, we extend the distributed directory service in order to make it resilient to failures exhibited by intermediary nodes, possibly containing forwarding pointers. This algorithm will be used by a fault-tolerant message router, which will be the object of another publication. Combining these algorithms will provide a reliable communication infrastructure for mobile agents.

We consider stopping failures, according to which processes are allowed to stop during the course or their execution [8]. 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 $N$.

We have formalised the algorithm and derived a fully mechanical proof of its correct-
ness using the proof assistant Coq [3]. The complete source code of the proof (involving some 25000 tactic invocations) may be downloaded from the following URL [10].

The paper is organised as follows. We present the intuition of the algorithm in Section 2, while we study its formalisation as an abstract machine in Section 3. The purpose of Section 4 is to investigate the correctness 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, we discuss our algorithm in Section 5, suggesting possible variants or extensions. Finally, our work is compared with related proposals in Section 6.

2 The Algorithm

In this section, we summarise the principles of our initial distributed directory service (and its associated message router) [11]. We explain why it is not fault-tolerant, and then we generalise it in order to make it tolerant to stopping failures.

We consider a number of sites able to host mobile agents. Each mobile agent is associated with a timestamp, which 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, we can decide which position information is the most recent, and therefore we can avoid creating cycles in the graph of forwarding pointers.

Sites rely on the information about agents’ positions in order to route messages; they use the following algorithm. 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 the agent’s location becomes known; otherwise, the message is forwarded to the agent’s known location.

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.

The intuition of our solution to this 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.

The cost of introducing such a redundancy remains linear in $N$. We will 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** The focus of the paper is a reliable communication mechanism for mobile
agents. In particular, we aim to design an algorithm which is resilient to failure
of intermediary nodes. We are not concerned with reliability of agents themselves.
Systems replicating agents and using failure detectors such as [9] may be used for
that purpose; they are complementary to our approach.

3 Formalisation

Using the same approach we adopted for other algorithms [11, 12], we formalise 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 algorithm
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 the 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
\( \text{agent}(s, l, \vec{M}) \), contains the site \( s \) 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,
typically of the form \( \text{ack}(s, l) \), 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 communi-
cation hypotheses are formalised in the abstract machine by point-to-point communication
links, which we define as queues. We shall see that the order preserving constraint
may be relaxed. We use the following notations and operations on queues:

\[
\begin{align*}
q, q_1, \ldots & : \text{denote queues;} \\
\text{first}(q) & : \text{head of a non empty-queue } q; \\
q \gets \{ m \} & : \text{queue } q \text{ after adding a message } m \text{ at its tail;} \\
q_1 \searrow q_2 & : \text{queue obtained after concatenating } q_1 \text{ and } q_2. \\
\end{align*}
\]

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
\[
S = \{s_0, s_1, \ldots, s_n\} \quad \text{(Set of Sites)}
\]

\[
L = \mathbb{N} \quad \text{(Mobility Counters)}
\]

\[
\Psi = \text{list } S \times L \quad \text{(Memory)}
\]

\[
N \in \mathbb{N} \quad \text{(Algorithm Parameter)}
\]

\[
M : \text{agent} : S \times L \times \Psi \to M \ | \ \text{ack} : S \times L \to M \quad \text{(Messages)}
\]

\[
K = S \times S \to \text{Queue}(M) \quad \text{(Message Queues)}
\]

\[
\mathcal{L}T = S \to \Psi \quad \text{(Location Tables)}
\]

\[
\mathcal{P}T = S \to \Psi \quad \text{(Present Tables)}
\]

\[
\mathcal{M}T = S \to L \quad \text{(Mobility C. Tables)}
\]

\[
\mathcal{A}T = S \to \Psi \quad \text{(Acknowledgement Tables)}
\]

\[
\mathcal{F}T = S \to \text{Bool} \quad \text{(Failure State)}
\]

\[
\mathcal{C} = \mathcal{L}T \times \mathcal{P}T \times \mathcal{M}T \times \mathcal{A}T \times \mathcal{F}T \times K \quad \text{(Configurations)}
\]

Characteristic variables:

\[
s \in S, \ m \in M, \ k \in K, \ c \in \mathcal{C}
\]

\[
\text{loc}_T \in \mathcal{L}T, \ \text{present}_T \in \mathcal{P}T, \ \text{mob}_T \in \mathcal{M}T, \ \text{ack}_T \in \mathcal{A}T, \ \text{fail}_T \in \mathcal{F}T.
\]

Figure 1: State Space

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 flag to indicate whether a machine is up and running. We will see that a site’s failure state is allowed to change from false to true, which indicates that the site is exhibiting a failure, and that there is no transition allowing a failure state to change from true to false. Consequently, we are modelling stopping failures [8].

