FATOMAS - A Fault-Tolerant Mobile Agent System Based on the Agent-Dependent Approach

Stefan Pleisch
IBM Research
Zurich Research Laboratory
CH-8803 Rüschlikon
spl@zurich.ibm.com

André Schiper
Operating Systems Laboratory
Swiss Federal Inst. of Technology (EPFL)
CH-1015 Lausanne
Andre.Schiper@epfl.ch

Abstract

Fault tolerance is fundamental to the further development of mobile agent applications. In the context of mobile agents, fault-tolerance prevents a partial or complete loss of the agent, i.e., it ensures that the agent arrives at its destination. In this paper, we present FATOMAS, a Java-based fault-tolerant mobile agent system based on an algorithm presented in an earlier paper. In contrary to the standard "place-dependent" architectural approach, FATOMAS uses the novel "agent-dependent" approach introduced in the paper. In this approach, the protocol that provides fault tolerance travels with the agent. This has the important advantage to allow fault-tolerant mobile agent execution without the need to modify the underlying mobile agent platform (in our case ObjectSpace's Voyager).

In our performance evaluation, we show the costs of our approach relative to the single, non-replicated agent execution. Pipelined mode and optimized agent forwarding are two optimizations that reduce the overhead of a fault-tolerant mobile agent execution.

1 Introduction

Mobile agents are computer programs that act autonomously on behalf of a user, and travel through a network of heterogeneous machines. Failures in a mobile agent system may lead to a partial or complete loss of the agent[1]. To achieve fault-tolerance, the agent owner (i.e., the person or application creating and configuring the agent) can try to detect the failure of its agent, and upon such an event launch a new agent. However, this requires the ability to *correctly* detect the crash of the agent, i.e., to distinguish between a failed agent and an agent that is delayed by slow processors or slow communication links. Unfortunately this cannot be achieved in systems such as the Internet. An agent owner who tries to detect the failure of his agent thus cannot prevent the case where he mistakenly thinks that the agent has crashed. In this case, launching a new agent leads to multiple executions of the agent, i.e., to the violation of the desired *exactly-once* property of agent execution. Even though this may be acceptable for certain applications (e.g., applications whose operations do not have side-effects, i.e., are idempotent), others clearly forbid it. Fortunately, multiple agent executions can be prevented in environments with unreliable failure detection by replicating agents with an adequate protocol. Indeed, if failure detection is unreliable, replication by itself does not ensure the exactly-once execution property. For example, exactly-once execution is not ensured by the protocol of [5], which assumes a perfect failure detection mechanism. Some systems have tried to address the exactly-once issue in the context of unreliable failure detection, and have proposed complex solutions based on transactions and leader election [2, 9]. Other approaches address the exactly-once execution problem by detecting duplicate agents at the end of the agent execution, and undoing at that moment the effects of multiple executions [6, 10]. However, undoing the effects of duplicate agents at the end of the execution is not simple, and often limits dramatically the overall system throughput. In contrast to these different approaches, we have presented in an earlier paper [8] an approach that ensures the exactly-once execution property using a very simple principle: the mobile agent execution is modeled as a sequence of agreement problems. In the current paper, we present FATOMAS, a Java-based FAult-TOlerant Mobile Agent System, that implements this approach.

In order to characterize the architecture of FATOMAS, we start by introducing two approaches called *place-dependent* and *agent-dependent*. Place-dependent is the standard approach that integrates fault tolerance into the mobile agent platform (the platform, that provides the support for mobile agents). Agent-dependent is the new approach introduced by FATOMAS. In this approach, the protocol that provides fault tolerance travels with the agent.

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

This has the important advantage to allow fault-tolerant agent execution without having to modify the underlying mobile agent platform. Currently, FATOMAS supports ObjectSpace's Voyager mobile agent platform [7]. However, our design enables to easily port FATOMAS to other mobile agent platforms.

The rest of the paper is structured as follows. Section 2 presents the model for fault-tolerant mobile agent execution. In Section 3, we explain the architectural choices in FATOMAS, in particular the agent-dependent approach. Section 4 discusses important implementation issues. The performance evaluation of FATOMAS is presented in Section 5. Section 6 summarizes other work published in this area and compares it with our work. Finally, Section 7 concludes the paper.

2 Fault-Tolerant Mobile Agent Execution: Background

In this section, we give a brief overview of the model of fault-tolerant mobile agent execution, and of the algorithm that ensures fault-tolerance. A detailed presentation is given in [8].

2.1 System Model

We assume an asynchronous distributed system, where processes communicate by message passing. Processes only fail by crashing, i.e., we exclude malicious processes. The system is asynchronous, i.e., we do not assume any bound on the transmission delay of messages and on relative process speeds. Because agreement problems are not solvable in an asynchronous system [4], we assume that the system is augmented with a failure detector allowing us to solve consensus [1], a particular agreement problem. Consensus is mentioned in Section 2.5.

2.2 Mobile Agent Model

A mobile agent executes on a sequence of machines, where a place p_i provides the logical execution environment for the agent (see Figure 1). 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 the agent *source* and *destination*. Logically, a mobile agent executes in a sequence of actions. Each action is represented by an agent $a_i (0 \leq i \leq n)$ at the corresponding stages S_i. Place p_i executes agent a_i at stage S_i, which results in a new internal state of the agent as well as potentially a new state of the place (if the operations of an agent have side effects)[2]. We denote the resulting agent a_{i+1}. Place p_i forwards a_{i+1} to p_{i+1} (for $i < n$).

