Communication and Tracking Infrastructure of a Mobile Agent System

Jocelyn Desbiens
INRS-Telecommunications
desbiens@inrs-telecom.uquebec.ca

Martin Lavoie
Alex Informatique Inc.
lavoie@alex.qc.ca

Francis Renaud
INRS-Telecommunications
renaud@inrs-telecom.uquebec.ca

Abstract

Objects are reactive in the sense that they change state only when one of their methods is invoked by other objects. Reactive objects are sufficient in a client-server model of computation with a remote invocation mechanism “a la RPC”. However, active objects are much more powerful mainly because they are autonomous entities. Epidaure’s objects are a particular type of active objects from which multi-agent systems can be built. In the context of mobility, migration becomes a necessity. For instance, a mobile client may disconnect from its server and reconnect to another server at a later time. In this paper, we describe in a detailed manner the communication mechanism of a particular multi-agent system, Epidaure, and how the communication algorithm and mobility protocols that we developed can be applied to address problems related to mobile objects (e.g., registration, transport, tracking, ...).

1. Introduction

Agent mobility is a natural extension of the concept of “stationary agent”. It also carries the extension of the client-server model even further: with mobile agents, clients send parts of themselves to the server for execution. Mobile agents provide a unique distributed architecture which functions differently from the static set-ups. It provides for an innovative way of doing distributed computation. However, some problems like location tracking, modifications to static communication systems and garbage collection of obsolete objects are inherently related to mobility. This paper shows how Epidaure tackles the problem of locating mobile agents and how is maintained nonetheless the “guarantee of delivery” principle.

The remainder of this paper is organized as follows. Sections 2-5 describe in details the multi-agents system Epidaure. Section 6 gives the migration and location algorithms. Section 7 explains why signposts are necessary. Section 8 compares Epidaure with existing systems.

2. Agents in Epidaure

One of the earliest models for distributed systems is the Actor model [1, 6], where an actor is a computational agent which has a mail address and a behavior. Actors communicate by message-passing and carry out their actions concurrently. The behavior of an actor consists in processing a communication, which can result in the creation of new actors, new tasks and a replacement behavior enabling the current actor to process the next communication. Epidaure [5], whose initial classes come from Act++ [7], is a hybrid implementation of the Actor model which is particularly well suited to the programming and implementation of distributed mobile agents.

In this Section, we describe the basic characteristics of an agent, mainly how an agent is i) identified, ii) created, iii) cloned and iv) deleted from the system.

2.1. Agent identification

In Epidaure, each agent is given at creation time a unique universal identification name which, in the actual implementation, is the local agent’s name prefixed by its task and node id’s. This identifier is immutable.

2.2. Agent creation

Creation of an agent is done by making up a universal id, if no such id is furnished to the creator agent, or, in case of migration, by using the old id, and checking in the database of the local communication agent to make sure that this id is not already registered. If not, then Epidaure creates a mailbox (actually, an instance of the Mailbox class) and assigns its address to the agent’s id. It marks this entry with the tag Alive. In case of migration, the variable NumberOfHops is incremented by one. The mailbox address and the agent’s name are then made public over the local network. If there is already an entry in the local database with the same id, then three cases can be distinguished:
• the entry is tagged Alive: this is an error if the entry points to a local agent. A message of type Agent‐
  IsAlive is returned to the sender of the creation message. If the entry points to a remote agent, then a mailbox is created and its address is assigned to the agent’s id. The entry now points to a local agent.
• the entry is tagged Migrated: the agent is back to square one after migration. The entry is tagged Alive and it now points to its old mailbox;
• the entry is tagged Dead: the agent is woken up. The entry is tagged Alive and it now points to its old mailbox.

It is important to notice that the creation of an agent in Epidaure can be done dynamically and from within any remotely located agent.

2.3. Agent cloning

Epidaure uses the available remote creation mechanism to realize agent cloning. It is a three steps process:

• making up of a universal id: use of the agent’s id in case of migration or a new universal id in case of a duplication;
• remote creation of an instance: creation of new empty instance of the agent on the target node with an initial behavior Non_Init;
• initialization of the agent: emission of a constructor message with a copy of the local class variables in case of duplication. The new remote agent is thus a copy of the source agent at the moment of the cloning. In case of migration, Epidaure waits for all active threads to terminate before sending the constructor message.

