Mobile Agent-Based Transactions in Open Environments

Flávio Morais de ASSIS SILVA† and Radu POPESCU-ZELETIN††, Nonmembers

SUMMARY This paper describes a transaction model for open environments based on mobile agents. Mobile agent-based transactions combine mobility and the execution of control flows with transactional semantics. The model presented represents an approach for providing reliability and correctness of the execution of distributed activities, which fulfills important requirements of applications in Open Environments. The presented transaction model is based on a protocol for providing fault tolerance when executing mobile agent-based activities. This protocol is outlined in this paper. With this protocol, if an agent executing an activity at an agency (logical “place” in a distributed agent environment) becomes unreachable for a long time, the execution of the activity can be recovered and continue at another agency. The fault tolerance approach supports “multi-agent activities,” i.e., activities where some of its parts are spawned to execute and migrate asynchronously in relation to other parts. The described transaction model, called the basic (agent-based) transaction model, is an open nested transaction model. By being based on the presented fault tolerance mechanism, subtransactions can be executed asynchronously in relation to their parent transactions and agent-based transactions can explore alternatives in the event of agent unavailability. The model fulfills requirements for supporting the autonomy of organizations in a distributed agent environment.

key words: distributed systems, open environments, mobile agents, transactional agents, fault tolerance

1. Introduction

Technological advancements in networking and in distributed processing are enabling the emergence of new types of distributed processing environments. Exemplified by electronic service markets or virtual enterprises, such environments are highly complex distributed systems that support corporations’ needs for integrating systems and that allow new forms of automated cooperation.

Some of the characteristics of such systems are: they are composed by a multitude of autonomous organizations cooperating or competing to achieve their own goals; massive geographical distribution; they encompass a huge diversity of types (qualities) of communication links; the execution of inter-organizational activities are typical in such environments; a multitude of services are offered to a multitude of clients of such services; different types of services exist and they may range from totally automated services to services executed by human beings; high dynamism with no global control; high heterogeneity; and coexistence of different types of hosts (personal digital assistants (PDAs), personal computers, powerful workstations or mainframes). Environments with these properties will be called here Open Environments.

The definition of open environments in the context of this paper tries to capture the types of requirements that environments impose on applications, in a similar way that the definition of Autonomous Decentralized Systems (ADSs) was used in for example, [11].

A mobile agent (or simply agent) is a self-contained software element responsible for executing a programmatic process, which is capable of autonomously migrating through a network. The mobile agent concept has been proposed to provide support for different types of applications, including electronic commerce [15], [20], [24], workflow management [4], [5], [16], network management [8], [13], implementation of telecommunication services [14], distributed information retrieval [8] and active networks [8]. Mobile agents have been considered a concept that can be explored to provide, among others, the following benefits: better use of communication resources (both in terms of costs and performance); flexible support for disconnected operation; flexibility for the management of software deployment and maintenance; and adequate support for interactions with human users [1]. These benefits are of particular relevance to open environments, due to the properties of this type of environment, as listed above. The support for disconnected operation, for example, addresses the need for supporting PDAs and executions on/from personal computers. The ability of mobile agents to move to a host to interact with a human user enables applications to trigger interactions with that user during the execution of a distributed application. Decentralizing processing with mobile agents is a form of circumventing limitations of centralized systems, promoting scalability of systems. These exam-

†The author is with LaSiD/CPD/UFBA, Av.Adhemar de Barros, S/N, CEP 40170-110, Salvador, Brazil.
††The author is with GMD FOKUS, Kaiserin-Augusta-Allee, 31, 10589, Berlin, Germany.

Manuscript received October 1, 1999.

*We use the expression open environments to be consistent with our previous work.

**In [4] the authors describe a model where workflows are executed by information carriers (INCAS). The authors do not mention the work on mobile agents, but an INCA exhibits the functionality of mobile agents.
amples illustrate the new possibilities mobile agents offer to the design of distributed applications and to adapt them to better meet requirements of open environments when compared to conventional client/server forms of distributed interactions.

Support for transactions represents an important functionality to be present in the underlying infrastructure of a distributed environment. At least for some application areas (workflow systems, electronic commerce), it is necessary that accesses to a subset of the services available in the environment can be combined as an unit of work that executes correctly and reliably in the presence of concurrency and failures. Integrating mobile agents with a suitable transaction model results in a concept for executing distributed transactions that can explore the potential advantages of mobile agents. Combining the asynchronous mode of operation of mobile agents with the enforcement of control flows with transactional semantics results in an adequate concept for reliable processing in open environments.

This paper presents a model for the execution of distributed transactions with mobile agents. The developed transaction model is built upon the concept for fault tolerance of mobile agent-based executions presented in [1], [3]. The presented transaction model, called the basic (agent-based) transaction model, is an open nested transaction model. The model supports that parts of a distributed transaction are executed asynchronously in relation to other parts of the same global transaction. Furthermore, the model is able to recover the execution of a transaction when an agent executing parts of this transaction becomes unavailable for a long period of time. This model, described in details in [1], associates mobile agents and the execution of distributed transactions and fulfills requirements not yet fulfilled by other related concepts presented in the literature.

The paper is structured as follows. In Sect. 2, the environment where mobile agents execute (the DAE) is presented. Section 3 outlines the approach for mobile agent fault tolerance, upon which the transaction model is based. Section 4 describes the agent-based transaction model. Section 5 describes how agent-based transactions are executed. In particular, it is described how recovery of agent-based transactions is performed. Section 6 presents related work and Sect. 7 concludes the paper.

2. Mobile Agents and the DAE

The whole environment where agents execute is referred to in this paper as the Distributed Agent Environment (DAE). Figure 1 shows an agent executing in the DAE. An agent migrates in the DAE between logical “places,” called agencies in this paper. When an agent migrates, its execution is suspended at the original agency and the agent is transported (i.e., program code, data, execution state and control information) to another agency in the DAE, where it then resumes execution.