[2]Interactions with remote places potentially also lead to modifications of their state. This paper does not explore these aspects further, as they are similar to the state modifications on the hosting place.

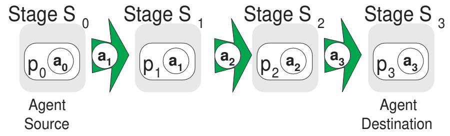

Figure 1. Model of a mobile agent execution with four stages.

2.3 Mobile Agent Model With Replication

Without replication, the forwarding scheme of the previous section leads to blocking, as a failure may cause the loss of the agent. To prevent blocking, the agent at stage S_i is sent to a set $\mathcal{M}_{i+1}$ of places at S_{i+1}, instead of only to one place p_{i+1}. In other words, the place p_i^j hosts the agent replica a_i^j of agent a_i. We will omit the superscripted index if the agent replica is understood from the context. If one place $p_{i+1}^k \in \mathcal{M}_{i+1}$ fails while executing the agent, another place $p_{i+1}^j \in \mathcal{M}_{i+1}$ $(j \neq k)$ takes over. Figure 2 depicts an example of a mobile agent execution.

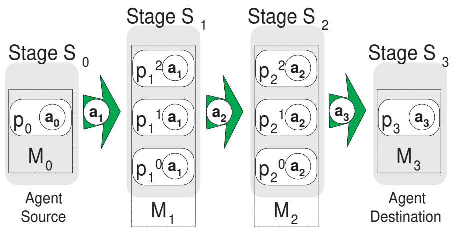

Figure 2. Example of an agent execution with three redundant places.

The so-called *pipelined mode* reduces the communication overhead caused by the agent forwarding. Instead of forwarding the agent at each stage S_i to a new set of places $\mathcal{M}_{i+1}$, previous stage places are reused as *witnesses* for the execution at the new stage (see Figure 3). 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 and the agent can then take corresponding actions such as notifying the user or backtracking. Any place that can execute the agent can act as a witness.

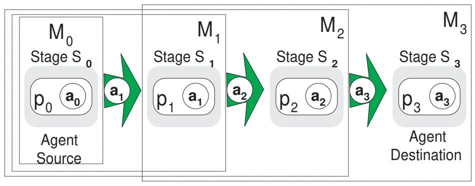

Figure 3. Model of the pipelined mode.

2.4 Algorithmic Issues

The redundancy illustrated in Figure 2 enables the mobile agent execution to proceed despite failures, i.e., prevents blocking. However, the algorithm that prevents blocking while ensuring a consistent execution is not as easy as one might guess. This is related to the fact that we assume a system model in which failure detection is unreliable. The solution presented in [8] consists, for all agent replicas at stage S_i, to solve the *stage agreement problem*, which leads them to agree on:

- the place that has executed the agent, called the *primary* and denoted p_i^{prim},
- the resulting agent a_{i+1}, and
- $\mathcal{M}_{i+1}$, the set of places for stage S_{i+1}

Hence, the fault-tolerant mobile agent execution leads to a sequence of agreement problems. Figure 4 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 $\mathcal{M}_2$.

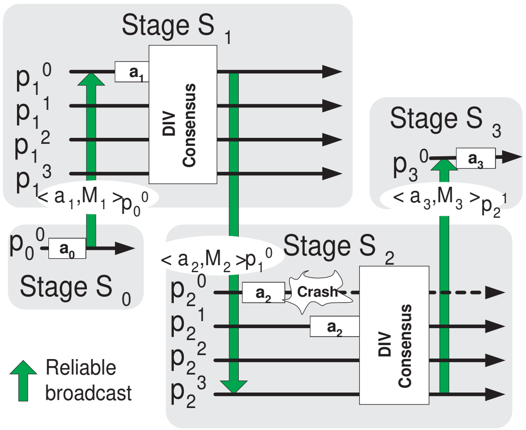

Figure 4. Agent execution where p_2^0 fails. An erroneously suspected place p_2^0 leads to the same situation.

2.5 Two Building Blocks: DIV Consensus and Reliable Broadcast

At every stage S_i (1) one (or potentially multiple) of the replica agents a_i^j executes the stage operation phase, then (2) solves an agreement problem with all replica agents, and (3) finally $< a_{i+1}, M_{i+1} >$ is sent to the next stage.

Items (1) and (2) are done together as part of a variant of the consensus problem, called Deferred Initial Value Consensus (DIV Consensus for short) [3]. DIV Consensus is the first building block of our FATOMAS system. In the consensus problem, each process needs an initial value at the beginning of consensus [1]. In our problem, the initial value at stage S_i for place p_i^j is obtained by executing agent a_i. Executing a_i on all the places of stage S_i is not wanted and too costly. DIV Consensus allows us to defer the computation of the initial value of some place p_i^j and only perform the computation when requested by the DIV Consensus algorithm. For example, if p_i^0 succeeds in computing its initial value and does not crash, no other place p_i^j, $j > 0$, will be required to provide (i.e., compute) an initial value. DIV Consensus assumes that a majority of participants does not fail.[3]

Item (3) is an instance of the reliable broadcast problem. Traditional reliable broadcast protocols assume a $1 - m$ communication scheme where one process broadcasts a message to m destination processes. In our case we have a $r - m$ communication schema: r sender have the same message to reliably broadcast to m destinations. This is discussed in Section 4.1.2.

