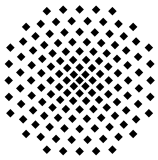


Universität Stuttgart
Fakultät Informatik

System Mechanisms for Partial Rollback of Mobile Agent Execution

Authors:
Dipl.-Inform. M. Straßer
Prof. Dr. rer. nat. K. Rothermel

Institut für Parallele und Verteilte
Höchstleistungsrechner (IPVR)
Fakultät Informatik
Universität Stuttgart
Breitwiesenstr. 20 - 22
D-70565 Stuttgart

System Mechanisms for Partial Rollback of Mobile Agent Execution

M. Straßer, K. Rothermel

Bericht 1999/10
Juni 1999

System Mechanisms for Partial Rollback of Mobile Agent Execution

Markus Strasser and Kurt Rothermel

Institute of Parallel and Distributed High-Performance Systems
University of Stuttgart, Breitwiesenstr. 20-22, D-70565 Stuttgart, Germany
markus.strasser@informatik.uni-stuttgart.de

Abstract: Mobile agent technology has been proposed for various fault-sensitive application areas, including electronic commerce, systems management and active messaging. Recently proposed protocols providing the exactly-once execution of mobile agents allow the usage of mobile agents in these application areas. Based on these protocols, a mechanism for the application-initiated partial rollback of the agent execution is presented in this paper. The rollback mechanism uses compensating operations to roll back the effects of the agent execution on the resources and uses a mixture of physical logging and compensating operations to rollback the state of the agent. The introduction of different types of compensating operations and the integration of an itinerary concept with the rollback mechanism allows performance improvements during the agent rollback as well as during the normal agent execution.

Keywords: mobile agents partial rollback fault-tolerance compensation itineraries

Technical areas most relevant to this paper:

- Distributed Fault-Tolerant Systems
- Multi-Agent Systems

1 Introduction

Throughout the past years, the concept of mobile agents has drawn a lot of attention in both academia and industry. However, only few “real” applications based on mobile agents exist today. The rather early stage of currently available agent platforms might be one reason for that. Functions critical for applications, such as security mechanisms, are often incomplete or missing entirely. Moreover, only little work has been done so far to integrate agent technology with legacy systems, such as TP-Monitors and transactional resource managers. In [11], the integration of mobile agent technology with transactional technology to provide fault-tolerant exactly-once execution of a mobile agent has been presented. Based on these results, this paper provides a mechanism for the application-invoked partial rollback of the mobile agent execution.

Agents are autonomous objects which may move from node to node to access services provided there. Agent execution proceeds in *steps*, where a new step is initiated whenever an agent migrates to the next node. When an agent decides to migrate to another node, the agent’s code, data and execution state is captured and transferred to the next node, where it is initiated after arrival.

The use of mobile agents has been proposed for many application areas, including electronic commerce, systems management, or active messaging. But only the use of protocols like the exactly-once execution protocols presented in [11] allows the usage of mobile agent technology in these application areas. However, the strict forward recovery of these protocols contradicts the autonomy of mobile agents. For situations where an abort and restart of the step transaction is not sufficient to deal with an error situation (e.g. if the agent lacks the permission to access a resource) or where the program logic of the agent detects that the current strategy does not lead to the agent’s goal, the agent needs the ability to initiate a partial rollback of its execution.

In this paper, we will first investigate which types of operations a mobile agent performs can be rolled back. Then, we will propose a rollback mechanism which uses compensating operations to rollback the effects of an agent on the

resources used and which uses a mixture of physical logging and compensating operations to rollback the state of the agent. This mechanism ensures, that the partial rollback will be executed eventually even in the presence of non-lasting node and network crashes. To increase the performance, an extension of the mechanism is presented which allows to prevent unnecessary agent transfers during the rollback. Integrating itineraries with the rollback mechanism provides a structured way to automatically define agent savepoints and provides the possibility to reduce the size of the log necessary for the rollback mechanism. The rollback mechanism is currently implemented in Mole [1][9], a mobile agent system developed at Stuttgart University.

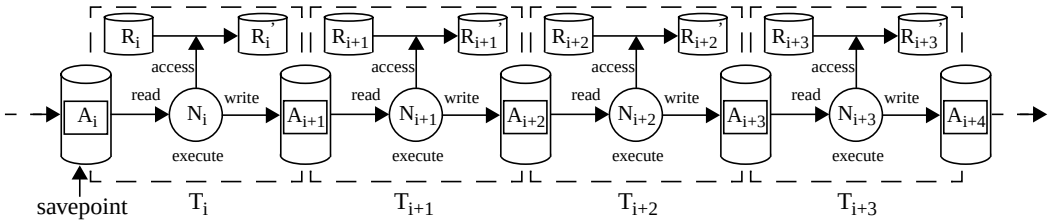
The remainder of the paper is structured as follows. In the next section, we will describe our mobile agent model and execution model. Section 3 investigates the different types of compensation. Section 4 is dedicated to the mechanism for partial rollback. Section 4.1 introduces two different types of private agent data allowing to roll back the private agent data belonging to one type using an image of the data while the data belonging to the other type has to be compensated using compensating operations. Section 4.2 presents the logging mechanism necessary for the partial rollback. Section 4.3 introduces a first version of the rollback mechanism which is extended in Section 4.4 to increase the performance of the algorithm. Related work is discussed in Section 5 before the paper concludes with a brief summary.

2 Mobile Agent Model and Execution Model

This section describes our model of mobile agents and their execution. Mobile agents are autonomous objects, which perform a job on behalf of their owner. While performing a job, an agent often has to visit several network nodes to access local resources by invoking operations on these resources. The set of actions an agent has to perform on a single node is called a *step* and is implemented as a single method of the agent object. Which step the agent has to perform on which node and the order in which the steps have to be performed is described by an *itinerary*, which may be adapted during the execution of the agent [15]. If an agent migrates to another node, the agent object with code and all private data belonging to the object is captured and transferred to the next node. There, it is re-instantiated and the step (i.e. the method implementing this step) to be performed on this node is executed.