Each agency is associated with a set of application services. Application services represent application-related services, which agents can access to achieve their goals. Examples of application services are banking services, flight-ticket reservation services, etc. An agent goes to the agency where a certain application service is in order to access it locally.

3. An Approach for Mobile Agent Fault Tolerance

Mobile agents carry together the code and data of the (part of the) activity they are executing while moving through the DAE. While executing at an agency, an agent is completely subject to the execution rules and conditions of that agency. If the agency where the agent is running fails, the execution of that agent remains blocked while the agency is failed. Since an agency may remain failed for a long time, it is desirable to have a concept to enable an agent-based execution to be recovered and be continued from another agency in the event of long-term failures of agencies.

This section outlines the approach for mobile agent fault tolerance described in [1] and [3]. The presented fault tolerance mechanism recovers agent-based executions from short- and long-term failures of agencies. This mechanism represents the underlying concept for the definition of the transaction model described in Sects. 4 and 5.

Failure Model. The fault tolerance mechanism described in this section is based on the following assumptions. Each agency has a local stable storage, i.e., a storage which survives failures. Agents execute in volatile memory. Agencies fail by crashing [9], i.e., when an agency fails, it immediately halts all processing, without performing incorrect actions. Agencies, however, eventually recover from failures. In particular, when an agency recovers, it creates a new copy of each
agent that was running at it when the failure occurred. Peer-to-peer communication channels are reliable, i.e., with no damaged, duplicated or out-of-sequence messages and a message will eventually reach the destination if the sender and the receiver do not fail and they are in the same network partition. There is no known limit for the time a message sent from a node takes to arrive at its destination. Network partitioning may occur, but the network eventually recovers from failures as well.

Making Mobile Agent-Based Activities Fault Tolerant. Making the execution of an agent-based activity tolerate long-term failures of agencies is achieved by replicating agents. With replication, an agent-based execution is seen as being performed in a sequence of stages (see Fig. 2). At each stage, copies of the agent are sent to a non-empty set of agencies. The first stage begins when the activity starts. A new stage then begins (and the previous terminates) when the agent-based execution reaches a movement operation. An agent-based execution is performed now by a moving agent group, instead of by a single agent.

Figure 2 represents the execution of an agent-based activity. Consider that the activity consists of performing a sequence of actions at agencies agency\(_1\).1, agency\(_2\).1, ..., agency\(_m\).1, i.e., at the uppermost agencies at each stage represented in Fig. 2. As can be seen, at each stage copies of the agent are sent to the desired agency and additionally to a set of \(n-1\) other agencies. For example, in stage 1, copies of the agent are sent to agencies agency\(_1\).1, agency\(_1\).2, ..., agency\(_1\).n. One copy of the agent starts executing the activity (consider that the copy of the agent at the agency with the smallest index begins executing). The copies at the other agencies taking part in the stage remain inactive. By the use of a monitoring process and a leader election protocol (such as the one described in [19]), whenever the current active copy fails, some other copy recovers and resumes the execution of the failed agent. When the execution terminates at a stage (i.e., the execution of the activity reaches a movement operation), copies of the agent with the current state of its execution are sent to the agencies that will form the next stage and the copies at the current stage are destroyed.

A copy currently in charge of executing the activity is called a leader of the stage. Since detection of failures of remote nodes cannot be done precisely under our failure model, at a given instant more than one of the agent’s copies might be executing considering to be a leader. However, the protocol for managing the execution of stages guarantees that the multiple agent’s copies taking part in a stage execute consistently [1], [3]. A distinct priority is assigned to each agency participating in a stage. This priority assignment is used by the leader election protocol (leaders are elected obeying the priority assignment) and for the recovery of agent-based activities (as will be described in Sect. 5).

Input Queues and the DCD. Each agency has an input queue. When an agent is sent to an agency, the information about the agent (its id, code, data and execution state and additional control information) is put in the input queue of that agency.

A distributed repository of recovery information is associated with each stage. The repository is called the DCD (Distributed Context Database) for that stage (see Fig. 3). The DCD is formed by local repositories existing at each agency taking part in the stage.

The DCD is used to store distributed checkpoints of the execution during a stage and some control information. A checkpoint of the agent-based execution is also stored at the input queues of agencies when a new stage is started.

A variable called current leader is stored at the DCD. This variable maintains the identifier of the copy of the agent that was last elected the leader to continue executing the activity of a stage. This variable is normally set whenever an agent’s copy is elected the leader (each agent’s copy has a unique identifier).
Fig. 3 DCD: Distributed repository of recovery information (stage i).

ages the creation and termination of stages. It is carried out when one of the copies of the agent achieves the end of the execution of a stage and wants to move to another agency (in order to continue executing the activity). First the termination protocol basically: puts the information about the agent (its id, code, data and execution state and additional control information) at the input queue of each agency of the next stage; and stores a termination flag at the DCD of the terminating stage. These two steps are performed as a single atomic action. If this atomic action succeeds, the next stage can begin and the actions to terminate the current stage will start, i.e., the copies of the agent and all recovery information at the terminating stage will be deleted from the agencies that took part in it. If the atomic action does not succeed, it may be retried (with the same or with another set of agencies). If it failed because another leader was elected, a failure indication will be issued. This failure will be handled as described at the end of this section.

Making the Execution of a Multi-agent Activity Fault Tolerant. During its execution, an agent group can create other agent groups. The new agent group is said a child agent group (or simply child) of the agent group that created it, its parent agent group (or, simply, its parent). Child agent groups can be created to execute subactivities asynchronously in relation to their parent agent groups. Figure 4 illustrates the creation of two child agent groups. In this figure, the child agent groups are created while the parent (agent group) was executing its stage with index $i$. The child agent groups continue the execution asynchronously in relation to their parent.

