RZ 3386 (# 93432) 12/10/01
Computer Science 17 pages

Research Report

TranSuMA: Non-Blocking Transaction Support for Mobile Agent Execution

Stefan Pleisch

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland
spl@zurich.ibm.com

also with:
Operating Systems Laboratory
EPFL
Lausanne
Switzerland

André Schiper

Distributed Systems Laboratory
Swiss Federal Inst. of Technology (EPFL)
1015 Lausanne
Switzerland
Andre.Schiper@epfl.ch

TranSuMA: Non-Blocking Transaction Support for Mobile Agent Execution

Stefan Pleisch

IBM Research, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland

also with: Operating Systems Laboratory, EPFL, Lausanne, Switzerland

André Schiper

Distributed Systems Laboratory, Swiss Federal Inst. of Technology (EPFL), 1015 Lausanne, Switzerland

Abstract

In the context of e-commerce, execution atomicity is an important property for mobile agents. A mobile agent executes atomically, if either all its operations succeed, or none at all. Assume, for instance, a mobile agent that books a flight and rents a hotel room at the destination. The hotel room is of no use if no flight is available. Consequently, either both operations need to succeed or none at all. Transactional mobile agents ensure execution atomicity on their entire execution. This requires to solve an instance of the atomic commitment problem. However, in the context of transactional mobile agent execution it is important that failures (e.g., of machines or agents) do not lead to blocking of the mobile agent execution. Blocking occurs, if the mobile agent execution cannot proceed because of a single failure. In this paper, we specify non-blocking atomic commitment in the context of mobile agent execution. We then show how transactional mobile agent execution can be built on top of earlier work on fault-tolerant mobile agent execution. Our implementation is the first implementation of non-blocking transactional mobile agents. Its performance evaluation shows that the overhead introduced by the transaction mechanisms is reasonable.

1 Introduction

Mobile agents are computer programs that act autonomously on behalf of a user and travel through a network of heterogeneous machines. So far, only few real applications rely on mobile agent technology. We believe that the lack for transaction support for mobile agents is one reason for this. Assume, for instance, an agent[1] whose task is to buy an airline ticket, book a hotel room, and rent a car at the flight destination. The agent owner, i.e., the person or application that has created the agent, naturally wants all three operations to succeed or none at all. Clearly, the rental car at the destination is of no use if no flight to the destination is available. On the other hand, the airline ticket may be useless if no rental car is available. The mobile agent's operations thus need to execute *atomically*. Execution atomicity needs to be ensured also in the face of failures to hardware or software components. Indeed, any component in a system is subject to failures. In this context, we distinguish between blocking and non-blocking solutions for transactional mobile agents, i.e., mobile agents, that execute as a transaction. Blocking occurs, if the failure of a single component prevents the agent from continuing its execution. In contrast, the non-blocking property ensures that the mobile agent execution can make progress any time, despite of failures. While other approaches [2, 21] block if the place running the mobile agent fails, the approach presented in this paper is non-blocking. A non-blocking transactional mobile agent execution has the important advantage, that it can make progress despite failures. In a blocking agent execution, progress is only possible when the failed component has recovered. Until then, the acquired locks cannot be freed. As no other transactional mobile agents can acquire the lock, overall system throughput is dramatically reduced.

Similar to [4], our work also reuses earlier work on fault-tolerant mobile agent execution to prevent blocking. In contrary to [4], we encompass both transaction support and fault tolerance in a common model of nested transactions. This allows us to specify non-blocking atomic commitment in the context of transactional mobile agents. Moreover, our approach also supports non-compensatable transactions (i.e., transactions that cannot be undone any more once they are committed). In this respect, it is thus more general than the approach in [4], which only supports compensatable transactions.

We have implemented the proposed approach and present preliminary evaluation results. To our knowledge, our implementation is the first to provide non-blocking transactional mobile agent execution. Our evaluation shows the important result that the overhead introduced by the commitment mechanisms is reasonable.

The rest of the paper is structured as follows: Section 2 presents our model. In Section 3, we present an overview on the atomicity property of mobile agent execution and specify the non-blocking atomic commitment problem for transactional mobile agents in Section 4. In Section 5, we discuss our approach for transactional mobile agent execution and show the implementation and preliminary performance results in Section 6. Section 7 summarizes other work published in this field and relates it to our contribution. Finally, Section 8 concludes the paper and indicates potential future work.

2 Model

2.1 Mobile Agent

A mobile agent executes on a sequence of machines. On each machine i, a place p_i $(0 \leq i \leq n)$ provides the logical execution environment for the agent. Executing the agent at a place is called a stage S_i of the agent execution. We call the places where the first and last stages of an agent execute (i.e., p_0 and p_n) the agent *source* and *destination*, respectively. The sequence of places between agent source and destination is called the *itinerary* of the mobile agent. Whereas a *static* itinerary is defined in its entirety at the agent source and does not change during the agent execution, a *dynamic* itinerary is subject to modifications by the agent itself.

Logically, a mobile agent executes in a sequence of stage actions (see Figure 1). Each stage action sa_i consists of potentially multiple operations $op_0, op_1, \ldots, op_l, \ldots$. Agent a_i $(0 \leq i \leq n)$ at the corresponding stage S_i represents the agent a that has executed the stage actions on places p_j $(j < i)$ and is about to

[1]In the following, the term *agent* denotes a *mobile agent* unless explicitly stated otherwise.

execute on place p_i. The execution of a_i results in a new internal state of the agent as well as potentially a new state of the place (if the operations of the agent have side effects, i.e., are non-idempotent). We denote the resulting agent a_{i+1}. Place p_{i+1} forwards a_{i+1} to p_{i+1} (for $i < n$).

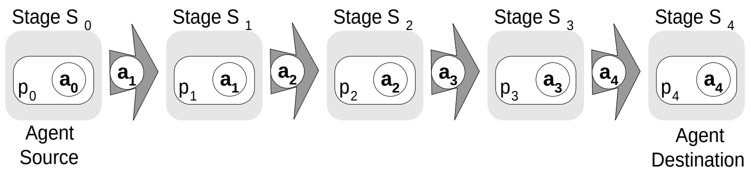


Figure 1: Logical agent execution.

2.2 Failures

2.2.1 Infrastructure Failures

Machines, places, or agents fail by crashing (i.e., we exclude malicious processes) and can recover later. A component that has failed but not yet recovered is called *down*, whereas it is *up* otherwise. In this paper, we focus on crash failures; malicious failures (i.e., Byzantine failures) are not discussed. A failing place causes the failure of all agents running on it. Similarly, a failing machine causes all places and agents on this machine to fail as well. Failures of machines, places, and agents are called *infrastructure failures*.

In an asynchronous system such as the Internet, no bounds on transmission delays of messages nor on relative process speeds exist. In such a system, reliable failure detection is impossible [7]. Consequently, p (a machine, place, or agent) can erroneously suspect q (another machine, place, or agent), although q has not failed.

2.2.2 Semantic Failures

A *semantic failure* is different from an infrastructure failure in the sense that neither machine, place, nor agent initiating the request crash. Rather, it occurs when a requested service is not delivered because of the application logic or because the process providing this requested service has failed. For instance, a request for an airline ticket is declined if no seats are available on a particular flight. Nevertheless in this case, the agent's operation, i.e., the request for a ticket, executes in its entirety.

3 Overview

3.1 The Problem of Execution Atomicity

An *atomic* mobile agent execution ensures that either all stage operations succeed or none at all. Assume, for instance, a mobile agent that books a flight to New York, books a hotel room there, and rents a car. Clearly, the use of the hotel room and the car in New York is limited if no flight to New York is available any more. On the other hand, the flight is not of great use if neither a hotel room nor a rental car is available. This example illustrates that either all three operations (i.e., flight ticket purchase, hotel room booking, and car rental) need to succeed or none at all, i.e., the operations have to be executed *atomically*. Execution atomicity ensures that all operations execute as an atomic action, i.e., either in their entirety or none at all. Note that infrastructure and semantic failures may lead to a violation of the atomicity. In the following, we focus on semantic failures only; infrastructure failures are discussed in Section 3.3.