To provide reliable agent execution, the agent is executed using one of the protocols for providing the exactly-once property of mobile agents presented in [11]. The basic idea of these protocols is to store the agent in stable storage between steps and execute each step of an agent inside a transaction, the *step transaction*. After the start of the step transaction, the agent object with all its data and code is read from the agent input queue of the node which resides on stable storage, the agent is re-instantiated and then the method implementing the step is invoked. At the end of the step, the agent object with all its data and code is captured and transferred to the next node where it is stored in the nodes' agent input queue on stable storage. Then the (distributed) step transaction is committed. It is important to note that all accesses to local resources are performed within the step transaction. Therefore, if the execution of a step aborts, all changes to resources during the step transaction are undone automatically and the agent still resides in the input queue of the node that executed the aborted step. In the case where a step transaction commits, our model is very similar to the saga transaction model [4] with the addition, that a temporary (stable) savepoint of the program execution (the agent state) is written after every step, allowing to abort and restart a step. For situations where an abort and restart of the step transaction is not sufficient to deal with an error situation (e.g. if the agent lacks the permission to access a resource) or where the program logic of the agent detects that the current strategy does not lead to the agent's goal, the agent has the ability to initiate a partial rollback of its execution. The points to which an agent can be rolled back are called *agent savepoints*, and have to be constituted by the agent program logic. Due to the fact that most transaction management systems do not support resource savepoints, agent savepoints can only be constituted at the end of a step.

Let us illustrate this: Figure 1 shows the execution of steps i through i+3 of an agent. The agent constituted a savepoint after step i-1 and is currently executing step i+3. If the agent commits this step, the agent is written into stable memory (A_{i+4} in the figure). However, if the agent decides to roll back its execution to the last savepoint, only the effects of step transaction T_{i+3} can be undone by the transaction management. The effects of steps i through i+2 on the resources accessed during those steps have to be undone by using compensating transactions [8]. We will show in Section 3.2, that the private data state of the agent cannot be rolled back by using A_i to resume the execution of the agent. Therefore, in contrast to common approaches (e.g. [4]), in our approach the private agent state is rolled back as well during the compensation transactions.



A_i: agent state before step i	R_i: resource state of node i
N_i: node executing step i	R_i': R_i after step i
T_i: step transaction i	

Figure 1: Execution of an agent

3 Classification of Compensation Types

The compensation of an operation aims at undoing the semantic effects of this operation. However, there are different operation types and not all types can be compensated to the same degree. After the introduction of some notations, this section introduces different types of compensation operations.

3.1 Notations and Definitions

The formal notations and definitions given in this sub-section are adopted from [8]. In order to be able to describe the rollback of the private data of an agent as well as the rollback of the resources accessed by it, we use the notion of the *augmented state*. The augmented state space is the state space of the resources accessed by the agent merged with the private data space of the agent object. This enables us to describe the execution of a step as a sequence of operations on this augmented state. Rolling back several steps of an agent execution requires the compensation of the operations executed during these steps. This compensation can also be described as a sequence of (compensating) operations on the augmented state.

We use the symbol S to denote a state and the symbol f to denote an operation. In contrast to [8], operations in our context may read and write any number of entities in the augmented state space. A *history* is a sequence of operations which defines a total order among the operations as well as a function mapping augmented states to augmented states. The notion $X=< f_1, f_2,..., f_n >$ describes a history in which f_i precedes f_{i+1}, $1 \leq i < n$, the notion $X=f_1 \bullet f_2 \bullet ... \bullet f_n$ denotes the function mapping augmented states to augmented states defined by the same history X. Upper case letters from the end of the alphabet will be used to denote the sequence as well as the function a history defines. Two histories X and Y are equal (denoted as $X \equiv Y$), iff for all augmented states S, $X(S)=Y(S)$. Two histories X and Y of operations *commute*, if $(X \bullet Y) \equiv (Y \bullet X)$ holds.

We use the symbol T for step transactions. The compensating operations for a transaction T are carried out inside a compensation transaction, denoted by CT. A transaction T_2 is dependent of a transaction T_1 if it reads data updated by T_1. The set of transactions dependent of T is denoted as $dep(T)$.

3.2 Types of Compensation

The idea behind compensation is to semantically undo the effects of already committed transactions. Unfortunately, compensation of operations is not always possible. Whether compensation is possible depends not only on the operation itself, but also on the application. In this section, different types of compensation are introduced.

The most comfortable type of compensation operations are those which create *sound* histories. A history is *sound* iff $X(S)=Y(S)$ with X being the history of T, CT and $dep(T)$, Y being the history of $dep(T)$ and S being the initial state [8]. In this case, the outcome of $dep(T)$ is not influenced by the compensation. If the compensation operations of CT commute with each of the operations contained in $dep(T)$, then the history X of T, CT and $dep(T)$ is sound. As an example, we consider a bank account with the operations *deposit*(x) and *withdraw*(x). If the account may be overdrawn, these two operations commute and therefore, as long as T, CT and $dep(T)$ only use those operations, the histories produced are sound. However, this type of compensation rarely occurs in real applications. Let us assume that one of the transactions in $dep(T)$ uses the current account balance to decide which actions to perform ("if I have enough money, then...."). Now we have created a very simple transaction that does not commute with *deposit*(x) and *withdraw*(x). Please note that the definition of soundness implies that $T \bullet CT \equiv I$ (with I as the identity operation).

For most applications it is acceptable, that an execution of only the dependent transactions $dep(T)$ without T and CT produces different results than those produced by T, CT and $dep(T)$ or even that $(T \bullet CT)(S) \neq S$. If, for example, a transaction T_1 tries to buy some goods from a shop and the desired good is out of stock because another transaction T_2 just bought these goods, T_1 - as in real life - buys the good from another shop. This transaction T_1 - buying from the other shop - is not affected if T_2 is compensated a short time later, even if the first shop would now be able to deliver the desired good. The reasons for accepting that the consecutive execution of the transaction and the compensating transaction does not result in the initial state are twofold. First, there are compensation operations which only produce a state equivalent to the initial state. If an agent uses digital cash [2] contained in its private data to buy some goods and compensates this operation, it (hopefully) gets back the same amount of cash. However, the representation of this digital cash is only an equivalent representation - the digital coins have different serial numbers. Second, if, in the same example, the seller of the goods charges a small fee for the compensation transaction or only agrees to give a credit note to the customer, the agent contains other information than before the purchase. In a more complicated scenario, the amount of reimbursement may vary with the time passed between purchase and compensation. The following policy is but one example: until x hours after the purchase, the seller returns cash but charges a small fee, after that, the customer only gets a credit note. In these cases, the agent must be able to deal with the changed situation.

