Approaches to Fault-Tolerant and Transactional Mobile Agent Execution—An Algorithmic View

STEFAN PLEISCH AND ANDRÉ SCHIPER

École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Over the past years, mobile agent technology has attracted considerable attention, and a significant body of literature has been published. To further develop mobile agent technology, reliability mechanisms such as fault tolerance and transaction support are required. This article aims at structuring the field of fault-tolerant and transactional mobile agent execution and thus at guiding the reader to understand the basic strengths and weaknesses of existing approaches. It starts with a discussion on providing fault tolerance in a system in which processes simply fail. For this purpose, we first identify two basic requirements for fault-tolerant mobile agent execution: (1) non-blocking (i.e., a single failure does not prevent progress of the mobile agent execution) and (2) exactly-once (i.e., multiple executions of the agent are prevented). This leads us to introduce the notion of a *local transaction* as the basic building block for fault-tolerant mobile agent execution and to classify existing approaches according to when and by whom the local transactions are committed. In a second part, we show that transactional mobile agent execution additionally ensures execution atomicity and present a survey of existing approaches. In the last part of the survey, we extend the notion of fault tolerance to arbitrary Byzantine failures and security-related issues of the mobile agent execution.

Categories and Subject Descriptors: A.1 [**General Literature**]: Introductory and Survey; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns (e.g., client/server, pipeline, blackboard)*; D.4.5 [**Operating Systems**]: Reliability—*Checkpoint/restart*; *fault-tolerance*; D.4.6 [**Operating Systems**]: Security and Protection—*Cryptographic controls*; *invasive software*

General Terms: Reliability, Algorithms, Design, Theory

Additional Key Words and Phrases: ACID, agreement problem, asynchronous system, Byzantine failures, commit, crash failures, fault tolerance, malicious places, mobile agents, replication, security, transaction

An important part of this work was done while S. Pleisch was working at the IBM Zurich Research Laboratory, Rüschlikon, Switzerland.

Authors' address: Distributed Systems Laboratory, École Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland; email: pstefan@acm.org; andre.schiper@epfl.ch.

1. INTRODUCTION

A mobile agent[1] is a computer program that acts autonomously on behalf of a user and moves through a network of heterogeneous machines.[2] Over the past years and originally triggered to a large extend by the work on Telescript [White 1995], the field of mobile agents has attracted considerable attention, and mobile agent technology has been considered for a variety of applications [Chess et al. 1995, 1998; Lange and Oshima 1999] such as systems and network management [Bieszczad et al. 1998], mobile computing [Takashio et al. 2001], information retrieval [Theilmann and Rothermel 2000], and e-commerce [Maes et al. 1999]. However, before mobile agent technology can appear at the core of tomorrow's business applications, reliability mechanisms for mobile agents have to be established. Among these reliability mechanisms, fault tolerance and transaction support are mechanisms of considerable importance. Currently, various approaches targeting different application fields exist. Owing to the respective strengths and weaknesses of these approaches, it is often difficult for the application developer to choose the one best suited to a given application. The article aims at structuring the field of fault-tolerant and transactional mobile agent execution and at examining the advantages and disadvantages of particular approaches.

In the first part, the article presents a survey of current approaches to fault-tolerant mobile agent execution. It focuses on the algorithmic aspects; implementations of the current approaches are not discussed. In this part, the crash failure model is assumed, that is, components such as agents, places (i.e., the logical environment that executes the mobile agent) and machines fail by prematurely halting their execution. We begin by identifying the requirements for fault-tolerant mobile agents: *nonblocking* and *exactly-once*. The nonblocking property ensures that the failure of an infrastructure component (e.g., a machine, place, agent, or communication link) does not prevent progress in the agent execution. A blocking execution is undesirable because it can lead the agent owner to potentially wait a long time for the return of the agent. Replication prevents blocking, but may result in multiple executions of the agent. The exactly-once property requires that the code of an agent be executed exactly once. This is particularly important for operations with side effects, for example, an agent withdrawing money from an account. Our survey proposes a novel classification of fault-tolerant mobile agent approaches. It is based on the time when the modifications of the agent become permanent and visible to other agents: *commit-after-stage vs. commit-at-destination*. In commit-after-stage approaches, the modifications are permanent immediately after the stage execution (i.e., after each execution step of the agent), whereas commit-at-destination approaches only commit the modifications when the agent has finished its entire execution. Within these two approaches we further distinguish between solutions where the agent and the commit decision execute on multiple places and those where they execute only on a single place. We show how these characteristics influence the exactly-once and nonblocking properties.

In the second part, we present a survey of transactional mobile agents. Transactional mobile agents execute the mobile agent transactionally. More specifically, a transactional mobile agent execution ensures atomicity, consistency, isolation, and durability (i.e., the so-called ACID properties [Härder and Reuter 1983; Gray and Reuter 1993]). Assume, for instance, that a mobile agent has to book (1) a flight from Zurich to New York, (2) a hotel room, and (3) a rental car in New York. Clearly, the agent owner, that is, the person or application creating and initializing the agent, wants to have either all of (1), (2), and (3)

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

[2] So far, the mobile agent research community has not agreed on a common definition for mobile agents. Hence, various definitions exist. For the purpose of this article, we adopt the definition given in Mattern [1998] and Object Management Group (OMG) [2000].

or none at all. A rental car in New York, for instance, is of limited use if no airline ticket is available. Hence, the three operations all have to succeed or otherwise none of them should be executed. This all-or-nothing property corresponds exactly to the atomicity property of a transaction. Note that this property is not ensured by fault-tolerant mobile agent execution discussed in the first part of this article.