When a child agent group is created, a distributed checkpoint of the activity of its parent is stored at the DCD of the parent’s stage. The storage of the checkpoint and the creation of the child agent group’s first stage are performed as an atomic action. With the checkpoint, if some failure involving agencies of the parent’s stage occurs after the creation of a child agent group, the execution of the parent’s stage can backtrack to the point in the execution where the child agent group was created (or to a point after that, if some more recent checkpoint was stored). This way, failures at a parent’s stage will not have effects on the execution of child agent groups.

Communication Support. The multiple agent groups executing a global activity must communicate to synchronize their actions. Communication between agent groups is considered to follow a specific pattern. After a parent agent group creates a child agent group, the child agent group will eventually return the results of its execution to its parent. The results are transmitted once, at the end of the child agent group’s execution.

A restriction is imposed on the movement of parent agent groups. If an agent group creates child agent groups during a stage, it only moves again (starts a new stage) after it has received the results of all these child agent groups. The communication support can be implemented using message multicasting (the results are multicast to the members of the parent agent group) [1].

Events. Consider a single group executing a stage. The behavior of an agent group is perceived by each copy of the agent during a stage through a set of events. By appropriately handling these events, the various copies of the agent can execute a stage consistently. The events that can be issued at each agency during a stage are:

- **leader_elected**: when the copy of the agent at the agency is elected the leader;
- **long_term_failure**: when it is realized (through the leader election protocol) at an agency that a new leader was elected at another agency;
- **processing_long_failure**: when the agency recovers from a failure and the agent’s copy was handling a long term failure (signalized previously by a long-term failure event);
- **short_term_failure**: when the agency recovers from a failure and the agent’s copy was not handling a long term failure.

An example of the use of such events is as follows. Con-
Consider a stage with three agencies, \textit{agency\_a}, \textit{agency\_b} and \textit{agency\_c} (Fig. 5). Consider that the current leader (agent painted white) is the copy at \textit{agency\_a} (Fig. 5(a)). Suppose that this leader stores a checkpoint in the DCD, produces later some effect locally (modifies a local database, for example) and then the agency where it is running (\textit{agency\_a}) fails for a long time. Eventually a new leader will be elected at \textit{agency\_b} (Fig. 5(b)). This leader reads the checkpoint from the DCD and continues executing the stage, i.e., it performs forward recovery. Suppose that this leader executes the stage until its end (Fig. 5(c)). Later \textit{agency\_a} recovers and creates a new copy of the failed agent (Fig. 5(d)). This agent continues to execute its local actions (accessing services at the agency), but eventually it realizes that the stage terminated$^\dagger$ (Fig. 5(e)). The agent at \textit{agency\_a} then cancels its effects backward until the checkpoint read by the leader at \textit{agency\_b} (Fig. 5(f))$^{\dagger \dagger}$.

In this example the following sequence of events will be issued. An event \textit{leader\_elected} will be raised at \textit{agency\_b} when the leader is elected at that agency (Fig. 5(b)). Afterwards, when \textit{agency\_a} recovers, a new copy of the agent will be created and it will receive a \textit{short\_term\_failure} event (Fig. 5(d)). The copy of the agent continues executing as if it were the leader of the stage. When it is realized that another leader was elected, a \textit{long\_term\_failure} event is issued (Fig. 5(e)). The agent then performs a partial backward recovery (Fig. 5(f)), i.e., it cancels the effects produced at \textit{agency\_a} after it has stored the checkpoint at the DCD (in the example, it cancels the effects produced at the accessed local database).

An event \textit{processing\_long\_failure} would be issued, for example, in Fig. 5(f), if \textit{agency\_a} fails while performing the local recovery actions.

These events summarize the behavior of the underlying fault tolerance mechanism.

4. The Basic Transaction Model

This section defines a transaction model that will be referred to as \textit{basic transaction model}. The definition of this transaction model aims at separating the proper-

\footnote{$^\dagger$A copy of an agent can realize the termination of the stage by receiving a message (sent as part of the termination protocol) or by verifying the existence of the termination flag (introduced previously in this section) when it accesses the DCD [1].}

\footnote{$^{\dagger \dagger}$In Fig. 5, parts (d) until (f), the stage has already terminated at \textit{agency\_c}. The stage terminates at \textit{agency\_b} (the agency from where the termination protocol is being controlled) after it has already terminated at all other agencies [1].}
ties that an agent-based transaction model should have due to the use of the underlying fault tolerance mechanism from possible extensions and alternatives that could be chosen. It is called basic because it was intentionally left as simple as possible. Variations of the model are discussed in [1].

The basic agent-based transaction model is an open nested transaction model. Open nested transaction models have been proposed for coping with long-running activities and with the autonomy of systems in multidatabases and thus take into consideration aspects of open environments.

A transaction that is submitted to be performed over the environment is called a global transaction. A global transaction is composed of a set of subtransactions. Each subtransaction may by its turn also contain subtransactions. The global transaction, therefore, has the form of a tree, called the transaction tree.

The root of this tree is called the root transaction. The term transaction will be used hereafter to denote both the root transaction and subtransactions. Other common terms for hierarchical structures will also be used hereafter, such as leaf transaction, parent transaction, etc.

The root transaction is an open nested transaction. Each of the subtransactions of the root transaction can be either a flat ACID (also called closed) transaction or an open transaction. Open subtransactions of the root transaction have the same structure as the root transaction, thus applying the transaction structure recursively. Each of the flat transactions represents a leaf of the transaction tree.

Each non-leaf transaction corresponds to a combination of its subtransactions, forming a potentially complex control flow. The control flow of a non-leaf transaction may include, for example, the specification of parallel and sequential execution of subtransactions, dynamic creation of subtransactions (instances) during the execution of a transaction and the definition of sets of alternative subtransactions (i.e., transactions that are equivalent, according to application semantics).