An agent’s clone is a carbon copy of the agent at the moment of the execution of the replacement behavior Migrated.

2.4. Agent deletion

The deletion of an agent is started by marking, in the database of the local communication agent, the agent’s id with the tag Dead. It signals to the other agents of the system that the services given by this agent are no longer available. Its mailbox is then flushed after sending a message of type AgentIsDead to all agents that sent a SendWithReply message to it. All threads are terminated before killing the agent.

3. Messages in Epidaure

3.1. Types of messages

The management of all messages is done entirely by the class Message. There are in Epidaure seven types of messages:

3.1.1. Plain messages

• Message: a one way message. By default, Epidaure messages are asynchronous. One way messages return control to the sender immediately and discard results; they do not allow return values;
• MessageWithReply: a message with a return value. Although messages are asynchronous by default, Epidaure supports synchronous and future messages as well [1]. Future messages return control to the sender without waiting for the message to complete. However, unlike one way messages, future messages return a placeholder that can be used to retrieve the return value by waiting for a callback. The MessageWithReply construction is used in conjunction with the Receive command. Placing it immediately after a MessageWithReply is equivalent to writing a blocking send;
• Reply: a message containing a return value; the unique identifier tagging the MessageWithReply is used in the callback to retrieve the destination of the message;

3.1.2. Control messages

• AgentIsAlive: an error message. It means that the agent already exists, is active and that its universal id cannot be duplicated;
• AgentHasMigrated: an updating message. It contains the target address left behind by the agent when it moved away from this node. It is not necessarily its actual address;
• AgentIsMigrating: an updating message. It signals the target communication agent to put in its local Pending_List all messages sent to the migrating agent (ref. Section 6.1.1.). Its main purpose is to prevent message cycling;
• AgentIsDead: end of activity message. It means that the agent is not answering any more messages.

3.2. Message handling

Messages coming from local agents, or remote agents either in another task or on distant node, are processed according to the same procedure. They all transit through a communication agent that can be thought of as a "messages dispatcher". Assuming that a message (containing the location of an addressee) has been handed to a communication agent, it is possible to distinguish four cases:

• the addressee is known and tagged Alive: the message is sent to the addressee. If the agent is local to the task, then it is put in its mailbox;
• the addressee is known and tagged Migrated: the message is forwarded to the target destination after pushing the node and task ids onto the StopList (a class variable of the object Message); if the StopList is empty, then the address of the emitting agent is pushed onto the list;
• the addressee is known and tagged Dead: this is an error. The message is flushed and a message of type AgentIsDead is returned to the sender;
• the addressee is unknown: the message is put in the local Pending_List and the address resolution algorithm is applied (ref. Section 5.4.).

3.3. Message sending

The sending of a message is a two steps process: i) its construction, and ii) its actual execution.

At the setup time of a message, i.e. when the class Message is instantiated, the arguments are encoded using XDR and RPC-GEN, and the message is uniquely identified. It is this unique identifier that is used to bind a return value to a message of type MessageWithReply. Once the message is setup, Epidaure hands it to the local communication agent and the above algorithm is applied.

3.4. Message receiving

The receiving of a message is a two steps process: i) its extraction from the mailbox, and ii) its actual execution.

When a message is received by a communication agent then the above algorithm is applied. Once the message is in the mailbox of the agent, it is the internal behavior algorithm which decodes the arguments and starts a new thread of execution.

4. Tasks in Epidaure

The task is the basic unit of services to the agents such as: processing of the flow of incoming/outgoing messages, encoding/decoding of arguments, scheduling of resources, creation and destruction of agents, and so forth. It contains also a variable number of user agents and three persistent stationary agents:

• AgentProg agent: an agent acting as an entry point to the task (similar to the main function in C++);
• a communication agent: an agent acting as a “mailman”;
• a creator agent: an agent whose role is the making up of new agents (ref. Section 2.3.).

The communication agent manages the flow of information exchanged between local and remote agents and, to this effect, has a local database of resolved agents’ addresses.