Similar to fault-tolerant mobile agent execution, the non-blocking property is also desirable for transactional mobile agents. We show how approaches to fault-tolerant mobile agent executions can help to achieve nonblocking in a transactional context. In particular, we discuss how the commit-at-destination approach can be extended to a transactional mobile agent approach.

In the third part, we extend our failure model from crash failures to more hostile environments, in which malicious places can access and tamper with the mobile agent's state and code. More specifically, the place may read confidential data of the agent (e.g., the credit card information) or modify previously collected data. We survey existing approaches and show why this issue is still not adequately resolved.

The rest of this survey is structured as follows: Section 2 presents our model of mobile agents. In Section 3, we specify fault-tolerant mobile agent execution in terms of two properties: nonblocking and exactly-once. The basic building blocks to ensure these properties are identified in Section 4. Section 5 defines the characteristics of fault-tolerant mobile agent approaches and provides a classification of these approaches. A survey of existing approaches in terms of our classification is given in Section 6. In Section 7, we discuss mechanisms to ensure execution atomicity for transactional mobile agents and we present a survey on the current approaches to transactional mobile agents in Section 8. Section 9 contains the third part of this survey: fault-tolerant mobile agent execution in the context of malicious places. Finally, Section 10 concludes this article.

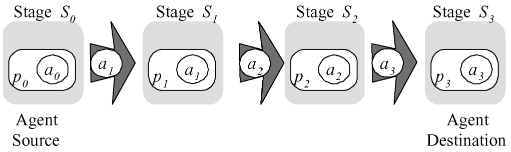

Fig. 1. Model of a mobile agent execution with four stages.

2. MODEL

We assume an asynchronous distributed system, that is, there are no bounds on transmission delays nor on relative processor speeds. An example of an asynchronous system is the Internet. Processors communicate via message passing.

2.1. Mobile Agent

A mobile agent executes on a sequence of machines, where a *place*[3] p_i $(0 \leq i \leq n)$ provides the logical execution environment for the agent [White 1996]. Executing the agent at a place p_i is called a *stage* S_i of the agent execution [Minsky et al. 1996]. 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 [Minsky et al. 1996]. 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 Figure 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).[4] We denote the

[3] Also called *landing pad* in Johansen et al. [1999].

[4] We assume that the mobile agent accesses only resources that are local to the place.

resulting agent a_{i+1}. Place p_i forwards a_{i+1} to p_{i+1} (for $i < n$).

In this article, we focus on the execution of a *single* agent. Hence, we denote by *agent execution* the execution of a single agent in a sequence of stages. The case of multiple agents coordinating to solve a higher-level task is briefly discussed in Section 3.3.

2.2. Infrastructure Failures

Machines, places, or agents can fail and subsequently recover. A component that has failed but not yet recovered is called *down*, whereas it is *up* otherwise. In this article, we first focus on crash failures (i.e., processes that halt prematurely). In Section 9, we extend this view to also accommodate malicious failures (i.e., Byzantine failures) of places and security-related issues. A failing place causes all agents running on it to fail as well. Similarly, a failing machine causes all places and agents on this machine to fail as well. We do not consider catastrophic failures such as deterministic, repetitive programming errors (i.e., programming errors that occur on all agent replicas or places) in the code or the place to be relevant failures in this context. In general, we call a failure catastrophic if it violates our failure assumption. To address catastrophic failures, Johansen et al. [1999] 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.

Finally, a link failure causes the loss of the messages or agents currently in transmission on this link. Failures of machines, places, agents, and links are called *infrastructure failures*.

The detection of infrastructure crash failures is generally encapsulated into a failure detector module [Chandra and Toueg 1996]. Failure detectors are defined in terms of completeness and accuracy properties. Completeness requires that failed processes be eventually suspected, whereas accuracy limits the number of false suspicions, that is, processes that are suspected but have not failed. In Chandra and Toueg [1996], the authors introduce several types of failure detectors defined by completeness and accuracy properties. Perfect failure detectors, which eventually detect all failures (strong completeness) and make no false suspicions (strong accuracy), are unrealistic in the Internet. Hence, assuming unreliable failure detectors—that can make false suspicions—is a more realistic assumption. $\diamond\mathcal{S}$ is an example of an unreliable failure detector [Chandra and Toueg 1996]: correct processes can falsely be suspected, but eventually some correct process is no more suspected.

2.3. Unfavorable Outcome

An *unfavorable outcome* is different from an infrastructure failure in the sense that neither machine, place, nor agent initiating the request fail. Rather, it occurs when a requested service is not delivered because of the application logic or because the service has failed. For instance, a request for an airline ticket is declined if no seats are available on a particular flight. Nevertheless in this case, the agent's operation, that is, the request for a ticket, executes in its entirety (although no ticket is issued). Actually, in this example no real "failure" has occurred, as the result is a valid outcome of the service. However, from the perspective of the agent (i.e., the client of the service), the outcome of the service request is undesired. Hence, we call this outcome an "unfavorable outcome". We assume that the request of a mobile agent to a service always returns. Therefore, an unfavorable outcome does not include programming errors such as infinite loops.

2.4. Transactional vs. Nontransactional Mobile Agents

The execution of two stage actions sa_i and sa_j is *atomic* if and only if both stage actions succeed or none at all. Atomicity addresses both infrastructure failures and unfavorable outcomes. Assume, for instance, that stage action sa_i books a flight, whereas sa_j reserves a hotel room at the flight destination. Clearly, there is no need for the hotel room if no seat is available on any flight to the destination. Hence, sa_i and sa_j must execute atomically, that is, we want both to succeed. If either one fails (because of an infrastructure failure or an unfavorable outcome such as no seat being available) then the other one has to be aborted as well. We call these mobile agents *transactional mobile agents*.