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 instance of an asynchronous distributed system [8]. In a distributed implementation, tables are not shared resources, but their contents can be distributed on each site.

The behaviour of the algorithm is represented by transitions, which specify how the state of the abstract machine evolves. Figure 2 contains all the transitions of the distributed directory service. For convenience, we use some notations such as post, receive or table updates, which give an imperative look to the algorithm; their definitions appear below. Given a configuration \( c = (\text{loc}_T, \text{present}_T, \text{mob}_T, \text{ack}_T, \text{fail}_T, k) \),
Given a configuration $c = \langle loc_T, present_T, mob_T, ack_T, fail_T, k \rangle$, five basic transitions are permitted:

**migrate_agent($s_1, s_2$)**:

$$s_1 \neq s_2 \land loc_T(s_1) = \emptyset \land present_T(s_1) \neq \emptyset \land ack_T(s_1) = \emptyset \land \neg fail_T(s_1)$$

$$\rightarrow \{ \text{let } \vec{M} = present_T(s_1) \}$$

$$in \quad present_T(s_1) := \emptyset$$

$$post(s_1, s_2, agent(s_1, mob_T(s_1), \vec{M})) \}$$

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

$$first(k(s_1, s_2)) = agent(s_3, l, \vec{M}) \land \neg fail_T(s_2)$$

$$\rightarrow \{ \text{receive}(s_1, s_2) \}$$

$$\text{let } \vec{S}' = add(N, s_3, l, remove(s_2, \vec{M}))$$

$$in \quad loc_T(s_2) := \emptyset$$

$$present_T(s_2) := \vec{S}'$$

$$mob_T(s_2) := l + 1$$

$$ack_T(s_2) := \vec{S}' \}$$

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

$$ack_T(s_1) = (s_2, l_2) \notin \vec{M} \land \neg fail_T(s_1)$$

$$\rightarrow \{ \text{ack}_T(s_1) := \vec{M} \}$$

$$post(s_1, s_2, \text{ack}(s_1, mob_T(s_1))) \}$$

**receive_ack($s_1, s_2, s_3, l$)**:

$$first(k(s_1, s_2)) = \text{ack}(s_3, l) \land \neg fail_T(s_2)$$

$$\rightarrow \{ \text{receive}(s_1, s_2) \}$$

$$loc_T(s_2) := add(N, s_3, l, loc_T(s_2)) \}$$

**inform($s_1, s_2, s_3, l$)**:

$$(s_3, l) \in loc_T(s_1) \land \neg fail_T(s_1)$$

$$\rightarrow \{ \text{post}(s_1, s_2, \text{ack}(s_3, l)) \}$$

**stop_failure($s$)**:

$$fail_T(s) = \text{false}$$

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

**msg_failure($s_1, s_2, m$)**:

$$first(k(s_1, s_2)) = m \land \text{fail}_T(s_2)$$

$$\rightarrow \{ \text{receive}(s_1, s_2) \}$$

Figure 2: Fault-Tolerant Distributed Directory Service
- \( \text{mob}_T(s) := V \) denotes \( c = \langle \text{loc}_T, \text{present}_T, \text{mob}_T, \text{ack}_T, \text{fail}_T, k \rangle \), such that \( \text{mob}_T'(s) = V \) and \( \text{mob}_T'(s') = \text{mob}_T(s') \), \( \forall s' \neq s \).

- a similar notation is used for other tables.

- \( \text{post}(s_1, s_2, m) \) denotes \( c = \langle \text{loc}_T, \text{present}_T, \text{mob}_T, \text{ack}_T, \text{fail}_T, k' \rangle \), with \( k'(s_1, s_2) = k(s_1, s_2) \), and \( k'(s_1, s_j) = k(s_1, s_j), \forall (s_1, s_j) \neq (s_1, s_2) \).

- \( \text{receive}(s_1, s_2) \) denotes \( c = \langle \text{loc}_T, \text{present}_T, \text{mob}_T, \text{ack}_T, \text{fail}_T, k' \rangle \), with \( k(s_1, s_2) = \{m\} \), and \( k'(s_1, s_j) = k(s_1, s_j), \forall (s_1, s_j) \neq (s_1, s_2) \).