Until now, it has been assumed implicitly that a compensation transaction has to be accomplished successfully eventually. But there are cases where a compensation operation might be impossible to execute. For example, if transaction T_1 deposits 20USD on an account, then CT_1 has to withdraw 20USD. If the account cannot be overdrawn, there have to be at least 20USD on the account for CT_1 to be successful. If there are less than 20USD (e.g. because another transaction T_2 withdrew all money in the meantime), the compensation transaction fails. Solutions to this problem are discussed in [4] and [10].

Finally, there are operations which cannot be compensated. If, for example, a transaction deletes a considerable amount of data in a database, it would be necessary to log all this data to be able to compensate the deletion. Therefore, if a step contains an operation which cannot be compensated, the step cannot be rolled back after its commit.

4 Rollback of the Agent Execution

Rollback in our model requires the compensation of actions performed on resources as well as the compensation of the agent's private data space. To support the rollback on the agent's private data space, we first identify two different types of agent data, which allows to rollback a part of the agent's data space without compensating operations. Then, we introduce the mechanism we use for logging and present the rollback mechanism. Introducing different types of compensation operations will allow the presentation of an optimized rollback mechanism.

4.1 Rollback of the Private Agent Data Space

The data objects contained in the private data space of the agent can be classified in two categories. The first category are data objects which can be compensated by means of an image of the objects. If a savepoint is established, an *image* of these data objects (*before-image* [6]) is written into the *agent rollback log* (see next section). If an agent has to be rolled back to a savepoint, these objects can be restored using the image stored in the log. For example, if an agent

collects information and stores this information into a vector, then this information can be rolled back to a savepoint without the need of a compensating operation. In this case, the vector can be restored using the original content contained in the vector at the time the savepoint has been taken. We call this type of objects *strongly reversible objects*. Strongly reversible objects (which have to be declared by the developer as such) will be restored by the system without the need for a compensating operation.

As described in Section 3.2, there are applications which accept augmented states produced by the compensation which differ from the augmented state produced without the execution of the step and the compensation of this step. The second category of data objects contains those objects in the private data space of the agent which may contain different data after the compensation, i.e. which cannot be compensated using before-images. The reason for not being able to use a before-image for rollback is that during the agent rollback, information originally not contained in the agent's private data space is produced (usually by the rollback of the state space of the resources). This new information has to be integrated into the private agent data. An example showing that electronic cash belongs into this category of data is the scenario presented in Section 3.2 where an agent orders some (digital) good using electronic cash. We call the objects contained in this second category *weakly reversible objects*. These objects cannot be compensated using a before-image of the objects, i.e. the agent developer has to provide code for the compensation.

4.2 Logging

The data necessary for the rollback of previous, committed steps of an agent is contained in the *agent rollback log*. It contains all information for the rollback of the private data space of the agent as well as for the rollback of the resource state space. The log is attached to the agent and hence migrates with the agent from node to node. Because the log only contains data for the compensation of already committed steps (backward recovery), this log is made persistent at transaction commit (i.e. at the end of step transactions or end of compensating transactions).

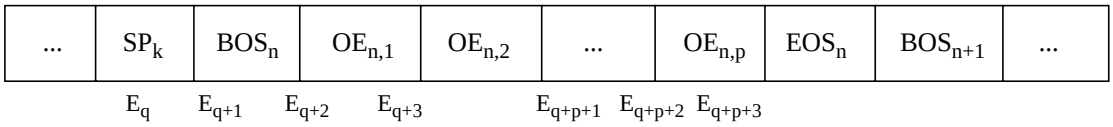
The advantages of attaching the log to the agent are twofold. Firstly, at the end of the agent execution, no global actions are necessary to delete the log. Secondly, the log is always available as long as the agent is available, enabling the agent to roll back its execution as long as the resources necessary for the rollback are available. The problem of this approach is, that the amount of data which has to be transferred to migrate the agent increases. A solution to this problem is introduced in Section 4.4.2.

The type of logging used in our approach is a mixture of physical logging and logical logging [6]. The images of the strongly reversible objects are logged using physical logging. This can be done either by writing a complete image of the objects into the log (state logging) or by writing differences of the object states between adjacent savepoints (transition logging). These informations to restore the strongly reversible objects are written to the log as part of a *savepoint entry* (SP). A savepoint entry is written to the log when an agent savepoint is constituted. Besides the image of the strongly reversible objects a savepoint entry contains a (unique) identifier for the savepoint. For reasons of simplicity, we further assume that state logging is used unless mentioned otherwise.

For the compensation of the weakly reversible objects as well as the state space of the resources, logical logging is used by writing the compensating operations and their parameters into the log. These entries will be called *operation entries*. An operation entry contains the code of one compensating operation and the parameters for this operation. As has already been mentioned in Section 3.1, the compensation of a step can be described as a sequence of operations on the augmented state. The compensating operations can be arbitrarily complex. Therefore, the number of compensating operations contained in the log which are associated to a step may vary from one (complex) operation, which compensates all the effects of the step on the weakly reversible objects and the state space of the resources, to several times the number of operations performed by the step if some of those operations need several compensating operations.

In addition to the savepoint and operation entries, the log contains entries to log the begin and the end of a step (*begin-of-step* (BOS), *end-of-step* (EOS)) These entries contain the identifier of the node on which the step has been executed. Figure 2 shows an extract of a rollback log. It contains the entries of the k-th agent savepoint which is located before step n and for the compensation of the n-th step. To rollback to this k-th agent savepoint, the log has to be executed beginning from the end of the log up to the savepoint entry. To compensate a step, all operation entries associated with this step are executed within a transaction in the reverse order they appear in the log. For example, to compensate

step n, the compensation operations are executed in the order $OE_{n,p}$, $OE_{n,p-1}$, ..., $OE_{n,2}$, $OE_{n,1}$. The next sections will present the details of the rollback process.



