Fault-Tolerant Mobile Agent Execution

Stefan Pleisch and André Schiper, *Member*, *IEEE*

Abstract—Mobile agents have attracted considerable interest in recent years. In the context of mobile agents, fault tolerance is crucial to enable the integration of mobile agent technology into today's business applications. This article identifies two important properties for fault-tolerant mobile agent execution: nonblocking and exactly-once. Nonblocking ensures that the agent execution can proceed despite a single failure of the agent or the machine, for instance. Replication is the generally adopted mechanism to prevent blocking, but may lead to multiple executions of the agent (i.e., a violation of the exactly-once property), which is undesirable with operations that have side effects. Hence, we propose that fault-tolerant mobile agent execution be modeled as a sequence of agreement problems. Our approach is nonblocking and ensures exactly-once execution. FATOMAS, our prototype fault-tolerant mobile agent system, implements our approach. Its performance evaluation illustrates the overhead of the replication mechanisms.

Index Terms—Mobile agents, fault tolerance, nonblocking execution, exactly-once execution, agreement problem.

1 Introduction

In recent years, the field of mobile agents, i.e., programs that act autonomously and travel through a network of heterogeneous machines, has attracted considerable attention. Mobile agent technology has been considered for a variety of applications [7], [8], [16] such as systems and network management [4], [14], mobile computing [28], information retrieval [29], and e-commerce [17]. However, before mobile agent technology can appear at the core of tomorrow's business applications, reliability mechanisms, among other things, need to be established for mobile agents. Of these reliability mechanisms, *fault tolerance* is a mechanism of considerable importance.

Any software or hardware component in a distributed system may be subject to failures. A single failing component (e.g., agent or machine) may prevent the agent from proceeding with its execution. Worse yet, the current state of the agent and even its code may be lost. Assume, for instance, that an agent has bought a book at a virtual book shop and currently executes on the server of a clothing shop, which fails by crashing. The agent execution thus cannot proceed because the agent has failed with the machine; the agent execution is blocked. In the absence of recovery mechanisms such as logging, even the information about the acquired book is lost. The agent owner, i.e., the person or application that has configured the agent, finds that its agent has not returned yet. In an asynchronous system such as the Internet, it is impossible to detect correctly whether the agent has failed or whether it is merely slow [10]. Indeed, no boundaries exist on relative processor speed and communication time in an asynchronous system. Consequently, the agent owner cannot determine whether the agent has failed or is delayed by slow processors or communication links.

Replication prevents blocking, but may lead to multiple executions of the agent, i.e., to a violation of the exactly-once execution property. Although multiple executions of the agent are allowed in the context of idempotent operations, they are undesirable with nonidempotent operations. Assume, for instance, that a failed agent replica has withdrawn money from a bank account. Although the bank account may still reflect the money withdrawal, the money itself is lost. Executing a second agent replica results in two money withdrawals, which is clearly undesirable to the agent owner. In contrast, reading the balance of a bank account is an idempotent operation: It can be executed multiple times without influencing the state of the bank account or the state of the agent (provided that the balance has not changed in the meantime).

Although fault-tolerant mobile agent approaches exist, we have found that current solutions are either complex and thus difficult to prove correct [23], [3], make limiting assumptions such as correct failure detection [15] or strict timing constraints [20], or only release resources at the end of the mobile agent execution [18], [26]. In this article, we model fault-tolerant mobile agent execution as a sequence of agreement problems. Our approach prevents blocking in the mobile agent execution and ensures the exactly-once execution property. We validate our approach with the implementation of FATOMAS, a fault-tolerant mobile agent system, and give important results of our performance evaluation.

The rest of this article is structured as follows: Section 2 presents the model used throughout the article. In Section 3, we identify the properties for fault-tolerant mobile agent execution: *nonblocking* and *exactly-once*. The specification of fault-tolerant mobile agent execution is presented in Section 4. Section 5 identifies two building blocks that solve the problem specified in the previous section. The prototype implementation of our approach and a performance evaluation are presented in Section 6. Our approach is also applicable to agent replicas executing on machines that are replicas among themselves (Section 7). Section 8 relates our

• *S. Pleisch is with IBM Research, Zurich Research Laboratory, CH-8803 Rüschlikon, Switzerland. E-mail: pstefan@acm.org.*
• *A. Schiper is with the Distributed Systems Laboratory, Swiss Federal Institute of Technology (EPFL), CH-1015 Lausanne, Switzerland. E-mail: Andre_Schiper@epfl.ch.*

Manuscript received 1 Feb. 2002; revised 7 Sept. 2002; accepted 20 Sept. 2002. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 117439.

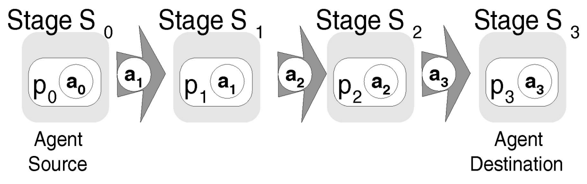


Fig. 1. Model of a mobile agent execution with four stages.

approach to existing work and, finally, Section 9 concludes the article.

2 Model

We assume an asynchronous distributed system, i.e., there are no bounds on transmission delays of messages or on relative process speeds. An example of an asynchronous system is the Internet. Processes communicate via message passing over a fully connected network.

2.1 Mobile Agent

A mobile agent executes on a sequence of machines, where a *place*[1] $p_i\ (0 \leq i \leq n)$ provides the logical execution environment for the agent. Each place runs a set of services, which together compose the state of the place. For simplicity, we say that the agent "accesses the state of the place," although access occurs through a service running on the place. Executing the agent at a place p_i 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 the agent source and destination (i.e., $p_0, p_1, \ldots, p_n$) is called the itinerary of a mobile agent. Whereas a *static* itinerary is entirely defined 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 Fig. 1). Each stage action sa_i consists of potentially multiple operations $op_0, op_1, \ldots$. 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 execute on place p_i. The execution of a_i on place p_i 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, i.e., are nonidempotent). We denote the resulting agent a_{i+1}. Place p_i forwards a_{i+1} to p_{i+1} (for $i < n$).

2.2 Failures

Machines, places, or agents can fail and recover later. A component that has failed but not yet recovered is called *down*; otherwise, it is *up*. If it is eventually permanently up, it is called *good* [2]. In this paper, we focus on crash failures (i.e., processes prematurely halt). Benign and 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. We do not consider deterministic, repetitive programming errors (i.e., programming errors that occur on all agent replicas or places) in the code or the place as relevant failures in this context.[2] Finally, a link failure causes the loss of the messages or agents currently in transmission on this link and may lead to network partitioning. We assume that link failures (and network partitions) are not permanent.

1. Also called *landing pad* in [15] or *agency* in [26].

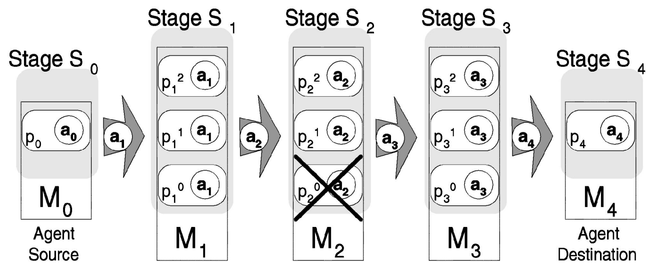


Fig. 2. Agent execution with redundant places, where place p_2^0 fails. The redundant places (i.e., p_2^1 and p_2^2) mask the place failure.

3 Overview

3.1 The Problem of Blocking

The failure of a component (i.e., agent, place, machine, or communication link) can lead to blocking in the mobile agent execution. Assume, for instance, that place p_2 fails while executing a_2 (see Fig. 1). While p_2 is down, the execution of the mobile agent cannot proceed, i.e., it is *blocked*. Blocking occurs if a single failure prevents the execution from proceeding. In contrast, an execution is *nonblocking* if it can proceed despite a single failure. The blocked mobile agent execution can only continue when the failed component recovers. This requires that a recovery mechanism be in place, which allows the failed component to be recovered. If no recovery mechanism exists, then the agent's state and, potentially, even its code may be lost. In the following, we assume that such a recovery mechanism exists (e.g., based on logging [13]).