Each transaction has associated with it a set of input and output parameters, allowing a definition of data flow between transactions. Additionally, each transaction has a set of internal data which represent its private variables (its private state space).

The control flow of a transaction may be determined with the use of values of internal data and output parameters or outcome of previously executed transactions. The control flow is, however, restricted in the basic transaction model so that, for each transaction:

- open subtransactions can execute in parallel;
- the execution of flat subtransactions must be a sequence;
- no flat and open subtransactions can execute in parallel.

All transactions in the basic model are compensatable. Each transaction, with exception of the root transaction, has a corresponding compensating transaction†. In case the effects of a compensatable transaction must be cancelled after its commitment, its com-

†The compensating transaction can be null.
A compensating transaction cancels the effects of the compensated-for transaction according to application semantics. Compensation is performed in the reverse order of execution of the compensated-for transactions.

The compensating transaction of a leaf transaction is another flat transaction defined by the transaction specifier. The compensating transaction of an intermediary (non leaf) transaction corresponds to another open transaction that compensates the committed subtransactions of the compensated-for transaction. The compensating transaction for an intermediary transaction is defined automatically at runtime, depending on the subtransactions that have committed. Values for parameters of compensating transactions are defined by the application when the compensated-for transaction is committed or can be determined at the moment the compensating transaction executes.

Each transaction is either vital or non-vital. A vital transaction is a transaction the failure of which determines immediately the failure of its parent transaction. A failure of a non-vital transaction does not have direct effects on the outcome of its parent transaction.

Each leaf transaction is restricted to be executed entirely at the same agency, i.e., only service components at the same agency are accessed as part of a leaf transaction. The control flow of a leaf transaction represents a combination of accesses to services at that agency. A compensating transaction for a local transaction is considered to be executed at the same agency where the compensated-for transaction executed.

The general recovery semantics of the basic transaction model is as follows. In the occurrence of failures the recovery process of a transaction tries to perform forward recovery. A recovery process is performed which resets the execution to a consistent state and the transaction continues to be executed from that state on, trying to achieve a successful termination state. Backward recovery, i.e., the cancellation of the effects of a transaction, however, may also occur. Backward recovery is performed when a vital transaction aborts. In this case the parent transaction of the vital transaction will be backward recovered.

Due to the behavior of the fault tolerance mechanism, upon which this transaction model is based, partial backward recovery may also occur. In this case some of the already committed subtransactions of an open transaction are compensated as a form of backtracking the execution to a previous consistent state. Forward execution of the transaction is then performed from that state on. Partial backward recovery was illustrated in Fig. 5(f), when the copy of the agent at agency_a cancels the effects it produced after having stored the checkpoint. The basic transaction model enforces semantic atomicity.

Figure 6 shows an example basic transaction. In the figure, the root transaction t has 7 subtransactions, denoted t_1 to t_7. Transactions t_1, t_2, t_3, t_4 and t_7 are closed. Transactions t_3 and t_6 are open. Transactions t_1 and t_2 should be executed at agency ag_1. Transactions t_3, t_4 and t_7 should be executed, respectively, at agencies ag_2, ag_3 and ag_4. The open transaction t_5 has two subtransactions, t_5_1 and t_5_2, to be executed, respectively, at agencies ag_5 and ag_6. Similarly, open transaction t_6 has two closed subtransactions, t_6_1 and t_6_2, to be executed, respectively, at agencies ag_7 and ag_8. Transactions t_2, t_3, t_4, t_7 and all the subtransactions of t_5 and t_6 are vital. If any of them fails, its parent transaction must be backward recovered.

Figure 7 shows the control flow defined for the open transaction t. According to the figure, t_2 will be exe-

---

†Backward recovery can also be triggered upon user request. In this case the whole global transaction is backward recovery. This case will not be considered in this paper. See [1].
Fig. 7 A representation of a control flow for the root transaction of Fig. 6.

cuted if \( t_1 \) succeeds. Transaction \( t_3 \) is executed if \( t_2 \) succeeds. Transaction \( t_4 \) is executed if \( t_1 \) fails. Transactions \( t_5 \) and \( t_6 \) are executed in parallel, after either \( t_3 \) or \( t_4 \) succeeds. Transaction \( t_7 \) will execute after \( t_5 \) and \( t_6 \) terminate. Similar definitions of control flow are supposed to exist for open transactions \( t_5 \) and \( t_6 \).

5. Execution of Agent-Based Transactions

This section describes how transactions defined according to the model described in the previous section will be executed. The transactions will be executed by agent groups, presented in Sect. 3, in order to tolerate long-term failures of agencies.

First the relationship between transactions of a transaction tree and agent groups is defined in Sect. 5.1. Section 5.2 presents then a form of representing the execution of each agent group. Section 5.3 presents a simple form of managing the replication of agents. Finally, Sect. 5.4 presents how recovery of transactions is performed.

5.1 Relationship between Transactions and Agent Groups

A set of agent groups is used in the following way to execute a global transaction. First an agent group starts executing the root transaction. According to Sect. 4, each subtransaction of the root (and of every non-leaf) transaction may be either a close or an open transaction. If during the execution of the root transaction a closed subtransaction should be executed, that agent group will be responsible for executing it. If an open subtransaction should be executed, a child agent group is created and the execution of the open subtransaction is delegated to it. Each created child agent group, by their turn, will execute in the same way, i.e., being responsible for executing closed subtransactions and creating new child agent groups for executing open subtransactions. The whole global transaction will thus be executed by an agent group tree, where each agent group is responsible for controlling the execution of an open transaction (either the root transaction or an open subtransaction in the transaction tree).