On the other hand, general fault-tolerant mobile agent executions do not require atomicity. Rather, they address infrastructure failures only and ignore unfavorable outcomes. For instance, a mobile agent execution that buys a book (i.e., sa_k) and shoes (i.e., sa_l) acquires the book even if no shoes are available, or vice versa. In this sense, sa_k and sa_l are independent.

Note that nontransactional fault-tolerant mobile agent executions can be implemented using transactions [Assis Silva and Popescu-Zeletin 1998; Rothermel and Strasser 1998]. However, the use of transactions still does not ensure atomicity in the entire mobile agent execution. In other words, our classification into "transactional" and "nontransactional" executions is related to the provided properties (e.g., atomicity), and not to the mechanisms used in the implementation. To distinguish between the properties and the mechanism, we refer to the properties using the word "transactional", whereas we use "transaction" to refer to the mechanism.

In the following, we first focus on nontransactional mobile agents, that is, general fault-tolerant mobile agent execution. Transactional mobile agents are discussed in Section 7.

3. SPECIFICATION OF FAULT-TOLERANT MOBILE AGENT EXECUTION

In this section, we specify the desired fault-tolerant mobile agent execution in terms of two properties: nonblocking and exactly-once execution.

3.1. Infrastructure Failures and the Blocking Problem

While a mobile agent is executing on a place p_i, an infrastructure failure of p_i might interrupt the execution of a_i and prevent any progress of the mobile agent execution. During the time p_i is down, the execution of a_i and consequently the entire mobile agent execution cannot proceed. We say that the execution of a_i is *blocked*. Provided the availability of suitable recovery mechanisms, the execution of a_i on p_i proceeds when p_i recovers from the failure. Generally, a mobile agent execution is called *blocking* if a single failure renders progress in the mobile agent execution impossible until the failed component (e.g., machine, place, agent, or communication link) recovers. In contrast, a *nonblocking* mobile agent execution can continue the execution despite a single failure. We generalize this definition to *t-blocking*, that is, an approach blocks if t or more failures occur. Correspondingly, an approach is *t-nonblocking* if it can sustain t failures and still continue its execution. Hence, t in t-nonblocking specifies the *degree of nonblocking*. Unless explicitly defined otherwise, we use *blocking* to refer to *1-blocking* or *0-nonblocking*, and *nonblocking* to refer to *1-nonblocking*. Generally, blocking mobile agent executions are undesired. In particular, if the failed component does not recover, then the agent is lost and never returns to the agent owner. Moreover, long downtimes of components lead to very long response times and may be unacceptable for the agent owner. Hence, mobile agent executions are preferably nonblocking. Note that unfavorable outcomes (see Section 2.3) do not lead to blocking.

3.2. Agent Replication and the Exactly-Once Execution Problem

3.2.1. Replication to Prevent Blocking. Blocking can only be overcome by introducing redundancy. More specifically, if a place fails, the agent is executed on another place. However, redundancy of execution may result in multiple executions of (parts of) the mobile agent.

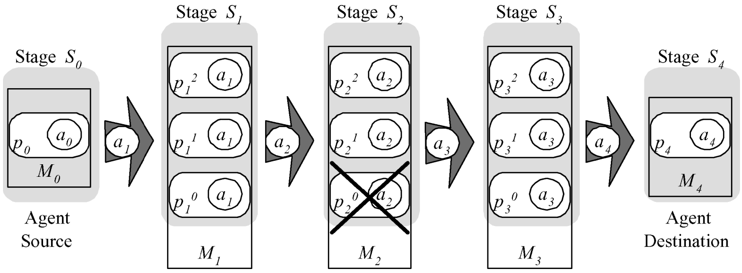

Fig. 2. Agent execution with redundant places, where a place fails. The redundant places mask the place failure.

While this is not a problem for idempotent operations (e.g., operations without side effects) it should not occur for nonidempotent operations. Take, for instance, an agent that retrieves money from the agent owner's bank account. This is clearly a nonidempotent operation and multiple executions of this operation have the undesired effect of multiple money retrievals. Therefore, nonidempotent stage actions must be executed exactly-once [Rothermel and Strasser 1998]. On the other hand, operations such as reading an account balance allow multiple executions. Clearly, blocking in a mobile agent execution consisting only of idempotent operations is easily prevented by sending multiple agents.

The redundancy introduced by replication masks failures and ensures progress of the mobile agent execution. Figure 2 illustrates the replication approach. At stage S_i, a set of places $M_i = \{p_i^0, p_i^1, p_i^2, ..\}$ executes the agent a_i. Even if place p_2^0 fails the agent a_2 is not lost, as the other places in M_2 have also received a_2 and can start executing it. Note that there is no need to replicate the agent at the agent source and destination. At the agent source, the agent is still under the control of the agent owner. The agent destination may be a mobile device, that is connected only intermittently to the network. Hence, mechanisms need to be implemented to store the agent until the mobile device connects again to the network. As the agent only presents the results to the agent owner at the agent destination, which is generally an idempotent operation, these mechanisms at the same time also address failures at the agent destination.