3.2 Introducing Replication

Replication prevents blocking. Instead of sending the agent to one place at the next stage, agent replicas are sent to a set $\mathcal{M}_i$ of places $p_i^0, p_i^1, \ldots$ (see Fig. 2). We denote by a_i^j the agent replica of a_i executing on place p_i^j, but will omit the superscripted index if the meaning is clear from the context. Although a place may crash (i.e., p_2^0 in Fig. 2), the agent execution does not block. Indeed, p_2^1 can take over the execution of a_2 and thus prevent blocking. Note that the execution at stages S_0 and S_4 is not replicated as the agent is under the control of the user. Moreover, the agent is only configured at the agent source and presents the results to the agent owner at the agent destination. Hence, replication is not needed at these stages.

Despite agent replication, network partitions can still prevent the progress of the agent. Indeed, if the network is partitioned such that all places currently executing the agent at stage S_i are in one partition and the places of stage S_{i+1} are in another partition, the agent cannot proceed with

2. Johansen et al. [15] introduce a so-called *rally point*. On detection of a catastrophic failure, the agent is sent to the rally point, where the agent owner can debug it.

its execution. Generally (especially in the Internet), multiple routing paths are possible for a message to arrive at its destination. Therefore, a link failure may not always lead to network partitioning. In the following, we assume that a single link failure merely partitions one place from the rest of the network. Clearly, this is a simplification, but it allows us to define blocking concisely (see Section 3.1). Indeed, in the approach presented in this article, progress in the agent execution is possible in a network partition that contains a majority of places. If no such partition exists, the execution is temporally interrupted until a majority partition is established again. Moreover, catastrophic failures may still cause the loss of the entire agent. A failure of all places in $\mathcal{M}_2$ (see Fig. 2), for instance, is such a catastrophic failure (assuming no recovery mechanism is in place). As no copy of a_2 is available any more, the agent a_2 is lost and, obviously, the agent execution can no longer proceed. In other words, replication does not solve all problems. The definition of nonblocking merely addresses single failures per stage as they cover most of the failures that occur in a realistic environment.

In the next section, we classify the places in $\mathcal{M}_i$ into iso-places and hetero-places according to their properties.

3.3 Place Properties

In Section 3.2, we have introduced replication as a way to overcome the problem of blocking in a mobile agent execution. Replication occurs on the agent level: The agent replicas execute on different places $p_i^j \in \mathcal{M}_i$ at a stage S_i (see Fig. 2). Depending on the relationship among these places, we distinguish between two classes of places: *iso-places* and *hetero-places*.

3.3.1 Iso-Places

Iso-places correspond to the traditional case of server replication: The set $\mathcal{M}_i$ consists of replica places, where all places reflect the same state. For instance, all places are provided by airline X and provide a ticket reservation system: Modifications to one place are visible to the others as well. Within the class of iso-places, we can further distinguish between places p_i^j where 1) the modifications are propagated by the places themselves or 2) where consistency is ensured by the agent replicas. In 1), the places are called *replicated iso-places*, whereas the places in 2) are called *independent iso-places*.[3] Executing a replicated agent on replicated iso-places leads to two levels of replication: server replication in the places and client replication on the agent level. The replication mechanisms can thereby be different at the two levels. Note that executing the mobile agent a_i on two replicated iso-places in $\mathcal{M}_i$ at stage S_i causes all iso-places in $\mathcal{M}_i$ to reflect the modifications twice.[4] Actually, the case of replicated iso-places is similar to a replicated client invoking a replicated server in traditional distributed systems. Indeed, the agent replicas correspond to a replicated client, whereas the replicated iso-places take the role of a replicated server.

In contrast, independent iso-places do not run any replication mechanism; rather, the agent replicas are executed on all independent iso-places in order to ensure that the independent iso-places are always in a consistent state.

3.3.2 Hetero-Places

Hetero-places are not replicas; rather, they correspond to a set $\mathcal{M}_i$ of places, of which all provide a similar service such as selling airline tickets from Zurich to New York. However, the places are provided by different airlines, e.g., airlines X, Y, and Z.

Finally, hetero-places with witnesses[5] are a generalization of hetero-places. Whereas hetero-places all provide the particular service (i.e., airline tickets from Zurich to New York), in hetero-places with witnesses, only a subset of the places provides the service. The others (i.e., the witnesses), although they can execute the agent replica, do not provide an airline ticket service to the agent replica and, thus, the service request of the agent replica fails on those places. The goal of witnesses is to help decide whether the execution on another (nonwitness) place has been successful. Witnesses allow the agent to be replicated, i.e., to ensure that the agent is not lost and proceeds with the execution. Assume, for instance, that $\mathcal{M}_i$ consists of a place of airline X and two witnesses. If airline X's place fails or is suspected, then the agent executes on a witness. The agent can then move to $\mathcal{M}_{i+1}$ that consists of a place of airline Y and two witnesses and can attempt to acquire a flight ticket from airline Y. Clearly, executing the agent on the witness could also cause the agent to decide to return immediately to its owner and no longer proceed with its execution.

In the following, we consider only hetero-places (and hetero-places with witnesses). The case of iso-places is discussed in Section 7.

3.4 Replication and the Exactly-Once Property

Replication allows us to prevent blocking. However, it can also lead to a violation of the *exactly-once* execution property. Indeed, the exactly-once property and nonblocking are closely related. Assume, for instance, that place p_i^0 fails after having partially executed agent a_i (see Fig. 3). After some time, p_i^1 detects the failure of p_i^0 and takes over the execution of a_i: The agent a_i has now (partially) executed multiple times. Consequently, upon recovery, place p_i^0 needs to undo the modifications performed by agent a_i. The same issues arise if only an agent replica fails, but not the place. In this case, modifications by the failed agent to the place state survive. As the agent is then executed on place p_i^1, modifications are applied twice (to p_i^0 and p_i^1). Replication of the agent thus leads to a violation of the exactly-once execution property of mobile agents. Consequently, the replication protocol of agents has to undo the modifications of a_i to the place p_i^0.

Another source for the violation of the exactly-once execution property is unreliable failure detection. Indeed, in an asynchronous system such as the Internet, no boundaries

3. In [21], replicated iso-places are called *nonintegrated* iso-places, whereas independent iso-places are called *integrated* iso-places.

4. Unless a mechanism (e.g., transaction IDs) is provided that prevents iso-places from executing the same operation twice. See Section 7 for a discussion of multiple executions of mobile agents in the context of iso-places.

5. Also called *exception nodes* in [27].

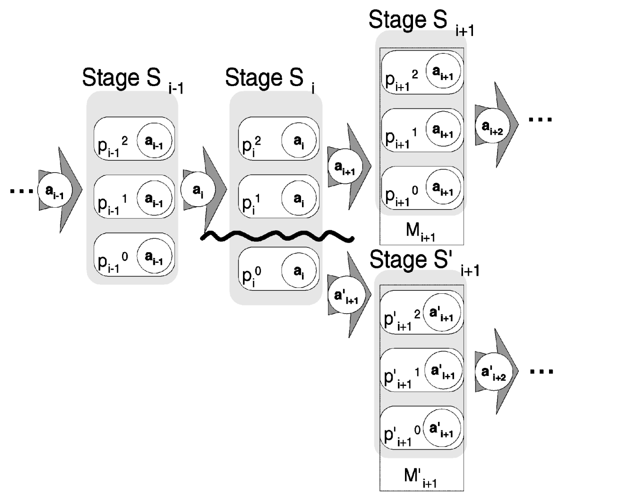


Fig. 3. Disagreeing stage agents potentially lead to a violation of the exactly-once property.

exist on communication delays or on relative process speeds. Hence, it is impossible to detect failures reliably [10]. Assume, for instance, that p_i^1 suspects p_i^0 has failed, when, in fact, p_i^0 has not (see Fig. 3). This may lead to two agents a_{i+1} and a'_{i+1}, which are potentially sent to different places for the next stages. Clearly, this is a violation of the exactly-once execution property.

In summary, a violation of the exactly-once execution property can occur 1) in the agent replicas and 2) at the places (or, rather, the services running on the places). Clearly, both instances are related in that a violation of the exactly-once execution property at the places is a consequence of multiple executions of the agent (e.g., a_i on p_i^0 and a_i on p_i^1).