Four essential transitions specify the behaviour of the directory service: \text{migrate}_\text{agent} models the initiation of an agent migration; \text{receive}_\text{agent} models the arrival of an agent at a new location; \text{send}_\text{ack} deals with the sending of an acknowledgement message, while its counterpart \text{receive}_\text{ack} is concerned with the reception of an acknowledgement. The fifth transition optimises the algorithm by reducing forwarding chains, whereas the last two transitions are concerned with failures.

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 \( \neg \text{fail}_T(s) \), which indicates that the site \( s \) where the rule occurs must be up and running. The right-hand side of a rule denotes the configuration that is reached after transition. We assume that the guard evaluation and new configuration construction are performed atomically.

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 on \( s_1 \) must be non-empty, which indicates that the agent is present on \( s_1 \). After transition, the present table on \( s_1 \) 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 \( \text{mob}_T(c, s_1) \), and the previous content of the present table on \( s_1 \). 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 transition is concerned with \( s_2 \) handling a message\(^1\) \text{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 \text{add}(N, s, l, \vec{M}) \) adds an association \((s, l)\) to the memory \( \vec{M} \), keeping at most \( N \) entries with the highest timestamps. 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.

According to the third transition, if the acknowledgement table on \( s_1 \) contains a pair \((s_2, l_2)\), then an acknowledgement message \text{ack}(s_1, (\text{mob}_T(s_1))) \) has to be sent from \( s_1 \)

\(^1\)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; such a possibility will be discussed in Section 5.2.
to $s_2$; the acknowledgement message indicates that the agent is on $s_1$ with a mobility counter $\text{mob}_T(s_1)$.

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. It is worth observing the following two facts. 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 mechanism for propagating more information about the agent’s location. Second, we should 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 $\text{add}$, a complete specification of which may be found in [10].

According to rule $\text{inform}$ of Figure 2, any site $s_1$ believing that the agent is located on 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 acknowledgement message. It is important to distinguish the roles of the $\text{send\_ack}$ and $\text{inform}$ transitions. The former is mandatory to ensure the correct behaviour of the algorithm, whereas the latter is optional. The purpose of $\text{inform}$ is to propagate information about the agent’s location in the system, so that the agent may found by involving less sites. As opposed to previous rules, the $\text{inform}$ rule is non-deterministic, as far as the destination and the location information in an acknowledgement message are concerned. 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 5.2.

A guard deserves a further explanation: similarly as [11], the acknowledgement table is required to be empty before allowing a transition $\text{migrate\_agent}$. Indeed, an agent is only allowed to leave a site after this site has emitted all acknowledgement messages to the agent’s previous locations (though these messages do not have to be received before allowing migration). This constraint simplifies the algorithm (and its proof) because a site has to remember at most $N$ sites to acknowledge (as opposed to an unbounded set of sites). We do not believe that this condition significantly changes the performance of the algorithm.

**Failure** The first five rules of Figure 2 require the site $s$ where the transition takes place to be up and running, i.e. $\neg \text{fail}_T(s)$. 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 [8]. We model a stopping failure by the transition $\text{stop\_failure}$, 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.

As far as distributed system modelling is concerned, it is unrealistic to consider that messages that are in transit on a communication link remain present if the destination of the communication link exhibits a failure. Rule $\text{msg\_failure}$ 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) on a site that stops. We could add an extra rule handling that 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.
In the initial configuration, noted $c_i$, we assume that the agent is at a given site $\text{origin}$ with a mobility counter set to $N+1$. Furthermore, we consider a set $S_i$ of $N$ different sites, which are acting as previously visited by the agent, and are each associated with a different mobility counter in the interval $[1, N]$. For each sites in $S_i$, the location table points to the origin and to sites of $S_i$ with a higher mobility counter; the location table on all other sites contains the origin and the $N-1$ first sites of $S_i$. The present table on $\text{origin}$ contains the sites in $S_i$. A detailed formalisation of the initial configuration is available from [10].

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:

$$
c_i \xrightarrow{t_1} c_1 \xrightarrow{t_2} c_2 \ldots \xrightarrow{t_n} c.
$$

We define $\xrightarrow{\ast}$ as the reflexive, transitive closure of $\xrightarrow{}$.

4 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. Its liveness guarantees that agent location information eventually gets propagated. We will primarily focus on the former property, because the proof of the latter is similar to the one derived in our previous work [11].