E_x - log entry
SP_x - savepoint entry
OE_x - operation entry
BOS_x - begin-of-step entry
EOSx - end-of-step entry

Figure 2: Example Log

4.3 The Rollback Mechanism

The idea of the rollback mechanism is to perform all compensating operations associated with a step on the node where the step has been executed. The steps are compensated in the reverse order of their execution. Similar to the execution of steps, the compensation operations associated with a step are executed within a compensation transaction. This ensures that other transactions see either a resource state affected by the step which has to be compensated or the resource state after the compensation has taken place (isolation of the compensation). The state of an agent between two compensation transactions is stored in stable storage. Until the savepoint is reached to which the agent has to be rolled back, only the changes to the weakly reversible objects of the agent and the changes to the state space of the resources accessed by the agent are compensated. The state of the strongly reversible objects is restored when the savepoint is reached using the information contained in the savepoint entry in the log. Hence, accessing the strongly reversible objects during the execution of the compensating operations is not allowed.

Figure 3 shows the execution of steps i through i+3 of an agent and the rollback initiated in step i+3 to the savepoint established before step i. After the abort, the transaction management undoes the changes performed during step i+3 to the resource state space and the agent space. Then the compensation transactions are executed on the nodes in the reverse order of the step execution. It is important to note that the strongly reversible objects are not restored until the savepoint is reached. If a compensating operation e.g. on node N_{i+1} accessed the strongly reversible objects it would read the (“old”) state established by step i+3.

Figure 4 shows the rollback algorithm. In Figure 4a, the part of the algorithm executed on the node where the rollback has been initiated (current node) is shown. This algorithm gets the identifier of the savepoint (`spID`) to which the agent has to be rolled back as parameter. After the abortion of a step transaction (hereafter called the aborting step transaction), a new transaction is initiated and the agent and the agent rollback log are read (and deleted) from stable storage. Please note that the state of the agent and the rollback log read from stable storage is the state before the execution of the aborting step transaction. Now two cases have to be distinguished. The first case is that the desired savepoint was set directly before the aborting step transaction. In this case, the rollback is already finished and the next step transaction has to be initiated. In the second case, the first compensation transaction has to be initiated. This is done by writing the agent, the agent rollback log and the savepoint identifier `spID` to the input queue of the node where the first compensation transaction has to be executed. This node can be determined by examining the last end-of-step entry contained in the agent rollback log (which is the last entry if no savepoint entry has been written after the last end-of-step entry). Then the transaction is committed. If this transaction commits successfully, then the second part of the algorithm given in Figure 4b is executed (or the next step transaction is initiated). If the transaction fails (e.g. due to a node failure), the agent and the log still resides in the input queue of the current node. In this case, the step which initiated the abort is restarted on the current node or, if the fault tolerant protocol for the execution of steps [11] is used,

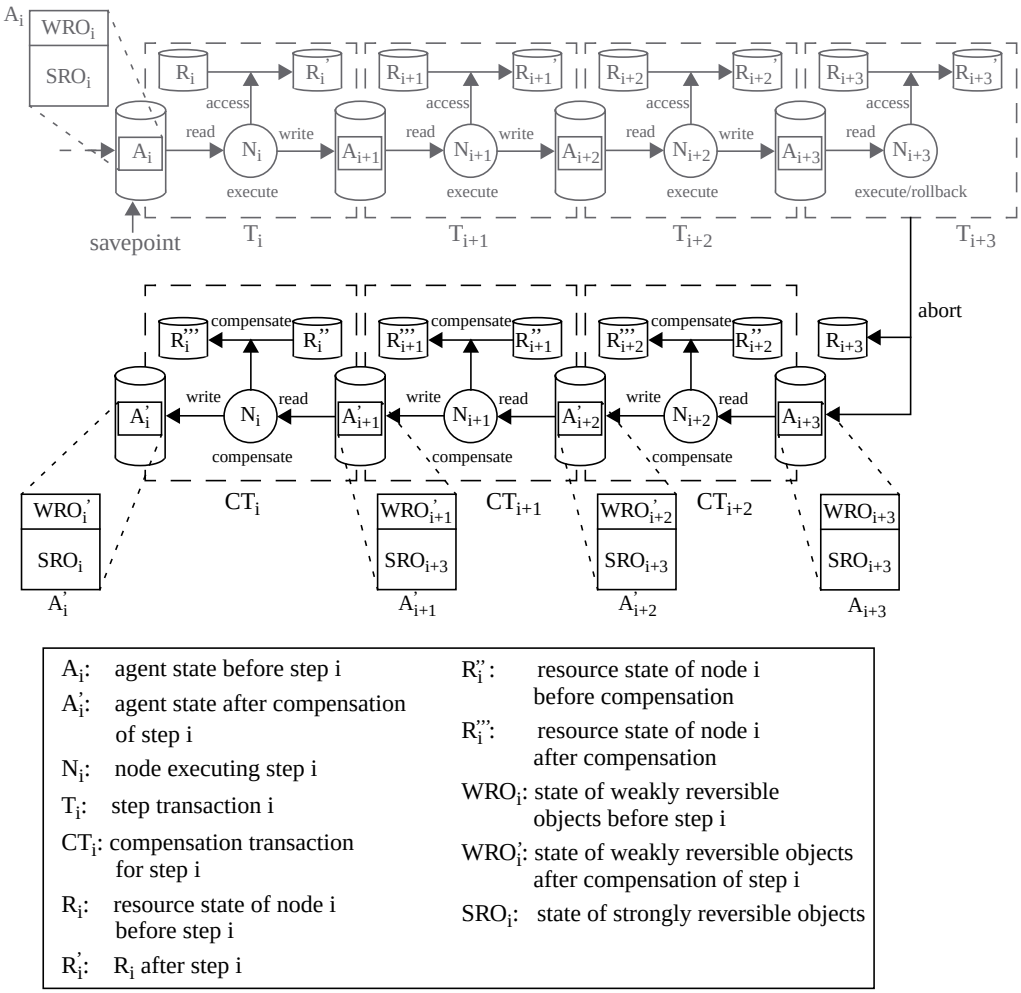


Figure 3: Partial Rollback of an Agent with the Rollback Mechanism

it may be even restarted on another node. This is still a correct execution since it has the same result as if the step transaction was aborted before the initiation of the rollback.

