Execution Atomicity for Non-Blocking Transactional Mobile Agents

STEFAN PLEISCH ANDRÉ SCHIPER
Distributed Systems Laboratory
Swiss Federal Institute of Technology (EPFL), CH-1015 Lausanne
{Stefan.Pleisch,Andre.Schiper}@epfl.ch

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. This requires to solve an instance of the atomic commitment problem. However, it is important that failures (e.g., of machines or agents) do not lead to blocking of transactional mobile agents, i.e., agents that execute as a transaction. In this paper, we give a novel specification of 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 and give preliminary performance results.

KEY WORDS
Mobile agents, non-blocking atomic commitment, transaction, replication

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 of 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 is an important property of a transactional mobile agent, i.e., a mobile agent, that executes as a transaction, and needs to be ensured also in the face of failures of 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. 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 may block if the place running the mobile agent fails [11, 12], 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 on data items cannot be freed. As no other transactional mobile agents can acquire the locks, overall system throughput is dramatically reduced.

In this paper, we give the first, concise specification of non-blocking atomic commitment in the context of transactional mobile agents. Furthermore, we show that the non-blocking property of atomic commitment is closely related to the non-blocking property of mobile agent execution. Similarly to the approaches in [1, 12], this allows us to build transactional support on top of fault-tolerant mobile agent execution in order to prevent blocking. For this purpose, we use previous work in the context of fault-tolerant mobile agents [10]. In contrast to [1, 12], our approach supports also non-compensatable transactions (i.e., transactions that can no longer be undone once they are committed). In this respect, it is more general than the approaches in [1, 12]. For simplicity of presentation, we assume in the following that the agent execution only consists of non-compensatable transactions. The extension to compensatable transactions is straightforward. A compensatable transaction can be undone by executing a compensating transaction which semantically undoes its effects.

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 that the overhead introduced by the commitment mechanisms is reasonable.

The rest of the paper is structured as follows: Section 2 presents our system and agent model. In Section 3, we discuss the problem of enforcing the atomicity property of mobile agent execution, and specify the non-blocking atomic commitment problem for transactional mobile agents in Section 4. Section 5 presents our approach for transactional mobile agent execution and Section 6 shows the implementation and preliminary performance results. Finally, Section 7 concludes the paper.

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

2 Model

Agent Model. On each machine i, a *place* p_i $(0 \leq i \leq n)$ provides the logical execution environment for the agent. We call the execution of the agent at a place a "*stage* of the agent execution". The entire mobile agent execution can thus be viewed as a sequence of stages $S_0, \ldots, S_n$. Agent a_i at the corresponding stage S_i represents the agent a that has executed the stage actions sa_j on places p_j $(j < i)$ and is about to execute on place p_i (Fig. 1, with $n = 4$). We assume that stage actions do not access any remote places. The execution of the agent a_i at stage S_i results in a new internal state of the agent (reflected by the resulting agent a_{i+1}) and also a new state of the place p_i (if the operations of the agent have side effects). In this paper, we assume a single agent, i.e., we do not consider multiple agents that collaborate to fullfill a particular task.[2]

System Model. We assume an asynchronous system such as the Internet, i.e., no bounds on communication delays nor on relative process speeds exist. Machines, places, or agents fail by crashing (i.e., we exclude Byzantine processes) and can recover later. For simplicity, we assume that communication channels are reliable. Note that a crashing machine leads to the crash of all places and all agents running on it, and a failure of a place also crashes the agents running on this place. A component that has failed but not yet recovered is called *down*, whereas it is *up* otherwise. We assume that components are eventually up forever.[3] This assumption is only needed to achieve consistency on the level of the application that has launched the transactional mobile agent. For instance, crashed databases accessed by the mobile agent need to properly terminate the transactions that have not yet committed or aborded. For this purpose, they need to eventually recover. A component is *correct*, if it does not crash during the execution of the transactional mobile agent.[4] Failures of machines, places, and agents are called *infrastructure failures*.