3.1.1 Traditional Distributed Transactional Systems: Background

In traditional distributed transactional systems, *atomic commitment* protocols such as 2PC and 3PC [5, 10] address the issues of execution atomicity. In the 2PC, for instance, a designated coordinator (also called transaction manager) queries all the participants in the execution of the distributed transaction (called the resource managers) on the state of the corresponding operations. The participants return either a YES-VOTE or a NO-VOTE, depending on whether their operations have succeeded or not. If all operations have been successful, i.e., all returned votes are YES-VOTEs, the coordinator decides COMMIT, otherwise ABORT. The decision is propagated to the participants, which then either commit or abort their operations.

3.1.2 Transactional Mobile Agent Execution

The operations of the participants of traditional distributed transactions can run in parallel. In a transactional mobile agent execution T_a, the operations of mobile agent a are executed *sequentially* in a sequence of stages S_i $(0 \leq i \leq n)$. The execution of a at stage S_i depends on the outcome of the previous stage S_{i-1} and implies that a has successfully executed on all previous stages S_j $(j < i)$, i.e., all a_j have issued a YES-VOTE. Consequently, the vote of a_i unilaterally determines whether the agent execution is continued (in case of a YES-VOTE) or aborted (NO-VOTE); the agent a_i only returns a NO-VOTE if a semantic failure has occurred[2]. Actually, the transaction T_a spans only over stages $S_1, \ldots, S_{n-1}$: the agent source p_0 and destination p_n are executing the agent outside of the transaction context T_a. On these places, the interaction with the agent owner (i.e., initialization of the agent and presentation of the results) takes place and a transaction context is not needed. Moreover, mobile users are often disconnected from the network and hence stage S_n may be temporarily unreachable. Consequently, executing *sa*$_n$ within the context of T_a may lead to blocking of the mobile agent execution until the mobile user reconnects to the network. While T_a is blocked, it maintains its locks on data items, thus reducing overall system throughput. Terminating transaction T_a already at stage S_{n-1} prevents blocking due to disconnections of mobile devices. The agent a_n is then kept at place p_{n-1} until p_n reconnects and is able to collect the result. At stage S_{n-1}, the agent a_{n-1} unilaterally decides either COMMIT (if the execution of a_{n-1} has succeeded), or ABORT (in case of a semantic failure).

In the case of a dynamic itinerary, any place p_i may become the final place of T_a (i.e., p_{n-1}), based on the outcome of the execution of a_i. In this case, the vote of a_i immediately becomes the outcome of the transaction T_a. In other words, a_i unilaterally decides the outcome of the transaction. This is different from traditional distributed transactions. Moreover, in traditional distributed transactions, a participant can only unilaterally abort a transaction, namely by issuing a NO-VOTE. In contrast, a_i can also unilaterally decide to continue T_a, in addition to the abort decision.

To summarize, the outcome of T_a at S_i solely depends on the result of the stage operations of a_i on S_i and on the value of i:

- At $\{S_i | i < n-1\}$: the agent a_i casts either a YES-VOTE or NO-VOTE. A NO-VOTE immediately results in an abort of the transactional mobile agent execution T_a (see Figure 2). It is cast when the stage action operations semantically fail. Successful operations lead to a YES-VOTE and allow the agent to proceed with the transaction execution. Only the ABORT decision is communicated to all the participants p_j $(0 < j < i)$.

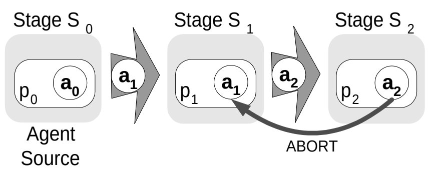


Figure 2: An agent execution that has aborted at stage S_2.

- At S_{n-1}: the agent a_{n-1} decides either ABORT or COMMIT, depending on whether the operations at stage S_{n-1} have failed or succeeded, respectively. A successful execution of the operations at S_{n-1} implies that all operations of the agent a have successfully executed (i.e., voted YES-VOTE) and the transaction is thus ready to commit. Figure 3 illustrates a successful transactional mobile agent execution, where place p_3 decides COMMIT.

[2]Remember that we are only addressing semantic failures at this point.

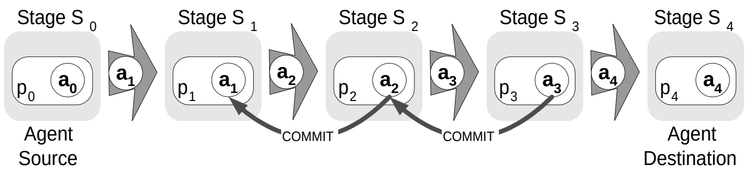


Figure 3: A committing agent execution.

3.2 Generalization of the Itinerary

3.2.1 Itinerary Choices

A transactional mobile agent execution permits itinerary choices. In particular, the agent owner may specify that the agent should sequentially visit various car rental companies such as Hertz or Avis. As soon as it receives the desired car from any one of them, the agent terminates. Failed requests for a car rental are ignored and the execution on this place locally aborted. The commitment only spans places that have successfully executed the service requests. Figure 4 illustrates the example, where the agent first books a flight from Swissair and then attempts to rent a car from Hertz. As Hertz does not have any more rental cars available, agent a (i.e., a_3) moves to the Avis server. The stage actions of a_2 on p_2 can be locally aborted without aborting the entire transactional mobile agent execution. The outcome of the mobile agent execution, i.e., the decision `COMMIT` or `ABORT`, will not be sent to p_2.

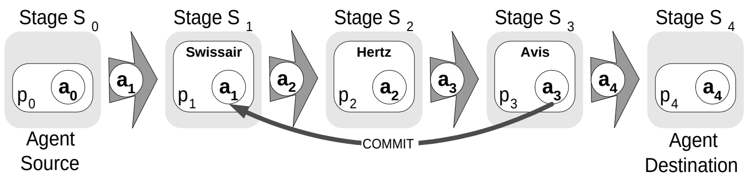


Figure 4: A committing agent execution with choices. Hertz does not have a rental car available and thus is not part of the global commitment.

3.2.2 Generalization To Non-Linear Itineraries

So far, we have only considered mobile agent executions with linear itineraries (see (a) in Figure 5). An agent, however, can spawn other agents, so-called child agents. Child agents lead to non-linear itineraries: itineraries that terminate in a single place (Figure 5 (b)), and itineraries that terminate in several endpoints (Figure 5 (c)). Case (b) contains itineraries where all parent and child agents meet again on a common place (e.g., at p_4 in Figure 5 (b)). Assume, for instance, that transaction T_b acquires clothes, while T'_b buys books. Both transactions can run in parallel, and their results are collected at place p_4. In contrast, parent and child agents finish the execution at different agent destinations in (c) (e.g., at p_5, p'_4, and p''_4). An example for (c) is a transaction that reconfigures routers in different subnets. At the occurrence of a subnet, a new child transaction is spawn recursively.

Compared to (a), atomicity is more difficult to ensure in cases (b) and (c). Assume, for instance, that child transaction T'_b aborts on p'_3. Transaction T_b continues the execution until is reaches p_4 and there waits for T'_b, which never arrives. The simplest approach is to wait for a certain time and then abort. However, this may lead to a prematurely unnecessary abort if T'_b is just slow but has not aborted. Another approach is to abort the transaction T'_b immediately, but still forward the agent to p_4, where it notifies T_b of the abort. T_b thus always waits for the agent to arrive. This has the drawback that additional communications are required, but the transaction never prematurely aborts.