3.2.2. Properties of Places M_i. In Section 3.2.1 we have introduced replication as a way to overcome the problem of blocking. Replication occurs at the agent level: the agent replicas execute on different places $p_i^j \in M_i$ at a stage S_i. Depending on the relation among these places, we distinguish among three different classes of places: *iso-places*, *hetero-places*, and *hetero-places with witnesses* [Pleisch and Schiper 2000; Strasser and Rothermel 1998]. Iso-places correspond to the traditional case of server replication: the set M_i consists of replica places, where all places are provided by the same company. Revisiting our airline example, all places are provided by Swiss International Air Lines: modifications to one place are visible to the others as well. Consequently, executing a fault-tolerant mobile agent on iso-places leads to two levels of replication: server replication in the places (i.e., Swiss International Air Lines' servers) and client replication on the agent level.

Within the class of iso-places, we can further distinguish between places p_i^j, where the modifications are propagated (1) by the places themselves or (2) by the agent replicas. In (1), called *replicated iso-places*, the places run a replication protocol, which ensures consistency among the place replicas. Note that executing the mobile agent a_i on two such iso-places in M_i at stage S_i causes all iso-places in M_i

to reflect the modifications twice.[5] In (2), on the other hand, the agent replicas update the iso-places in M_i. The mobile agent thus ensures consistency of the replicas. However, this requires that an instance of the agent (i) execute on all replicas and (ii) must not fail as long as the service is up and running. We refer to this case as *independent iso-places*.[6] Independent iso-places are assumed for instance in [Schneider 1997].

Hetero-places correspond to a set M_i of places (see Figure 2) that all provide a similar service such as selling airline tickets from Geneva to New York. However, the places are provided by different airlines, e.g., Swiss International Air Lines, Delta Airlines, and Lufthansa.

Finally, hetero-places with witnesses are a generalization of hetero-places. Whereas hetero-places all provide the particular service (i.e., airline tickets from Geneva 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, do not provide an airline ticket service to the agent and thus the service request of the agent fails. However, the agent is not lost and proceeds with the execution, while potentially reporting the failed ticket acquisition to the agent owner. In general, a witness is a place that can execute the mobile agent (e.g., provides a Java Virtual Machine capable of executing the mobile agent), but does not provide the particular service required by the agent. Hence, the agent request to the service fails (for instance, raises an exception that is caught by some exception handler), but the agent can continue its execution despite an infrastructure failure of a place in M_i. Note that the execution of an agent replica on a witness generally results in an unfavorable outcome, as the requested service is not installed on the witness place.

Approaches for fault-tolerant mobile agent executions generally address replicated[7] iso-places, hetero-places, and hetero-places with witnesses. Clearly, it is desirable that the agent execute on only one place per stage unless a failure occurs. This allows a significant reduction of the overhead and improves the performance of the system.

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, then the agent cannot proceed with 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 concisely define *blocking* (see Section 3.1).

Moreover, catastrophic failures may still cause the loss of the entire agent. A failure of all places in M_2 (see Figure 2), for instance, is such a catastrophic failure. As no copy of a_2 is available any more, the agent a_2 is lost and, obviously, the agent execution cannot proceed. In other words, replication does not solve all problems. It is important to keep this in mind.

3.2.3. Replication and the Exactly-Once Problem.

Replication allows executions to be nonblocking. However, it may also lead to multiple agent executions. Assume, for instance, that p_i^0 fails (see Figure 3). Place p_i^1 starts executing a_i, which results in agent a_{i+1} and M_{i+1}. In the meantime, p_i^0 recovers and continues the execution of a_i. Clearly, this requires that the agent's state and code have been checkpointed to stable storage upon arrival of the agent on p_i^0. If

[5] This is true, unless a mechanism (e.g., transaction IDs) is provided that prevents iso-places from executing the same operation twice. See Section 3.2.3 for a discussion on multiple executions of mobile agents.

[6] In Pleisch and Schiper [2000], replicated iso-places are called *nonintegrated iso-places*, whereas independent iso-places are called *integrated iso-places*.

[7] In the following, the term "iso-places" refers to replicated iso-places unless explicitly stated otherwise.

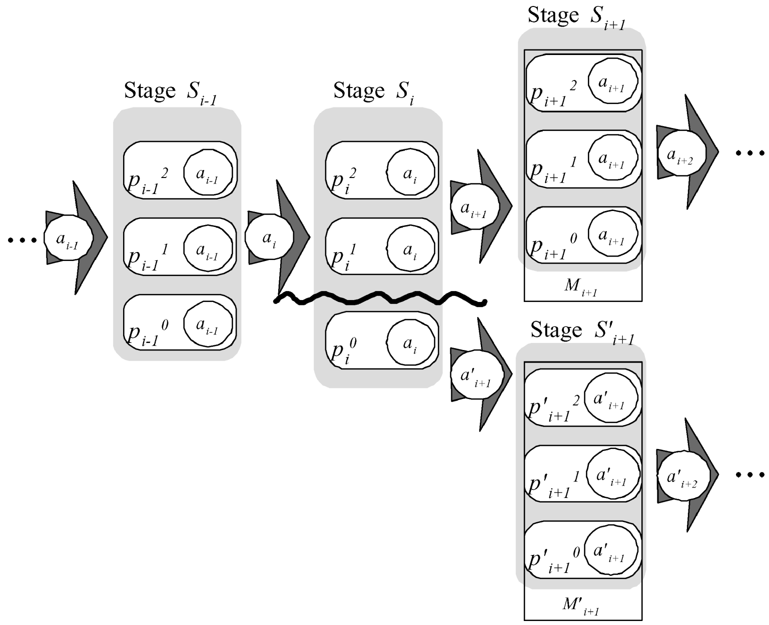

Fig. 3. Replication potentially leads to a violation of the exactly-once property.

p_i^0 and p_i^1 commit the agent's stage action, the agent is executed multiple times and results in duplicate agents a_{i+1} and a'_{i+1}. Although blocking of the agent execution because of a failure to p_i is prevented, the mechanism to prevent blocking results in multiple agent executions. Consequently, the problem of multiple agent executions and blocking are related problems in the sense that preventing blocking may lead to multiple agent executions.