In an asynchronous system, reliable failure detection is impossible. Consequently, the approach presented in this paper does not rely on correct failure detection. More specifically, it allows p (a machine, place, or agent) to erroneously suspect q (another machine, place, or agent), although q has not failed. However, we assume our failure detection mechanism to satisfy the weak properties of $\diamond S$, which allows us to solve agreement problems [3].

3 The Problem of Execution Atomicity

An *atomic* mobile agent execution ensures that either all stage actions succeed or none at all. If a stage action is not successfull (e.g., no flight to the destination is available in the example of Section 1), we speak of a *semantic failure*. More generally, a semantic failure occurs if a requested service is not delivered because of the application logic or because the process providing this requested service has failed.

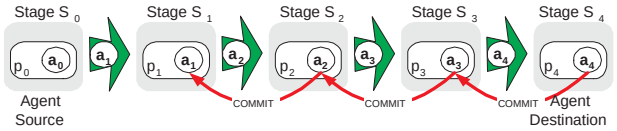

Figure 1. Successful transactional mobile agent execution T_a.

To achieve execution atomicity in a transactional mobile agent execution T_a, place p_i only forwards agent a_i to the place p_{i+1} at stage S_{i+1} if the execution of a_i has been successful (Fig. 1). Otherwise, p_i immediately aborts T_a. At stage S_n, the successful execution of a_n triggers a commit to all places p_j $(0 < j \leq n)$. Consequently, only place p_n can decide to commit T_a. However, every place p_i can decide to unilaterally abort T_a, when it is executing a_i.

Infrastructure failures may lead to blocking or to a violation of the atomic execution of the transactional mobile agent. Assume that place p_2 fails while executing a_2 (Fig. 1). In an asynchronous system, p_0 and p_1 are left with the uncertainty of whether p_2 has actually failed or is just slow [4]. In addition, it is impossible for p_0 and p_1 to detect the exact point where p_2 failed in its execution. More specifically, they cannot detect whether p_2 has succeeded in forwarding the agent to the next stage or not. If p_2 has failed before forwarding a_2, the agent execution is blocked. To prevent blocking, another place such as p_1 could monitor place p_2. If it detects the failure of p_2, it could then abort T_a. However, unreliable failure detection potentially leads to a violation of the atomicity property. Indeed, assume that p_1 detects the failure of a_2. Place p_1 thus assumes the responsibility for the decision and decides to abort transaction T_a. However, because of unreliable failure detection, p_1 may erroneously suspect p_2. Actually, even if p_2 has failed, it may have succeeded in forwarding the agent to p_3, resulting in potentially conflicting decisions on the outcome of the transaction. Indeed, while p_1 decides to abort the transaction, p_3 may decide to commit T_a. This conflicting outcome clearly violates the atomic execution property of the transaction's operations, as certain operations are aborted (i.e., on p_1), whereas others are committed (on p_3) or may be committed later (on p_2).

Note that traditional distributed transaction mechanisms based on 2-phase or 3-phase commit (2 or 3PC) [5] are not well suited to ensure execution atomicity in the context of mobile agents. Indeed, having a central coordinator contradicts the autonomy property of mobile agents. As the places of the mobile agent often are not known beforehand, but are dynamically determined along the way, the coordinator of the 2PC needs to be updated on the new partici-

[2] Our approach can be generalized to also encompass multiple agents.

[3] Actually, components only need to be up sufficiently long to finish their tasks.

[4] This definition simplifies the discussion. Actually, a place that crashes at stage S_i can recover and again participate in the execution of the agent at a later stage S_k $(k > i)$.

pants. Moreover, the voting phase of the 2PC is not needed at all. Rather, the outcome of the atomic commitment is known to the agent at S_i $(0 < i \leq n)$.

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 standard ACID properties [5]:

- (*Atomicity*) The stages 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.
- (*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 $sa_1, \ldots, sa_n$ is executed as a transaction. Every stage action is itself a transaction in order to ensure that it can be undone. Consequently, T_a can be modeled as *nested transactions* [7]. 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.