Case (c) is handled by applying the following transformation: instead of terminating the child agents

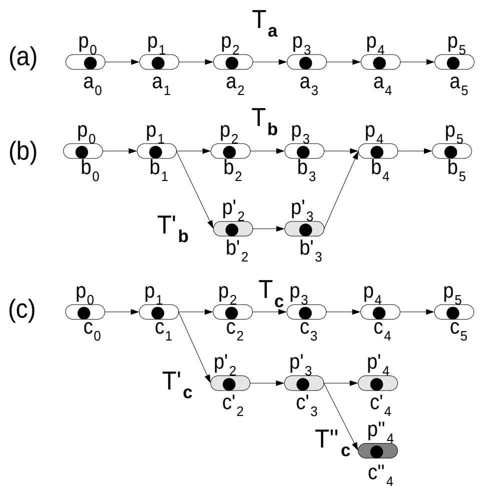

(a)
(b)
(c)
T_a
T_b
T'_b
T_c
T'_c
T''_c
p_0 p_1 p_2 p_3 p_4 p_5
a_0 a_1 a_2 a_3 a_4 a_5
b_0 b_1 b_2 b_3 b_4 b_5
p'_2 p'_3
b'_2 b'_3
c_0 c_1 c_2 c_3 c_4 c_5
p'_2 p'_3 p'_4
c'_2 c'_3 c'_4
p''_4
c''_4

Figure 5: Classification of mobile agent executions.

on p'_4 and p''_4, they report back to their parent agent on place p_4. This transformation converts case (c) into case (b) and allows to reuse the approach discussed before.

For simplicity, we only address linear itineraries in the rest of this paper. However, all presented concepts can easily be extended to itineraries of classes (b) and (c).

3.3 The Problem of Failures

So far, we have not considered infrastructure failures. Any hardware and software component in a computing environment is potentially subject to failures. We assume the failure model presented in [16], i.e. either an agent, a place, or a machine may fail by crashing. A crashing machine leads to the crash of all places and all agents running on it, whereas a failure of a place also crashes the agents running on this place (Section 2.2.1).

Failing components in the system may lead to blocking or to a violation of the atomic execution of the transactional mobile agent. Figure 6 illustrates a crash at stage S_3. In an asynchronous system, where no bounds on communication delays nor on relative processor speed exist, p_0, p_1, and p_2 are left with the uncertainty of whether p_3 has actually failed or is just slow [8]. In addition, it is impossible for p_0, p_1, and p_2 to detect the exact point where p_3 failed in its execution. More specifically, they cannot detect whether p_3 has succeeded in forwarding the agent to the next stage or not. Assume, for instance, that the agent a_i (i.e., a_3 in Figure 6) issues a `YES-VOTE` but then crashes. If S_i has not succeeded to forward the agent to S_{i+1}, the agent execution is blocked. During this time, all locks acquired by transaction T_a at the previous places p_j $(j < i)$ remain with T_a and another transaction T_b has to wait. This dramatically reduces overall system throughput.

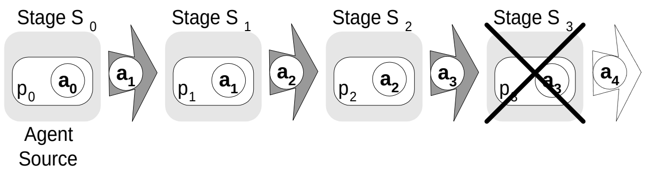


Figure 6: An agent execution crashes at stage S_3

To prevent blocking, another place such as p_{i-1} could monitor place p_i. If it detects the failure of p_i, it could then issue a `NO-VOTE`, which causes the transaction to abort. However, unreliable failure detection potentially leads to a violation of the atomicity property. Indeed, assume that p_j detects the failure of a_i. Place p_j thus assumes the responsibility for the decision and decides to abort transaction T_a. However, because of unreliable failure detection, p_j may erroneously suspect p_i. Actually, even if p_i has failed, it may have succeeded in forwarding the agent to p_{i+1}, resulting in potentially conflicting decisions on the outcome of the transaction. Indeed, while p_j decides to abort the transaction, p_{i+1} may decide `COMMIT` if $a_{i+1} = a_{n-1}$, or cast a `YES-VOTE` otherwise. This conflicting outcome clearly violates the atomic execution property of the transaction's operations, as certain operations are aborted, whereas others are committed or may be committed later.

The approach we advocate in this paper uses replication to prevent blocking of the transactional mobile agent execution. At any time, the places know that the agent is still progressing and thus it is worth to wait for the result. This alleviates the need to monitor the execution and prevents potentially conflicting outcomes to the atomic execution of the transactional mobile agents.

4 Specification

In this section, we specify the properties of the transactional mobile agent execution T_a associated with a mobile agent a. The entire execution T_a is specified in terms of the ACID properties [10]:

- (*Atomicity*) The stage executions of T_a are executed atomically, i.e., all of them or none are executed.
- (*Consistency*) A correct execution of T_a on a consistent state of the system (encompassing the places, the services running on them, and the agents) must result in another consistent system state.
- (*Isolation*) Updates of a stage execution of T_a on a place p_i are not visible to another transactional mobile agent T_b until T_a has committed in its entirety.
- (*Durability*) Committed changes by T_a are reflected in the system and are not lost any more.

Specifying the transactional mobile agent T_a in terms of the ACID properties implies that the sequence of stage actions $sa_1, \ldots, sa_{n-1}$ is executed as a transaction. Every stage action is itself composed of a set of operations $op_0, \ldots, op_l, \ldots$, which have to run as a transaction as well. Consequently, T_a can be modeled as *nested transactions* [14]. A nested transaction is a transaction that is (recursively) decomposed into *subtransactions*. Every subtransaction forms a logically related subtask. A successful subtransaction only becomes permanent, i.e., commits, if all its parent transactions commit as well. In contrast, a parent transaction can commit (provided that its parent transactions all commit) although some of its subtransactions may have failed. In a transactional mobile agent execution, the top-level transaction (i.e., the transaction that has no parent) corresponds to the entire mobile agent execution. The first level of subtransactions is composed of stage actions sa_i. Note that subtransactions may be aborted, but parent transactions still commit. Indeed, if a service request fails on one place, the subtransaction sa_i can be aborted and retried at another place, without aborting the top-level transaction (see Section 3.2.1).

In the context of transactional mobile agents, consistency is ensured by the application composed of the mobile agent and the services running on the places. Isolation is discussed in Section 5.3. The properties we are mainly concerned with are atomicity and durability[3]. To ensure the atomicity property, all the places participating in the execution of the transactional mobile agent T_a need to solve an instance of the atomic commitment (AC) problem [5]. Informally, T_a commits if all stage actions sa_i $(0 < i < n-1)$ have executed successfully. However, blocking occurs if a place p_i fails while executing the stage action sa_i of a mobile agent a. Progress of the transactional mobile agent execution T_a is interrupted until p_i recovers. Deciding `ABORT` because of infrastructure failures is not admissible, as this potentially causes a violation of the atomicity property. Such a violation occurs if the agent has already moved to the next stage while the execution at the previous stages is aborted. Clearly, this could result in conflicting outcomes of

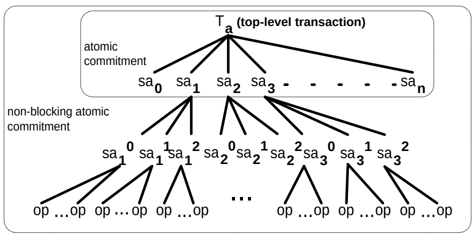
[3]Actually, atomicity and durability are tightly coupled. Assume a transaction that executes `write[x]` and `write[y]`. Assume further that the transaction commits, but a crash causes the modification to y to be lost, whereas the operation to x is made permanent. It is difficult to say whether atomicity or durability has been violated.

the transaction (see Section 3.3). Consequently, an infrastructure failure blocks the transactional mobile agent execution. However, blocking is undesirable, as it dramatically limits overall system throughput (see Section 3.3).