5. Communications in a distributed agents system

5.1. Communication agent

As its name suggests, the communication agent is a persistent agent handling the flow of messages going in and out of the tasks. In Epidaure, it is possible to replace this agent by a user-defined communication agent in order to tailor the system to other needs. To this effect, the following four methods totally define the interface:

• Register_Agent: puts an agent’s address in the local database;
• Forward: redirects the current message sent to a not yet registered agent. The message is then rerouted to the local communication agent of the target destination;
• Delete_Agent: informs the communication agent that the sought after agent no longer exists;
• Exit: informs the communication agent that the job is over and that it must quit.

In order to facilitate the exchange of information between dedicated communication agents, we added three new methods that, unlike of the above methods, which are part of the standard library, are implementation dependent:

• Add_Entry: is equivalent to Register_Agent but at a much higher level. When an agent is created locally, the communication agent may send this information to a remote communication agent;
• Inquire_Entry: looks for the address over the network of a particular agent;
• Update_Entry: answers to a request of the type Inquire_Entry.

Furthermore, the communication agent must provide methods to handle the four types of messages defined above (ref. Section 3.1.) which are: AgentIsAlive, AgentHasMigrated, AgentIsMigrating, and AgentIsDead.

5.2. Communication network

In the context of a distributed memory architecture system, references between objects on different nodes based on pointers is not possible. Consequently, we need a new way to reference objects no matter where they are located in the system. Generally speaking, operating systems for distributed memory architecture do not support direct references between the object’s name and its memory location to nodes other than the one where the object is stored. This situation is related to the fact that most operating systems for this type of machine still have to implement the concept
of uniform address space for local memories. For systems that need this uniform address space, it is possible to create tables that keep records of the correspondence between the name of objects and their node location. The implementation could be done through a unique global table or many local tables, or a mix of both strategies. If a unique global table is used, the implementation will be easy but accessing this table will create a bottleneck. The use of many correspondence tables distributed in the system will solve the bottleneck problem but coherence problems will render this strategy difficult to implement. In Epidaure we have chosen the global table strategy and decided to reduce the impact of the bottleneck access on performances by creating cache memories in each task of the application. Cache memories prevent global table access for objects that are local and also prevent repetitive access to the same object (since the first access copies the corresponding entry into the local cache memory).

It is the global database (whose role is similar to a DNS) in the Epidaure system that records the correspondence between agents and their node location. This global database sits on a node whose location is known to every agent in the application. Some of the fields in this database are (see Figure 1 where the communication structure field is also displayed):

- the agent’s id;
- the identification of the task and the node to which the agent belongs;
- a tag indicating whether the agent is Alive, Migrated, or Dead.

![Communication structure](image)

Figure 1: Communication structures

This global database is aware of all the agents existing in the system at any given moment. It is updated by local communication agents as soon as an agent is either created, migrated or deleted. Local cache memories consist of a network of local communication agents, one for each task created by an application. These local databases are updated as soon as an agent is created locally or when the global database is consulted. The fields record of these databases are roughly similar to the fields of the global database (see Figure 1). They are:

- the agent’s id;
- the identification of the task and the node to which the agent belongs;
- a tag indicating whether the agent is Alive, Migrated, or Dead;
- an integer NumberOfHops giving the number of migrations made by the agent up to the current node; its value defaults to 0 and is incremented each time a migration occurs;
- a structure giving the type of communication protocols used by the agents (RPC or socket under Unix, send/receive under MPI or PVM).

The global database does not have to know whether an agent has migrated or not. It knows only the location where it was last created. It is important to structure both internally and externally the network of communication agents in such a way that:

- it is relatively easy for the user to specify where the main database is, and what is the relation between the local and the main databases;
- the relation existing between loosely coupled networks can be maintained;
- the overall traffic generated by the messages can be minimized as much as possible.

Figure 2 is an illustration of the tree-like structure that is used by Epidaure to deal with these constraints. The options \(-r\) (for receiver) and \(-s\) (for sender) are there to make the clear distinction between a server task and a client task.