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. The properties we are mainly concerned with are atomicity and durability.[5] 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 [2]. Informally, T_a commits if all stage actions sa_i $(0 < i \leq n)$ 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. Because the places can unilaterally decide whether to continue T_a (on p_i $(0 < i < n)$), commit (p_n), or abort (p_i $(0 < i \leq n)$) (see Section 3), blocking in the transactional mobile agent execution also leads to blocking in the atomic commitment problem. Blocking in the atomic commitment is thus a consequence of blocking in the mobile agent execution. Hence, non-blocking atomic commitment can be achieved by ensuring the non-blocking property in the execution of the mobile agent. In other words, it is sufficient to ensure that the decision on commit or abort is always possible.

4.1 Non-Blocking Atomic Commitment Problem for Transactional Mobile Agents

Blocking can be prevented by executing replicas of the mobile agent on multiple places (Fig. 2). Instead of forwarding the agent to only one place, the agent is sent to the set M_i of places $p_i^0, \ldots, p_i^m$ of the next stage S_i. Although one place at stage S_i may crash (e.g., p_2^0), other places can take over and prevent blocking of the mobile agent execution. As we do not assume reliable failure detection, redundant agents potentially lead to multiple executions of the agent code. The places of a stage (i.e., $p_i^0, \ldots p_i^2$) can either be [9]:

- *(Iso-places)* All places at a stage are exact replicas, which leads to two levels of replication: replication of the agent and replication of the place. As an example, all the replica places are provided by Lufthansa. As the places replicate their state among themselves, only one agent replica must eventually be executed. The other agent replicas need to undo their potential executions. Otherwise, the execution of the agent at the stage is reflected multiple times in the state of the place replicas.
- *(Hetero-places)* The places belong to different companies that provide the same service (e.g., Swiss, Lufthansa, and American Airlines provide flights to a particular destination).
- *(Hetero-places with witnesses)* Some of the places among the hetero-places are *witnesses*. 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. Hence, a witness helps to handle infrastructure failures at a stage.

Recall the example of the agent that buys a flight ticket, and books a hotel room and rents a car at the destination, given in Section 1. Such an agent could be defined

[5] 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.

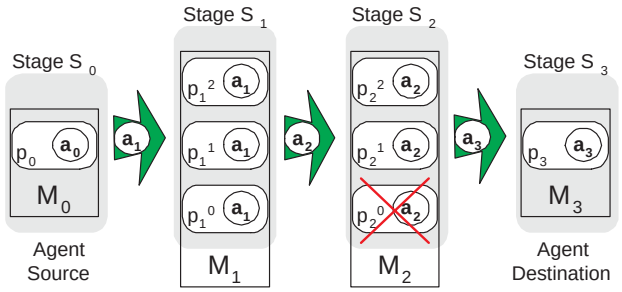

Figure 2. Example of an agent execution with three (i.e., $m = 3$) redundant places.

as follows: it starts at the agent source and eventually returns there again. The first stage S_1 consists of three hetero-places, one respectively from Lufthansa, Swiss, and American Airlines. Stage S_2 contains three iso-places from a particular hotel, say Hyatt. Finally, S_3 contains two hetero-places (e.g., Avis and Hertz) and one witness.

Replication of the agent at a stage and thus non-blocking adds another level of subtransactions to a transactional mobile agent execution. Indeed, subtransaction sa_i, in turn, can be modeled by yet another level of subtransactions $sa_i^0, \ldots, sa_i^m$, which correspond to the agent replicas $a_i^0, \ldots, a_i^m$ running on places $p_i^0, \ldots, p_i^m$. 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. Indeed, assume that more than one agent replica commits at stage S_i. The execution of the agent a_i is thus reflected in multiple places (in the case of hetero-places) or more than once on each place replica (in the case of iso-places). We call the place that has executed the stage action sa_i^{prim} the *primary* and denote it p_i^{prim}.

So, the non-blocking atomic commitment (NB-AC) problem[6] 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*) $p_i^{prim} \in \mathcal{M}_i$ 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.