Figure 8 represents the way three agent groups would execute the transaction represented in Fig. 6. According to this figure, agent group 1 will be responsible for executing the root transaction. The agent group 1 will execute the closed subtransactions of the root transaction \((t_1, t_2, t_3, t_4 \text{ and } t_7)\) according to the control flow and will create child agent groups to execute the open subtransactions \((t_5 \text{ and } t_6)\). Child agent group 2 will be responsible for executing the open subtransaction \(t_5\) (and its subtransactions) and child agent group 3 will be responsible for executing the open subtransaction \(t_6\) (and its subtransactions). When agent group 2 and agent group 3 complete their executions, they return the results of, respectively, \(t_5\) and \(t_6\) to agent group 1.

The uppermost transaction executed by an agent group is called that agent group’s base transaction. In the basic model each open transaction corresponds to the base transaction of a mobile agent group. An agent group executes a single open transaction (its base transaction) and its closed subtransactions and it delegates the execution of the open subtransactions to its child agent groups. The base transaction of an agent group is an open subtransaction of the base transaction of that agent group’s parent. After an agent group executes its base transaction, it returns the results and outcome of its base transaction (as well as some control information) to its parent agent group. In Fig. 8, transactions \(t, t_5\) and \(t_6\) are the base transactions of, respectively, agent group 1, agent group 2 and agent group 3.

5.2 Representing a Transaction with Basic Actions

Considering the definition of the basic transaction model and the previous section, an execution of an open
transaction can be represented as a sequence of actions. At each time during the execution of an open transaction the next action to be performed (if there is any) is one of the following:

- the execution of a flat transaction;
- the execution of an open transaction (creation of a child agent group); or
- a waiting-action, i.e., an action where the results of open subtransactions are being waited for.

Each action representing the execution of a flat transaction has an indication of the agency where the flat transaction should be executed. The exact sequence of actions is determined during runtime. Figure 9 represents the sequence of actions corresponding to an execution of transaction $t$, according to the control flow defined in Fig. 7, considering that subtransactions $t_1$, $t_2$, $t_3$, $t_5$, $t_6$ and $t_7$ succeeded.

During forward execution, the sequence of actions is determined while the transaction is being executed. In the case of backward recovery, the sequence of actions is completely determined when backward recovery starts.

Observe that the sequence of actions represent the actions performed by an agent group, since an agent group executes an open transaction.

5.3 Management of Replication

In order to use the fault tolerance mechanism, a policy must be defined for managing the replication of agents at each stage, i.e., for determining the number of agencies to be used at a given stage (degree of replication) and which agencies should be chosen. The number of agencies to use in each stage depends on the level of fault tolerance that should be maintained by the system. The choice of agencies to take part in a stage depends on the following aspects: which agencies will be potentially visited by the execution; semantics of the transactions (for example, the existence of sets

\[ \text{Although child agent groups execute in parallel, they are created in sequence, since creating a child agent group involves storing a checkpoint in the DCD (as described in Sect. 3).} \]
of alternative transactions); availability properties of
agencies; and the current flow of execution (forward or
backward).

In this paper a very simple replication manage-
ment policy will be described. It depends on if an agent
group is executing its base transaction forward or if it
is performing backward recovery:

- **forward execution:** a fixed number of agencies is
  used for each stage, say \( n \). When a request for
  movement is made, the following agencies will take
  part in the new stage: the agency to which the
  movement is desired; and a set of \((n-1)\) agencies
  chosen among the agencies of the previous
  stage or (in the case some of them is not avail-
  able) from the set of agencies that took part or
  will potentially take part in the execution (assume
  here, for simplicity, that they are known and that
  there is always a sufficient number of them to be
  used for fault tolerance). In particular the agency
  from where the new stage is being created should
  take part in the new stage. The agency with ser-
  vices that the agent-based activity wants to access
  will have the highest priority. Figure 10 illustrates
  the management of replication for the transaction
  execution represented in Fig. 9, considering
  \( n = 3 \).

  As can be seen, for the first stage, replication is
doing using agencies \( ag1, ag2 \) and \( ag3 \). The copy of
  the agent at \( ag1 \) has the highest priority, since the
  transaction \( t_1 \) should be executed at this agency.
  Agencies \( ag2 \) and \( ag3 \) are used for providing fault
tolerance. During the first stage, \( t_1 \) and \( t_2 \) are
executed. At the end of the first stage, a move-
ment request to \( ag2 \) is performed, since the applica-
tion wants now to execute a transaction there,
\( t_3 \). Agency \( ag2 \) will then be the agency with the
highest priority of the second stage. Agencies \( ag1 \)
and \( ag3 \) are used for fault tolerance since they are
agencies used in the terminating stage. During this
stage, child agent groups are created to execute \( t_5 \)
and \( t_6 \). Afterwards, stage 3 is constructed for the
execution of \( t_7 \), with \( ag4 \) as the agency with the
highest priority;

- **backward recovery:** during backward recovery,
  compensating transactions must be executed. At
  any given moment, if the next compensating tran-
sactions to execute are either a set of open compen-
sating transactions or such a set preceded by a sin-
gle closed compensating transactions, then repli-
cation using \( n \) agencies is used, as in the forward
execution case. Otherwise, no replication is used.
The set of compensating transactions to be exe-
cuted is known when backward recovery is started.
Figure 11 illustrates the management of replica-
cation for the transaction execution represented in
Fig. 9, considering \( n = 3 \) and that backward recov-
eries was started because \( t_7 \) has failed during stage
\( i \). Since \( t_1, t_2, t_3, t_5 \) and \( t_6 \) have been successfully
executed, their respective compensating transac-
tions, \( ct_6, ct_5, ct_3, ct_2 \) and \( ct_1 \), must be executed
in the reverse order of execution of the correspond-
ing compensated-for transactions (\( ct_5 \) and \( ct_6 \)
are executed in parallel). The backward recovery is
started using replication, since child agent groups
will be created to execute the open subtransactions

![Fig. 10](image-url)  
**Fig. 10** Example of replication management during forward
execution.