2.6 Asynchronous Agent Propagation

We have assumed (Section 2.1) an asynchronous system, where there is no bound on the transmission delay of messages. This has an impact on the different instances of the agreement protocol (i.e., DIV Consensus) that run at each stage S_i of an agent execution. Because of the asynchrony, the agent a_i may not arrive simultaneously at the different places p_i^j of stage S_i. Assume for instance that the agent replicas a_i^0, a_i^1, a_i^2 are sent respectively to $p_i^0, p_i^1, p_i^2 \in \mathcal{M}_i$ (see Figure 4), and assume that a_i^2 arrives late at p_i^2. DIV Consensus may have already started executing for agent replicas a_i^0 and a_i^1 when a_i^2 arrives. The execution of DIV Consensus may even have terminated when a_i^2 arrives. The late arrival of a_i^2 at p_i^2 is indistinguishable to a_i^0 and a_i^1 from the crash of a_i^2 followed by the recovery of a_i^2. Thus, the asynchrony assumption forces us indirectly to support the recovery of agents after a crash.

3 Architecture

3.1 Isolation of the Fault Tolerance Mechanisms

Conceptually, a mobile agent executes in three phases: (1) an *initialization phase*, (2) *stage operation phases*, and (3) a *termination phase*. The initialization phase takes place on the agent source S_0, while the termination phase is executed on the agent destination S_n. Between the agent source and destination, the stage operation phase is run at each stage S_i $(0 < i \leq n)$. Hence, it is executed multiple times.

Ideally, fault tolerance should be orthogonal to mobile agents and its mechanisms transparent to the agent owner. Unfortunately, complete transparency is difficult to achieve and the user-defined agent, i.e., the part that defines the application-specific operations of the agent, needs to interact with the fault tolerance mechanisms. While in single-agent execution, for instance, an agent just needs to specify the next place it moves to, our fault-tolerant agent execution generally[4] requires a set of destination places for the next

[3] In the following, the term "consensus" denotes DIV Consensus unless explicitly stated otherwise.

[4] Except in the particular case of the pipelined mode (see Section 2.3).

stage ($\mathcal{M}_{i+1}$). Clearly, the agent is aware of the replication and complete transparency is no more possible. Moreover, ensuring fault tolerance adds another phase to the agent execution: the *commit/abort phase*. In Section 2, we mention that imperfect failure detection may lead to a violation of the exactly-once property of mobile agent execution. Solving an agreement problem prevents multiple executions of the agent by deciding on a primary p_i^{prim}. Only the primary commits the operations, while all other places that have executed the agent as well must abort/undo the agent operations. Hence we need a commit/abort phase. The semantics of this phase depend on the agent operations. For instance, database transactions need to be committed or aborted (or rolled back, depending on the database), while idempotent operations generally do not require any further action.

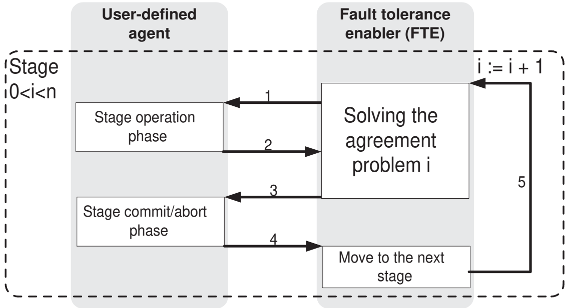

Figure 5. Phases of a fault-tolerant mobile agent execution and interaction with the FTE.

We propose an architecture that isolates the fault tolerance mechanisms in a component called *Fault Tolerance Enabler* (FTE). Figure 5 shows the flow of interaction between the FTE and the user-defined agent. This interaction occurs during the stage operation and the commit/abort phase. The FTE groups the fault tolerance mechanisms, while the user-defined agent contains the application-specific part. At each stage S_i $(0 < i < n)$, the FTE solves the stage agreement problem. Depending on the outcome of the agreement, the operations performed in the stage operation phase (see Figure 5, arrow 1 and 2) are either committed or aborted (arrow 3 and 4). Finally, the FTE moves the agent to the set of places in $\mathcal{M}_{i+1}$ (arrow 5), which are computed by the user-defined agent and returned as the result of the stage operation phase (see Section 2.4).

We can identify two approaches related to the location of the FTE: the *agent-dependent* (FTE with the agent) and the *place-dependent* approach (FTE with the places). Our system uses the agent-based approach, which is presented in more detail in the next section. We only briefly discuss the place-dependent approach.

3.2 Agent-Dependent Approach

In the agent-dependent approach, the FTE is integrated into the agent and travels with it. Only one instance of the FTE exists per agent. It is initialized by the user-defined agent at the agent source, and triggers the termination phase of the user-defined agent at the agent destination. The interaction of a user-defined agent with the FTE creates a fault-tolerant mobile agent. Hence, the replication mechanisms are completely transparent to the places; the agent appears to the place as a normal agent. Consequently, existing mobile agent platforms do not need to be modified. However, we redefine the way agents are created and moved. Instead of programming the agent against the proprietary mobile agent platform API, the agent uses the functionality of the FTE-API (see Figure 6). The FTE then addresses issues such as fault tolerance and mobility. For instance, in ObjectSpace's Voyager mobile agent platform [7], an agent moves with a call to the `move` method. Beside the destination, the `move` method also accepts the name of the method to be called upon arrival on the destination place. In our approach, the callback method `doStageOperation` in the FTE-API is invoked whenever the agent arrives at a new destination. The next destination places are returned as a result of the execution of method `doStageOperation`. These changes are straightforward and, in our opinion, simplify the notion of an agent.