A domain is a set of tasks that are bounded to the same global database. The agents they serve are therefore linked to the same global communication agent. In Figure 2, agent 0 and agent 5 belong to domain B, while agent 7 and agent 11 belong to domain G. There is only one global database per domain in any Epidaure application. On the other hand, there are other ways to let global databases exchange information.

5.3. Launching an application

Still referring to Figure 2, task B contains the global database of the current application because it was launched, at run time, with the \(-r\) option, whereas the other tasks (namely, A and C) were launched with the \(-s\) option, making them clients of task C. Furthermore, it is permitted in Epidaure to start a task both with the \(-r\) and \(-s\) options, allowing it to be concurrently server and client in the same application. In Figure 2, task D is both a server for tasks E and F and a client for task B. This strategy reduces the traffic load at the global communication agent level and induces a natural mapping between the logical network of agent making up the application and its physical counterpart. This would be the case, for example, if some of the agents were
located on a physically remote sub-network (see Figure 2).
To establish the communication with other domains, the \(-r\) option is followed by a list of global tasks which, once parsed,
will inform \textit{Epidaure} where to look if any unknown reference
is to be resolved.

5.4. Resolution algorithm
Suppose now that we have two agents, let’s say \textit{agent}_0
and \textit{agent}_1, sitting respectively on nodes A and C (A and C
might not be distinct, but at least \textit{agent}_0 and \textit{agent}_1
belong to different tasks) and that \textit{agent}_0 is sending a
message to \textit{agent}_1. Assume furthermore that the main
communication agent belongs to task B and other communication
agents in task A and C are clients (see Figures 3 and 2,
and how the \(-r\) and \(-s\) options are used) of B. At creation
time, \textit{agent}_0 registers to the local communication agent’s
database (ref. Section 2.2.). The information recorded in the
database pertains mainly to the physical location of the agent
and to its communication protocol. For instance, referring
to Figure 1, \textit{agent}_0 sits on node A, in task id. #1243,
and its communication protocol is given by the corresponding
entry in the communication structures table. As soon as
the local registration is done, a message \textit{Add_Entry}
"\textit{agent}_0" "Alive" is sent to the main communication
agent informing it to perform the same action on its side. The

\begin{itemize}
\item \textit{agent}_0 \textit{knows the address of agent}_1: this is the case,
for instance, if \textit{agent}_1 had a prior communication
with \textit{agent}_0. The message is therefore sent directly to
\textit{agent}_1 and the message handling algorithm is applied
(ref. Section 3.2.);
\item \textit{agent}_0 \textit{does not know the address of agent}_1: this is the case,
for instance, if \textit{agent}_1 does not exist yet, or
\textit{agent}_1 is not recorded in the main database, or if there
were no previous communications between \textit{agent}_0
and \textit{agent}_1. The message is first recorded in the lo-
cal \textit{Pending_List} and marked with the tag \textit{agent}_1.
Then a message \textit{Inquire_Entry} "\textit{agent}_1" is sent
to the main communication agent, upon which the re-
quested address is returned to \textit{agent}_0, naturally if
such an address is known to the global communica-
tion agent. In the latter case, the address is resolved
by running the message \textit{Update_Entry} "\textit{agent}_1"
"C:1755", updating the local database, sending to
\textit{agent}_1 the pending messages, and flushing the lo-
cal \textit{Pending_List}. On the other hand, if the ad-
dress of \textit{agent}_1 is unknown, then the global \textit{In-
quire_List} is updated with a new entry meaning
that \textit{agent}_0 needs the address of \textit{agent}_1 (see Figure
3). At \textit{agent}_1 registration time, all such requests will
be satisfied and its address will be propagated to all
agents in the list. The list of all the entries referring
to \textit{agent}_1 is then flushed. In both cases, the message
handling algorithm applies (ref. Section 3.2.).
\end{itemize}
6. Migration and tracking algorithms

6.1. Agent migration

In Epidaure [5], an agent can migrate independently - that is, an agent can migrate itself. It only has to carry out the replacement behavior Migrated. Epidaure is one of the few agent systems available today in which clients can communicate with agents while the agents are moving.