We can intuitively explain the proof of the safety property as follows. An acknowledgement message results in the creation of a forwarding pointer that points towards the agent’s location. We can formalise these forwarding pointers by a relationship $\text{parent}$; we can establish that the relationship $\text{parent}$ defines a directed acyclic graph leading to the agent’s location.

In the presence of failures, we need to show that the relationship parent contains sufficient redundancy in order to guarantee the existence of a path leading to the agent, without involving any stopped site. We can establish the following properties. (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 $N$, there is at least one parent that is closer to the agent’s location; by repeating this argument, we can find the agent’s location.

The rest of this section will precisely formulate this intuitive reasoning. All proofs were mechanically derived using the proof assistant Coq [3]; complete proofs may be downloaded from [10]. The notation and properties formulation presented here are pretty-printed versions of the mechanically established ones.

When a mobile agent is not in transit, we refer to the site currently hosting it as the agent host.

Definition 1 (Agent host)

For any configuration $c = \langle \text{loc}_T, \text{present}_T, \text{mob}_T, \text{ack}_T, \text{fail}_T, k \rangle$, and for any site $s$, $\text{agent.host}(c, s)$ holds if $\text{present}_T(s) \neq \emptyset$. □
In the abstract machine, we can establish that memories are well-formed.

**Lemma 2 (Well Formed Memory)**

For any memory $\vec{M}$ in a configuration, (i) the length of $\vec{M}$ is bounded by $N$; (ii) there is at most one entry in $\vec{M}$ for a given timestamp; (iii) there is at most one entry in $\vec{M}$ for a given site. □
Proof of Lemma 2
The proof appears in files consistent.v and bounded.v and proceeds by induction on the legal transitions, and by a case analysis of the different transitions. (We shall note that the operator \textit{add} adding an entry to a list preserves such a property.) \hfill \Box

The agent can either be located on its agent host or be in transit; furthermore, in either case, there is a single instance of the agent.

Lemma 3 (Situated or Migrating)
For any configuration \(c_i \rightarrow^* c\):
\[
\{ \exists s \in S \mid \begin{align*}
\text{agent}_\text{host}(c, s) \\
\land \forall s_i \in S, \text{agent}_\text{host}(c, s_i) \rightarrow s = s_i \\
\land \forall s_1, s_2, s_3 \in S, \forall l \in \mathbb{N}, \forall M \in \Psi, \text{agent}(s_3, l, M) \notin k(s_1, s_2)
\end{align*} \}
\lor
\{ \exists (s_1, s_2, s_3, l, M) \in S \times S \times S \times \mathbb{N} \times \Psi \mid \begin{align*}
\text{agent}(s_3, l, M) \in k(s_1, s_2) \\
\land \text{agent}_\text{count}(l, k) = 1 \\
\land \forall s_i \in S, \neg\text{agent}_\text{host}(c, s_i) \\
\land \forall l_i \in \mathbb{N}, l_i \neq l \rightarrow \text{agent}_\text{count}(l_i, k) = 0
\end{align*} \}
\]
with \text{agent}_\text{count}(l, k) defined as the number of agent messages with mobility counter \(l\) in communication queues \(k\). \hfill \Box

Proof of Lemma 3
The proof appears in file situated.v [10] and proceeds by induction on the legal transitions, and by a case analysis of the different transitions. \hfill \Box

In this paper, our focus is to make a directory service fault-tolerant; in particular, we want to make this algorithm resilient to sites previous visited by a mobile agent. However, as opposed to [9], we are not attempting to make mobile agents resilient to failures; this is a different problem which requires different solutions. We shall therefore assume the following hypothesis.

Hypothesis 4 (No Agent Failure)
For any configuration \(c = \langle \text{location}_T, \text{present}_T, \text{mob}_T, \text{ack}_T, \text{fail}_T, k \rangle\), we assume the following constraints:
\[
\begin{align*}
(1) \quad \forall s \in S, \text{agent}_\text{host}(c, s) \rightarrow \neg\text{fail}_T(s). \\
(2) \quad \forall (s_1, s_2, s_3, l, M) \in S \times S \times \mathbb{N} \times \Psi, \\
\text{agent}(s_3, l, M) \in k(s_1, s_2) \rightarrow \neg\text{fail}_T(s_2).
\end{align*}
\]
\hfill \Box

The first condition specifies that the agent host must be up and running. The second condition specifies that an agent cannot be in transit to a site that has stopped. Furthermore, we said that our abstract machine models a system with buffered inputs and
unbuffered outputs. If outputs were made buffered, we would have to exclude an agent in transit on a communication channel originating from a site that stopped.