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. We call this abort stage-ABORT, while the abort related to the global agreement problem is called global-ABORT. Consequently, the decision on p_i^{prim} is implicitly also a decision stage-ABORT for all places $p_i^j \neq p_i^{prim}$. However, the agreement only occurs on the decision about the primary, not on stage-ABORT or stage-COMMIT. Indeed, subtransactions $sa_i^j \neq sa_i^{prim}$ abort (if they are down, they abort upon recovery), while sa_i^{prim} only aborts if its parent transaction aborts (i.e., if a global-ABORT occurs). However, the decision on global-COMMIT or global-ABORT 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:[7]

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 global-ABORT (semantic ABORT). Primary p_{n-1}^{prim} decides either global-ABORT or global-COMMIT. Decision global-COMMIT is a consequence of no semantic abort up to and including stage S_{n-1}. In all other cases the decision is global-ABORT.
- (*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 global-ABORT. Rather, infrastructure failures cause the agent to execute on another place at the same stage (i.e., may lead to stage-ABORTs). 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 global-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. An (infrastructure) 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 a global-ABORT is given. In

[6] A specification for NB-AC in the context of traditional distributed systems (i.e., without mobile agents) is given for instance in [6].

[7] We consider the case in which stage S_n is not replicated (Fig. 2). To prevent blocking, the transaction (and thus the global agreement problem) spans only stages $S_1, \ldots, S_{n-1}$.

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

The approach we advocate to provide non-blocking transactional mobile agents builds upon earlier work on fault-tolerant mobile agent execution [10]. The solution presented in [10] 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. At the same time, the agreement on (1) solves the stage agreement problem (see Section 4.1). However, in fault-tolerant mobile agent execution a_i decides unilaterally whether to commit subtransaction sa_i^{prim}. More specifically, the decision is taken independently of the parent transaction, as no such transaction exists.

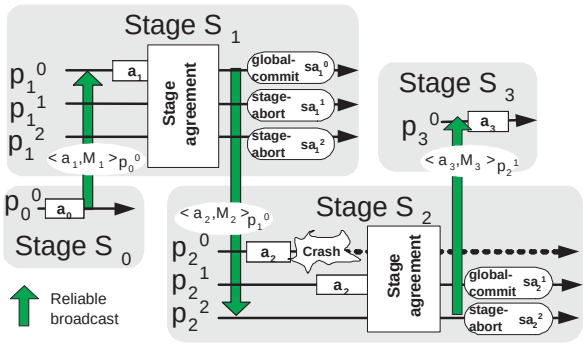

Figure 3. Non-blocking transactional mobile agent with crash of p_2^0.

To achieve non-blocking atomic commitment, 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 [10]. The decision to commit rather depends on the outcome of the transaction T_a.

5.1 Terminating T_a

To terminate a pending transactional mobile agent execution such as T_a, each primary place runs a *stationary* (i.e., not mobile) *stage action termination (SAT)* agent. During its execution, agent a maintains a 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. As soon as the outcome of T_a is decided, all SAT agents in the list are notified of the outcome. It is important that this message eventually arrives at all destination (i.e., the SAT agents participating in T_a). Indeed, a destination SAT agent 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 that ensures the eventual arrival of the message at all destinations. This requires the participation of all places at stage S_{n-1}.

The other ACID properties of a transactional mobile agent execution (i.e., consistency, isolation, and durability) are enforced using standard approaches [5], as well as for deadlock resolution, where we apply the timeout-based approach.

5.2 Pipelined Mode

To reduce the communication overhead among stages, the so-called *pipelined mode* reuses places of previous stages as *witnesses* for the current stage (Fig. 4), instead of using three new places for every stage [9]. For instance, stage S_2 is composed of the new place p_2 and places p_1 and p_0. At stage S_1, we assume the existance of one (dummy) place that acts as a witness (not shown in Fig. 4) in addition to p_0.

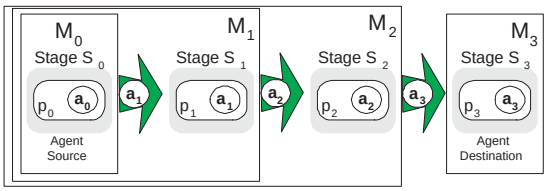

Figure 4. Model of the pipelined mode.