Another source of a violation of the exactly-once execution property is unreliable failure detection. In asynchronous systems, such as the Internet, it is impossible to detect failures correctly (see Section 2.2). Even if a place p_i^k suspects the failure of another place p_i^j (i.e., believes that p_i^j has failed), p_i^j may not have failed in reality. Indeed, slow communication or processor speeds, or network partitioning may have caused p_i^k to erroneously suspect p_i^j. Therefore, when place p_i^1 suspects the failure of p_i^0, it starts executing a_i (see Figure 3). If the suspicion of p_i^1 was erroneous, the execution of a_i at stage S_i results in two agents a_{i+1} and a'_{i+1}; a violation to the exactly-once property.

In summary, we require that a fault-tolerant mobile agent execution satisfy the following liveness and safety properties [Alpern and Schneider 1985]:

t-Nonblocking. t failures must not prevent the termination of the agent execution (liveness).

Exactly-once. The mobile agent's stage actions are executed exactly-once (safety).

3.3. Agent Coordination

So far, we have not discussed the case of coordinating multiple agents to solve a higher-level task. For instance, an agent could split its task into subtasks and assign them to new agents, called child agents. Eventually, these child agents and the parent agent meet again to share the results. This case is more difficult to handle than the case of a single mobile agent execution and involves the creation of child agent(s) and, later, the coordination among them and the parent agent again.

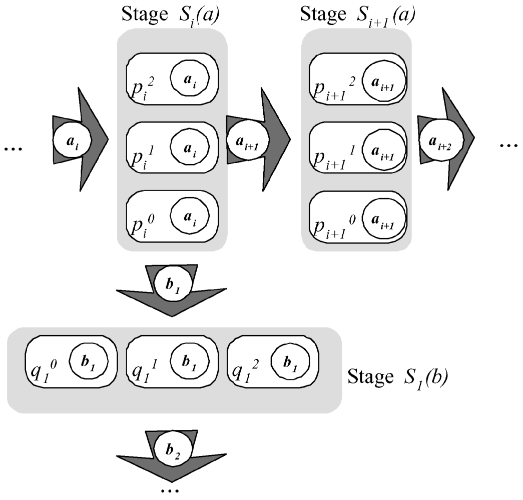

Fig. 4. Agent a_i spawns a new agent b_1 at stage $S_i(a)$.

3.3.1. Spawning Child Agents. The agent a_i at stage S_i can spawn a new agent b, which causes two agents to be forwarded by stage $S_i(a)$ (the identifier within the parenthesis distinguishes the stages of agent a from those of b): a_{i+1} and b_1 (see Figure 4). Assume, for instance, that the execution of a_i on place p_i^0 has led to agent b. If a_i on p_i^0 fails, then all its modifications have to be undone. In particular, the spawned agent b has to be undone.

3.3.2. Rejoin. The agent b spawned by agent a may rejoin agent a at a later stage in a's execution, say $S_k(a)$ $(k > i)$. In other words, the agents a and b meet again at stage $S_k(a)$. Between the stage at which a has created b and $S_k(a)$, a and b may follow different itineraries. As agent a is replicated at stage $S_k(a)$, a replica of agent b must be received on every place in $M_k(a)$ as well.[8] This ensures that no replica agent a_k^j waits forever for the arrival of agent b. Indeed, assume that a replica of agent b only arrives on place p_k^0 and that this place fails. In this case, the other replicas a_k^1 and a_k^2 still await the arrival of b and thus the execution of a cannot proceed. Clearly, agents a and b have to agree on their meeting point, that is, $M_k(a)$.

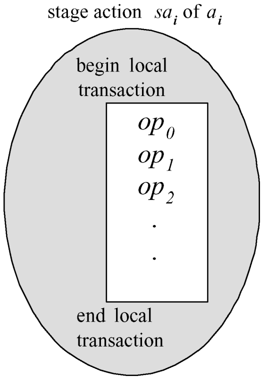

Fig. 5. Stage action of agent a_i runs as a local transaction.

4. BASIC BUILDING BLOCK: LOCAL TRANSACTION

In Section 3.1 we have specified the fault-tolerant mobile agent execution in terms of the non-blocking and exactly-once properties. In this section, we define a basic building block that is fundamental to enforce the exactly-once property and thus implicitly also the non-blocking property: the local transaction.

4.1. Local Transaction

The stage action sa_i of mobile agent a_i encompasses a set of operations $op_0, op_1, \ldots,$ that act on the local services (see Figure 5). Locally, on the place p_i, the agent executes the set of operations, thereby transforming a consistent state of the agent and the place into another consistent state[9] (consistency). The effects of executing sa_i have to be durable, that is, reflected by the place (new state of the place) as well as by the agent a_{i+1}, and not to be lost anymore (durability). Moreover, we require that sa_i execute entirely or not at all (atomicity). Only when sa_i has completed its execution should the results, including the modifications to the place, generated messages, or spawned child agents, be visible to other

[8] Actually, it may be sufficient under certain conditions that a majority of places in $M_k(a)$ receive a replica of b.