The effect of an acknowledgement message is to update the location table of its receiver. Any pair \((s, l)\) in a location table indicates that the agent was on site \(s\) with timestamp \(l\). In other words, this information can be regarded as a forwarding pointer; a trail of such forwarding pointers is left by the mobile agent as it migrates. We can formally define a relation between sites that captures the notion of forwarding pointer. We say that \(s_2\) is the \textit{parent} of \(s_1\), if there is a pair \((s_2, l_2)\) in \(locT(s_1)\) for some \(l_2\). However, agent migration is not atomic: the location table of the agent’s previous location is only updated after transitions \texttt{send_ack} and \texttt{receive_ack} are performed. Therefore, we define a \textit{parent} relation, which is stable when these transitions are executed.

\textbf{Definition 5 (Parent)}

For any configuration \(c = \langle location_T, present_T, mob_T, ack_T, fail_T, k, k_0 \rangle\), for any sites \(s_1, s_2\), \(\text{parent}(c, s_1, s_2)\) holds if:

\(\exists l(s_2, l) \in loc_T(s_1)\)
\(\land \ s_1 \neq s_2\)
\(\land \ l > mob_T(s_1)\)

\(\exists l \in \mathbb{N}, s_3 \in S,\)
\(\land \ \ s_1 \neq s_2\)
\(\land \ \ l > mob_T(s_1)\)
\(\land \ \ ack(s_2, l) \in k(s_1, s_3)\)

\(\exists l \in \mathbb{N},\)
\(\land \ \ present_T(s_1) = \emptyset\)
\(\land \ \ (s_1, l) \in ack_T(s_2)\)

\(\square\)

The second case of the definition considers an \texttt{ack} message in transit, while the third case is about an acknowledgement message remaining to be sent. We shall note that, at this stage, we do not take the failure status of either \(s_1\) or \(s_2\) into account in the definition of \textit{parent}. We can prove that mobility counters are strictly increasing along edges of the parent relationship.

\textbf{Lemma 6 (Increasing Parent)}

For any configuration \(c = \langle location_T, present_T, mob_T, ack_T, fail_T, k, k_0 \rangle\) such that \(c \rightarrow^* c\), for any sites \(s_1, s_2\), if \(\text{parent}(c, s_1, s_2)\) holds, then:

\(mob_T(s_1) < mob_T(s_2)\).

\(\square\)

\textbf{Proof of Lemma 6}

The proof appears in file \texttt{increasing.v} and proceeds by induction on the legal transitions, and by a case analysis of the different transitions. \(\square\)

We now have all the ingredients to show that the parent relation leads to the agent’s location. The latter is formally defined by the concept of \textit{terminal site}, which is the agent host if the agent is not in transit, or the agent’s previous site if the agent is in transit.
Definition 7 (Terminal Site)
For any configuration $c$, the terminal site, written $\text{terminal\_site}(c)$, is a site $s$ such that $\text{agent\_host}(c, s)$ holds if the agent is not in transit, or such that there exists a message $\text{agent}(s, l, \vec{M})$. □

If we follow a sequence of forwarding pointers, we reach a site without successor. Formally, we can define a function that maps a site $s$ onto a site without successor.

Lemma 8 (Root) There exists a function $\text{root}: \text{Site} \rightarrow \text{Site}$ mapping any site onto a site without parent. □

Proof of Lemma 8
For a given abstract machine configuration, we can define a well-founded relation on sites: $s_2$ is a successor of $s_1$ if $\text{parent}(c, s_1, s_2)$ holds. There is a strictly decreasing measure associated with sites $s_i$, given by the difference between the mobility counters of $s_i$ and the terminal site: $\text{mob}_T(\text{terminal\_site}(c)) - \text{mob}_T(s_i)$. Using a fixed point on the well-founded relation, we can define $\text{root}$ as a function that maps a site $s$ onto a site without successor. □

The function $\text{root}$ is the basis of the safety property in the absence of failure: for any site, the value returned by $\text{root}$ is the terminal site, as specified by Theorem 9.