Figure 4b shows the part of the rollback algorithm executed on the nodes where the compensation transactions have to be executed. After the start of the compensation transaction, the agent, the agent rollback log and the savepoint identifier (`spID`) of the savepoint to which the agent has to be rolled back are read (and deleted) from stable storage. First, the end-of-step entry of the step which has to be compensated and, if existent, a savepoint entry are deleted from the log (LOG.pop() reads and removes the last entry from the agent rollback log). This savepoint entry can be deleted because it cannot be the savepoint to which the agent has to be rolled back (this is tested before the agent is written to stable storage). Then, all operation entries are read from the log and the compensating operations contained in those entries are executed until the begin-of-step entry is reached (which is also deleted from the log). It has to be mentioned, that a savepoint can only be written after the execution of a step (see Section 2) and therefore, no savepoint entries can be found between an BOS entry and an EOS entry. Now, two cases have to be distinguished. If the last entry now contained in the log is the savepoint to which the agent has to be rolled back, the states of the strongly reversible objects have to be restored using the information contained in the savepoint entry (without deleting the savepoint entry from the log) and the next step transaction has to be initiated. If the last entry is another savepoint or the end-of-step entry

```
rollback(spID){
   abort-transaction();
   begin-transaction();
   read(agent,LOG) from
      stable storage;
   if savepoint spID reached{
      start next step transaction;
   }else{
      write(spID,agent,LOG) to
         input queue of next node
   }
   commit-transaction();
}
```

Figure 4a: Start of the Rollback Algorithm

```
begin-transaction();
read (spID, agent, LOG) from
   node input queue;
if (last log entry is savepoint)
   LOG.pop(); // remove savepoint
LOG.pop(); // read end-of-step
entry=LOG.pop(); // next entry
while (entry<>begin-of-step){
   execute entry;
   entry = LOG.pop();
}
if savepoint spID reached{
   restore strongly reversible
      objects;
   start next step transaction;
}else{
   write(spID,agent,LOG) to
      input queue of next node;
}
commit-transaction();
```

Figure 4b: Rollback Algorithm Executed on each Node

of the next step which has to be compensated, the agent, the agent rollback log and the savepoint identifier `spID` are written to the input queue of the node where the next compensation transaction has to be executed (determined from the end-of-step entry contained in the log). Finally, the compensation transaction is committed. If the commit is successful, the effects of the compensating operations invoked on the resources are permanent and the new agent state (reflecting the compensating operations performed on the weakly reversible objects) is contained in the input queue of the node where the next compensation transaction has to be executed or, if the desired agent savepoint is reached, the next step transaction is initiated. If the compensation transaction aborts (node failures, deadlocks,...), the effects of compensating operations invoked on the resources are undone by the transaction management and the agent (including log and savepoint identifier) still resides in the input queue of the node enabling the algorithm to restart this compensation transaction.

Discussion: The algorithm presented in this subsection rolls back an agent to an agent savepoint by moving the agent back along the way it moved during the execution of the steps which have to be rolled back. On each node, the compensating operations to compensate the resource state of the node as well as the state of the weakly reversible objects are executed within a compensation transaction. Assuming that node crashes and network crashes are only temporary and further assuming, that the network provides reliable data transfer, the algorithm ensures that all steps which have to be rolled back are eventually rolled back and finally, the state of the strongly reversible objects is restored as well. If, instead of state logging, transition logging is used, the state of the strongly reversible objects has to be updated every time an agent savepoint entry is read during the rollback process.

The algorithm as presented above has two problems. The first problem is, that if a node on which a compensation transaction has to be executed is permanently unreachable, the rollback cannot proceed and the agent is blocked. A solution to this problem is to provide the information, on which nodes the rollback of a step can be performed alternatively (if there are any) e.g. in the end-of-step entry. Then a fault-tolerant execution of the rollback similar to the fault-tolerant step execution described in [11] can be realised.

The second problem of the algorithm is, that the agent is transferred to a node during the rollback process although maybe no resource compensation has to be performed on this node. If, for example, the agent only gathered some information during a step and stored this information in strongly reversible objects, no compensating operations at all are necessary to roll back this step. A solution to this performance problem is introduced in the following subsection.

4.4 Optimizing the Partial Agent Rollback

Two possibilities for optimizations regarding the partial agent rollback are to avoid unnecessary agent transfers during the rollback on the one hand and to reduce the transfer size of an agent by reducing the size of the agent rollback log on the other hand. This section introduces mechanisms for these optimizations.

4.4.1 Optimizing the Number of Agent transfers

To be able to decide if the compensation transaction of a step has to be executed on the node where the step was executed, the agent rollback log has to contain the appropriate information. To allow a flexible and efficient rollback, we define three different types of operation entries.

Types of Operation Entries

The first type of operation entries contains a compensating operation which rolls back only resource state space and which needs no access to the private agent state space. All information necessary for this compensating operation has to be contained in the operation entry as parameters; the compensating operation must not access the private agent state space. For example, if an agent invoked a fund transfer between two accounts of a bank, all information necessary to compensate this fund transfer is the two bank accounts and the amount of money transferred between these two accounts. This type of operation entry is called a *resource compensation entry.* Resource compensation entries have to be executed on the node on which the resource resides (i.e. on the node where the step was executed). Because the compensating operation contained in a resource compensation entry is not allowed to access the agent, it is possible to send only the operation entry (without the agent) to the node where the compensating operation has to be executed (within the compensation transaction)

The second type of operation entries contains a compensating operation which rolls back only weakly reversible objects (i.e. private agent state space) and which needs no access to the resource state space. All information necessary for this compensating operation has to be contained in the operation entry as parameters and in the weakly reversible objects of the private agent state space. As described in Section 4.3, the compensating operation may not access the strongly reversible objects. We call this type of operation entry *agent compensation entry.* An agent compensation entry always has to be performed on the node where the agent resides. Because no resource access is allowed, this may be an arbitrary node.

The third type of operation entries contains a compensating operation which needs access to the private agent state space (only weakly reversible objects, see above) and to the resource state space. An example scenario is a step, where the agent changes digital cash from one currency into another (e.g. from USD into Euro) at the bank. To compensate this (i.e. to change the money back from Euro to USD), the compensating operation needs access to the weakly reversible object containing the cash in Euro (it cannot be contained in the rollback log, see Section 4.1), to the object where the received USD have to be stored (it is not possible to restore the digital cash in USD by using a copy of the original cash, see Section 4.1 also) and to the resource which changes the money. We call this type of operation entry *mixed compensation entry.* To execute a mixed compensation entry, the agent has to reside on the node where the step to which the mixed compensation entry belongs was executed.