![Fig. 11](image-url)  
**Fig. 11** Example of replication management during backward
execution.
ct_5 and ct_6. After ct_5 and ct_6 are executed, replication is no more used (ct_3, ct_2 and ct_1 must be executed and they do not have alternatives in case of failure during their execution). Compensating transactions ct_3, ct_2 and ct_1 will be executed with stages containing a single agent’s copy (ct_3 will be executed in stage \( i + 1 \)). Transactions ct_2 and ct_1 will be executed during stage \( i + 2 \).

When a child agent group is created the same rules are followed to create its first stage. Examples of replication management policies are defined in [1],[20],[10].

5.4 Recovery

In this section it is described how failures (or failure suspicions) are handled during the execution of an agent group. Each agent group handles failures in the described way until it returns its results to its parent agent group, when it then terminates execution.

It was described in Sect. 3 that special events are issued at each agency taking part in a stage so that the multiple agent’s copies can synchronize their actions. The description of the recovery actions is thus done by describing the actions performed at the agencies taking part in a stage when those events are issued.

Recovery uses checkpoints that are stored during the execution of the agent groups. Checkpoints can be stored at the DCD or, during the creation of a new stage, at the input queues of the agencies that will take part in the stage. Checkpoints are read by a new leader when it is elected to perform recovery of the agent-based transaction. A checkpoint is stored at the DCD when:

- a child agent group is created, i.e., when the execution of an open subtransaction is spawned;
- right before a leader starts backward execution of its base transaction;
- right before a child agent group returns its results to its parent agent group.

Let the more recent checkpoint be called the last checkpoint of the stage. This checkpoint might be stored either at the DCD or at the input queues of the agencies of the stage. Additionally, the last checkpoint accessed by a leader will be defined as either the last checkpoint this leader stored at the DCD, or, if it did not store any checkpoint, the checkpoint that it read (from the DCD or from the input queue) when it was elected leader.

In general terms, an agent group works as follows. At any time each leader will try to proceed with the execution of the transaction through a certain path, i.e., through a certain sequence of actions, from the last checkpoint it accessed. If the leader succeeds in proceeding with the execution of the transaction, i.e., if it succeeds in storing a future checkpoint reflecting the path it executed, the stage execution will achieve a new stable (“frozen”) state. If a leader does not succeed in storing the next checkpoint, it will eventually cancel the effects it produced at its agency after the last checkpoint it accessed (i.e., the new effects the leader was trying to store with the new checkpoint). A specific control is performed by the agent’s copies to guarantee that an agent succeeds in storing a new checkpoint if and only if that checkpoint is consistent†.

The way each event is processed depends on if an agent group is executing a transaction forward or backward. All agent’s copies execute in the same way. While reading Sects. 5.4.1 and 5.4.2 the reader should keep in mind the representation of an open transaction with basic actions (Sect. 5.2).

5.4.1 Recovery during Forward Execution

Handling a leader-elected event. When a leader-elected event is issued at an agency, the value of the current_leader variable in the DCD is updated, the last checkpoint of the stage is read and the local copy of the agent will start executing as a new leader. While the transaction is being executed forward, the leader will execute in the following way, according to the next action to execute:

- execution of a closed flat transaction: in this case the leader will verify if the closed transaction should be executed at the agency where it is running. If yes, the local transaction is executed. If the local transaction must be executed at another agency, there are then two cases. If the transaction should be executed at one of the agencies of the stage that has a higher priority than the leader’s agency, then the transaction is considered to have failed. Those agencies are suspected to have failed by this leader’s agency, otherwise the agent’s copy at one of those agencies would be elected the leader (higher priority). If the transaction should be executed at an agency with a lower priority than the leader’s agency or on an agency not taking part in the stage, then a new stage will start with the agency where the transaction should be executed as the agency with the highest priority. If the transaction is considered to have failed and it is vital, backward recovery of the parent transaction is started;
- execution of an open transaction: the leader tries to start the execution of an open transaction (create a child agent group). If the leader succeeds in executing this action, the leader continues executing the next actions. Starting the execution of

†Inconsistencies such as two leaders starting from the same checkpoint, executing through different control paths and the checkpoint of one of them replacing the checkpoint of the other are avoided by a control mechanism [1],[3].
an open transaction may fail because a new leader was elected (and the agent copy did not know that yet). In this case, this agent’s copy starts canceling its local effects until the last checkpoint it accessed. If the action fails for some other reason, the control flow of the transaction continues to be evaluated normally (considering that transaction to have failed or retrying it);

- waiting-action: the leader executes this action, waiting for the results of open subtransactions. After it receives the results from all the started open subtransactions (created child agent groups), it simply continues executing the transaction.

Handling a short_term_failure event. When an agency recovers from a failure, it reinstates the agents that were executing in it at the moment of the failure. Each of the leaders receives a short_term_failure event.

When a leader receives this event, it must reestablish a consistent state and continue executing its activity. If the leader was executing a local transaction, the state of the local transaction must be reestablished. If the transaction was being committed, the commitment process continues. If the transaction has rolled back, it is retried.

If the leader was either starting an open subtransaction or initiating a new stage at the moment of the failure, the outcome of these operations will be determined. If they did not terminate successfully due to the failure, they can be retried.

If the leader was executing a waiting-action, it simply continues waiting.

Handling a long_term_failure event. When a leader receives a long_term_failure event, it must cancel the effects of its actions backward until the last checkpoint it accessed (as previously defined). The recovery action must interrupt the current action and compensate local transactions that were committed after that checkpoint. Then, for example, if the agent has executed a local transaction successfully and was trying to create a new stage when a long-term failure is indicated, the stage creation process is interrupted and the local transaction is compensated.