In contrary, *non-blocking* atomic commitment (NB-AC) does not block if a machine, place, or agent fails. Rather, progress of the transactional mobile agent execution is ensured by potentially executing replicas of the mobile agent on multiple places. We give the specification of NB-AC for transactional mobile agents in Section 4.1 [4].

4.1 Non-Blocking Atomic Commitment Problem for Transactional Mobile Agents

Non-blocking adds another level of subtransactions to a transactional mobile agent execution. To prevent blocking, the agent at stage S_i is not executed on one place, but replicas of the agent are potentially executed on multiple places p_i^j. Consequently, subtransaction sa_i, in turn, can be modeled by yet another level of subtransactions sa_i^j, which correspond to the agent replicas $a_i^0, \ldots, a_i^m$ running on places $p_i^0, \ldots, p_i^m$ and executing the set of operations $op_0, \ldots, op_l, \ldots$ (see Figure 7). Of the subtransactions sa_i^j at stage S_i, only one, called sa_i^{prim}, is allowed to commit (if all its parent transactions commit): all others have to abort. This way it is ensured that the stage action sa_i is not executed multiple times. We call the place that has executed the stage action sa_i^{prim} the *primary* and denote it p_i^{prim}. We further define $\mathcal{M}_i$ as the set of places $p_i^0, \ldots, p_i^m$.



T a (top-level transaction)
atomic commitment
sa0 sa1 sa2 sa3 - - - - - san
non-blocking atomic commitment
sa1 0 sa1 1 sa1 2 sa2 0 sa2 1 sa2 2 sa3 0 sa3 1 sa3 2
op ...op op ...op op ...op ... op ...op op ...op op ...op

Figure 7: Scope of NB-AC specification. Compared to AC, NB-AC adds an additional layer of subtransactions sa_i^j, that execute the operations $op_0 \ldots op_l, \ldots$. In AC, subtransaction sa_i directly executes these operations without subtransactions sa_i^j.

The non-blocking atomic commitment (NB-AC) problem for transactional mobile agents consists of two levels of agreement problems: (1) the *stage agreement problem*, and (2) the *global agreement problem*. Agreement problem (2) ensures atomicity in the top-level transaction. On the other hand, (1) specifies the agreement problem among the places that execute the stage actions sa_i^j $(0 \leq j \leq m)$, where they decide on which subtransaction may potentially commit. We begin by specifying the stage agreement problem (1) at each stage S_i:

Stage agreement problem:

- (*Uniform agreement*) No two places $p_i^j \in \mathcal{M}_i$ at stage S_i decide on a different primary p_i^{prim}.
- (*Validity*) The decision value p_i^{prim} is in the set $\mathcal{M}_i$ and p_i^{prim} has executed stage action sa_i (more specifically, sa_i^{prim}).
- (*Uniform integrity*) Every place p_i^j of stage S_i decides at most once.
- (*Termination*) Every correct place p_i^j of stage S_i eventually decides.

[4]A specification for NB-AC in the context of traditional distributed systems (i.e., without mobile agents) is given for instance in [11].

The decision on p_i^{prim} in the stage level agreement causes all other places $p_i^j \neq p_i^{prim}$ to abort the subtransactions *sa*$_i^j$. Consequently, the decision on p_i^{prim} is implicitly also a decision ABORT for all places $p_i^j \neq p_i^{prim}$. However, the agreement only occurs on the decision about the primary, not on ABORT or COMMIT. Indeed, subtransactions *sa*$_i^j \neq$ *sa*$_i^{prim}$ abort, while *sa*$_i^{prim}$ only aborts if its parent transaction aborts. However, this decision is again part of another agreement problem that is to be solved and is only taken by a_{n-1} at the end of the agent execution and specified in the global agreement problem as follows:

Global agreement problem:

- (*Uniform agreement*) No two primaries p_i^{prim} and p_k^{prim} participating in the execution of T_a decide differently.
- (*Uniform validity*) Primary p_i^{prim} $(0 < i \leq n-1)$ can decide ABORT. Primary p_{n-1}^{prim} decides either ABORT or COMMIT. Decision COMMIT is a consequence of successfully executing the agent up to stage S_{n-1} and successfully executing *sa*$_{n-1}$ at stage S_{n-1}. In all the other cases the decision is ABORT. Place $p_i^j \neq p_i^{prim}$ always decides ABORT (see stage agreement problem).
- (*Uniform Integrity*) Every place decides at most once.
- (*Termination*) Every correct place eventually decides.

It should be noted that an infrastructure failure does not allow to immediately decide ABORT. Rather, infrastructure failures cause the agent to execute on another place at the same stage. If this place provides the same service, the agent execution can proceed. Otherwise, a semantic failure occurs that, contrary to infrastructure failures, immediately results in an ABORT decision. Assume, for instance, that the agent a_i at stage S_i is entrusted with buying an airline ticket from Zurich to New York. Assume further that it executes on place p_i^j, that sells such tickets. A failure of p_i^j does not immediately abort subtransaction *sa*$_i$. Rather, a_i can be executed on another place p_i^k $(k \neq j)$ at stage S_i. If p_i^k provides the same service as p_i^j, i.e., also sells the same airline tickets, then *sa*$_i$ succeeds, T_a can proceed and no reason for an ABORT is given (see Section 3.2.1). In other words, infrastructure failures are masked by the redundancy of the agent at a stage (see the specification of the stage agreement problem).

5 Non-Blocking Transactional Mobile Agents

In this section, we show how earlier work in fault-tolerant mobile agent execution [16] can help to provide non-blocking transactional mobile agents. Indeed, fault tolerance in the context of mobile agents prevents that the execution of a mobile agent blocks because of the failure of a single component (e.g., an agent, place, or machine). Hence, it solves a problem similar to the stage agreement problem of NB-AC (see Section 4.1). However, fault-tolerant mobile agent execution generally only addresses infrastructure failures: semantic failures (see Section 3.1) are not handled. In other words, it does not solve the global agreement problem of NB-AC. Revisiting the example in Section 3.1, fault-tolerant mobile agent execution prevents that the agent fails, but allows it to book a hotel room and rent a car, although no seat is available on a flight to New York. Hence, the approach in [16] is not sufficient to ensure atomicity of a mobile agent execution. Indeed, both infrastructure as well as semantic failures need to be covered. However, building on top of the approach in [16] allows us to easily solve the stage agreement and the global agreement problem (see Section 4.1) and provide non-blocking transactional mobile agent execution. We begin by summarizing the approach presented in [16].

5.1 Fault-Tolerant Mobile Agent Execution: Background

In earlier work [16], we have presented an approach to fault-tolerant mobile agent execution, which prevents blocking in the mobile agent execution and ensures that the mobile agent is executed exactly-once. Fault tolerance is enabled by agent replication; instead of sending the agent from one place to the other, it is sent to the set $\mathcal{M}_i$ of places p_i^j at stage S_i (see Figure 8). This redundancy enables the mobile agent execution to proceed despite infrastructure failures, i.e., prevents blocking [5]. As we do not assume reliable failure detection, redundant agents potentially lead to multiple executions of the agent code. The solution presented in [16] consists, for all agent replicas at stage S_i, to agree on (1) the place p_i^{prim} that has executed the agent, (2) the resulting agent a_{i+1}, and (3) the set of places of the next stage $\mathcal{M}_{i+1}$. In the context of fault-tolerant mobile agent execution, (1), (2), and (3) are important to prevent multiple executions of the agent, i.e. ensure the exactly-once property. All the places that have potentially started executing a_i, except p_i^{prim}, abort. Only p_i^{prim} commits the modifications of a_i. This corresponds to the stage actions sa_i^j in Figure 7. However, in fault-tolerant mobile agent execution, sa_i decides unilaterally which subtransaction to commit. More specifically, the decision is taken independently of the parent transaction, as no such transaction exists. As we will see later (see Section 5.2), the agreement on item (1) corresponds to the stage agreement problem of NB-AC and is reused by NB-AC. On the other hand, (2) and (3) are not relevant for NB-AC.