Optimization

As can be seen from the description of the operation entry types above, the agent has to be transferred to a node during the rollback only if a compensation transaction has to execute a mixed compensation entry there. If the operation entries associated to a step are only agent compensation entries or resource compensation entries, it is not necessary to transfer the agent to the node on which the step was executed for the compensation. In this case, the agent compen-

sation entries are executed on the node where the agent currently resides and the resource compensation entries are sent to the node where the step was executed for execution.

To decide whether an agent has to be transferred to another node to perform the next compensation transaction, the agent rollback log must be examined. One possibility is to read all entries for the next step (i.e. the step which has to be compensated next). The other possibility is to include a flag in the end-of-step entry indicating whether a mixed compensation entry is contained in the step. In this case, only this end-of-step entry has to be examined. To integrate this optimization, the changes shown in Figure 5 have to be made to the algorithm.
Instead of always writing the agent to the "next" node, the agent has to be transferred only if the flag in the end-of-step entry indicates that one of the operation entries which has to be executed in the next compensation transaction is a mixed compensation entry.

The loop of the original algorithm executing the operation entries has to be replaced completely. If the compensation transaction has to execute a mixed compensation entry, all operation entries are executed locally in the order defined by the rollback log. In the other case, the agent compensation entries can be executed concurrently to the resource compensation entries because the definition of those operation entry types ensure that they operate on disjoint data. Therefore, the log is read and two lists are made - one containing the agent compensation entries and the other containing the resource compensation entries (always in the order in which the entries have to be executed). Now the list containing the resource compensation entries and the identifier of the compensation transaction is sent to the resource node. Then, the agent compensation entries contained in the other list are executed (in the order in which they are contained in the list). On the resource node, the resource compensation entries are executed in the order in which they are contained in the list. All the operations performed on the resource node are performed inside the compensation transaction (by using the transaction identifier obtained along with the list). After the last resource compensation entry is executed, an acknowledgement is sent back to the node where the agent resides. Only then, the compensation transaction can be committed.

Discussion: The optimization described in this subsection avoids agent transfers for compensation transactions in which no mixed compensation entries have to be performed. This reduces the network load during the agent rollback, because only the resource compensation entries have to be transferred to the node on which the step being compensated was executed. Additionally, the resource compensation entries can be executed concurrently to the agent compensation entries which may save execution time.

Further optimizations are possible, if the access to resources within the mixed compensation entries and the resource compensation entries may be performed using RPC. In this case, a performance model similar to that introduced in [16] can be used to determine if the agent or the resource compensation objects should be transferred to the node where the resources reside or if RPC should be used to access the resources.

4.4.2 Reducing the Log Size by Integrating Itineraries and Rollback

Attaching the rollback log to the agent introduces some overhead to the migration because the log has to be transferred additionally to the agent state. This subsection introduces solutions for reducing the rollback log size.

There are two possibilities to reduce the size of the log. The first possibility is to reduce the number of savepoint entries contained in the log. As described above, savepoint entries are written when an agent savepoint is established. This can be influenced by the application developer by giving up the possibility to roll back an arbitrary number of steps i.e. by not establishing an agent savepoint before each step. The second possibility is to discard rollback information which is not needed any more. The information which rollback information can be discarded has also to be provided by the application developer. In this subsection, we will present an itinerary concept which allows the application developer to define in a structured way to which points an agent can be rolled back and when rollback information can be discarded.

The basic idea is, that an application can be structured into sub-tasks. If a rollback is necessary, always the complete sub-task currently executed has to be rolled back. To provide enhanced flexibility, each (sub-)task may be partitioned into further sub-tasks, allowing hierarchies of sub-tasks. In this case, the application developer may specify whether only the nested sub-task currently executed has to be rolled back or (one of) the surrounding sub-tasks that contains the current sub-task. To describe such a hierarchy of sub-tasks, an itinerary concept similar to that described in [14]

```
rollback(spID){
   abort-transaction();
   begin-transaction();
   read(agent,LOG) from
      stable storage;
   if savepoint spID reached{
      start next step transaction;
   }else{
      if (next EOS entry indicates
         mixed compensation entry){
         write(spID,agent,LOG) to input
            queue of next node;
      }else{
         write(spID,agent,LOG) to input
         queue of current node;
      }
   }
   commit-transaction();
}
```

Figure 5a: Start of the optimized rollback algorithm

```
begin-transaction();
read (spID, agent, LOG) from
   node input queue;
if (last log entry is savepoint)
   LOG.pop(); // remove savepoint
EOSEntry = LOG.pop() // read end-of-step
entry=LOG.pop(); // next entry
if (EOSEntry indicates mixed compensat.
   entry){ // execution on agent node
   while (entry<>begin-of-step){
      execute entry;
      entry = LOG.pop();
   }
}else{ // group operation entries
   resourceNode = EOSEntry.execNode();
   ACEList = {}; // empty list of agent
                 // compensation entries
   RCEList = {}; // empty list of resource
                 // compensation entries
   while (entry<>begin-of-step){
      if (entry is agent compensation
         entry){
         ACEList.add(entry);
      }else{ // res. compensation entry
         RCEList.add(entry);
      }
      entry = LOG.pop();
   }
   send (TransactionID, RCEList) to
      resourceNode; // only if not empty
   execute the entries in ACEList in
      the order they appear in the list;
   wait for ACK from resourceNode;
}
if savepoint spID reached{
   restore strongly reversible
      objects;
   start next step transaction;
}else{
   if (next EOS entry indicates
      mixed compensation entry){
      write(spID,agent,LOG) to input
         queue of next node;
   }else{
      write(spID,agent,LOG) to input
      queue of current node;
   }
}
commit-transaction();
```

Figure 5b: Optimized rollback algorithm executed on each node

can be used. An itinerary, which describes a (sub-)task, is a set which contains itinerary entries and specifies in which order those itinerary entries have to be executed. An entry of an itinerary is either a nested sub-itinerary or a *step entry.* A step entry in an itinerary is a tuple (`meth()`/`loc`) which describes that the agent has to execute the step specified

by the method `meth()` on the node specified by `loc`. The term "execute an entry *e*" describes the execution of the step if the entry *e* is a step entry or the (recursive) execution of all entries contained in *e* if *e* is a sub-itinerary. The order defined between the entries of a (sub-)itinerary may be partial, allowing the system to choose which entry to execute as the next entry. It is even possible to allow the application to specify entries which have to be executed alternatively, or to define complex rules which specify under which conditions an entry has to be executed. An approach using pre-conditions for the entries to decide whether and when an entry can be executed is presented in [14].