3.5 Undoing Agent Actions

As mentioned in Section 2.1, executing an agent action on a place generally results in a modification of the agent state as well as the state of the place. Because of the particular failure dependency between agent and place, a crash of the agent may leave the machine and the place in an incorrect state (see Section 3.4). The same issue arises with incorrect failure suspicions.

Consequently, it must be possible to undo the agent actions on the place and adequate protocols must be devised that ensure the consistent state of machine and place even when the agent fails. Generally, two approaches are used: *optimistic* and *pessimistic* execution. With pessimistic execution, the stage actions of the agent are tentatively executed. The modifications are only made permanent when it is ensured that the agent neither crashes nor is erroneously suspected. Undoing the modifications in this context is simple: The modifications are ignored. Generally, pessimistic execution is based on the use of transactions, which lock the accessed data items. A lock protects a data item and grants exclusive access to the data item. All other agents that try to access the same data item have to wait until the agent that holds the lock has committed (i.e., made permanent) its modifications.

In contrast, with optimistic execution, modifications to the place are immediately made permanent and visible to other agents. Undoing modifications becomes a more complex task. For instance, undoing agents is difficult as the agent may have moved on in the meantime. In an asynchronous system, a message or undo agent sent after an agent may trace this agent, but never actually catch up with it. Undoing modifications performed by an agent becomes more difficult if multiple agents access the same services. Generally, modifications are visible to other agents when they are made permanent. Assume that agent a has modified the state of its local place and agent b has retrieved (part of) this state. Undoing the modifications of agent a may invalidate also the state of b. In the context of transactions, the notion of a *compensating transaction* [12], [11] has been introduced. Compensating transactions semantically undo transactions. With this concept, modifications to the data can immediately be committed. If they need to be aborted later, simply the compensating transaction is run.

Note that the approach presented in this article can support both optimistic and pessimistic execution. Without loss of generality, we focus on pessimistic execution in the rest of this article.

3.6 Exactly-Once in the Context of Hetero-Places

Hetero-places run no replication mechanism among themselves (see Section 3.3.2). Indeed, hetero-places are generally competitors that have no interest in sharing any information (i.e., communicating) among themselves. The hetero-places in $\mathcal{M}_i$ need not all execute the agent request; in contrast, the exactly-once execution property of the agent request needs to be enforced. Hence, if no failure and no false suspicions occur, only one hetero-place at a stage reflects the execution of the mobile agent at this stage and only its services have been invoked. However, although no replication occurs among hetero-places, the state of the agent is replicated among the agent replicas: All agent replicas know the place that has executed the agent replica.

Failures and false suspicions can lead to the execution of agent replicas on multiple hetero-places of $\mathcal{M}_i$ (see Section 3.4). Hence, multiple hetero-places reflect the modifications of the agent a_i. As the exactly-once property requires that only one place has executed the agent, the modifications on all but one place need to be undone (see Section 3.5). Because of the lack of communication, hetero-places cannot prevent multiple executions of the agent request themselves; they generally have no means to detect that another hetero-place has already executed the same request. Hence, the exactly-once property needs to be ensured by the agent replicas. More specifically, the agent replicas need to guarantee that, at the end of the agent execution, only one agent replica has executed the operations, while all others undo potential executions. For this purpose, recovering agent replicas a_i have to retrieve the identity of the place that has executed the agent replica at stage S_i. If this identity denotes a place other than the one running the recovering agent replica, all modifications by this agent replica prior to the crash are undone. However, this requires that agent, places, and machines be good, i.e., that they eventually be permanently up (see Section 2.2).

Executing an agent replica on a witness does not modify the state of the witness (see Section 3.3.2). Hence, a witness behaves similarly to a stateless server and partial executions of the agent replica do not need to be undone. In other words, the exactly-once execution property of the agent replicas can be violated on witnesses as the execution is idempotent. Failures of a witness or an agent running on a witness are thus handled much more efficiently than on a hetero-place or iso-place.

4 Fault-Tolerant Mobile Agent Execution As a Sequence of Agreement Problems

We claimed in Section 3 that replication of agents prevents blocking without depending on reliable failure detection. However, to enforce the exactly-once property of mobile agent execution, the replicas have to decide on a place that has executed the agent. This decision is modeled as an agreement problem.

4.1 Basic Agreement Problem

Despite the differences of hetero-places and hetero-places with witnesses, we give a specification of the problem that encompasses the two cases. The idea is to model the execution of each stage S_i as an *agreement* problem. By $AgrPb_i$ we denote the agreement problem of stage S_i. The problem $AgrPb_i$ is to be solved by the agent replicas a_i^j running on the places in $\mathcal{M}_i$ and the solution is the decision on which all agent replicas running on the places in $\mathcal{M}_i$ agree. We denote by dec_i the decision (i.e., the solution) of $AgrPb_i$, with the following properties:

- (*Agreement*) No two agent replicas of stage S_i decide differently.
- (*Uniform validity*) If an agent replica of stage S_i decides dec_i, then dec_i was proposed by some agent replica a_i^j of S_i and is the result of executing a_i^j on place p_i^j.
- (*Uniform integrity*) Every agent replica of stage S_i decides at most once.
- (*Termination*) Every agent replica of stage S_i decides eventually.

The decision dec_i is as follows for the two cases identified in Section 3.3:

Hetero-places. The decision dec_i has three parts: 1) the single place $p_i^{prim} \in \mathcal{M}_i$, called *primary*, that has executed the agent in stage S_i, 2) the resulting agent a_{i+1}, and 3) the places $\mathcal{M}_{i+1}$ for a_{i+1}.

Hetero-places with witnesses. Similar to the previous case. Place p_i^{prim} can potentially be a witness.

The agreement problem is fundamental to enforce the exactly-once property of an agent execution (see Section 3.4).

4.2 Sequence of Agreement Problems

Having defined the basic agreement problem $AgrPb_i$, we now define the entire mobile agent execution as a sequence of agreement problems. This is done as follows:

- The initial problem $AgrPb_0$ of stage S_0 is solved by a_0 only. This can be regarded as a trivial agreement problem (only one agent replica has to decide). The decision is 1) p_0, 2) a_1, and 3) the places in $\mathcal{M}_1$. The agent a_1 is then sent to the places in $\mathcal{M}_1$. In practice, this agreement problem is reduced to a configuration problem. The agent owner configures the agent before sending it off to stage S_1.
- The problem $AgrPb_1$ of stage S_1 is solved by a_1^j running on the places $p_1^j \in \mathcal{M}_1$. The decision is p_1^{prim}, a_2, and the places in $\mathcal{M}_2$. The agent a_2 is then sent to the places in $\mathcal{M}_2$.
- ...
- The problem $AgrPb_i$ of stage S_i is solved by a_i^j running on the places $p_i^j \in \mathcal{M}_i$. The decision is p_i^{prim}, a_{i+1}, and the places in $\mathcal{M}_{i+1}$. The agent a_{i+1} is then sent to the places in $\mathcal{M}_{i+1}$.
- ...
- Similar to the problem $AgrPb_0$, $AgrPb_n$ of stage S_n is solved by only one agent replica. At this stage, the agent's results are presented to the agent owner or to another designated destination.

5 Two Building Blocks for Fault-Tolerant Mobile Agents

The previous section has shown that fault-tolerant mobile agent execution can be expressed as a sequence of agreement problems. In this section, we identify two building blocks for fault-tolerant mobile agent execution: 1) *consensus* and 2) *reliable broadcast*. Building block 1) is used to solve the agreement problem at stage S_i, whereas 2) allows the agent to be forwarded reliably between consecutive stages. Our approach encompasses various system models such as process recovery, depending on the implementation of consensus and of the reliable forwarding of agents.

Fig. 4 depicts a fault-tolerant mobile agent execution. The execution at stage S_i consists of 1) one (or, in case of a failure or false suspicions, multiple) place(s) executing the agent, 2) the agent replicas running on the places in $\mathcal{M}_i$ reaching an agreement on the computation result, and 3) the reliable forwarding of the result a_{i+1} to the next stage S_{i+1}. The computational result contains the new agent a_{i+1} and the set of places executing the agent at stage S_{i+1} (i.e., $\mathcal{M}_{i+1}$), as well as the place p_i^{prim} that has executed the agent. Note that the latter relates to stage S_i, whereas the former two results provide information about the next stage S_{i+1}.