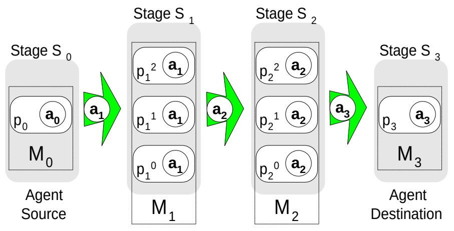


Figure 8: Example of an agent execution with three redundant places.

Overall, the entire fault-tolerant mobile agent execution leads to a sequence of agreement problems. Figure 9 shows an example of a mobile agent execution spanning 4 stages (S_0 to S_3). Note that at stage S_2, place p_2^0 fails, which causes p_2^1 to take over the execution. Solving an agreement problem leads all places in $\mathcal{M}_2$ to agree on p_2^1 as the place that has executed a_2. This would be of particular importance if p_2^0 had been erroneously suspected by the other places in $\mathcal{M}_2$.

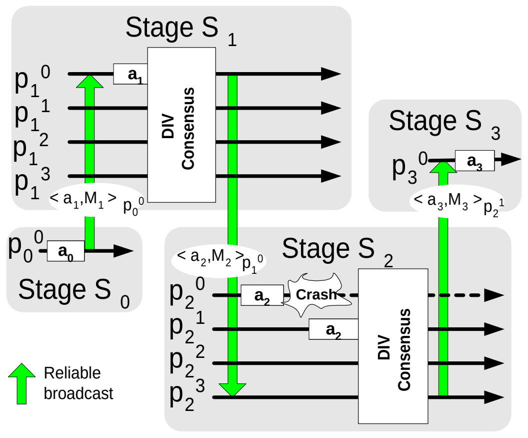


Figure 9: Agent execution where p_2^0 fails. An erroneously suspected place p_2^0 leads to the same situation.

To reduce the communication overhead among stages, the so-called *pipelined mode* reuses places of previous stages as *witnesses*, see Figure 10 [16]. A witness is a place that can execute the agent, but cannot deliver the requested service to the agent: the call to the service fails (i.e., a semantic failure) and the agent can then take corresponding actions, such as aborting the transaction.

[5]Note that the approach depicted in Figure 8 is different from case (b) in Figure 5. While the former uses agent replication to provide fault tolerance, the latter starts a new child transaction T_b'. Child transaction T_b' is generally not identical to T_b; rather it executes different operations. Sher et al. [21] use this approach. We give a more in-depth comparison with our approach in Section 7.

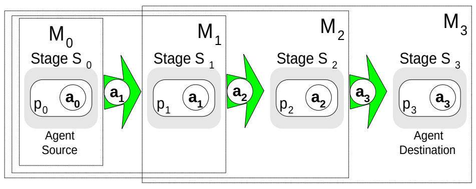


Figure 10: Model of the pipelined mode.

5.2 From Fault-Tolerant to Transactional Mobile Agent Execution

5.2.1 Solving the Stage Agreement Problem

An important property of the fault tolerance algorithm summarized in Section 5.1 is non-blocking. This property is also desired for transactional mobile agent executions. Indeed, a transactional agent execution that blocks because of a failure on place p_i has probably acquired a large amount of locks on previous places p_j $(j < i)$. Holding these locks prevents other agents from accessing this data items and thus dramatically reduces overall system throughput. Non-blocking transactional mobile agent execution does not suffer this problem. Progress is assured even in case of failures. Hence, locks are released earlier and overall system throughput improves. Consequently, we reuse our work on non-blocking fault-tolerant mobile agents to prevent blocking in a transactional mobile agent execution. More specifically, we use the approach in [16] to solve the stage agreement problem and thus build transactional mobile agents on top of it.

5.2.2 Solving the Global Agreement Problem

To solve NB-AC (see Section 4.1), the modifications of a_i on the primary $p_i^{prim} \neq p_{n-1}^{prim}$ (i.e., the subtransaction *sa*$_i^{prim}$) are not immediately committed after the stage execution. In other words, stage action *sa*$_i^{prim} \neq$ *sa*$_{n-1}^{prim}$ cannot unilaterally decide `COMMIT`. This is fundamentally different from the fault-tolerant mobile agent execution approach in [16]. The decision to `COMMIT` rather depends on the outcome of the top-level transaction (i.e., the result of the global agreement problem, see Figure 11). Here, a commit occurs when a has successfully executed all the stage actions a_i $(i = 0, 1, .., n-1)$. On the other hand, an abort may occur as soon as the execution of any a_j $(0 < i \leq j \leq n-1)$ semantically fails.

To terminate a pending transactional mobile agent execution, each primary place runs a *stationary* (i.e., not mobile) *stage action termination (SAT)* agent (see Figure 12). While agent a_{i+1} moves to p_{i+1}, the SAT agent *sat*$_i$ waits for the outcome of the entire transactional agent execution, either (1) a commit message from a_{n-1} or (2) an abort message from a_j $(1 \leq j \leq n-1)$. Upon reception of an abort message, *sat*$_i$ aborts the pending transaction *sa*$_i^{prim}$, otherwise commits it. Hence, SAT agent *sat*$_i$ can be viewed as the transaction manager [10] of the local transaction represented by stage action *sa*$_i$. While the outcome of the transaction is undetermined, all data items accessed by a_i on place p_i^{prim} (i.e., the place that has executed *sa*$_i$) remain locked and are not accessible by other agents.

To improve the performance, the place p_i itself can offer the SAT service. The agent a_i registers subtransaction *sa*$_i^{prim}$ that needs to be aborted or committed, and receives an ID. Using this ID, the agent can later contact the SAT service on p_i and initiate either a commit or an abort on the transaction. This service approach prevents the overhead of instantiating a SAT agent.

5.2.3 Terminating T_a

During its execution, agent a maintains a *SAT list* of all the SAT agents that it needs to contact in order to commit or abort the transaction. At every primary place p_i^{prim} ready to commit, a new entry is appended to this list. Unless the agent execution has failed on a previous stage, the execution at stage S_{n-1} decides whether to commit or abort the agent transaction. This decision is based on the outcome of the execution of a_{n-1} on place p_{n-1}^{prim}. If successful, the decision is `COMMIT`, otherwise `ABORT`. It is then communicated to $sat_i(1 < i \leq n-1)$, based on the SAT list. It is important that this decision eventually arrives at

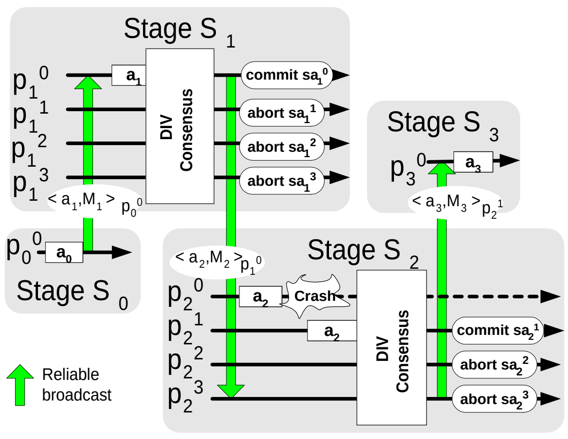


Figure 11: Transactional mobile agent that commits. Aborts on non-primary places are executed immediately, while primary places p_1^0 and p_2^1 only commit after a_2 has successfully executed on p_2^1.

all destinations. Indeed, a destination sat_i that does not receive the decision message does not learn the outcome of the transaction T_a and still retains all locks on the data items. Hence, the decision message is distributed using a reliable broadcast mechanism [6] that ensures the eventual arrival of the message at all destinations. All correct places in $\mathcal{M}_{n-1}$ participate in the reliable broadcast to prevent that a failure of p_{n-1}^{prim} causes the loss of the decision message. Figure 12 depicts an example agent execution with 5 stages.