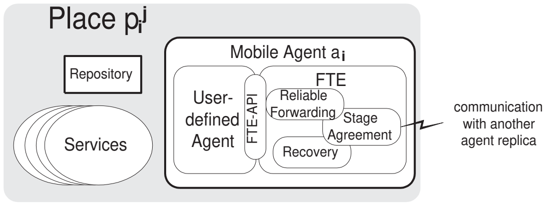

Figure 6. Agent-dependent approach: architecture of the fault-tolerant mobile agent framework.

Figure 6 shows the architecture of the agent-dependent approach. The FTE is composed of a *stage agreement* component (implementing consensus), a *reliable forwarding* component (responsible for the agent forwarding to the next stage), and a *recovery* component. The latter handles the recovery in case the agent fails or arrives late at a place (see Section 2.6). Finally, the *repository* is a location where place specific fault tolerance information can be stored temporarily. This location is agent platform dependent, but typically corresponds to some sort of local repository, such as the Voyager directory. For convenience, we require that such repositories allow other agents at place p_i to remotely access some information at another place p_k $(k \neq i)$. If this is not the case, an agent needs to be defined that acts as a proxy between the local directory and the fault-tolerant agents.

3.3 Place-Dependent Approach

In the place-dependent approach, the FTE is provided by the mobile agent platform, e.g., [6, 10]. Here, fault tolerance is built into the places, and a new instance of the FTE is created and executes at every stage of the agent execution. A disadvantage of the place-dependent approach is the need to modify existing proprietary mobile agent platforms. In particular, the installed base of mobile agent platforms needs to be replaced by platforms that all use the same fault tolerance mechanisms, which is problematic. Moreover, providing the fault tolerance mechanisms locally on a place may lead to versioning problems.

On the other hand, the FTE can be reused if two agents a and b execute on a similar set of places ($\mathcal{M}_i$) at stage S_i. However, the performance gain is small, as we believe that the sets $\mathcal{M}_i$ for an agent a and $\mathcal{M}_j$ for an agent b are generally not identical. Another advantage of the place-dependent approach is that it allows the places to selectively instantiate the agent replicas a_i^j when needed. Indeed, only agent replicas are instantiated whose stage operation phase is really executed. Nevertheless, each place runs an instance of the FTE per agent replica a_i^j, whether the agent replica a_i^j itself is instantiated or not, in order to participate in the stage fault tolerance protocol for a_i (i.e., the consensus algorithm). Since the FTE is located at the places, it does not need to be transported with the agents, thus limiting the size of the agent and improving transmission performance.

4 Implementation Issues

This section describes the implementation of FATOMAS. As indicated in Section 3, we build fault tolerance on top of an existing mobile agent platform, without modifying existing code. We use ObjectSpace's Voyager v3.1.2 [7] as the Java mobile agent platform. In the following, we first discuss the implementation of the FTE in Section 4.1. Section 4.2 then explains the problem of deadlock and shows a way to prevent it.

4.1 FTE

The interface of the FTE to the user-defined agent, i.e., the FTE-API (see Figure 6), consists of a single method, called `start`. Invoking this method causes the FTE to take over the responsibility for executing the agent. As shown in Section 3, the FTE interacts with the user-defined agents in the stage operation phase and in the commit/abort phase. For this purpose, the user-defined agent implements the callback method `doStageOperation`, which represents the stage operation phase and returns the set of destination places at the next stage. The callback methods `commit` and `abort` implement the commit/abort phase. The next subsection discusses in more details how a stage execution works.

4.1.1 Stage Execution

A stage execution works as follows: On arrival on place p_i^j, the agent replica a_i^j (more specifically the FTE) immediately starts executing consensus.

The communication among the agent replicas a_i^j is currently based on the communication means of Voyager, more specifically VoyagerSpaces for multicasts, and Voyager remote method invocation for point-to-point communication [7]. Remote method invocations are synchronous calls in FATOMAS and return an exception if the communication link is broken or the receiver is not available. These exceptions are caught and the message is confided to a dedicated sender thread that keeps on resending the message until it is successfully delivered or the stage execution has terminated. Replica agents that arrive when the other replica agents are already done with the stage execution run the recovery protocol.

When the consensus algorithm decides, the FTE stores the decision value in a local repository (see Figure 6). Actually, only part of the decision is stored, i.e., the primary's ID p_i^{prim}. This information must be kept until all participants in a stage execution, i.e., $\mathcal{M}_i$, are aware of the result. In particular, participants that have crashed during consensus and are assumed to recover again need to learn about the primary to decide whether to commit or abort the agent's operation on their place. However, it is not necessary to forward the agent to the next stage, as the agent execution may well have terminated in the meantime. After a certain time, the decision value is discarded. Selecting a timeout value that is sufficiently large, the probability of erroneously discarding an entry becomes very small and thus negligible.

Having stored the decision value in the repository, the FTE either calls `commit` or `abort` on the user-defined agent, depending on the decision value, more specifically, on the p_i^{prim} value of the decision (Section 2.4). Finally, the FTE forwards the agent to the next stage as described in the next section.

4.1.2 Reliably Forwarding the Agent

Having solved consensus at stage S_i, the agent needs to be forwarded reliably to members M_{i+1} of the next stage. To assure reliable forwarding, each participant of stage S_i sends a clone of the agent to the participants in $\mathcal{M}_{i+1} \setminus \mathcal{M}_i$.