Stage 2 in Fig. 4 illustrates the case of a place failure. When a_2^1 detects the failure of a_2^0, it starts executing and tries to impose its computation as the decision value of the agreement protocol to all $p_2^j \in \mathcal{M}_2$. Upon recovery, a_2^0 learns the outcome of the agreement (i.e., dec_2). If $p_2^0 = p_2^{prim}$, the modifications of a_2^0 on p_2^0 become permanent; otherwise (i.e., $p_2^0 \neq p_2^{prim}$), they are undone/aborted.

In the following, we present the Consensus with Deferred Initial Value (DIV consensus) as building block 1 (Section 5.1) as well as a protocol to forward the agent reliably to the next stage (building block 2) in Section 5.2.

5.1 Building Block 1: Solving the Agreement Using DIV Consensus

5.1.1 DIV Consensus Problem

The consensus problem is a well-specified and studied problem in fault-tolerant distributed systems research. It is

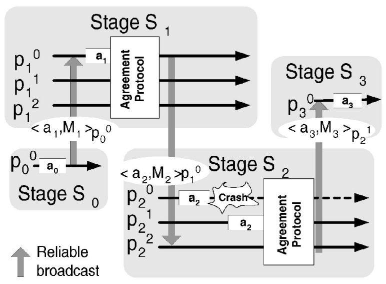


Fig. 4. Agent execution with p_2^0 failing. An erroneously suspected place p_2^0 leads to the same situation. The notation $< a_{i+1}, \mathcal{M}_{i+1} >_{p_i^{prim}}$ means that p_i^{prim} has executed agent a_i (which leads to a_{i+1} and M_{i+1}).

defined in terms of the primitive $propose(v)$. Every process p_k in a set of processes Ω calls this primitive with an initial value v_k as an argument. Informally, the consensus allows an agreement on a certain value to be reached among the correct processes in Ω. This value, called decision value, is an element of the set of initial values v_k. The formal specification of the consensus problem is given in [6].

The algorithm in [6] solves the consensus problem with the unreliable failure detector $\diamond S$ and a majority of correct processes. DIV consensus [9] modifies the consensus problem such that all processes need not have an initial value. The initial value is computed during the execution of the consensus algorithm, whenever needed. Specifically, in the absence of failures, only one process computes the initial value. For this purpose, the participants do not invoke the consensus by passing their initial value as an argument. Rather, they pass a handler $\mathcal{H}(x)$ that allows the protocol to compute the initial value only when needed.

5.1.2 Applying DIV Consensus

At each stage S_i, an instance of DIV consensus is solved and determines the outcome of the stage execution. Using DIV consensus requires the following transformations:

Initial handler $\mathcal{H}(x)$. The initial handler $\mathcal{H}(x)$, passed as an argument to the function $propose$, is the agent a_i or, more precisely, a method of a_i. It is executed only when needed during the execution of DIV consensus. In particular, in the absence of failures, it is executed only once.

Decision value dec. The execution of consensus decides on the tuple $dec_i = \langle a_{i+1}, \mathcal{M}_{i+1} \rangle_{p_i^{prim}}$ (see Section 4.1).

DIV consensus ensures that all a_i^j running on $p_i^j \in \mathcal{M}_i$ agree on the p_i^{prim} that has executed a_i, on the new agent a_{i+1}, as well as on the places of the next stage S_{i+1}.

The version of DIV consensus presented in [9] assumes reliable communication channels. As stated in [9], the algorithm can easily be extended to handle unreliable communication channels as well by using an approach along the lines of [1]. As long as the network is partitioned in such a way that one partition contains a majority of places of a stage, the execution is not blocked. Moreover, it makes the assumption that a majority of a_i^j does not fail, i.e., is correct. In our system model, agents are good (see Section 2.2). However, the termination of the agreement does not depend on the recovery of the agents. Rather, we assume that a majority of them does not fail while DIV consensus executes. When they recover, they no longer participate in consensus. But, they undo their modifications (if needed) to ensure exactly-once. Assuming good agents maintains consistency from the point of view of the agent owner (or application) who has launched the mobile agent by bringing all accessed places to a consistent state.

However, the protocol presented could easily be extended to also encompass recovery by using a corresponding version of consensus along the lines of [2]. Indeed, we argue in the next section that recovery needs to be supported to a certain degree because of asynchronous agent propagation.

Note that the order of the places in $\mathcal{M}_i$ determines the order in which the places attempt to execute the agent replicas. For instance, if $\mathcal{M}_i$ contains the set of places $\{p_i^0, p_i^1, p_i^2\}$, the agent execution is first performed on p_i^0. If p_i^0 is suspected, then p_i^1 starts executing its agent replica. Hence, the places given first in the set $\mathcal{M}_i$ have a higher probability of executing the agent replica than the ones given later. Witnesses always appear last in $\mathcal{M}_i$.

5.1.3 Asynchronous Agent Propagation

We have assumed (see Section 2) 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 Fig. 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. Agent a_i^2 thereby always uses a model of recovery, where no partial state survives a crash. In summary, the asynchrony assumption thus forces us indirectly to support the recovery of agents after a crash.

5.2 Building Block 2: Reliably Forwarding the Agent between S_i and S_{i+1}

Having solved the problem of executing the agent at a stage, we must address the issue of reliably forwarding the agent to the next stage. A naive approach leads to a protocol, where every place in $\mathcal{M}_i$ broadcasts the result dec_i to every place in $\mathcal{M}_{i+1}$. However, this incurs significant overhead in terms of message number as well as number of communication steps,[6] depending on the protocol selected. Our approach reduces this overhead considerably. For this purpose, only a majority of the places in $\mathcal{M}_i$ broadcast to all places in $\mathcal{M}_{i+1}$. As DIV consensus assumes that a majority of places in $\mathcal{M}_i$ do not fail, it is ensured that at least one place actually sends the agent.

5.3 Optimization: Pipelined Mode

In our discussion so far, we have assumed that $\mathcal{M}_{i-1}$ and $\mathcal{M}_i$ are a disjoint set of places. However, this is not a

6. A communication step is identified as the sending of a message that is in the critical path of the protocol, i.e., the protocol cannot proceed until it has received this message.

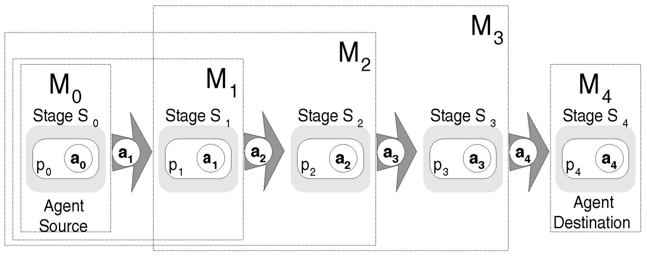


Fig. 5. Pipelined mode without failures.

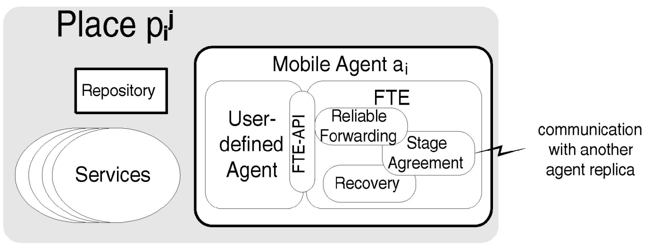


Fig. 6. Agent-dependent approach: architecture of FATOMAS.

requirement [15], [23]. On the contrary, reusing places of stage S_{i-1} as witnesses for S_i (see Section 3.3.2) improves the performance of the protocol and prevents high messaging costs; the pipelined mode thus assumes hetero-places with witnesses. At a limit, every stage S_i merely adds another place to $\mathcal{M}_{i-1}$, while removing the oldest from the set $\mathcal{M}_{i-1}$. In this mode, forwarding costs are minimized and limited to forwarding the agent to the new place (see Fig. 5). We call this mode *pipelined*. Note that, for set $\mathcal{M}_1$, we assume the existence of a place that acts as a witness for the stage execution (not displayed in Fig. 5). The execution at stage S_0 (or S_4) is not replicated (see Section 3.2) and no witnesses are needed for $\mathcal{M}_0$ ($\mathcal{M}_4$).