[9] The resulting agent is called a_{i+1} (see Section 2.1).

agents (isolation). These four properties correspond to the specification of an ACID transaction. Hence, sa_i has to run transactionally. This is ensured using a *local transaction* to execute sa_i. The concept of a local transaction is an important building block for fault-tolerant mobile agent execution.

The local transaction consisting of operations $op_0, op_1, \ldots$ terminates either by a commit or an abort decision. If the decision is to commit, the effects of executing $op_0, op_1, \ldots$ become durable, otherwise, all the modifications are undone. We classify the approaches for fault-tolerant mobile agent execution according to when and by whom this commit/abort decision is taken (see Section 5).

4.2. Enforcing Exactly-Once Execution Property

Executing the stage action sa_i transactionally is the basic mechanism allowing us to enforce the exactly-once property for mobile agent executions. Actually, whereas the entire mobile agent is executed exactly-once, the local operations $op_0, op_1, \ldots$ of a stage action are executed *at-most-once*.

4.2.1. At-Most-Once Ensured on Place p_i. Failures during the execution of an agent's stage action sa_i potentially leave the execution in an inconsistent state. More specifically, some of the operations $op_0, op_1, \ldots$ that correspond to the stage action may have been executed, whereas others have not. The agent a_i as well as the place (or rather, its services) are thus in an incorrect, transitory state. Executing sa_i transactionally prevents such inconsistent states, as either all the operations $op_0, op_1, \ldots$ are executed or none at all.

4.2.2. Exactly-Once for the Entire Mobile Agent Execution. In Section 3.2, we have shown how replication can prevent blocking. Replication may lead to multiple executions of a stage action sa_i on different places p_i^j and p_i^k. To prevent a violation to the exactly-once execution property, only one of the executions must be committed, whereas the other(s) have to be aborted. This is why stage actions are executed exactly-once. In Figure 3, for instance, the execution of a_i on place p_i^0 has to be aborted, whereas the execution of a_i on p_i^1 is committed. Running the stage executions as local transactions allows us, by controlling the commit/abort decision, to enforce the exactly-once property.

Note that terminating local transactions at stage S_i (i.e., issuing either abort or commit) requires that the place running the local transaction eventually recover from a failure and that potential link failures (i.e., network partitions) be resolved. However, such a link failure or place failure should not prevent the continuation of the agent execution, that is, they should not lead to blocking.

4.3. Handling Isolation of Local Transactions

Local transactions related to the execution of some agent can either make their results immediately visible to other mobile agents, or they only make the results visible when the outcome of the mobile agent execution is known. The former local transactions are called *open* local transactions, while the latter are *closed* local transactions. Undoing open local transactions related to the execution of some agent a potentially requires one to undo the operations of other agents that have used a's results in their computations, thus resulting in cascading undo operations. In practice, this is generally avoided. In the context of mobile agents, an open local transaction is immediately committed after its execution. If at a later stage this local transaction has to be undone, a so-called *compensating transaction* [Gray 1981; Korth et al. 1990; Garcia-Molina and Salem 1987] is run, which semantically undoes the effects of the corresponding local transaction. However, compensating transactions are not always possible. For instance, operations that send a message, print a check, or launch a rocket generally cannot be compensated. Moreover, if the local transaction has spawned a child agent (see Section 3.3.1), then this child agent may already have moved off. Hence, another agent has to be sent after this child agent to compensate all its activities. This requires that the compensating agent can deterministically recompute the exact itinerary of the original agent, and that all actions of the child agent are also compensatable. Note that sending an undo message to the child agent to trigger its rollback is not always successful either. Indeed, a slow undo message may never reach a fast-moving mobile agent, causing the undo to be delayed and increasing dependencies.[10] In summary, open local transactions are only suited for particular applications in a mobile agent environment.

In contrast, closed local transactions make their results only visible to other mobile agents when the outcome of the mobile agent execution is known, that is, when it is ensured that the local transaction will not be undone any more. Until the local transaction is committed, other agents can generally not access the data items that are accessed by the local transaction. Assume, for instance, that the local transaction uses a pessimistic concurrency control scheme based on locking. Any other agent can only access the locked data items when the local transaction commits and releases the locks on the data items. Besides locking, multiple concurrency control schemes exist. The reader is referred to Weikum and Vossens [2002] for an overview on existing concurrency control schemes.

5. CLASSIFICATION OF FAULT-TOLERANT MOBILE AGENT APPROACHES

In Section 4, we have identified the local transaction as the basic building block for fault-tolerant mobile agent execution (i.e., for addressing infrastructure failures). The stage actions of the mobile agent are executed as local transactions. Once the operations of the stage action are executed, the local transaction is either committed or aborted. We call this decision about which local transaction of a given stage to commit and which local transaction to abort the *commit decision*. This commit decision can happen at different moments in the execution of the mobile agent: (1) at the end of the stage execution (called *commit-after-stage*), or (2) at the end of the mobile agent execution, that is, at the agent destination (called *commit-at-destination*). Whereas in case (2), this decision is only made once for the entire mobile agent execution, case (1) requires one decision for every intermediate stage.

We first discuss the commit-after-stage and commit-at-destination cases in detail in Sections 5.1 and 5.2, and then compare them in Section 5.3.