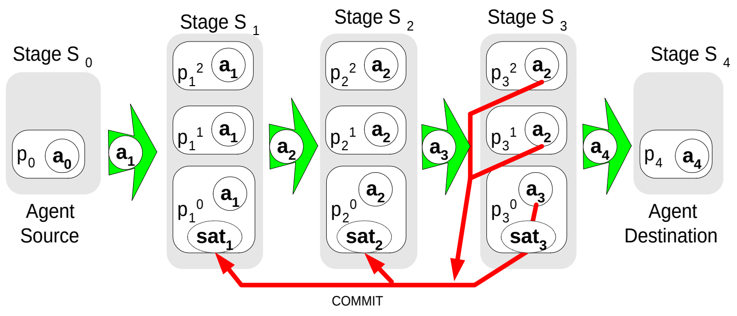


Figure 12: Committing transactional execution of a mobile agent execution with 5 stages.

5.3 Multiple Concurrent Transactions

In our discussion so far we have focused on enforcing the atomicity property of a transactional mobile agent execution. In a real system, however, a transactional mobile agent does not execute in an isolated environment. Rather, it executes concurrently with other transactional mobile agents. Multiple transactions accessing concurrently the same data items may lead to a violation of the isolation property. Isolation is enforced locally by locking all accessed data items until the outcome of the transactional mobile agent execution, i.e., `COMMIT` or `ABORT`, is determined. Clearly, the isolation property limits the possible level of concurrency. As a remedy, services decide themselves whether they allow concurrent access to their data. For this purpose, they design a so-called commutativity matrix [18], which shows potential conflicts among operations of this service and only allow operations that do not conflict to be executed concurrently.

Isolation also needs to be ensured on a global level, i.e., among places. Here, isolation is more difficult to achieve. One approach is to require that the stage actions on different places p_i and $p_j (j \neq i)$ are *independent* with respect to the execution order of stage actions. Revisiting the example agent execution in Section 3.1, two transactional mobile agents T_a and T_b can execute the airline ticket purchase, the hotel room booking, and the car rental in any order, i.e., not necessarily in a serializable order, without violating the isolation property. Consequently, because the three services are independent, no global serializibility is required to preserve isolation. This improves the concurrency of the transactional mobile agent and hence overall system performance. Clearly, service independence is a property of the application as well as the services. In the Internet, independence among services of different service providers is usually given.

We use the standard approaches [10] to enforce the other ACID properties, as well as for deadlock resolution, where we apply the timeout-based approach.

6 Implementation and Preliminary Evaluation

This section introduces TranSuMA (Transaction Support for Mobile Agents), a system that implements the ideas developed in Sections 4 and 5. We first present the architecture of TranSuMA, before discussing preliminary performance evaluation results.

6.1 Architecture

TranSuMA is based on an agent-based approach, where the transaction support mechanisms travel with the mobile agent [17]. This has the important advantage, that underlying mobile agent platforms do not need to be modified. On the other hand, the agent-based approach results in an increased communication overhead, as the size of the agent has also increased.

Our system is based on FATOMAS [17], the fault-tolerant mobile agent system presented in Section 5.1. This dependency is also reflected in the architecture of TranSuMA (see Figure 13). Here, a mobile agent a_i is composed of a TranSuMA user-defined agent, i.e., the agent developed by the agent owner, and the *transaction support module (TSM)*. The TSM provides the mechanisms for transactional mobile agent execution. It is based on the *fault tolerance enabler* (FTE) of FATOMAS. Indeed, from the viewpoint of the FTE, the TSM and the TranSuMA user-defined agent are just another FATOMAS user-defined agent. This has the advantage that the FTE can be reused with only minor modifications to it. These modifications are a generalization in that they allow FATOMAS user-defined agents to add user-defined state to the decision value in the stage agreement (see Section 5.1). The TSM uses this generalization to add the SAT list to the decision value, which enables the implementation of the reliable broadcast to terminate T_a.

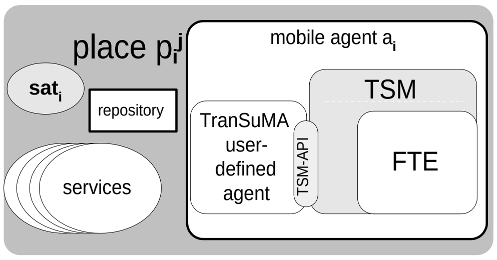


Figure 13: Architecture of TranSuMA.

The TranSuMA user-defined agent either interacts with the TSM (through the `TSM-API`) or with the services local to the place. The latter act as resource managers (RM) [10] (e.g., exporting an XA interface [12]). The `TSM-API` provides functions to begin a subtransaction, to abort the subtransaction (i.e., `abort`), and to commit (i.e., `commitCurrentTransaction`) or abort (i.e., `abortCurrentTransaction`) the current transactional mobile agent execution. From the view point of the user-defined agent, the TSM assumes the role of a transaction manager. However, the TSM does not really act as the transaction manager; rather, it forwards the calls of the user-defined agent to *sat*$_i$.

Typically, a stage action *sa*$_i$ of the user-defined agent in a_i^j consists of (1) a call to begin the transaction and (2) potentially multiple requests to local services, and (3) the end of stage. The stage may end with a call to `abortCurrentTransaction`, upon which the entire transactional mobile agent execution is terminated. The method `abortCurrentTransaction` is called, for instance, if a service request has terminated unsuccessfully and hence the transactional mobile agent execution cannot succeed any more. Note that the unsuccessful service request may not trigger the abort of the entire transactional mobile agent execution (see Section 3.2.1); rather, only the service requests of this stage action are aborted. This is achieved by calling the method `abort` in the `TSM-API`.

A call to method `commitCurrentTransaction` (or `abortCurrentTransaction`) triggers the commit (abort) of all unterminated subtransactions of the current transactional mobile agent. As shown in Section 5.2.3, transaction termination is achieved using reliable broadcast. We use a reliable broadcast with linear message cost [20], which has the advantage of a lower number of messages compared to other strategies. Hence, the TSM first commits (aborts) the local subtransaction and then contacts sat_{i-1}. The SAT agent sat_{i-1} recursively does the same, i.e., commits (aborts) local subtransactions and contacts sat_{i-2}.

6.2 Preliminary Performance Evaluation

To measure the performance of TranSuMA, we have implemented a Java-based prototype using ObjectSpace's Voyager platform [15]. Each place provides a simple counter service that offers a method to increment the value of the counter, in addition to the standard methods to commit and abort/rollback the modifications.

Our performance tests consist in sending a number of agents that atomically increment a set of counters, one at each stage S_i. Each agent starts at the agent source and returns to the agent source (i.e., the agent source is identical to the agent destination). This allows to measure the round trip time of the agent. Between two agents, the places are not restarted. Consequently, the first agent needs considerably longer for its execution, as all classes need to be loaded into the cache of the virtual machines. Consecutive agents benefit from already cached classes and thus execute much faster. We do not consider the first agent execution in our measurement results. As in [17], we assume that the Java class files are locally available on each place[6].