6 FATOMAS

In Sections 4 and 5, we have introduced an approach for fault-tolerant mobile agent execution. In this section, we present FATOMAS,[7] a FAult-TOlerant Mobile Agent System that implements this approach. FATOMAS addresses hetero-places, but also works with replicated iso-places (see Section 7). We first present the architecture (Section 6.1) followed by implementation issues and performance results (Section 6.2). Our implementation is based on ObjectSpace's Voyager 3.1.2 Java mobile object platform [19]. To validate our architecture, we have also ported FATOMAS to Mopros, an experimental mobile process platform developed in our laboratory. A more in-depth discussion of FATOMAS can be found in [22].

6.1 Architecture

6.1.1 Isolation of Fault Tolerance Mechanisms

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. Whereas, in single-agent execution, for instance, an agent simply needs to specify the next place it moves to, our fault-tolerant agent execution generally[8] requires a set of destination places for the next stage ($\mathcal{M}_{i+1}$). Clearly, the agent is aware of the replication and complete transparency is no longer possible.

We propose an architecture that isolates the fault tolerance mechanisms in a component called the *Fault Tolerance Enabler* (FTE). The FTE interacts with the user-defined agent through a well-defined interface, called FTE-API (see Fig. 6). At each stage S_i $(0 < i < n)$, the FTE solves the stage agreement problem, i.e., it runs an instance of DIV consensus (see Section 5.1.1). Depending on the outcome of the agreement, the operations performed during the execution of $\mathcal{H}(x)$ (i.e., the agent's stage action) are either committed or undone. Indeed, in Section 3.4, 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 undo the agent operations. The semantics of the undo depend on the agent operations. For instance, database transactions need to be committed or aborted (or rolled back, depending on the database), whereas operations without side-effects generally require no further action (see also Section 3.5). Finally, the FTE moves the agent to the set of places in $\mathcal{M}_{i+1}$, which are computed by the user-defined agent and returned as the result of executing $\mathcal{H}(x)$ (see Section 5).

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). FATOMAS uses the agent-dependent approach, which is presented in more detail in the next section. We only briefly discuss the place-dependent approach here.

6.1.2 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 terminates the execution 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 Fig. 6). The FTE then addresses issues such as fault tolerance and mobility.

Fig. 6 shows the architecture of the agent-dependent approach. The FTE is composed of a *stage agreement* component (implementing DIV consensus), a *reliable forwarding* component (responsible for the agent forwarding to the next stage), and a *recovery* component. The latter

7. Not to be confused with FANTOMAS [20]. See Section 8 for a discussion.
8. Except in the particular case of the pipelined mode (see Section 5.3).

handles the recovery in case the agent fails or arrives late at a place (see Section 5.1.3). 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 agents at place p_i to access some information remotely 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.

6.1.3 Place-Dependent Approach

In the place-dependent approach, the FTE is provided by the mobile agent platform, e.g., [18], [26]. 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 instantiate the agent replicas a_i^j selectively when needed. Indeed, only agent replicas are instantiated whose stage action sa_i (i.e., the initial function $\mathcal{H}(x)$) is actually 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). As 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.

6.2 Implementation and Performance Evaluation

In this section, we first give a brief overview of the most important implementation issues of FATOMAS (Section 6.2.1). The performance of FATOMAS is then evaluated using the example agent presented in Section 6.2.2. The goal of this performance evaluation is to identify the overhead introduced to a mobile agent by our replication mechanisms.

6.2.1 Implementation

This section describes the implementation of the FTE. As indicated in Section 6.1, we build fault tolerance on top of an existing mobile agent platform (i.e., Voyager) without modifying existing code.

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 DIV Consensus.

When the consensus algorithm decides, the FTE stores the decision value in a local repository (see Fig. 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. The decision value can only be discarded from the local repository when all the failed agents/places of the stage have recovered and know the outcome of the decision. In a simple approach, the decision value is discarded after a certain time. In more elaborate approaches, the places of a stage notify each other when the decision value is no longer needed.

Having stored the decision value in the repository, the FTE either commits or aborts the actions of the user-defined agent, depending on the decision value or, more specifically, on the p_i^{prim} value of the decision (see Section 5). Finally, the FTE forwards the agent to the next stage as described in the next section.

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 ensure 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., if the places at two consecutive stages S_i and S_{i+1} form disjoint sets, the simplest solution to reliable forwarding consists of sending $|\mathcal{M}_i| * |\mathcal{M}_{i+1}|$ agents. However, to reduce the communication overhead, we chose the following optimistic approach: The agent a_{i+1} is sent to each place in $\mathcal{M}_{i+1}$ only by the agent a_i at 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 successfully 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 because 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. 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.

Recovery. Although recovery and nonsimultaneous agent arrival can be handled in the same way (see Section 5.1.3), our prototype distinguishes between the following 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 part again 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.

TABLE 1
Costs of Replication Degree 1 and 3 in Milliseconds Compared to the Single Agent

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%

6.2.2 Example: A Fault-Tolerant Agent Accessing Counters

To measure the performance of FATOMAS, we assume hetero-places (see Section 3.3.2) and 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 of sequentially 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 us 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 additional costs because the classes have to be transported with the agent and then loaded into the virtual machine. However, once the classes are loaded in the class loader, other agents can take advantage of them and do not need to load these classes again.

6.2.3 Experimental Setup

Our performance tests are run on seven AIX machines (PowerPC 233 MHz processor, 256 MByte of RAM). The machines are connected by either a 100 Mbps Ethernet or 16 Mbps Tokenring; they are on three 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.

The 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 much lower than 5 percent. However, for very few results, it went up to 15 percent. 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.

6.2.4 Costs of Replication

We measure two aspects of the replication costs: 1) the overhead of the replication mechanism incurred 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 does not fail (see Section 5), replication degree 3 handles one failure and replication degree 5 would handle two failures. The results of these measurements are given in Table 1.

The first line in Table 1 shows the costs for a single agent, 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 percent overhead compared to a single agent. The overhead is lower (18 percent) 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 caused mainly by the additional communication costs of agent forwarding and consensus. Indeed, consider, for instance, the single agent execution on four stages, where there are three 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 overhead of the fault tolerance mechanisms seems reasonable, considering the guarantees the fault tolerance mechanisms provide: nonblocking and exactly-once mobile agent execution. Moreover, in our experiment, the execution time of the agent's stage operation is less than 5 ms and therefore not significant. Clearly, the longer the execution time of the agent's stage operation, the smaller the ratio of the overhead between the single agent and the replicated agent.

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 Fig. 4 (stage 2).

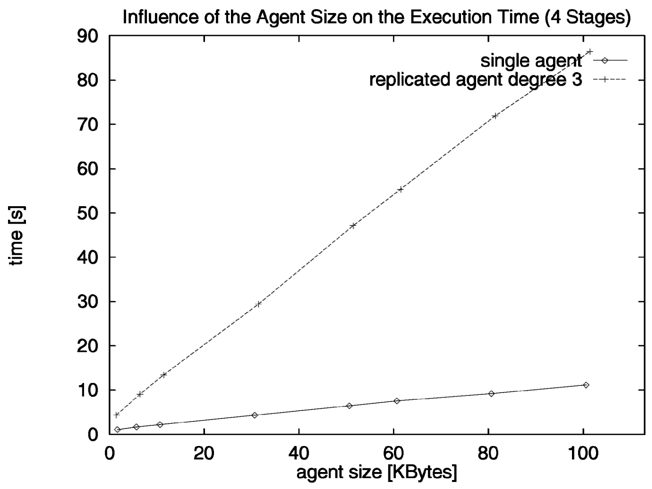


Fig. 7. Costs of single and replicated agent execution with increasing agent size (four stages).

The main part of the costs stems from the selected timeout value in the failure detection mechanisms for consensus (timeout = 10,000 ms). A more aggressive timeout value considerably speeds up the agent execution, but it also increases the risk of false suspicions.

6.2.5 *Influence of the Size of the Agent*

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 used to increase the size of the agent. As the results in Fig. 7 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.

6.2.6 *Optimization: Pipelined Mode*

We have introduced the pipelined mode in Section 5.3. 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 up 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, Fig. 8 shows that the pipelined mode has a lower execution time than the normal replicated agent.