5.1. Commit-After-Stage Approaches

The commit-after-stage approaches commit the stage actions at the end of every stage S_i before the agent moves to the next stage S_{i+1}. The commit is of particular importance if the mobile agent execution is replicated at stage S_i (see Figure 2). More specifically, the commit decision prevents multiple executions of the agent and thus ensures the exactly-once property. In this context, we need to distinguish between two cases: the execution of a_i (1) on a single place (i.e., a nonreplicated agent execution) and (2) on a set of places M_i (that is, a replicated agent execution). Moreover, the commit decision can be made by a single place or it can be distributed, that is, the decision can be made by multiple places. Finally, the commit decision can be collocated with the execution of a_i or not. Combinations of these three criteria lead to eight solutions, which can be represented in a three dimensional space (see Figure 6): (1) location of the agent execution, (2) location of the commit decision, (3) collocated/distributed.

In Section 3, we have shown how blocking can occur in the agent execution. Blocking also occurs in the commit decision. In particular, if the commit decision is made by only a single place, there is a risk of blocking or violating the exactly-once execution property to the mobile

[10] Murphy and Picco [1999] provide an approach to ensure reliable message delivery to an agent. However, this approach only works in an environment without failures and at a considerable cost.

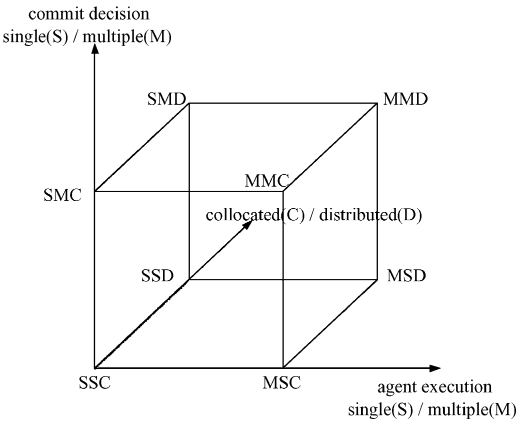

Fig. 6. Classification of fault-tolerant mobile agent approaches along three axes: (x) location of the agent execution, (y) location of the commit decision, and (z) collocated or distributed.

agent execution. For instance, in a two-phase commit (2PC) protocol, blocking occurs if the coordinator fails at a certain point in the protocol [Bernstein et al. 1987]. Moreover, network partitions may also prevent progress in the commit decision. Some protocols implementing the commit decision require the participation of a majority of places to reach a decision. However, the network may partition in such a way that no partition contains a majority of places and thus the commit decision protocols can only terminate when the partitions are merged again. We discuss now all eight solutions.

5.1.1. Single/Single/Collocated—SSC. The SSC solution encompasses the approaches where the stage action of an agent executes on a single place p_i, commits, and then the agent moves to the next place p_{i+1}. In other words, both the execution of the stage action and the commit decision occur on the same place (see Figure 7(a)). Actually, the outcome of the commit decision is always commit; abort is never decided, as there is no reason from the perspective of p_i to abort the agent execution. However, a failure of p_i causes blocking of the agent execution (see Figure 8). The SSC solution is thus 1-blocking. Moreover, if p_i does not recover, the agent (i.e., its code and state) is lost. Interestingly, the loss of the agent also leads to blocking, as the agent owner awaits the return of the agent. The use of standard logging and checkpointing mechanisms [Gray and Reuter 1993] for the state and code of the agent on the current place prevents the loss of the agent. However, it is still a blocking approach, as a failing place causes blocking of the mobile agent execution. Progress of the agent execution is only possible again when the failed place recovers. The recovering place thereby uses the latest local checkpoint to recover the agent a_i.

A SSC solution is most suited to environments in which failures are rare or where blocking is not a problem, either because of the nature of the application or because failed components recover quickly. The exactly-once execution property is ensured if a suitable recovery mechanism is used. Moreover, the agent must execute on a particular place in some applications. For instance, an agent that increments the value of a configuration parameter on a set of network switches must visit these switches. As network switches rarely fail (if they do, they are restarted quickly), paying the overhead of replication may not be appropriate in this context.

In SSC solutions based on checkpointing, the loss of the agent is generally prevented even in case of catastrophic failures. Both single and catastrophic failures prevent the progress of the agent, but the agent's code and state are preserved.

5.1.2. Single/Multiple/Collocated—SMC. Similar to the SSC solution, the stage action of the agent a_i is executed on one place p_i only. The commit decision, on the other hand, is distributed over multiple places (see Figure 7(b)). Consequently, a failure of p_i leads to blocking of the mobile agent execution. On the other hand, the commit decision is nonblocking, as it is distributed. The degree of non-blocking thereby is either constant or depends on the number of replicas that participate in the commit. It seems strange to distribute the commit decision, whereas the execution happens

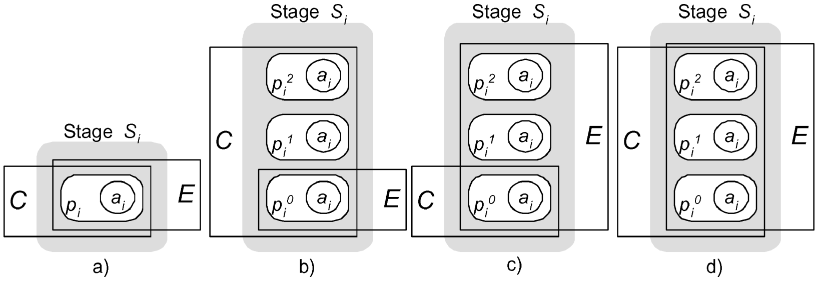

Fig. 7. Four solutions of collocated commits. E and C specify the places involved in the agent execution and the commit decision, respectively.