If $\mathcal{M}_{i+1} \cap \mathcal{M}_i = \emptyset$, i.e., the places at two consecutive stages S_i and S_{i+1} form disjoint sets, the simplest solution to reliable forwarding consists in sending $|\mathcal{M}_i| * |\mathcal{M}_{i+1}|$ agents (see Figure 7). However, to reduce the communication overhead, we chose the following optimistic approach: the agent a_{i+1} is sent to each places in $\mathcal{M}_{i+1}$ only by the agent a_i at the place p_i^{prim}. The other agents $a_i^j \neq a_i^{prim}$ simply verify whether the agent a_{i+1} has arrived at the places in $\mathcal{M}_{i+1}$ by remotely accessing the corresponding value in the repository on the places in $\mathcal{M}_{i+1}$. If an entry for the agent a_{i+1}^j already exists, the agent a_{i+1}^j has suc-

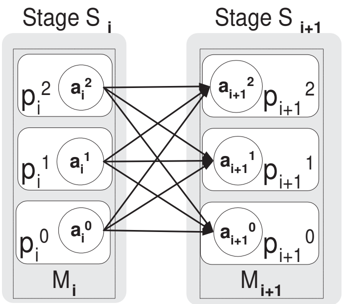

Figure 7. Reliably forwarding the agent from S_i to S_{i+1}.

cessfully arrived, otherwise, the agent a_{i+1}^j is cloned and sent to this place. In other words, instead of a priori always sending the entire agent, a small message is sent to check the need for sending the entire agent. This approach is optimistic, since it assumes that in most cases the agents arrive at their destinations. Even though the performance gain for a single agent is not great, the communication overhead is reduced for large agents, as discussed in Section 5.5.2. If an agent fails to arrive at its destination, because either (1) the sender place failed or (2) the agent was lost during transmission, agent forwarding leads to additional latency.

4.1.3 Recovery

Even though recovery and non-simultaneous agent arrival can be handled in the same way (see Section 2.6), our prototype distinguishes between the two problems: a delayed agent a_i^j takes part in the running instance of consensus (except if consensus has finished already), whereas a recovering agent does not. A recovering agent a_i^k requests the decision value of the consensus, more specifically p_i^{prim}, once it is available. Based on p_i^{prim}, a_i^k either commits or aborts its stage operations and can thus recover into a consistent state.

A recovering place that failed in stage S_i takes again part in the mobile agent execution at any stage $S_l (l > i)$ (if it is in $\mathcal{M}_l$) as well as in the execution of any other agent. For this case, no particular recovery algorithm is needed.

4.2 Interaction of FTE and User-Defined Agent: The Deadlock Problem

A deadlock may occur between two different agents a_i and b_i, if they happen to share the same places. This deadlock is a consequence of the interaction between the DIV Consensus algorithm and the stage operations. Consider for example the two replicas a_i^0, a_i^1 of agent a_i and the two replicas b_i^0, b_i^1 of agent b_i, were a_i^0, b_i^0 share the place p_i^0 and a_i^1, b_i^1 the place p_i^1 (Figure 8).[5] Assume further that agent a_i performs stage operation O_a (either on p_i^0 or on p_i^1), and b_i performs stage operation O_b (either on p_i^0 or on p_i^1), and that the two operations O_a and O_b access the same data item. Accessing the data item requires locking of the data, and the data must remain locked until the decision of the consensus of stage S_i is known.

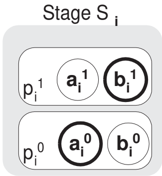

Figure 8. Deadlock between the agents a and b at stage S_i.

Let us assume first the solution where we have one single thread per agent a_i^j, denoted by $Thr(a_i^j)$, for (1) executing the consensus on place p_i^j and (2) performing the operation O on p_i^j. This can lead to the following wait-for dependencies:

- Because of the distributed consensus algorithm, we can have $Thr(a_i^0) \rightarrow Thr(a_i^1)$, where $\rightarrow$ stands for "waits-for".[6]
- For the same reason, we can have $Thr(b_i^1) \rightarrow Thr(b_i^0)$.
- Because of the data locking, we can have $Thr(b_i^0) \rightarrow Thr(a_i^0)$ (if $Thr(a_i^0)$ locks the data item of place p_i^0 before $Thr(b_i^0)$).
- For the same reason, we can have $Thr(a_i^1) \rightarrow Thr(b_i^1)$.

We have here a cycle in the wait-for-graph, i.e., a deadlock, as shown in Figure 9 a). Deadlocks can be prevented by having two threads per agent a_i^j: one thread, denoted by $ConsThr(a_i^j)$, for executing the consensus on place p_i^j, and another thread, denoted by $OpThr(a_i^j)$ for executing the stage operation on place p_i^j. The above wait-for dependencies become:

- $ConsThr(a_i^0) \rightarrow ConsThr(a_i^1)$.
- $ConsThr(b_i^1) \rightarrow ConsThr(b_i^0)$.
- $OpThr(b_i^0) \rightarrow OpThr(a_i^0)$.
- $OpThr(a_i^1) \rightarrow OpThr(b_i^1)$.

Obviously the cycle in the wait-for-graph, and the deadlock, have disappeared. Another solution that requires only one thread is the use of timed locks[7] in data accesses.

[5] Actually, to be accurate, we should have three replicas in our example. However, for simplicity, we consider only two replicas. Anyone familiar with the consensus algorithms based on the rotating coordinator paradigm and $\diamond S$ should mentally replace everywhere "two" by "three".

[6] a_i^0, the coordinator of the first round has been suspected, and a_i^1 is the coordinator of the second round.

[7] A *timed lock* is a lock where a thread is blocked until either it is granted the lock, or a timeout value is reached. The thread can then take the corresponding actions and potentially retry to acquire the lock.