6.2.7 *Discussion*

In [26], Silva et al. measure the performance overhead of their approach, called James, and also compare it to a partial implementation of the approach in [23] that is, similarly to our approach, based on replication. For an agent of size of about 1 KByte, they measure an overhead of 20 percent for James, while the approach in [23] introduces an overhead of 300 percent. This latter overhead is comparable to the overhead introduced by FATOMAS (see Table 1). However, the approach in [23] is blocking, whereas our approach is not, and Silva et al. mention having made only a partial implementation. James uses an approach with different characteristics which we call the commit-at-dest approach

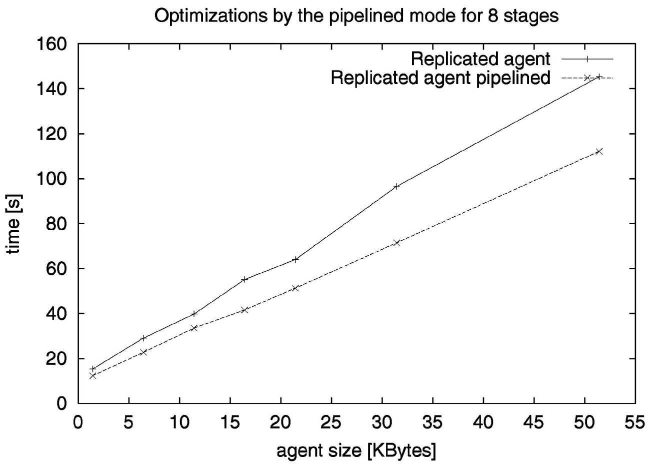


Fig. 8. Performance gain with the pipelined mode for eight stages.

(see Section 8) and it is not clear to us what exactly has been taken into account in their performance measurements (i.e., overhead of fault-tolerant lookup directory and of locking). Hence, it is difficult to compare the performance of the two approaches. Moreover, our measurements are more detailed as we also isolate the cost of consensus at a stage.

7 Fault-Tolerant Mobile Agent Execution in the Context of Iso-Places

In Section 3.3, we briefly introduced replicated and independent iso-places, but so far have limited the discussion to hetero-places and hetero-places with witnesses. In this section, we present fault-tolerant mobile agent execution in the context of iso-places. In particular, we address the modifications that are required by our approach in order to handle iso-places as well. We first address the case of replicated iso-places (Section 7.1). Then, we discuss independent iso-places (Section 7.2) and show that, because of particularities in the case of independent iso-places, the agent must know beforehead whether it is dealing with independent iso-places.

7.1 Replicated Iso-Places

In contrast to hetero-places, replicated iso-places run a replication mechanism among themselves (see Fig. 9a). Consequently, executing the replicated agent on replicated iso-places leads to two levels of replication: 1) *place replication* (replication running among the iso-places[9]) and 2) *agent replication* (replication among the agent replicas). The replication technique used for agent replication is based on DIV consensus (see Section 5), whereas place replication can use either *active* [24], *passive* [5], or *semi-passive* [9] replication. Actually, the case of replicated iso-places corresponds to a replicated client invoking a replicated server in traditional distributed systems.

The agent replicas need not run on all replicated iso-places; rather, they can run on a subset of these places.

9. Strictly speaking, the replication mechanism occurs among the services running on the replicated iso-places (i.e., U^j in Fig. 9a). For simplicity, we do not make this distinction in our discussion (see also Section 2.1).

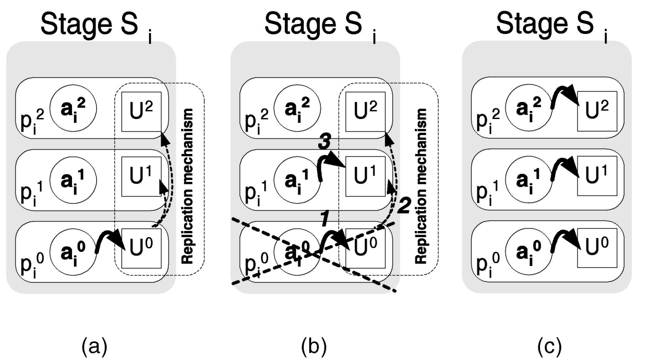


Fig. 9. Agent replicas execute on (a) replicated iso-places and (c) independent iso-places. U^j represents the replica of the service running on place p_i^j and provides access to (part of) the state of p_i^j (see Section 2.1). The failure of p_i^0 in the case of replicated iso-places is shown in (b).

Indeed, the replication mechanism among replicated iso-places ensures that all iso-places eventually reflect the changes, whether they run a replica agent or not. Revisiting the example in Section 3.3.1, although airline X may provide five iso-places, $\mathcal{M}_i$ may contain only three of them.

To isolate the replication technique used by the replicated iso-places from the agent replicas, i.e., to achieve transparency, the agent replicas access the service on the iso-place through a local proxy. This proxy then forwards the request to the service. In the case of actively (semi-passively) replicated iso-places, the proxy atomically (reliably) broadcasts the request to all iso-places [6]. With passively replicated iso-places, the request is sent to the primary place. To exploit locality, it is desirable that the primary of agent replication and the primary of place replication be identical. However, they may denote different places at stage S_i, in particular if failures or false suspicions occur.

In the following, we show how exactly-once execution of the agent at stage S_i can be achieved on replicated iso-places. Assuming deterministic execution,[10] the exactly-once property in the case of replicated iso-places is achieved immediately (Section 7.1.1). Without this assumption, ensuring the exactly-once property requires recovery of failed agents (Section 7.1.2).

7.1.1 Exactly-Once and Determinism

As shown in Section 3.4, replication can lead to multiple executions of the agent's code and, thus, also multiple instances of the agent request to the place. However, multiple executions of the agent replica's requests on the place can easily be prevented. Indeed, the use of request IDs allows replicated iso-places to detect whether the same request has already been processed. If this is the case, the iso-places simply return the result; otherwise, the agent replica's request is executed. However, this requires that the agent replicas use identical request IDs when issuing the same request. Assume, for instance, that the agent replica executing on p_i^0 sends request rq_0 to service U^0 on p_i^0 (see Fig. 9b, step (1)). Service U^0 executes rq_0 and, because of the replication mechanism among the replicated iso-places, all other iso-places reflect the result of this execution (Fig. 9b, step (2)). Before the result of rq_0 is communicated to the agent replica, p_i^0 fails. Another agent replica (e.g., the one executing on p_i^1) takes over the execution and sends request rq_1 to service U^1 (step (3) in Fig. 9b). If the request IDs of rq_0 and rq_1 are equal, then the service on p_i^1 detects the duplicate request and simply returns the previously computed result. Identical request IDs on different agent replicas are generally only possible if the agent replicas execute deterministically. Interestingly, in this case, the stage agreement (see Section 4.1) may no longer be required. Indeed, if the agent performs invocations only to replicated iso-places, then no agreement is required among the agent replicas. The exactly-once execution is ensured by the replicated iso-places. Moreover, the requests of two agents a and b are processed in the same order on all replicated iso-places.

7.1.2 Nondeterminism Requires Recovery

Nondeterminism in the execution of the agent replicas requires that the agent replicas be good, i.e., that they are eventually permanently up. Indeed, revisiting the example in the previous section, assume that the request IDs of rq_0 and rq_1 are not equal. Although p_i^1 already reflects the execution of rq_0 on place p_i^0 (via the replication mechanism among the replicated iso-places, i.e., step (2) in Fig. 9b), it still executes rq_1. Because of the different request IDs, p_i^1 considers rq_1 a new request. Actually, a_i^0 and a_i^1 act like two different clients from the point of view of the iso-places and not as the replicas they are. This leads to a violation of the exactly-once execution property as the iso-places reflect the result of executing the agent request twice.

Ensuring exactly-once is thus only possible when a_i^0 recovers, learns that another agent replica has executed, and undoes its stage action, in particular the modifications to the iso-places caused by rq_0 (see Section 3.5). Consequently, the case of nondeterministic execution in the context of replicated iso-places is similar to the case of hetero-places. Moreover, the decision dec_i in the basic agreement problem is equal to the case with hetero-places (see Section 4.1). Hence, FATOMAS also handles the execution of the agent on replicated iso-places.