6.1.1. Migration algorithm

When an agent is told to migrate or decides on its own to migrate, the following steps are taken:

- send a message UpdateEntry to the local communication agent with the target address;
- send a message AgentIsMigrating to the target communication agent informing it to buffer all incoming messages forwarded to the migrating agent;
- in the local database, mark the agent’s address with the tag Migrated;
- wait for all active threads to terminate;
- remotely clone the agent on the target node [5] passing as arguments to the creator agent the agent’s id and the variable NumberOfHops (an integer giving the number of migrations made by the agent up to the current node; its value defaults to 0 and is incremented each time a migration occurs);
- forward all messages left in the mailbox to the new address and flush the mailbox.

As soon as an agent knows it has to move, its tag is changed from Alive to Migrated in order to forward to the new address the incoming messages. The message handling algorithm [5] insures that they will be properly executed when the migrated agent will resume execution on the other side. Also, all non read messages in the agent’s mailbox are forwarded to the new address. This makes sure that no information is lost in the meantime.

What will be left behind after an agent moved is a kind of pointer, or signpost, to forward messages (see Figure 4, where an agent A moved from node N0 to node N1). M0 is a message forwarded to A).

6.1.2. Lazy forwarding and updating algorithm

Assume that an agent, let’s say A has migrated from node N0 to node N3, migrating in between by nodes N1 and N2. When moving from node N0 to N1, A leaves behind a signpost indicating that it is now located on node N1. The same thing is done when migrating from node N1 to node N3, and finally from N2 to N3. The situation is depicted at Figure 5.

![Figure 5: Multiple hops](image)

Agent B, which is on node N4 and has a previous address of A, sends a message M0 to A. Because A is not on N0 and its signpost points to N1, M0 is automatically forwarded to N1. Ultimately, M0 will reach node N3 where agent A currently resides. The same scenario happens to a message M1 sent by agent C on node N5 to agent A. The number inside each agent or signpost is the number of migration, or hops, a mobile agent has done so far. By default, its value is 0.

Generally speaking, suppose that an A is migrating from node N0 to node Np. Denote by \( \tau(A) = N_0 N_1 \ldots N_{p-1} N_p \), the trajectory followed by A. Suppose furthermore that a message \( M_0 \) sent by an agent B is forwarded successively to its addressee A. Epidaure uses the path \( \tau(A) \) followed by the message to update the signposts met along the trajectory to A. The algorithm is as follows:

1. when a message M reaches its destination (i.e., a node where the target agent is Alive), the StopList (a class variable of the object Message containing a list of visited nodes) is examined; the first entry on the
list is popped up and discarded, and the actual address of the agent, its NumberOfHops, and the chopped Stop_List are sent to the communication agent of the task containing the signpost on the top of the list (actually, an Update_Entry message $M'$);

2. when a message Update_Entry arrives on a node, $\text{Epidaure}$ discards the top entry on the Stop_List and checks whether it is empty or not: if the list is, then step 3 is performed, otherwise the local NumberOfHops of the signpost is checked against the value NumberOfHops of $M'$. If it is smaller, then the signpost is updated and thereafter points to the location from where the message $M'$ was emitted. If not, the signpost is left unchanged. Afterwards, the actual address of the agent, its NumberOfHops, and the chopped Stop_List are sent to the communication agent of task containing the signpost on the top of the list (messages $M''$ and $M'''$ in Figure 6). Step 2 is then repeated over;

3. the address of agent $A$ in the local database of the of the initial sender is updated.

The algorithm is depicted at Figure 6 and the net result at Figure 7.

### 6.1.3. Guarantee of delivery

Given that a communication may be delayed for an arbitrarily long period of time, and consequently that the updating of the signposts can overlap over time, the question arises whether it is reasonable to assume that a message will reach its destination even when an agent has moved throughout the network in an unpredictable manner. $\text{Epidaure}$ guarantees that a message will eventually reach its target agent or, at least, that the message will be correctly routed to its destination.