An example of an itinerary (without the definition of the order between entries) is given in Figure 6. This itinerary *I* contains three sub-itineraries SI_1, SI_2 and SI_3. The sub-itineraries SI_1 and SI_2 contain only step entries, SI_3 contains, besides the step entry s_6, two additional sub-itineraries SI_4 and SI_5.

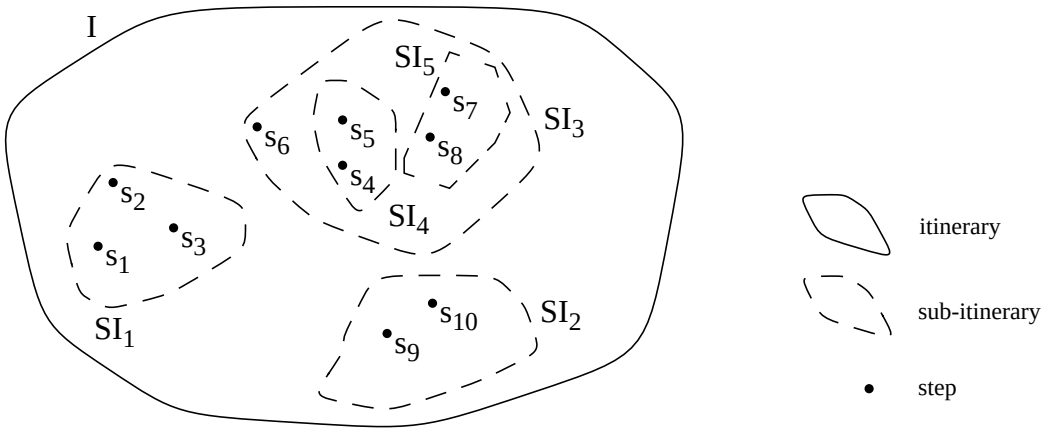


Figure 6: Sample Itinerary

To illustrate the integration of rollback and itineraries, we assume the following scenario: an agent begins its execution with sub-itinerary SI_3. It executes the step entry s_6 and continues with sub-itinerary SI_4 by executing step entries s_5 and s_4 (in this order). If it decides to roll back during the execution of s_4, it can either roll back only sub-itinerary SI_4 (by aborting step transaction s_4 and compensating s_5) or it can also roll back the enclosing sub-itinerary SI_3 (by additionally compensating s_6).

In this scenario, only two savepoint entries are necessary - one containing the state of the strongly reversible objects before the execution of SI_3 starts and one before the execution of SI_4 starts. Those savepoints can be written automatically by the system. If we change our scenario slightly and omit the execution of entry s_6, i.e. the agent begins with the execution of SI_3 and immediately continues with the execution of SI_4, only one agent savepoint is really necessary. In this case, the rollback mechanism has to read the savepoint entry, but must not delete it from the log as long as only SI_4 is rolled back (because the savepoint entry is necessary for a possible rollback of SI_3). To make this visible in the log, a special savepoint entry may be inserted (without data for the strongly reversible objects) for the "savepoint" written for SI_4. Due to reasons explained later in this section, savepoints are never needed for the main itinerary *I*.

On closer examination it shows, that it is not necessary to have one savepoint for each sub-itinerary in the log but only one savepoint for the sub-itinerary currently executed and one for each sub-itinerary which (directly or indirectly) contains the currently executed itinerary (i.e. for each sub-itinerary currently being executed). If, in our first scenario from above (agent begins with SI_3 by executing s_6 and continuing with SI_4), the sub-itinerary SI_4 has been executed and the agent now executes the sub-itinerary SI_5, it can either rollback SI_5 or it can rollback SI_3, but the rollback of SI_5 and SI_4 without the compensation of s_6 is not possible (per definition). The savepoint entry written for SI_4 is no longer needed. Therefore, the savepoint written for a sub-itinerary (but not the operation entries) can be removed from the rollback log as soon as the sub-itinerary has been executed completely. However, this may be a non-trivial task if transition logging is used for the strongly reversible objects.

Further reductions of the size of the rollback log may be reached by discarding rollback information which is no longer needed. Discarding rollback information is a far-reaching event in the "life" of an agent which can only be done, if it is sure that the results achieved do not have to be rolled back. Achieving such results very likely corresponds with the completion of one of the agents main sub-tasks, i.e. with the completion of a sub-itinerary directly contained in the main itinerary. Therefore, sub-itineraries directly contained in the main itinerary have the additional semantics, that, upon their completion, all information contained in the rollback log is deleted. To provide a clear semantics, no step entries are allowed in the main itinerary. So, if the main itinerary of an agent contains *n* sub-itineraries, the execution of the agent is split into *n* parts, where each of this parts cannot be rolled back as soon as that part is completed. It is important to note, that an abort of the agent by performing a complete rollback of the agent is possible only during the execution of the "first" sub-itinerary of the main itinerary.

Discussion: The itinerary concept introduced in this subsection provides a structured way to automatically constitute agent savepoints and provides the possibility to remove an agent savepoint associated with a sub-itinerary from the log as soon as the sub-itinerary is finished. This reduces the log size without losing the possibility to roll back the agent. Additionally, discarding the complete rollback log if a point in the agent execution is reached where it is sure that the agent will not rolled back beyond this point reduces the log size considerably.

5 Related Work

In the field of mobile agents, only few research groups have considered aspects of transaction management and fault-tolerance so far. Most of those groups provide mechanisms to increase the fault-tolerance of mobile agent execution [3][7][12][17] but offer no mechanisms to (partially) roll back the agent's execution.

In [13], an agent-based transaction model is presented. Similar to our model, the use of compensation to roll back the effects of already committed transactions is proposed. However, this paper purely concentrates on modelling transactional aspects, protocols or algorithms are not given.