Theorem 9 (Safety in the Absence Failure)
For any configuration $c = \langle \text{location}_T, \text{present}_T, \text{mob}_T, \text{ack}_T, \text{fail}_T, k \rangle$ such that $c_i \xrightarrow{*} c$, for any site $s$ such that $\neg \text{fail}_T(\text{root}(s))$, if $\neg \text{fail}_T(\text{terminal\_site}(c))$, then $\text{root}(s) = \text{terminal\_site}(c)$. □

Proof of Theorem 9
The proof appears in file root.v. It relies on a lemma stating the uniqueness of sites without successor. □

An important consequence of this theorem is that when a site has several forwarding pointers to parents, any of them inexorably leads to the terminal site. Therefore, we intend to use this redundancy of pointers to find a path that leads to the terminal site, without involving any site that has stopped. In order to investigate the algorithm’s tolerance to failures, we need to give an accurate account of the degree of forwarding pointer redundancy in the system. To this end, we define a function counting the number of parents of a given site.

Definition 10 (Number of parents)
For any configuration $c$ and for any site $s$, the number of parents, noted $\text{sum\_parent}(c, s)$, is defined as follows:

$$\text{sum\_parent}(c, s) = \sum_{s_i \in S} \text{Int}(\text{parent}(c, s, s_i))$$

with $\text{Int}(\text{true}) = 1$ and $\text{Int}(\text{false}) = 0$. □

We want to establish that the number of parents is greater than the maximum number of tolerated failures. However, this property does not hold for the sites that have recently been visited by an agent: in this particular case, however, we can show that they have the terminal site as a parent. In order to formulate this property precisely, we define the set of sites recently visited by the agent.
Definition 11 (Recently Visited Sites) For any configuration \( c \), the set of recently visited sites, noted \( \text{recently.visited}(c) \) is defined as:

\[
\begin{align*}
(1) \quad & \forall s \in S, \ \text{agent.host}(c, s) \rightarrow \text{recently.visited}(c) = (s, \text{mob}.T(s)) \# \text{present}.T(s) \\
(2) \quad & \forall(s_1, s_2, s_3, l, \vec{M}) \in S \times S \times S \times N \times \Psi, \quad \text{agent}(s_3, l, \vec{M}) \in k(s_1, s_2) \rightarrow \text{recently.visited}(c) = (s_3, l) \# \vec{M}
\end{align*}
\]

\( \square \)

We can derive the following property, giving an inferior bound on the number of parents of a site.

Lemma 12 (Bound on the Number of Parents)
For any configuration \( c = \langle \text{location}.T, \text{present}.T, \text{mob}.T, \text{ack}.T, \text{fail}.T, k \rangle \), for any site \( s \) such that \( \neg \text{fail}.T(s) \), the following statements hold:

\[
\begin{align*}
(1) \quad & s \in \text{recently.visited}(c) \rightarrow \text{sum.parent}(c, s) \geq \text{index}(s, \text{recently.visited}(c)) \\
(2) \quad & s \notin \text{recently.visited}(c) \rightarrow \text{sum.parent}(c, s) \geq N
\end{align*}
\]

with \( \text{index}(s, l) \) returning the position of \( s \) in a list \( l \), starting to count from 0. \( \square \)

Proof of Lemma 12
The proof appears in file \texttt{sigma.parent.v}. It proceeds by induction on the legal transitions and by an analysis of the possible transitions. \( \square \)

Note that this property is valid for a site \( s \) which is up and running. The property would no longer hold if we were introducing a rule changing a site’s failure state from true to false, since the inductive hypothesis would no longer be applicable for that transition.

Like \texttt{root} which followed a sequence of forwarding pointers, we can define \texttt{froot} which follows a sequence of forwarding pointers, without including a site that has stopped.

Lemma 13 (Failure Safe Root) There exists a function \texttt{froot} : \texttt{Site} \rightarrow \texttt{Site} mapping any site onto a failure-free site without parent. \( \square \)

Proof of Lemma 13
As in Lemma 8, we define \texttt{froot} by a fixed point on a well-founded relation on sites. However, now \( s_2 \) is a successor of \( s_1 \) if \( \text{parent}(c, s_1, s_2) \) holds and \( \neg \text{fail}.T(s_2) \). \( \square \)

Definition 14 (Number of failures) For any configuration \( c \) and for any site \( s \), the number of failures, \( \text{sum.failure}(c, s) \), is defined as follows:

\[
\text{sum.failure}(c) = \sum_{s_i \in S} \text{Int}(\text{fail}.T(s_i))
\]

with \( \text{Int}(\text{true}) = 1 \) and \( \text{Int}(\text{false}) = 0 \). \( \square \)
Assuming that the number of failures is smaller than \( N \), for a site without failure, \( f_{root} \) returns the terminal site (if it is up and running) or a site \( s' \) such that the terminal site is a parent of \( f' \).