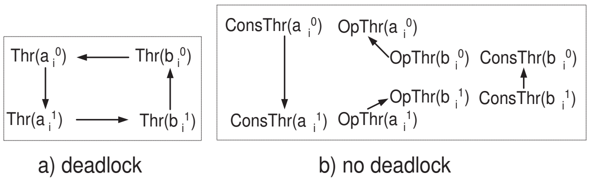

Figure 9. Wait-for-graphs for (a) the single thread and (b) the double thread case.

5 Performance Evaluation

This section evaluates the performance of FATOMAS. Our evaluation uses the example agent presented in the next section.

5.1 Example: A Fault-Tolerant Agent Accessing Counters

To measure the performance of FATOMAS, we use a simple service running on every place: a counter. Accesses to this counter are performed as local transactions, via the three methods: `increment` (to increment the value of the counter), `commit` (to commit the modifications), and `abort` (if the modifications need to be undone). A call to method `increment` locks the counter; the lock is only released after a call to either `commit` or `abort`.

Our test consists in sending a number of agents that increment the value of the counter at each stage of the execution. Each agent starts at the agent source and returns to the agent source, which allows to measure its round-trip time. 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. For a fair comparison, we used the same approach for the single agent case (no replication).

Moreover, we assume that the Java class files are locally available on each place. Clearly, this is a simplification, as the class files do not need to be transported with the agent. Remote class loading adds an additional costs because the classes need to be transported with the agent and then loaded into the virtual machine. The size of the class files for a single agent is about 8KBytes, for the replicated agent 50KBytes (including the classes for the FTE). An upper bound on these costs is modeled in our experiments by increasing the size of our agent. However, these experiments also contain the additional costs of executing consensus for an agent of the corresponding size. These are costs that are not relevant in the case of remote class loading. The class files are only transported once between the places of consecutive stages, but not between places of the same stage. As our chosen platform does not properly support remote class loading, we plan to port FATOMAS to another mobile agent platform to test the performance with remote class loading enabled.

5.2 Experimental Setup

Our performance tests are run on seven AIX machines (PowerPC 233 MHz processor, 256MByte of RAM). The machines are connected by either 100Mbps Ethernet or 16Mbps Tokenring; they are on 3 different subnets. As our evaluation results are in the area of hundreds of milliseconds, the difference in network bandwidth is negligible. The influence of the different subnets does not turn out to be significant either.

5.3 Costs of Replication

We measure two aspects of the replication costs: (1) the overhead of the replication mechanism, by considering replication degree 1, and (2) the costs of replication degree 3. The replication degree denotes the number of places at a stage and is an indicator of the number of failures the algorithm tolerates at a stage. Because of the assumption for our consensus algorithm, i.e., a majority of agent replicas do not fail (see Section 2.5), replication degree 3 handles one failure, and replication degree 5 would handle 2 failures. The results of these measurements are given in Table 1. They 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 much lower than 5%. However, for very few results, it went up to 15%. As a mobile agent execution combines agent forwarding and consensus, minor variations on network load and load on the AIX machine have a considerable influence on the execution time of a mobile agent.

The first line in Table 1 shows the costs for a single agent, i.e., a traditional Voyager agent, that performs exactly the same task as the replicated agent. The single FTE-agent (line 2 in Table 1) uses the replicated agent's code to execute in a single agent mode. Compared with the previous line, the second line shows the overhead of the replication mechanism (increased agent state adding to the communication costs, increased computing time). The results show that the replication mechanisms add about a 30% overhead compared to a single agent. The overhead is lower (18%) in the case of three stages, as no communication between intermediate stages occurs.

A replicated agent that is able to tolerate one failure at a stage is three to four times more expensive than a single agent (line 3). The increase in the agent execution time is mainly caused by the additional communication costs of agent forwarding and consensus. Indeed, consider for instance the single agent execution on 4 stages: there are here 3 messages in the critical path. On the other hand, with replication there are 12 messages in the critical path in the most favorable scenario. We suspect Voyager communication to be rather inefficient. Nevertheless, the over-

Type of Agent	3 stages		4 stages		5 stages	
Single agent (666 bytes)	793	100%	1089	100%	1546	100%
Single FTE-agent, degree 1 (1440 bytes)	939	118%	1427	131%	2004	130%
Replicated FTE-agent, degree 3	2369	290%	4375	402%	6470	418%
Replicated FTE-agent, degree 3, with failure (timeout = 10000)	10000 + 2445	1569%	10000 + 4631	1344%	10000 + 6299	1054%

Table 1. Costs of replication degree 1 and 3 in milliseconds compared to the single agent.

head of the fault tolerance mechanisms seems reasonable, considering the guarantees the fault tolerance mechanisms provide: non-blocking and exactly-once mobile agent execution. Moreover, in our experiment the execution time of the agent's stage operation is less than 5ms and therefore not significant. Clearly, the larger the execution time of the agent's stage operation is, the smaller the ratio of the overhead between the single agent and the replicated agent becomes.

Finally, the last line shows the execution costs when the coordinator fails. For this purpose, we force agent replica a_i^1 to crash in exactly the situation presented in Figure 4 (stage 2). The main part of the costs stems from the selected timeout value in the failure detection mechanisms for consensus (timeout = 10000ms). A more aggressive timeout value allows to considerably speed up the agent execution, but increases the risk of false suspicions.