6 TranSuMA

We have implemented a Java-based prototype system, called TranSuMA (Transaction Support for Mobile Agents), using the Voyager mobile agent platform [8]. It is build on top of FATOMAS, our approach for fault-tolerant mobile agent execution [10]. Similarly to FATOMAS, all transaction support mechanisms travel with the mobile agents. This has the important advantage that the underlying mobile agent platform does not need to be modified. On the other hand, the communication overhead is increased, as the size of the agent is larger.

To measure the performance of TranSuMA, all places offer a simple "counter" service with a method to increment the counter, and the standard methods to commit and rollback/abort. 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., $p_0 = p_n$). This

allows us to measure the round trip time of the agent. As in [10], we assume that the Java class files are locally available on each place.[8]

The test environment consists of seven AIX machines (Power PC 233 MHz processor, 256MByte of RAM) connected by a LAN (100MBit Ethernet). 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.

The results in Figure 5 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 5.2) FATOMAS and TranSuMA agents.

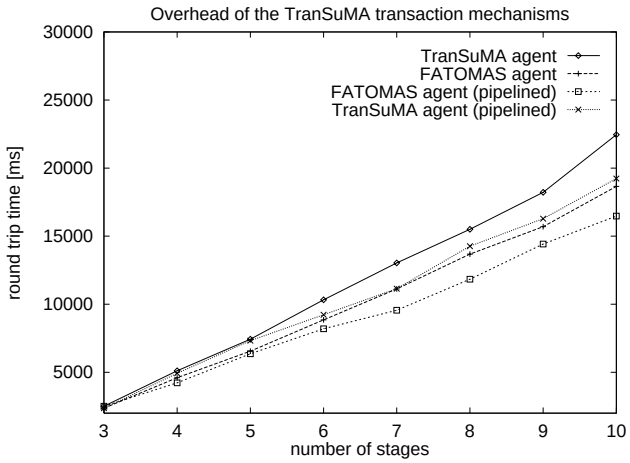

Figure 5. Round trip time [ms] of a TranSuMA agent compared to a FATOMAS agent for itineraries between 3 and 10 stages.

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 no impact on the performance of TranSuMa relative to FATOMAS. The reader is referred to [10] for the influence of infrastructure failures to FATOMAS.

[8] Indeed, our chosen platform does not seem to properly support remote class loading in our test environment (see also [10]). We plan to port TranSuMA to another mobile agent platform to test the performance with remote class loading enabled.

7 Conclusion and Future Work

In this paper we have presented a specification for non-blocking atomic commitment for transactional mobile agent execution. To our knowledge, we are the first to present such a specification in this context. Our specification uses nested transactions as a common model for integrating fault-tolerant mobile agent execution and transaction support. Our implementation, which is the first for non-blocking transactional mobile agents, satisfies this specification and we have given preliminary performance measurement results.

In the future, we plan to improve the performance of our approach and investigate its behavior in a system with multiple concurrent transactions.

References

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

[2] P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems* (Reading, MA: Addison-Wesley, 1987).

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

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

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

[6] 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 (Ed.), *Advances in Distributed Systems*, LNCS 1752 (Heidelberg, Germany: Spinger Verlag, 2000) 33–47.

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

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

[9] S. Pleisch and A. Schiper, Modeling fault-tolerant mobile agent execution as a sequence of agreement problems, *Proc. of 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, Nuremberg, Germany, 2000, 11–20.

[10] S. Pleisch and A. Schiper, FATOMAS: A fault-tolerant mobile agent system based on the agent-dependent approach, *Proc. of Int. Conference on Dependable Systems and Networks (DSN'01)*, Goteborg, Sweden, 2001, 215–224.

[11] R. Sher, Y. Aridor, and O. Etzion, Mobile transactional agents, *Proc. of 21st IEEE Int. Conference on Distributed Computing Systems (ICDCS'01)*, Phoenix, Arizona, 2001, 73–80.

[12] M. Strasser and K. Rothermel, System mechanisms for partial rollback of mobile agent execution, *Proc. of 20th IEEE Int. Conference on Distributed Computing Systems (ICDCS'00)*, Taipei, Taiwan, 2000, 20–28.