The test environment consists of seven AIX machines (Power PC 233 MHz processor, 256MByte of RAM). These machines are connected by either 100MBit Ethernet or 2MBit Tokenring; they are on 3 different subnets. As our evaluation results are in the area of hundreds of milliseconds, the difference in network bandwidth and the influence of the different subnets are negligible.

Our results represent the arithmetic average of 10 runs, with the highest and lowest values discarded to eliminate outliers. The coefficient of variations is in most cases lower than 5%. However, for few results, it went up to 15% because of variations in the network and machine loads.

We measure the costs of TranSuMA compared to FATOMAS. Each stage S_i $(0 < i < n)$ is composed of three places. In the experiment with two intermediate stages S_x and S_y, each intermediate stage uses three AIX machines, while another machine hosts the agent source and destination. If the number of stages exceeds four, S_x corresponds to all odd intermediate stages, i.e., S_{2k+1} $(2k+1 \leq n-1)$ and S_y to the even intermediate stages S_{2k} $(2k \leq n-1)$. The results in Figure 14 show that TranSuMA adds an overhead of 6 to 20% compared to a FATOMAS agent. This overhead is caused by the transaction support mechanisms such as the communication with the local SAT agent and the commitment when the agent has reached stage S_{n-1}. Results are similar for pipelined (see Section 5.1) FATOMAS and TranSuMA agents.

Note that the costs for an abort of T_a at S_{n-1} are the same as for commit. Indeed, the only difference between abort and commit is the content of the message reliably broadcasted to all SAT agents.

Clearly, the relative overhead decreases if the execution time of the stage action increases. Indeed, the time needed to increment the counter is negligible. If this time becomes more important, the relative overhead of TranSuMA compared to FATOMAS decreases considerably. Moreover, an increased agent size also decreases the relative overhead of the transaction mechanisms.

Infrastructure failures during the execution of stage action sa_i have a negative impact on the performance of the transactional mobile agent. However, they only influence the performance of FATOMAS. The overhead introduced by the transactional support mechanisms is the same. The influence of infrastructure failures on FATOMAS is shown in [17].

[6]Indeed, our chosen platform does not seem to properly support remote class loading in our test environment (see also [17]). We plan to port TranSuMA to another mobile agent platform to test the performance with remote class loading enabled.

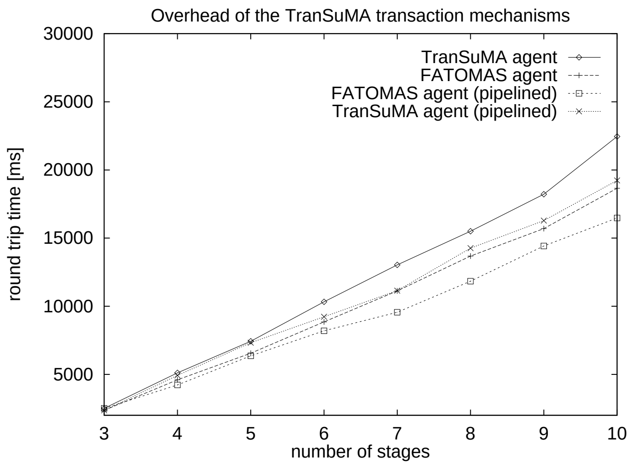


Figure 14: Round trip time [ms] of a TranSuMA agent compared to a FATOMAS agent for itineraries between 3 and 10 stages.

7 Related Work

Recently, mobile agents have been a very active field of research. In particular, a large body of work [3, 13, 16, 17, 19, 22, 23] has been published about fault-tolerant mobile agent execution, a field related to transaction support for mobile agents. However, fault-tolerant mobile agent execution generally does not address execution atomicity. Transactional mobile agent execution has received less attention. Exceptions are [2, 4, 21, 24]. The approaches in [2, 21, 24] are blocking; a single failure of a place currently executing the mobile agent may prevent progress of the transactional mobile agent execution. In contrary, [4] and our approach suggest an approach for non-blocking transactional mobile agent execution. We first relate [4] to our work, before briefly discussing [2, 21, 24]

7.1 Non-Blocking Transactional Mobile Agent Approach

The approach presented by Assis Silva and Popescu [4] is closest to our approach; they also build transaction support on top of fault-tolerant mobile agent execution. For this purpose, [4] reuses the approach in [3]. In contrary to our model of fault-tolerant mobile agent execution, the approach in [3] is based on a more complex model of leader election and transactions. Our model of fault-tolerant mobile agent execution [16], on the other hand, only relies on an agreement problem. This allows us to present a model for transactional mobile agent execution that integrates fault-tolerant mobile agents into a common model of nested transactions. We also provide a specification of non-blocking atomic commitment in the context of transactional mobile agents.

Assis Silva and Popescu [1, 4] do not describe any implementation. In contrary, our approach has been implemented and quantitatively evaluated.

Finally, [4] relies on compensatable transactions [9]. With compensatable transactions, stage action sa_i of an agent a can be committed before the outcome of the top-level transaction T_a is known. In the meantime, another mobile agent b can access data items modified by sa_i. If T_a eventually aborts, then a compensating transaction is run, which semantically undoes the modifications performed by sa_i. Agent b may have now read an inconsistent value. Consequently, T_b also needs to be aborted, leading to cascading aborts. For this purpose, another agent or an undo message have to be sent after agent b to notify b of the abort. Unfortunately, a slow undo message or agent may never reach a fast moving mobile agent, causing the undo to be delayed and increasing dependencies. Hence, compensatable transactions work best in an environment, where compensation transactions can be run without causing cascading aborts. However, this seriously limits the applicability of [4]. To our knowledge, the use of compensatable transactions is caused by the particular approach used for fault tolerance [3]. Indeed, the approach for fault tolerance is based on transactions, that have to commit before the agent execution can proceed. Consequently, if the agent execution is aborted at a later stage, compensation transactions have to run in order to undo the effects of the stage transactions. In contrast, TranSuMA relies on pessimistic concurrency control; subtransactions sa_i are only committed or aborted once the outcome of the top-level transaction is known. However, TranSuMA can also support compensatable transactions and thus is of more general use[7].

Note that the use of compensation transactions makes an abort very expensive. Each place p_i^{prim} needs to run the compensation transaction. Moreover, all the compensation transactions must eventually commit. Consequently, failures during the compensation transactions lead to blocking. In contrary, an abort in our approach is as expensive as a commit in the sense that the message sent to all SAT agents now contains the directive to abort.

7.2 Blocking Transactional Mobile Agent Approaches

In their work, Strasser and Rothermel [24] do not address the problem of execution atomicity for the entire mobile agent execution. Rather, they suggest a mechanism to partially rollback mobile agents that execute based on the protocol given in [19]. The use of compensation transactions, as suggested in this mechanism, leads to the isolation problems already discussed in Section 7.1. The stage actions of T_a are immediately committed and their effects thus visible to other transactional mobile agents, such as T_b. A partial rollback executes these compensation transactions in order to restore the state before the execution of the stage action of T_a.

In contrast to [24], Assis and Krause [2] address execution atomicity. However, they only present a model for transactional mobile agent execution. No algorithm or implementation is given. However, part of this model has been reused in [4].