**Theorem 15 (Safety in the presence of Failure)**

For any configuration \( c = (\text{location}_T, \text{present}_T, \text{mob}_T, \text{ack}_T, \text{fail}_T, k) \), for any site \( s \) such that \( \neg \text{fail}_T(s) \), if \( \text{sum\_failure}(c) < N \)

\[
\begin{align*}
(1) & \quad \neg \text{fail}_T(\text{terminal\_site}(c)) \rightarrow f_{root}(s) = \text{terminal\_site}(c) \\
(2) & \quad \text{fail}_T(\text{terminal\_site}(c)) \rightarrow \text{parent}(c, f_{root}(s), \text{terminal\_site}(c))
\end{align*}
\]

\( \square \)

**Proof of Theorem 15**

The proof is available in *failure_root.v*. \( \square \)

Similarly as our initial algorithm [11], the parent relationship can only be decided by examining the whole distributed system; it is not conveniently implementable, because it requires us to know if there are ack messages in transit, or if some acknowledgement tables are non empty. However, once acknowledgement messages have been processed, the parent relationship is given by the contents of the location table. Therefore, the algorithm will realistically be implementable, only if it has the liveness property, which ensures that location tables get updated to reflect the parent relationship.

A liveness result similar to the one in [11] may be established, which we now summarise. 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, then location tables will eventually be updated, which proves the liveness of the algorithm. Let us note that if delivery of acknowledgement messages is slower than the agent migration speed, then the information provided by the directory service is not up to date.

**5 Discussion**

Generalizing our initial algorithm [11] to support \( N \) acknowledgements has simplified its formalisation: two types of messages instead of three are now required and the number of transitions was reduced from 8 to 5. When \( N \) is equal to 1, the algorithm has the same observable behaviour as [11].

The complexity of the algorithm is linear in the number of sites for which we tolerate a failure. Indeed, we have \( N \) acknowledgement messages per migration, and spatial complexity is \( O(N) \), because all memories were proved to have a maximum length \( N \) (cf. Lemma 2).

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.
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 is dependent on the type of distributed system, and potentially the algorithms using the distributed 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. We can devise a lazy strategy, which needs cooperation with a message router. The recipient of a message may inform its emitter, when the recipient observes that the emitter has out-of-date routing information. In such a strategy, tables are only updated when user messages are sent.

5.1 Message Router

The purpose of this paper was the study of a distributed directory service; its use for message routing is beyond the scope of this paper. However, in this section, we sketch two possible variants.

Simple Routing. We can adapt our initial message router [11] 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 its complexity has increased; furthermore, the algorithm needs a mechanism to clear messages that have been delivered and are still held by intermediate nodes.

5.2 Algorithm Extensions

In Section 3, 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.

We explicitly raised the reader’s attention on rule receive_agent, where $s_3$ is not required to be identical to $s_1$. This allows us to introduce further rules by which a site may decide to reject an incoming agent, or may forward it to another site. Such rules would model cases where “firewalls” or “domain gateways” negotiate the entry of an agent and allow it to enter the domain or not.

While our algorithm focuses on stopping failures, we conjecture that loss of messages could also be tolerated. Indeed, in our abstract machine, we can model a lossy commu-
cation channel by a failure of the receiver-side of the channel. Our distributed directory service therefore tolerates up to \( N - 1 \) lossy channels.

6 Related Work

Wojciechowski and Sewell [16] formalise and prove the correctness of a message router for mobile agents. They propose two programming languages: the first one uses high-level location-independent primitives, whereas the second one contains low-level location-dependent primitives; the higher-level language can be translated into the lower-level one. Their proof of correctness is based on a suitably defined notion of program equivalence. Their approach, which focused on a centralised message router, may also be applicable to our algorithm. However, we believe that the constructive proof that we carried out gives us a better insight of the algorithm, which may help us to design new extensions.

Amadio and Prasad [2] study an idealised version of the IPv6 protocol, using a home agent, and use a notion of barbed bisimulation for proving the equivalence of different implementations.

Murphy and Picco [13] 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. [7] 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.

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.

References