Table 2 indicates an upper bound on the costs that can be attributed to consensus. It shows the costs of instantiating the consensus objects and of running the consensus algorithm. The test measures the costs of consensus at the intermediate stage of a mobile agent execution with three stages. The agent source sends the agent sequentially to the three places of the intermediate stage. As in agent forwarding between intermediate stages one can assume some degree of concurrency these results show an upper bound value. They highlight a significant difference of the costs among the places, caused by asynchronous agent arrival as discussed in Section 2.6 and by the intrinsic properties of our consensus algorithm.

The results confirm the obvious expectation that the size of the agent has an impact on the costs of consensus. We discuss this impact in more detail in Section 5.4.

agent size in byte	place p_i^0	place p_i^1	place p_i^2
1440	30718	570	1264
11440	4949	3237	6156
51440	25846	15819	26446
101440	49855	31127	51222

Table 2. Costs of consensus in milliseconds for a replication FTE-agent of degree 3.

A particular case arises for small agents (first line in Table 2). Because of asynchronous agent propagation, p_i^1 and p_i^2 solve consensus and are forwarded to the next stage before p_i^0 establishes the communication with them. Therefore, a_i^0 uses the recovery mechanisms to learn about the result of consensus. This explains the high value for p_i^0, which is mainly caused by timeouts.

5.4 Influence of the Size of the Agent

As already indicated in the previous section, the size of the agent has a considerable impact on the performance of the fault-tolerant mobile agent execution. To measure this impact, the agent carries a Byte array of variable length, that is used to increase the size of the agent. As the results in Figure 10 show, the execution time of the agent increases linearly with increasing size of the agent. Compared to the single agent, the slope of the curve for the replicated agent is steeper. Table 2 indicates the part of the costs that can be attributed to consensus. For instance, with the agent size of 11440 bytes, consensus needs 3.3 seconds at each intermediate stage. Figure 10 indicates that the costs for the entire agent execution is 13.4 seconds. From this we conclude that the communication overhead is about 7 seconds. The case of the forwarding optimization will be discussed in Section 5.5.2.

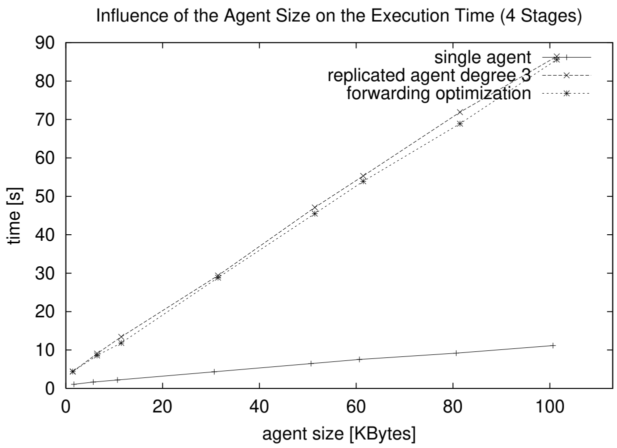

Figure 10. The costs of single and replicated agent execution with increasing agent size (4 stages).

5.5 Optimizations

In this section we present two optimizations: pipelined mode and forwarding optimization.

5.5.1 Pipelined Mode

We briefly introduced the pipelined mode in Section 2.3. It results in a reduced number of messages (i.e., forwarded agents), as the agent only needs to be forwarded to one new place of the next stage. This reduced number of messages does not entirely show in the performance gain, because our algorithm waits only for the reception of the first message. Reducing the number of messages, however, has a great impact on the underlying communication infrastructure.

Nevertheless, Figure 11 shows that the pipelined mode has a lower execution time than the normal replicated agent. The performance gains increase with increasing agent size, as one would expect. In this test, we used an agent that visits 8 stages, including agent source and destination.

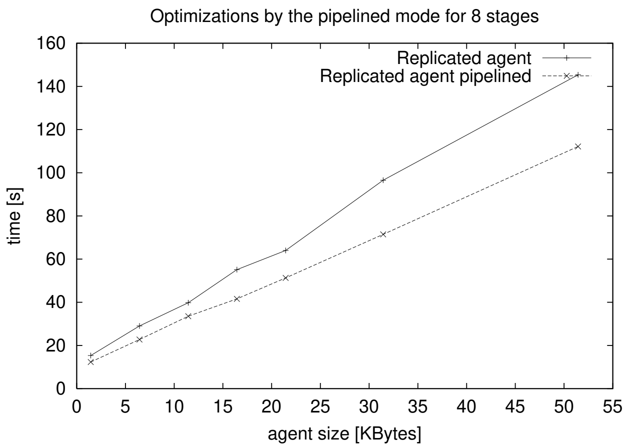

Figure 11. Performance gain with the pipelined mode for 8 stages.

5.5.2 Forwarding Optimization

Similar to the pipelined mode, this optimization addresses also the communication overhead (Section 4.1.2). Even though the number of bytes transfered is reduced with this approach, in particular for large agents, the performance gain for the agent execution itself is small (see Figure 10), as our algorithm waits only for the arrival of the first copy of the agent.

6 Related Work

Recently, fault-tolerant mobile agent execution has been a very active field of research. We distinguish between *spatial-replication-based (SRB)* approaches [2, 5, 9, 11] and *temporal-replication-based (TRB)* [6, 10] approaches; our paper advocates a SRB approach.