Let $\tau$ be a finite subset of the set $\mathcal{S} = \{N, M, O, \ldots\}$ and $(\tau, \prec)$ be a total ordering of $\tau$. Put $\tau = \{N_0, N_1, \ldots, N_9\}$ (when the context is clear, we write $\tau = N_0 N_1 \ldots N_9$) where $N_0 < N_1 < \cdots < N_9$ and $\tau_N = \{M \in \tau \mid N < M\}$. Consider a multivalued function $\sigma: \tau \rightarrow 2^\tau$ such that $\sigma(N) \subset \tau_N$. Naturally, $\sigma(N_p) = \emptyset$.

**Definition 6.1** $N$ is a source if $\sigma^{-1}(N) = \emptyset$ and a sink if $\sigma(N) = \emptyset$.

**Definition 6.2** The *signposts field* (or field) defined by $\tau$ and $\sigma$ is the directed graph $G(\tau, \sigma) = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ (= vertices), and $\mathcal{E}$ (= edges) are given by:

\[ \mathcal{V} = \{N_0, N_1, \ldots, N_9\}, \]
\[ \mathcal{E} = \{(N, M) \mid N \in \tau; M \in \sigma(N)\}. \]

For instance, if we have $\tau = N_0 N_1 N_2 N_3 N_4 N_5, \mathcal{V} = \{N_0, N_1, N_2, N_3, N_4, N_5\}$, and $\mathcal{E}$ is defined by the mapping $\sigma: \tau \rightarrow 2^\tau$, where:

\[ \sigma(N) = \begin{cases} \{N_2, N_4, N_5\} & \text{if } N = N_0, \\ \{N_2, N_3, N_4\} & \text{if } N = N_1, \\ \{N_3\} & \text{if } N = N_2, N_3, N_4, \\ \emptyset & \text{if } N = N_5, \end{cases} \]

then the graph $G(\tau, \sigma)$ is a signposts field. $N_5$ is a sink and $N_0, N_1$ are sources.

We omit the proof of the following two simple Lemmas.

**Lemma 6.1** Let $\tau$ and $\sigma$ be defined as above. Then, $G(\tau, \sigma)$ is a directed acyclic graph. Furthermore, take $N_{k_0} \in \tau$ and assume that, $\forall N \neq N_p, \sigma(N) \neq \emptyset$. Then any path starting at $N_{k_0}$ will end up at the sink $N_9$. 

Figure 6: *Updating signposts*

Figure 7: *Updated signposts*
Lemma 6.2 Let $\tau$ and $\sigma$ be defined as above.

1. If a source is removed from $G(\tau, \sigma)$, let’s say vertex $N_s$, then the new directed graph $G(\tau', \sigma')$, where $\tau' = N_0 \ldots N_{s-1} N_{s+1} \ldots N_p$ and $\sigma' = \sigma|_{\tau\setminus\{N_s\}}$, is a field having vertex $N_p$ as sink;

2. if a new node $N_{p+1} \notin V$ is added to the graph $G(\tau, \sigma)$ by defining: $V' = V \cup N_{p+1}$, $E' = E \cup (N_p, N_{p+1})$, and $\tau', \sigma'$ accordingly, then the new directed graph $G(\tau', \sigma')$ is a field having vertex $N_{p+1}$ as sink.

Theorem 6.3 The algorithm defined at Section 6.1.2. guarantees that all messages sent to an agent will be correctly routed to their destination.

Proof: If the agent is stationary, then the resolution algorithm [5] proves that any message sent to an agent will eventually reach its destination. A fortiori, in this case, messages are correctly routed. Assume now, as above, that agent $A$ is moving from node $N_0$ to node $N_p$, passing successively through nodes $N_1, N_2, \ldots, N_{p-1}$. Its path is then $\tau(A) = N_0 N_1 \ldots N_p$. We don’t, for the moment, put any constraints on the nodes $N_i$, and we assume that no messages have been sent yet to $A$. The first message sent to $A$ will certainly reach its destination because the trajectory $\tau(A)$ contains a subpath leading to $A$. We may now distinguish two cases:

- there are no cycles in the trajectory $\tau(A)$: this case is exemplified by Figure 6. When a message $M_0$ is received by agent $A$, then Epidaure sends back an UpdateEntry message to all signposts that were met along the way up (ref. Algorithm 6.1.2.). After updating, they all point to a place (not necessarily the actual location because, in the meantime, $A$ might have moved to another node) where $A$ has been. So, if we define $V = \{N_0, N_1, \ldots, N_p\}$ and $\sigma(N) := \sigma(N) \cup \{M\}$, $\forall N \in \tau \setminus \{N_p\}$, where $M$ is the new location pointed to by $N$ after updating, then all the assumptions of Lemma 6.1 are satisfied. Consequently, $G(\tau(A), \sigma)$ is a field having vertex $N_p$ as sink. This insures that all messages sent afterwards to $A$ will be correctly routed;

- there is a cycle in the trajectory $\tau(A)$: this case is exemplified by Figure 8. Without loss of generality, we may suppose that we are in the context of Figure 8, and that agent $A$ is closing the loop by moving to node $N_0$. The first step taken by Epidaure is the removal of the signpost located on $N_0$ (see Figure 9), which is, mutatis mutandis, equivalent to the removal of a source from the field. Therefore, by Lemma 6.2.1, the graph $G(\tau(A'), \sigma')$ is a field having $N_p$ ($p = 6$, in the example) as sink and, obviously, all messages will be forwarded correctly to $A$. If the migration is made in the direction of $N_1$ instead of $N_0$, we may still consider $N_1$ as a source of $G(\tau(A), \sigma)$, because after the move, node $N_0$, the sink of a subfield pointing initially to $N_1$, will point to the sink of the new field $G(\tau(A'), \sigma')$. In any case, the routing is correct.

The second step to occur is the creation of $A$ on $N_0$, and the construction on $N_0$ of a signpost pointing to $N_0$ (see Figure 10). This is obviously equivalent to the adjunction of a new node at the sink end of the signpost graph. Therefore, by Lemma 6.2.2, the graph $G(\tau(A'), \sigma')$ is a field having $N_{p+1}$ ($p = 6$, in the example) as sink and, again, all messages will be forwarded correctly to $A$.

Furthermore, the algorithm guarantees that no obsolete updating messages UpdateEntry can change the value of a signpost pointing to a location newer than the one specified in the message (ref. Algorithm 6.1.2., step 2).

In conclusion, in all cases, messages are routed correctly to their destinations. QED.
### 7. Signposts are necessary

*Epidaure* uses two levels of knowledge in order to ensure that all messages will be transmitted to the right addressee, whether it is mobile or not. The first level is a set of communicating local and global databases [5], the second level being the collection of signposts. What we want to put forward is that, under the assumptions we have made, all these features are necessary and that removing any one of them can invalidate the “guarantee of delivery” principle.

#### 7.1. The need for signposts

Let us then suppose that there are no signposts in the system and that all localizations of agents are done through local and global databases inquiries (in *Epidaure*, the migration mechanism operates through the invocation of the remote creation feature, which, in turn, registers the agent in both local and global communication agents). This registration is done once in the lifetime of a stationary agent and, for a mobile agent, repeated at the start of each migration. For the sake of the discussion, assume that agent A, which was on node N₀ and locally registered on this node, moved to node N₁ and registers on this node (see Figure 11).

The registration messages A₀ and A₁ are sent to the global communication agent G on node N₂. At the same time, agent B on node N₃ notifies the sending of a message M to A. A request is made to agent G, with inquiry message I₂, to get the address of B but, message A₀ having arrived at N₂ after message A₁, the value returned through the update message U₀ is a pointer p₀ pointing to N₀, wrongly indicating to B to send its message to an obsolete location. When M arrives on N₃, it is buffered in the local Pending List and, by the global address resolution algorithm, the wrong pointer p₀ (instead of p₁) is passed again to N₀, and so forth. Moreover, the registration being done once at creation time, the address of A in G will never change unless agent A is moved to another node.