A problem that is particular to the case of replicated iso-places arises with the use of pessimistic execution by the services of the replicated iso-places (see Section 3.5): the problem of blocking. Indeed, assume that the replicated iso-places in $\mathcal{M}_i$ use locking to achieve pessimistic execution, i.e., they execute the agent requests as transactions. Consequently, the execution of rq_0 on place p_i^0 has locked the accessed data items. Executing rq_1 on p_i^1 accesses the same (replicated) data items, currently locked by a_i^0. The execution of a_i^1 can generally only proceed when a_i^0 recovers, aborts rq_0, and thus releases the locks.

7.2 Independent Iso-Places

Independent iso-places are exact replicas with no replication mechanism running among them (see Section 3.3.1). So, the agent replicas ensure that all places in $\mathcal{M}_i$ learn about the agent replica's request and about the new state of the

10. An execution is *deterministic* if all agent replicas execute the same steps and return the same results. Moreover, these results are reproducible. Multiple threads, for instance, lead to nondeterministic execution as the thread scheduling cannot be controlled and is somewhat arbitrary.

iso-place that has executed the request (see Fig. 9c). This can be done by either 1) executing the agent on all places or 2) executing the agent on one place and sending the state update information of the place to the other agent replicas, which then update their local iso-place. In approach 1), agreement (i.e., DIV consensus) between the agent replicas is not required. However, approach 1) requires deterministic execution of the agent replicas. Indeed, to ensure that all iso-places p_i^j have the same state after the execution of a_i, the agent replicas a_i^j need to execute the same sequence of steps. Clearly, the execution of the agent replicas on all places p_i^j leads to a higher computation overhead. Moreover, the replicas of two mobile agents a_i and b_i need to be executed in the same order on all independent iso-places p_i^j, i.e., the execution of agent replicas needs to be totally ordered. Hence, reliable broadcast is not sufficient to forward the agent between two stages; rather, the agent is atomically (total order) broadcast to $\mathcal{M}_{i+1}$ by the places in $\mathcal{M}_i$. Note that, with this approach, the exactly-once property of agent execution is, in a strict sense, no longer required; rather, all agent replicas are executed, i.e., we have multiple executions of agent a_i at stage S_i. However, we still need to ensure that agent replica a_i^j executes exactly-once. In other words, the exactly-once property now refers to the execution of replica a_i^j and not to the execution of agent a_i as with hetero-places and replicated iso-places.

Approach 2) also requires atomic broadcast to forward the agent between two consecutive stages. However, approach 2) is more generic in that it also addresses nondeterministic execution of the agent replicas. For this purpose, the basic agreement problem (see Section 4.1) is used to decide on the iso-place that has executed the agent replica. Consequently, approach 2) has a lower computation overhead if no failures and false suspicions occur. Indeed, in this case, only one agent replica executes and sends the state update information to the other agent replicas. However, this state update information might be large or difficult to obtain. With independent iso-places, the decision value dec_i (see Section 4) is different: 1) the state update information for all the places in $\mathcal{M}_i$, 2) the resulting agent a_{i+1}, and 3) the places $\mathcal{M}_{i+1}$ for a_{i+1}.

Whereas executing the agent replicas on independent iso-places prevents blocking, failures of the agent replicas may lead to inconsistencies of the state of the independent iso-places. Indeed, assume that agent replica a_i^0 at stage S_i fails. As it has not received the state update information, place p_i^0 is not aware of the latest agent replica request, whereas places p_i^1 and p_i^2 have already updated their state. To avoid inconsistency, place p_i^0 can only execute another agent replica b_k^l when a_i^0 has recovered and successfully executed.

A summary of this section is given in Table 2, in which we identify the building blocks required to achieve fault-tolerant mobile agent execution in the context of independent and replicated iso-places. Table 2 also presents the case of hetero-places. Note that deterministic execution of agents and places is not meaningful with hetero-places.

TABLE 2
Summary of the Required Building Blocks Needed to Achieve Fault-Tolerant Mobile Agent Execution

place properties	agent/place deterministic	agent/place non-deterministic
hetero-places	-	DIV consensus Reliable Broadcast
independent iso-places	Atomic Broadcast	DIV consensus Atomic Broadcast
replicated iso-places	Reliable Broadcast	DIV consensus Reliable Broadcast

The building blocks are given with respect to the place properties and determinism in the agent/place execution. The case of hetero-places with deterministic execution of the agent/place is not meaningful as no replication mechanism runs between hetero-places.

8 Related Work

Recently, fault-tolerant mobile agent execution has been a very active field of research. We distinguish between the *commit-after-stage* and *commit-at-dest* approaches. Commit-after-stage approaches make the modifications at the agent and place permanent and visible to other agents during or immediately after every stage execution. In contrast, commit-at-dest approaches generally commit the modifications only at the end of the entire agent execution. Our approach, as well as the approaches in [3], [15], [20], [23], is commit-after-stage approaches, whereas [18], [26] are part of commit-at-dest approaches.

A commit-after-stage approach sends multiple replicas of the agent to a set $\mathcal{M}$ of places of the next stage. Having multiple replicas of the agent allows it to mask failures and to proceed despite failing replicas. To ensure the exactly-once execution property, commit-after-stage schemes need to solve an agreement problem at every stage. Current commit-after-stage schemes assume reliable failure detection [15], [20], are based on complex models using transactions and leader election [3], [23], or block even on a single place failure [23]. Our approach, which is based on an easily understandable model, does not assume reliable failure detection, prevents blocking, and ensures exactly-once execution. To our knowledge, our work is the first to perform an evaluation of a commit-after-stage approach for fault-tolerant mobile agents. In [26], Silva et al. compare their commit-at-dest approach with some basic performance results from their partial implementation of [23]. However, no details about the implementation are presented.

To our knowledge, Schneider [25] is the only one to advocate a commit-after stage approach to independent iso-places. However, his approach addresses Byzantine failures.

A commit-at-dest approach, on the other hand, attempts to execute the agent on a place. If this 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 detected only at the agent destination [18], [26]. Consequently, accessed data items have 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 have to be committed, whereas those of the redundant duplicate agents have 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 to have been taken into account in the performance evaluation in [18], [26], results in reduced overall system throughput. In contrast, commit-after-stage approaches do 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, performing the undo operation on a language level [18] requires specific knowledge about the way applications handle rollbacks. To our understanding, [18] assumes standard interfaces for rollbacks and undo operations. In contrast, our approach leaves the undo operation to the application, which has the best knowledge about the services it is using and their way to handle rollbacks. The JAMES platform [26] uses a replicated lookup directory to prevent duplicate agents and thus also has some elements of a commit-after-stage approach. Duplicates caused when a place cannot reach the lookup directory (e.g., because of network partitioning) are generally detected only at the agent destination. However, as network partitioning can occur at any point in time, JAMES needs to behave as a commit-at-dest approach. In our approach, network partitions may, in the worst case, lead to blocking, but the lifetime of duplicate agents is still limited to the stage execution. Moreover, such a lookup directory violates, to some extent, the autonomy property of a mobile agent.

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 6.1.2). The work in [3], [15], [18], [23], [26] all uses a place-dependent approach. As these approaches for fault-tolerant mobile agent execution are inherently different from ours, the place-dependent architecture may be better suited. Indeed, Mohindra [18], for instance, defines a new scripting language, with fault tolerance tightly integrated into language constructs. The commit-at-dest approach in [26] 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 of requiring modifications to the existing mobile agent platform.

9 Conclusion

In this paper, we have identified two important properties for fault-tolerant mobile agent execution: nonblocking and exactly-once. Nonblocking ensures that the agent execution proceeds despite a single failure of either agent, place, or machine. Blocking is prevented by the use of replication. However, replication may lead to multiple executions of the agent and thus to a violation of the exactly-once property. Whereas this is not a problem for idempotent operations, it may lead to an incorrect agent and place state with nonidempotent operations. Our approach consists of modeling fault-tolerant mobile agent execution as a sequence of agreement problems. In contrast to the more complex model consisting of transactions and leader election [3], [23], it is simple and does not rely on reliable failure detection such as [15]. Moreover, accessed data items are released immediately after the stage execution and not locked until the agent has reached its destination [18].

Our approach requires two building blocks: 1) DIV consensus [9] and 2) reliable broadcast. We have implemented these building blocks in a prototype system called FATOMAS (FAult-TOlerant Mobile Agent System). The performance measurements of FATOMAS show the overhead introduced by the replication mechanisms with respect to a nonreplicated agent. Not surprisingly, they also show that this overhead increases with the number of stages and the size of the agent. A performance improvement can be achieved with the pipelined mode.