A SRB approach sends the replicas of the agent at the same time to a set $\mathcal{M}$ of places of the next stage. Having multiple replicas of the agent allows to mask failures and to proceed despite of failing replicas. To ensure the exactly-once execution property, SRB schemes need to solve an agreement problem. Current SRB schemes assume reliable failure detection [5], are based on complex models [2, 9, 11], or block even on a single place failure [9, 11] (see [8] for an in-depth discussion). Our approach, which is based on an easily understandable model, does not assume reliable failure detection and prevents blocking. To our knowledge, our work is the first to perform an evaluation of a SRB approach for fault-tolerant mobile agents. In [10], Silva et al. compare their TRB approach with some basic performance results from their partial implementation of [9]. However, no details about the implementation are presented.

A TRB approach, on the other hand, attempts an agent execution on one place. If the execution fails, the agent is sent to another place. Incorrect failure suspicion may lead to duplicate agents and thus to a violation of the exactly-once execution property. Generally, duplicate agents are only detected at the agent destination [6, 10]. Consequently, accessed data items need to remain locked until the execution of the current agent finishes. If this is not the case, other agents may read incorrect data. At the end of the agent execution, the operations of the valid mobile agent need to be committed while those of the superfluous duplicate agents need to be aborted or undone. As incorrect failure suspicions may happen at any point, the commit/abort mechanism is always needed, even though at the end of the agent execution no duplicate agent may have occurred. In other words, when the agent execution is done, either a number of messages or another agent need to be sent to commit and/or abort all operations. This additional overhead, that, to our understanding, seems not taken into account in the performance evaluation in [6, 10], results in reduced overall system throughput. The JAMES platform [10] uses a replicated lookup directory to prevent certain duplicate agents. However, such a lookup directory violates to some extent the autonomy property of a mobile agent. In contrary, our approach does not need to run a global commit/abort protocol. Instead, undoing failed operations happens on a per-stage-basis and therefore does not create dependencies among stages. Locked data items are freed earlier, improving system throughput. In addition, undoing duplicate agents is difficult and involves undoing the agent's operations. Performing the undo operation on a language level [6] requires specific knowledge about the way applications handle rollbacks. To our understanding, [6] assumes standard interfaces for rollbacks and undo operations. In contrary, we leave the undo operation to the application, which has the best knowledge about the services it is using and their way to handle rollbacks.

To the best of our knowledge, our approach is the first to propose a platform-independent architecture, the so-called agent-dependent approach (see Section 3.2). The work of [2, 5, 6, 9, 10] all use a place-dependent approach. As their approaches for fault-tolerant mobile agent execution are inherently different from ours, the place-dependent architecture may be better suited for them. Indeed, Mohindra [6], for instance, defines a new scripting language, with fault tolerance tightly integrated into language constructs. The TRB approach of [10] also seems to benefit from a place-based approach. Indeed, a failure causes the previous place to send the agent to another place to circumvent the failed place. It is difficult, even though probably not impossible, to achieve the same behavior with an agent-dependent approach. Nevertheless, contrary to our agent-dependent approach, the place-dependent approach has the major drawback to require modification to the existing mobile agent platform.

7 Conclusion and Future Work

In this paper we have presented FATOMAS, a Java-based fault-tolerant mobile agent system. Contrary to [6, 10] we use a spatial-replication-based (SRB) approach to fault-tolerant mobile agent execution. Our paper presents the first detailed evaluation of such a SRB approach, as [2, 5, 9] do not give any performance results.

We have also introduced a novel architectural approach for fault-tolerant mobile agents: the agent-dependent approach. Instead of integrating the fault tolerance mechanisms into the mobile agent platform, the fault tolerance mechanisms travel with the mobile agent. This has the important advantage that existing mobile agent platforms do not need to be modified. Indeed, our framework builds entirely on top of ObjectSpace's Voyager mobile agent platform. Therefore, mobile agents can even travel to places that do not support fault tolerance. We think that these advantages outweight the performance penalty FATOMAS pays compared to the place-dependent approach.

Our results show the costs of fault tolerance compared to a single, non-replicated, agent. These costs depend on the number of stages as well as on the size of the agent. Increasing the size of the agent and the number of stages also increases the execution time for a mobile agent. These costs are reasonable considering the guarantees FATOMAS provides: non-blocking and exactly-once mobile agent execution. Finally, we have presented the pipelined mode and the agent forwarding optimization, that allow to alleviate these costs to a certain extent.

In the future, we plan (1) to port FATOMAS to other mobile agent platforms to further validate our architecture and (2) to improve the performance of FATOMAS. To achieve the latter, we plan to investigate the efficiency of other communication packages. As a large part of the overhead is caused by an increased number of communication messages (see Section 5), the efficiency of the communication package is instrumental to the performance of FATOMAS. Finally, we want to study the effect of remote class loading on the performance of our system.

References

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

[2] F. de 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.

[3] X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proc. of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 43–50, West Lafayette, IN, Oct. 1998.

[4] 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.

[5] D. Johansen, K. Marzullo, F. B. Schneider, K. Jacobsen, and D. Zagorodnov. NAP: Practical fault-tolerance for itinerant computations. In *Proc. of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS)*, Austin, Texas, June 1999.

[6] A. Mohindra, A. Purakayastha, and P. Thati. Exploiting nondeterminism for reliability of mobile agent systems. In *Proc. of the International Conference on Dependable Systems and Networks*, pages 144–153, New York, June 2000.

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

[8] 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.

[9] 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.

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

[11] M. Strasser, K. Rothermel, and C. Maihöfer. Providing reliable agents for electronic commerce. In W. Lamersdorf and M. Merz, editors, *Proc. of the Int. Conference on Trends in Distributed Systems for Electronic Commerce (TREC)*, LNCS 1402, pages 241–253, Hamburg, Germany, June 1998. Springer Verlag.