![Figure 10: Agent A is moved](image_url)

![Figure 11: Pointers inconsistency](image_url)

### 7.2. The need for a logical clock

Let us consider the following sequence of trajectories $\Gamma = \{T_{i,k}\}_{k=1}^{m}$, where the index $k$ is a time index. That is, if $i < j$ then $\tau_i$ is performed before $\tau_j$ (see Figure 12).

$$
\tau_1(A) = N_0 N_1 N_2 N_3 N_4,
\tau_2(M) = N_0 N_1 N_2 N_3 N_4,
\tau_3(U_4) = N_4 N_2,
\tau_4(A) = N_4 N_5 N_6 N_1,
\tau_5(M_2) = N_6 N_4 N_5 N_6 N_1,
\tau_6(U_1) = N_1 N_5 N_4 N_8,
\tau_7(A) = N_1 N_2 N_7,
\tau_8(U_4) = N_1 N_6.
$$

The arguments of the variables $\tau_i$ are the objects performing the trajectories. A is a mobile agent initially on node $N_0$, B and C are stationary agents sitting on $N_0$ and $N_8$. $M_1$ and $M_2$ are messages sent from, respectively, B and C to A, while $U_1$ and $U_4$ are updating messages (of the type $\text{UpdateEntry}$) stating that agent A can be found on $N_4$ and $N_1$. We may assume that it is perfectly possible for $U_4$ to arrive at node $N_1$ after A has visited $N_1$ for the second time and has migrated to $N_2$ and then $N_7$. The newer signpost left by the agent on $N_1$, stating that it is on node $N_2$, will be erased by $U_4$. On the other hand, updating message $U_1$ makes $N_4$ points to $N_1$, leading to the situation depicted in Figure 12. The arrows denote a cyclic signpost subgraph while the dotted arrows denote the trajectories followed by the objects. A message arriving from another node, let say $N_8$, to either $N_1$ or $N_4$ will be trapped forever in this cycle and never will it reach its target destination, agent A. The trajectory it covers, in Figure 12, is $\tau_7(M) = N_8 N_1 N_4 N_1 N_4 N_1 N_4 N_8 \ldots$

Let us see how the conditional updating of the signposts as described by Algorithm 6.1.2. is a solution to this problem. The context depicted at Figure 12 and the above scenario $\Gamma = \{T_{i,k}\}_{k=1}^{m}$ are assumed. When updating message $U_4$ arrives at node $N_1$, the value of the local variable Num

---

**Note:** The text is formatted to ensure readability and to highlight key points. The diagrams are placeholders and should be replaced with actual images. The content is a continuation of the discussion on the need for signposts and logical clocks in distributed systems, as referenced in *Epidaure*. The examples and algorithms are illustrative of the principles discussed in the text.
able NumberOfHops (= 4). Since 4 < 7, then the signpost, which was pointing to N2, is left unchanged. Hence, Message M, sent by agent C to A, will cover the trajectory \( \tau_{10}(M) = N_8 N_1 N_2 N_7 \), and will be delivered to A (see Figure 12).

8. Related works

Charlotte[2, 3], a message-based distributed operating system intended as a testbed for developing techniques and tools for large-grain parallelism, uses links, bound duplex communication channels between processes. When a process migration occurs, the channel ends of the involved process move with it. It thus avoids the problems related with leaving stubs behind while moving processes and updating them. In Charlotte, redirecting messages is done by sending channel ends between processes. The communication infrastructure of Epidaure differs fundamentally from Charlotte’s in that we chose to make our system connectionless, for openness purposes.

Voyager [8] and [4] are two systems where mobile and stationary objects can coexist. In Voyager, a moving agent leaves behind a trail of “secretaries” that are, when the agent dies, garbage-collected by the system. It is not clear how it is done. The signposts inconsistency mentioned at Section 7.1. can be a problem.

In [4], “Stub-Scion Chain Pairs” are used as a remote referencing mechanism to access objects stored on mobile and stationary hosts. SSP Chains support chain extension and short-cutting. Extension corresponds to adding stub(s) or scions(s) to either end of a chain and short-cutting to the creation of a new stub and scion pair to replace an indirect chain with a direct one.

There are no dynamic updatings of secretaries or SPP pairs in either systems.

9. Conclusion

We have described in this paper the communication and tracking infrastructure of a mobile agent system, Epidaure, which is currently developed and tested at the INRS. Furthermore, we have described in full details the tracking algorithm it implements and shown formally that all messages in the system were correctly routed to their destinations. Some problems related to the wrong routing schemes used by some systems were also pinpointed. Epidaure is a first step toward the building of a full-fledged mobile agent system.

10. References