In [21], Sher et al. present an approach for transactional mobile agents, which ensures the ACID properties on the entire mobile agent execution. However, [21] may be subject to blocking if a stage execution sa_i is executed on a single place. The probability of blocking is reduced by allowing parallel transactions to run over different parts of the itinerary that are combined again using so-called *mediators*, which govern how the parallel transactions are processed further. For instance, with the mediator `ANDjoin`, all parallel transactions have to arrive, or with `XORjoin`, only one has to arrive. Figure 15 depicts the example of an `XORjoin` mediator. The transactional mobile agent execution of a splits into two parallel transactions represented by agents b and c. For instance, b_{i-1} tries to book a flight with Swissair, while c_{i-1} books a flight with Delta Airlines. At stage S_i, the mediator `XORjoin` only keeps one of the subtransactions (represented by c_i and b_i), while the other is aborted. The agent a then continues to reserve a hotel room at stage S_{i+1}. The places that run a join mediator (i.e., p_i) must be visited by the partial mobile agents executing in parallel. This generally limits the itinerary to a (partially) static itinerary. Moreover, failures of non-parallel transactions and mediators result in blocking of the execution. Eliminating non-parallel transactions thus prevents blocking, i.e., a split mediator resides at the agent source and a join mediator at the agent destination. The entire mobile agent execution then runs as parallel transactions. However, executing parallel transactions from which only one is committed at the end, even if no failure occur, causes a considerable overhead.

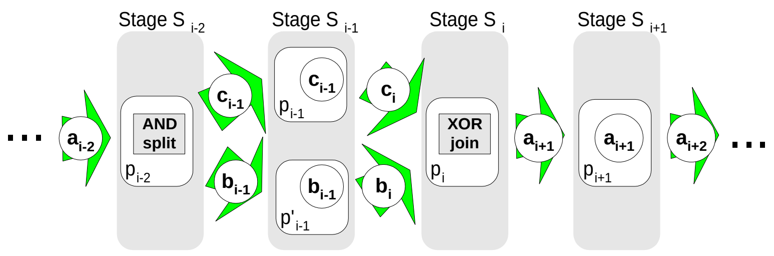


Figure 15: Mediators (rectangles) allow to execute parallel transactions.

[7]In this case, however, the isolation property in Section 4 needs to be relaxed.

8 Conclusion and Future Work

In this paper we have presented a novel approach for transactional mobile agent execution. In contrary to the approaches [2, 21], our approach does not block although places executing the agent potentially fail. While a mobile agent execution is blocked, it retains all locks. During this time, other transactional mobile agents cannot access the locked data item, resulting in reduced overall system throughput.

Similar to the approach in [4], we also suggest to build transaction support on top of fault-tolerant mobile agent execution in order to prevent blocking. For this purpose, we have introduced nested transactions as a common model for integrating fault-tolerant mobile agent execution and transaction support. This is a fundamental difference to the work in [4], where nested transactions only model the transaction support for mobile agents; fault tolerance is modeled separately. This common model has allowed us to formally specify the non-blocking atomic commitment (NB-AC) problem in the context of transactional mobile agents. NB-AC is fundamental to ensure atomicity of a mobile agent execution, the most tricky of the ACID properties [10]. In contrary to [4], our approach does not rely on compensatable transactions. Rather, it is more generic and can also rely on pessimistic concurrency control.

We have also shown that part of NB-AC is already solved by earlier work presented in [16]. Based on this work, we show a solution to the NB-AC problem. Our implementation, which is the first for non-blocking transactional mobile agents, builds on top of FATOMAS [17], the system that provides fault-tolerant mobile agent execution as modeled in [16]. Consequently, we use an agent-based approach [17], where the transaction support mechanisms travel with the mobile agent. Our preliminary performance results show that the overhead introduced by the transaction support mechanisms is reasonable compared with the performance of FATOMAS. The measurements have also been performed for pipelined agents with similar results.

In the future, we plan to improve the performance of our approach, as well as test it on a mobile agent platform other than Voyager.

References

[1] F. Assis Silva. *A Transaction Model based on Mobile Agents*. PhD thesis, Informatik, Technische Universität Berlin, June 1999.

[2] F. Assis Silva and S. Krause. A distributed transaction model based on mobile agents. In K. Rothermel and R. Popescu-Zeletin, editors, *Mobile Agents, Proceedings of the First International Workshop, MA'97*, LNCS 1219, pages 198–209. Springer Verlag, Apr. 1997.

[3] F. Assis Silva and R. Popescu-Zeletin. An approach for providing mobile agent fault tolerance. In K. Rothermel and F. Hohl, editors, *Proc. of the Second International Workshop on Mobile Agents (MA)*, LNCS 1477, pages 14–25. Springer Verlag, Sept. 1998.

[4] F. Assis Silva and R. Popescu-Zeletin. Mobile agent-based transactions in open environments. *IEICE Trans. Commun.*, E83-B(5), May 2000.

[5] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts, USA, 1987.

[6] R. Boichat and R. Guerraoui. Reliable broadcast in the crash-recovery model. In *Proc. of 19th Symposium on Reliable Distributed Systems (SRDS) 2000*, Nürnberg, Germany, Oct. 2000.

[7] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J ACM*, 43(2):225–267, Mar. 1996.

[8] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. In *Proc. of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 1–7, Atlanta, Georgia, Mar. 1983.

[9] H. Garcia-Molina and K. Salem. Sagas. In *Proc. of ACM SIGMOD Int. Conference on Management of Data and Symposium on Principles of Database Systems*, pages 249–259, 1987.

[10] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, USA, 1993.

[11] R. Guerraoui, M. Hurfin, A. Mostefaoui, R. Oliveira, M. Raynal, and A. Schiper. Consensus in asynchronous distributed systems: A concise guided tour. In S. S. S. Krakowiak, editor, *Advances in Distributed Systems*, number LNCS 1752, pages 33–47. Spinger, 2000.

[12] ISO/IEC. *Information Technology - Distributed Transaction Processing - The XA Specification*, 1st edition, 1996. ISO/IEC 14834.

[13] A. Mohindra, A. Purakayastha, and P. Thati. Exploiting non-determinism for reliability of mobile agent systems. In *Proc. of the IEEE Int. Conference on Dependable Systems and Networks (DSN)*, pages 144–153, New York, June 2000.

[14] J. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, MA, USA, 1985.

[15] ObjectSpace. *Voyager: ORB 3.1 Developer Guide*, 1999. http://www.objectspace.com/products.

[16] S. Pleisch and A. Schiper. Modeling fault-tolerant mobile agent execution as a sequence of agreement problems. In *Proc. of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 11–20, Nuremberg, Germany, Oct. 2000.

[17] S. Pleisch and A. Schiper. FATOMAS: A fault-tolerant mobile agent system based on the agent-dependent approach. In *Proc. of the IEEE Int. Conference on Dependable Systems and Networks (DSN)*, pages 215–224, Goteborg, Sweden, July 2001.

[18] A. Rakotonirainy. Exploiting transaction and object semantics to increase concurrency. In C. Girault, editor, *Proceedings of IFIP*, pages 155–164. Elsevier Science B.V., 1994.

[19] K. Rothermel and M. Strasser. A fault-tolerant protocol for providing the exactly-once property of mobile agents. In *Proc. of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 100–108, West Lafayette, Indiana, Oct. 1998.

[20] F. Schneider, D. Gries, and R. Schlichting. Fault-tolerant broadcasts. *Science of Computer Programming*, 4(1):1–15, Apr. 1984.

[21] R. Sher, Y. Aridor, and O. Etzion. Mobile transactional agents. In *Proc. of Int. Conference on Distributed Computing Systems (ICDCS)*, pages 73–80, Apr. 2001.

[22] L. Silva, V. Batista, and J. Silva. Fault-tolerant execution of mobile agents. In *Proc. of the IEEE Int. Conference on Dependable Systems and Networks (DSN)*, pages 135–143, New York, June 2000.

[23] M. Strasser and K. Rothermel. Reliability concepts for mobile agents. *International Journal of Cooperative Information Systems*, 7(4):355–382, 1998.

[24] M. Strasser and K. Rothermel. System mechanisms for partial rollback of mobile agent execution. In *Proc. of Int. Conference on Distributed Computing Systems (ICDCS)*, pages 20–28, Taipei, Taiwan, Apr. 2000.