Handling a processing_long_failure event. This event is received if the agency fails while the local copy of the agent was canceling the effects of its activities at the agency (handling a previously received long_term_failure event). If an agent receives this event, it simply continues processing the long_term_failure event, using recovery information stored locally.

To illustrate the recovery actions, consider that in Fig. 10, agency ag1 fails after transaction t1 has been executed, but before the execution of transaction t2 and that it remains failed for a long time. Eventually, a leader is elected at ag2. At ag2 thus an event leader_elected will be issued. This leader will read the last checkpoint of the stage, which in this case is the checkpoint stored at the input queue of the agencies when the stage was created (no checkpoints were stored at the DCD yet). It will realize that the next transaction to execute is t1, to be executed at ag1 (according to the checkpoint read). Since ag1 has a higher priority than ag2, t1 is considered to have failed. The next transaction to execute is then t4 (see Fig. 7). A new stage then is created with ag3 (where t4 is to be executed) as the agency with the highest priority. When the agent’s copy at ag1 recovers, it will try to proceed executing the transaction t1. It will then try to execute t2. Consider that t2 is successfully executed. Considering its own results, the next transaction to execute is t3 to be executed at ag2 (for this agent’s copy, t1 and t2 were successfully executed). When it tries to create a new stage, it will try to store a new checkpoint in the DCD. Then it realizes that the stage has already terminated (the termination_flag was stored by the leader at ag2—recall Sect. 3). The copy of the agent at ag1 then cancels the effects it produced after the last checkpoint accessed by it. This is the checkpoint that was read from the input queue of the agency. Therefore, transactions ct1 and ct2 are executed to cancel the effects of t1 and t2.

5.4.2 Recovery during Backward Execution

As previously stated, replication is used during backward recovery for stages during which child agent groups will be created, i.e., when open subtransactions will be compensated. When no replication is used, the fault tolerance mechanism will issue only short_term_failure events, since there will be only one agent’s copy. In this case, the recovery actions will consist of simply the single agent (leader) continuing the execution of the transaction locally from the point it was interrupted by the failure.

During backward execution the recovery process tries to continue executing the control flow (analogously to the case of forward execution). However, special care must be taken because: each compensating transaction must eventually commit; after a compensating transaction is executed by a leader, the effects of doing that cannot be cancelled to return the execution to a previous checkpoint. During forward execution, transactions can be considered to have failed and the effects of transactions can be cancelled.

As a consequence, let, for example, agency_x be the agency where the leaf compensating transaction ckt must be executed. The agency agency_x must be the agency with the highest priority of the stage during which transaction ckt will be executed. If during a stage in which replication is used the agency with the highest priority fails, it is only possible to perform recovery from some other agency of the stage when the next action to be performed is not the execution of a
leaf compensating transaction at the agency with the highest priority of the stage (nor a leaf compensating transaction at the agency with the highest priority preceded by a waiting-action).

The events are then handled in the following way, considering these restrictions.

**Handling a leader elected event.** When a leader elected event is issued, the last checkpoint of the stage is read. If the leader is at the agency with the highest priority, it simply continues executing the transaction normally from the checkpoint. If the leader is not at the agency with the highest priority, it verifies the next action to be performed and acts as follows:

- if the next action to execute is a leaf compensating transaction at the agency with the highest priority, then this leader gives its leader condition up. It does that by not changing the value of the variable current_leader at the DCD, as it is normally done when a leader is elected (see Sect. 3). In this case, the execution of the transaction can only be continued by the copy of the agent at the agency with the highest priority. If the agent’s copy at the agency with the highest priority was already executing as leader, it will continue being the leader (and, in particular, after executing a leaf compensating transaction, the leader will not be asked for returning the execution to a previous checkpoint);
- if the next action to execute is a leaf compensating transaction at another agency than the agency with the highest priority, then the leader sets the value of the current_leader variable and continues executing the transaction, i.e., it starts a new stage with that agency as the agency with the highest priority. Observe that this happens even if the next action is the execution of a compensating transaction at the agency where this leader is executing;
- if the next action to execute is an open subtransaction (creation of a child agent group) preceded or not by a waiting-action, then the leader updates the value of the current_leader variable at the DCD and continues executing the transaction;
- if the next action to execute is a waiting-action, then the decision about what to do depends on the action to execute after having received the results of the child agent groups. If the action is to execute a leaf compensating transaction at the agency with the highest priority, then the leader gives up its leader condition, as described previously. Otherwise (the action is either to execute a leaf compensating transaction at another agency than the agency with the highest priority, execute an open subtransaction (create another child agent group) or simply terminate the stage), the value of current_leader variable is set and the leader continues executing the transaction.

Additionally, whenever the leader at an agency which does not have the highest priority generates a checkpoint, it continues executing in the same way described right above. The only difference is that, in the situations when such a leader should give up its leader condition, it simply stops executing and waits for the copy of the agent at the agency with the highest priority to be elected again.

**Handling a short_term_failure event.** In the case of a short-term failure, the leader must simply continue executing its activity. The event is handled exactly as in the case of forward recovery, except that local leaf compensating transactions must be retried until they commit.

**Handling a long_term_failure event.** Whenever the next action to be executed after the last checkpoint of a stage is either a leaf compensating transaction to be executed at the agency with the highest priority or such a compensating transaction but preceded by a waiting-action, then the leaders that are not at the agency with the highest priority will either give up their leader condition or will wait for the copy of the agent at the agency with the highest priority to be elected leader again. The leader at the agency with the highest priority is the only that will execute the activity in this case and, therefore, will not receive a long_term_failure event while performing such actions.

Leaders at the agency with the highest priority might then receive long_term_failure events only when, after a checkpoint, the next actions to be performed are either:

- the execution of an open subtransaction (creation of a child agent group) preceded or not by a waiting-action;
- the execution of a leaf compensating transaction at another agency preceded or not by a waiting-action; or
- a waiting-action and then terminate the stage.