Finally, we have shown that our approach is also applicable to the case of replicated iso-places. In the particular case of deterministic execution of the agent replicas and places, building block 1) (i.e., DIV consensus) is no longer needed (although its use is still correct). Consequently, FATOMAS can also handle replicated iso-places. In contrast, independent iso-places require the implementation of atomic broadcast in FATOMAS. Nevertheless, our model of fault-tolerant mobile agent execution is still valid.

References

[1] M.K. Aguilera, W. Chen, and S. Toueg, “Quiescent Reliable Communication and Quiescent Consensus in Partitionable Networks,” Technical Report TR 97-1632, Cornell Univ., June 1997.

[2] M.K. Aguilera, W. Chen, and S. Toueg, “Failure Detection and Consensus in the Crash-Recovery Model,” *Distributed Computing,* vol. 13, no. 2, pp. 99-125, 2000.

[3] F.M. Assis Silva and R. Popescu-Zeletin, “An Approach for Providing Mobile Agent Fault Tolerance,” *Proc. Second Int’l Workshop Mobile Agents (MA ’98),* K. Rothermel and F. Hohl, eds., pp. 14-25, Sept. 1998.

[4] A. Bieszczad, B. Pagurek, and T. White, “Mobile Agents for Network Management,” *IEEE Comm. Surveys,* Sept. 1998.

[5] N. Budhirja, K. Marzullo, F.B. Schneider, and S. Toueg, “The Primary-Backup Approach,” *Distributed Systems,* S. Mullender, ed., second ed., pp. 199-216, Reading, Mass.: Addison-Wesley, 1993.

[6] T.D. Chandra and S. Toueg, “Unreliable Failure Detectors for Reliable Distributed Systems,” *J. ACM,* vol. 43, no. 2, pp. 225-267, Mar. 1996.

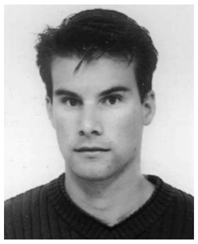
[7] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, and G. Tsudik, “Itinerant Agents for Mobile Computing,” *IEEE Personal Comm. Systems,* vol. 2, no. 5, pp. 34-49, Oct. 1995.

[8] D. Chess, C.G. Harrison, and A. Kershenbaum, “Mobile Agents: Are They a Good Idea?” *Mobile Agents and Security,* G. Vigna, ed., pp. 25-47, Springer Verlag, 1998.

[9] X. Défago, A. Schiper, and N. Sergent, “Semi-Passive Replication,” *Proc. 17th IEEE Symp. Reliable Distributed Systems (SRDS ’98),* pp. 43-50, Oct. 1998.

[10] M.J. Fischer, N.A. Lynch, and M.S. Paterson, “Impossibility of Distributed Consensus with One Faulty Process,” *Proc. Second ACM SIGACT-SIGMOD Symp. Principles of Database Systems,* pp. 1-7, Mar. 1983.

[11] H. Garcia-Molina and K. Salem, "Sagas," *Proc. ACM SIGMOD Int'l Conf. Management of Data and Symp. Principles of Database Systems,* pp. 249-259, 1987.
[12] J. Gray, "The Transaction Concept: Virtues and Limitations," *Proc. Int'l Conf. Very Large Databases,* pp. 144-154, 1981.
[13] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques.* San Mateo, Calif.: Morgan Kaufmann, 1993.
[14] T. Gschwind, M. Feridun, and S. Pleisch, "ADK—Building Mobile Agents for Network and Systems Management from Reusable Components," *Proc. First Int'l Conf. Agent Systems and Applications/Mobile Agents (ASAMA '99),* Oct. 1999.
[15] D. Johansen, K. Marzullo, F.B. Schneider, K. Jacobsen, and D. Zagorodnov, "NAP: Practical Fault-Tolerance for Itinerant Computations," *Proc. 19th Int'l Conf. Distributed Computing Systems (ICDCS '99),* June 1999.
[16] D.B. Lange and M. Oshima, "Seven Good Reasons for Mobile Agents," *Comm. ACM,* vol. 45, no. 3, pp. 88-89, Mar. 1999.
[17] P. Maes, R.H. Guttman, and A.G. Moukas, "Agents that Buy and Sell," *Comm. ACM,* vol. 42, no. 3, pp. 81-91, Mar. 1999.
[18] A. Mohindra, A. Purakayastha, and P. Thati, "Exploiting Non-Determinism for Reliability of Mobile Agent Systems," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '00),* pp. 144-153, June 2000.
[19] ObjectSpace, *Voyager: ORB 3.1 Developer Guide,* 1999. http://www.objectspace.com/products.
[20] H. Pals, S. Petri, and C. Grewe, "FANTOMAS—Fault Tolerance for Mobile Agents in Clusters," *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS) 2000 Workshop,* J.D.P. Rolim, ed., pp. 1236-1247, 2000.
[21] S. Pleisch and A. Schiper, "Modeling Fault-Tolerant Mobile Agent Execution as a Sequence of Agreement Problems," *Proc. 19th IEEE Symp. Reliable Distributed Systems (SRDS '00),* pp. 11-20, Oct. 2000.
[22] S. Pleisch and A. Schiper, "FATOMAS: A Fault-Tolerant Mobile Agent System Based on the Agent-Dependent Approach," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '01),* pp. 215-224, July 2001.
[23] K. Rothermel and M. Strasser, "A Fault-Tolerant Protocol for Providing the Exactly-Once Property of Mobile Agents," *Proc. 17th IEEE Symp. Reliable Distributed Systems (SRDS '98),* pp. 100-108, Oct. 1998.
[24] F.B. Schneider, "Replication Management Using the State-Machine Approach," *Distributed Systems,* S. Mullender, ed., second ed., pp. 169-198, Reading, Mass.: Addison-Wesley, 1993.
[25] F.B. Schneider, "Towards Fault-Tolerant and Secure Agentry," *Proc. 11th Int'l Workshop Distributed Algorithms,* invited paper, Sept. 1997.
[26] L.M. Silva, V. Batista, and J.G. Silva, "Fault-Tolerant Execution of Mobile Agents," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '00),* pp. 135-143, June 2000.
[27] M. Strasser and K. Rothermel, "Reliability Concepts for Mobile Agents," *Int'l J. Cooperative Information Systems,* vol. 7, no. 4, pp. 355-382, 1998.
[28] K. Takashio, G. Soeda, and H. Tokuda, "A Mobile Agent Framework for Follow-Me Applications in Ubiquitous Computing Environment," *Proc. Int'l Workshop Smart Appliances and Wearable Computing (IWSAWC '01),* pp. 202-207, Apr. 2001.
[29] W. Theilmann and K. Rothermel, "Optimizing the Dissemination of Mobile Agents for Distributed Information Filtering," *IEEE Concurrency,* pp. 53-61, Apr. 2000.



Stefan Pleisch received the MS degree in computer science from the Swiss Federal Institute of Technology in Lausanne (EPFL) in 1997. After his studies, he was working for ELCA Informatik AG, a Swiss IT supplier. Since 1998, he has been working at the IBM Zurich Research Laboratory and is about to finish his PhD thesis at EPFL. His research interests include distributed systems, fault tolerance, and mobile agents.

André Schiper has been a professor of computer science at the EPFL (Federal Institute of Technology in Lausanne) since 1985, leading the Distributed Systems Laboratory. During the academic year 1992-1993, he was on sabbatical leave at the University of Cornell, Ithaca, New York. His research interests are in the areas of fault-tolerant distributed systems, middleware, group communication, and, recently, mobile ad hoc networks. He has taken part in the following European projects: ESPRIT Basic Research BROADCAST (1992-1995), ESPRIT R&D OpenDREAMS (1996), ESPRIT Working Group BROADCAST (1996-1999), ESPRIT R&D OpenDREAMS II (1997-1999), IST REMUNE (2002-2004), IST MIDAS (2002-2003). He is a member of the IST Network of Excellence in Distributed and Dependable Computing Systems (CABERNET). From 2000 to 2002, he was the chair of the steering committee of the International Symposium on Distributed Computing (DISC). He is a member of the IEEE and the IEEE Computer Society.