In addition, there has been related work in the field of transaction processing. Our model is based on [4], where *sagas* as a transaction model for long-living activities are introduced. In a saga, a long-living activity is partitioned into several steps. Each step is executed within an ACID transaction. The commit of a step automatically begins the next step transaction. For each step, a compensation step has to specified. The runtime system of a saga guarantees, that eventually, either all steps of the saga are committed, or, if the saga has to be aborted, for all committed steps the compensation step has been executed. To allow backward/forward recovery, savepoints of the program state of the transaction program can be taken. However, savepoints are only used when a transaction aborts e.g. due to a deadlock or a system crash. A partial rollback initiated by the transaction program is not supported. Additionally, the use of a savepoint to restore the complete execution state of the saga prevents the usage of sagas in applications where the execution state of the saga also has to be compensated. Extensions of sagas like *nested sagas* and *non vital sub-sagas* as presented in [5] can be realized in our model by using flexible itineraries as described in [14] and Section 4.4.2.

The ConTract model [10] certainly comes closest to our approach. It also aims at the exactly-once execution of a task and similar to our approach allows the partial rollback of an execution of a task using compensation. The main difference to our approach lies in the underlying system mechanisms. A ConTract, which is defined by a script is given to a ConTract manager, controlling the entire execution of the ConTract. In other words, ConTract scripts are not mobile so far.

6 Conclusions and Future Work

We have investigated how the partial rollback of mobile agents which are executed using one of the exactly-once protocols presented in [11] can be realized. Due to the fact that these protocols use transactions to realize the exactly-once property of mobile agents, compensation is necessary to partially roll back an agent's execution. The classification of types of compensations showed, that - besides the operations performed on resources - the private state of the agent also has to be rolled back using compensating operations. Introducing two different types of private agent data allowed to roll back the private agent data belonging to one type using an image of the data while the data belonging to the other type has to be compensated using compensating operations. The rollback mechanism presented ensures,

that the rollback is eventually executed even in the presence of (non-lasting) node and network crashes. By introducing different types of compensating operations and integrating the rollback with an itinerary concept, performance optimizations of the rollback mechanism were presented. Currently, the protocol is under implementation in the Mole system [9] and will be evaluated in terms of performance.

Future work will concentrate on further performance optimizations of the rollback mechanism and a fault-tolerant rollback mechanism. Furthermore, an enhanced agent execution model supporting exactly-once executions comprising more than one agent will be investigated.

7 References

[1] J. Baumann, F. Hohl, K. Rothermel, M. Schwehm, and M. Strasser, “Mole 3.0: A Middleware for Java-Based Mobile Software Agents”, *Proc. Middleware'98*, Springer Verlag London, 1998

[2] D. Chaum, “Security Without Identification: Transaction Systems to Make Big Brother Obsolete”, C*ommunications of the ACM*, 28(10), October 1985, p. 1030-1044

[3] M. Dalmeijer, E. Rietjens, M. Soede, D. Hammer, and A. Aerts, “A Reliable Mobile Agent Architecture”, *Proc. 1st IEEE Int. Symp. on Object-oriented Real-time distributed Computing (ISORC’98)*, 1998, p. 64-72

[4] H. Garcia-Molina, and K. Salem, “SAGAS”, *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1988, pp. 249-259

[5] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem, “Modeling Long-Running Activities as Nested Sagas”, *Bulletin of the IEEE Technical Commitee on Data Engineering*, 14(1), 1991, pp. 14-18

[6] T. Härder, and A. Reuter, “Principles of Transaction-Oriented Database Recovery”, *ACM Computing Surveys*, 15(4), December 1983, pp. 287-317

[7] D. Johansen, R. van Renesse, and F. Schneider, “Operating System Support for Mobile Agents”, *Proc. 5th IEEE Workshop on Hot Topics in Operating Systems*, 1995, pp. 42-45

[8] H. Korth, E. Levy, and A. Silberschatz, “A Formal Approach to Recovery by Compensating Transactions”, *Proc. 16th Very Large Data Bases Conf.*, Brisbane, Australia, 1990, pp. 95-106

[9] Project Mole, http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole.html

[10] A. Reuter, K. Schneider, and F. Schwenkreis, “ConTracts Revisited”, S. Jajodia and L. Kerschberg (eds.), *Advanced Transaction Models and Architectures (ATMA)*, Kluwer Academic Publ., Boston, 1997, pp. 127-151

[11] K. Rothermel and M. Strasser, “A Fault-Tolerant Protocol for Providing the Exactly-Once Property of Mobile Agents”, *Proc. 17th IEEE Symp. on Reliable Distributes Systems*, West Lafayette, IN, October 1998, pp. 100-108

[12] F. Schneider, “Towards Fault-Tolerant and Secure Agentry”, *Proc. 11th Int. Workshop on Distributed Algorithms*, 1997

[13] F. Morais de Assis Silva, and S. Krause, “A Distributed Transaction Model Based on Mobile Agents”, K. Rothermel, and R. Popescu-Zeletin (eds.), *Mobile Agents, Proc. 1st Int. Workshop (MA’97)*, LNCS 1219, Springer, Berlin, 1997, pp. 198-209

[14] M. Strasser, and K. Rothermel, “Reliability Concepts for Mobile Agents”, *International Journal of Cooperative Information Systems (IJCIS)*, 7(4), World Scientific, 1998, pp. 355-382

[15] M. Strasser, K. Rothermel, and C. Maihöfer, “Providing Reliable Agents for Electronic Commerce”, W. Lamersdorf, M. Merz (eds.), *Trends in Distributed Systems for Electronic Commerce (TREC'98)*, LNCS 1402, Springer-Verlag, 1998, pp. 241-253

[16] M. Strasser, and M. Schwehm, “A Performance Model for Mobile Agent Systems”, H. Arabnia (ed.), *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, Vol II, CSREA, 1997, pp. 1132-1140

[17] H. Vogler, T. Kunkelmann, and M. Moschgath, “An Approach for Mobile Agent Security and Fault Tolarance Using Distributed Transactions”, Proc. 1997 Int. Conf. on Parallel and Distributed Systems (ICPADS’97), 1997, pp. 268-274