If a long_term_failure event is issued in these cases, then the leader simply cancels its activity currently being performed.

If a leader at an agency that does not have the highest priority receives a long_term_failure event in the situations above, it simply cancels its activities as well. Such a leader might receive a long_term_failure event also when the next action to be performed is a leaf compensating transaction at the agency with the highest priority preceded or not by a waiting-action. In these situations, nothing needs to be done.

**Handling a processing_long_failure event.** If an agent receives this event, it simply continues the processing of the long_term_failure event previously received, as described right above.

6. Related Work

This work is related to a series of areas: mobile agent
fault tolerance, transactional agents, multidatabases, extended transaction models, among others. In this section some of the more relevant related work is presented.

Mobile agent fault tolerance has been considered in the context of some mobile agent projects [6], [10], [17], [18], [22]. The presented approaches, however, either do not tolerate network partitioning, do not support that subactivities of an agent-based activity can be spawned to be executed asynchronously or do not provide a concept to recover agent-based transactions from long-term failures of agencies. The protocol described in Sect.3 fulfills these requirements. A first version of this protocol was presented in [3] and a detailed description of it appears in [1].

In [2] a model for executing transactions with a single mobile agent is presented. The transaction model presented supports compensatable and noncompensatable transactions and the specification of so-called ACID groups. An ACID group is a combination of subtransactions that is executed isolated from other parts of the same transaction and from other agent-based transactions. The model supports that ACID groups or the set of noncompensatable transactions span more than a single agency. In this paper the execution of distributed transactions can be based on more than a single mobile agent. Additionally, it is not allowed here that isolated parts of an agent-based transaction span more than one agency, in order to facilitate recovery from long-term failures.

In [21] a concept is presented for executing open and closed nested transactions with multiple mobile agents. The paper, however, does not consider long-term failures of agencies.

Mobile agents have been proposed for being used in workflow management systems [4]†, [5], [16]. A mobile agent knows the definition of a workflow and controls its execution. A transactional support for the execution of the workflows is, however, not described in these works.

Recovery from long-term failures of the nodes from where a transaction is being controlled and mobility of the control flow of a transaction execution were also considered in the development of two transaction models, respectively, in the transaction model of ConTracts [23] and in migrating transactions [12]. In the ConTracts, if the node from where a ConTract is being executed fails, it can be reinstatitated at another node. A ConTract, however, does not move during its execution. In migrating transactions, the flow of control of a transaction migrates in a distributed environment. Executing transactions with mobile agents extends the notion, providing more flexibility for the distribution of code and for the movement of the transaction control flow in the environment.

The DAE can be modeled as a multidatabase. The definition of the correctness criterion used in the agent-based transaction model described took into considera-

tion the transaction models developed for multidatabases. Examples of these models are presented in [7].

In [1] the fault tolerance protocol and the transaction model presented here are described in details. In particular, in [1] aspects of the presented approach are further discussed, such as: extensions to the basic transaction model; replication policies considering the availability properties of agencies; how autonomy of systems is supported by the model; among others.

7. Conclusion

Autonomous decentralized systems represent examples of environments for which the use of mobile agents is quite convenient. For example, designing highly scalable distributed systems in a massive, heterogeneous and multi-organizational distributed environment seems to benefit much from mobile agents, given their ability to decentralize processing; to adapt to the autonomy of systems; to flexibly allow the management of installed code; and their support to the interaction with human users.

In this work a transaction model based on mobile agents was described that takes into consideration properties and requirements of open environments and their applications. The model represents a concept that integrates the mobility of agents with the execution of control flows with transactional semantics. Agent-based transactions can be used as an approach for providing reliability and correctness of distributed activities in an open environment that provides the benefits of mobile agents. The resulting concept exhibits important features that should be supported by an underlying infrastructure to fulfill requirements of applications running on open environments.

The effectiveness of the applicability of mobile agents to open environments is, however, subject to or influenced by the development of appropriate solutions to a set of issues. The main set of such issues relates to what can be called controllability of agent-based activities. Users or organizations issuing mobile agent-based activities in an open environment should have control over the execution of their mobile agents at any time. Reliability of the execution of mobile agents is one of the aspects of the controllability of mobile agent-based activities. Other aspects are security, accounting and testing. The scope of applicability of mobile agents will be dependent on the achievements reached to these issues.

The described model represents a step towards developing controllable agent-based activities. This model is currently being extended to incorporate more functionality and to decrease some of the implied costs.

†See footnote ** on p.973.
References


Flávio Morais de Assis Silva works as a researcher and lecturer at LaSiD, the distributed systems laboratory at the Federal University of Bahia (UFBA), Brazil. His main research interest areas are mobile agent systems and applications and reliable distributed systems. He received his BS, MS and PhD in Computer Science, respectively, from the Federal University of Minas Gerais, Brazil (1989), University of Campinas (Unicamp), Brazil (1993), and Technical University Berlin, Germany (1999).

Radu Popescu-Zeletin is professor at the Technical University Berlin and Director of the GMD Research Institute for Open Communication Systems (FOKUS). He led the research and development department of the BERKOM project of the German Telekom pilot project for the development of new applications in broadband ISDN environment. He published many papers on distributed computing systems and applications. He was active in standardization committees (DIN, ISO, EURESCOM) and had contributed to the development of telecommunication standards. Prof. Popescu-Zeletin graduated at the Polytechnical Institute Bucharest, Romania, got his PhD from the University of Berlin, Germany and his habilitation from the Technical University Berlin. He is Senior Member of IEEE, Doctor honoris causa of the Polytechnical Institute Bucharest and Professor honoris causa of the Catholic University of Campinas, Brazil.