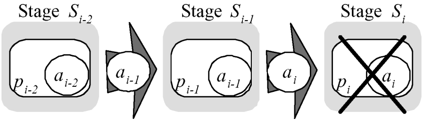

Fig. 8. Agent execution where place p_i fails while executing a_i. While p_i is down, the execution of a_i is blocked.

on a single place. Thus, the SMC solution has not been discussed in the literature.[11]

5.1.3. Multiple/Single/Collocated—MSC. The stage action of agent a_i is executed by multiple places, whereas the commit decision is made by a single place p_i^k (see Figure 7c). Revisiting Figure 3, p_i^0, p_i^1, and p_i^2 execute the agent, while p_i^2, for instance, executes the commit protocol. In the MSC solution, the commit decision determines the place that has executed the agent, called *primary* and denoted p_i^{prim}. All other places $p_i^j \neq p_i^{prim}$ abort all the modifications of a_i. For instance, p_i^2 decides that p_i^1 can commit the agent's operations, whereas p_i^0 and p_i^2 must abort them (if a_i has started execution on this particular place). This prevents multiple executions of a_i and thus a violation to the exactly-once property.

Although the execution of the stage action is non-blocking, blocking may occur in the commit protocol. This is because a single place executes the commit protocol. If this place fails, the commit decision blocks and thus the entire mobile agent execution is blocked. Consequently, MSC solutions decrease the probability of blocking compared to SSC solutions. This is especially true as the execution time for the commit decision is generally much shorter than the time needed for the agent execution.

Moreover, MSC solutions have practical relevance in environments where some places rarely fail. The commit decision can be executed on these places to reduce the probability of blocking.

5.1.4. Multiple/Multiple/Collocated—MMC. The MMC solution is a generalization of the solutions where the execution of the stage action and the commit decision are collocated. In other words, both the execution and the commit decision are distributed over multiple places (see Figure 7(d)). This distribution avoids blocking, but leads to the danger of violating the exactly-once execution property. To preserve the exactly-once property, the places that have executed the agent need to agree on the primary p_i^{prim} that commits the modifications performed by the agent while all other places abort them. In other words, unless an agreement is reached, that is, unless a so-called agreement problem is solved, among these places, multiple executions of the mobile agent cannot be prevented.

This solution makes the fewest assumptions about the environment. The overhead added by replication only makes

[11] The SMC solution may be applicable for scenarios where a mobile agent has to execute on very specific places. For example, the agent owner may want to fly only with Swiss International Air Lines, which only supports a non-replicated service.

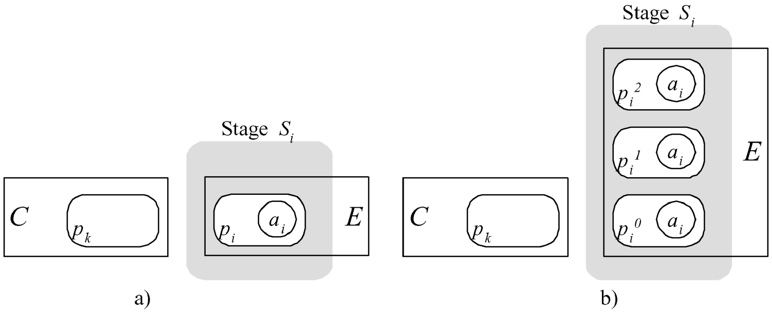

Fig. 9. Two solutions of distributed commits by place p_k. E and C specify the places involved in the agent execution and the commit decision, respectively.

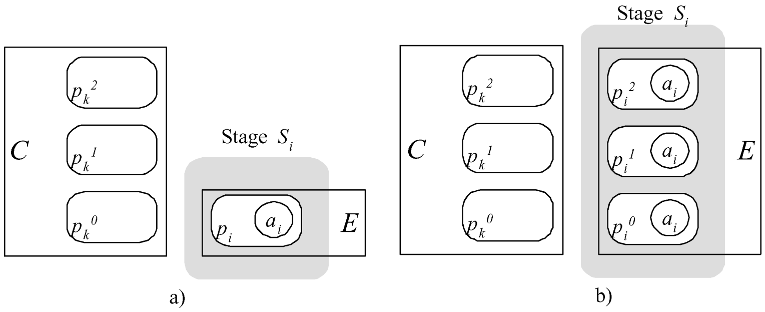

Fig. 10. Two solutions of distributed commits by multiple places. E and C specify the places involved in the agent execution and the commit decision, respectively.

sense for applications that have stringent requirements regarding fault tolerance and nonblocking, such as e-commerce applications.

5.1.5. Single/Single/Distributed—SSD. The SSD solution corresponds to SSC, except that the execution of the agent and the commit decision are not collocated (see Figure 9(a)). In other words, the place that executes the agent and the place that makes the commit decision are not the same. More specifically, any place p_k can make the commit decision, which then has to be communicated to p_i. This communication is prone to link failures. Moreover, the separation of the execution of a_i and the commit decision actually weakens the fault tolerance of the agent execution. Indeed, the probability that p_i and p_k do not fail is smaller than the probability that p_i does not fail. Consequently, the probability of blocking is higher, and this solution is less interesting than SSC.

To our knowledge, the SSD approach has not been implemented. However, we show below that this solution is of considerable interest to transactional mobile agents (see Section 7).

5.1.6. Single/Multiple/Distributed—SMD. This solution is similar to SMC and is not discussed further. It is depicted in Figure 10(a).

5.1.7. Multiple/Single/Distributed—MSD. A set of places executes the stage action of a_i, whereas the commit decision is located on any single place p_k (see Figure 9(b)). The execution of the stage action is non-blocking, but the commit decision can block. Failures in the communication channel between p_k and the places that execute the stage action of a_i may also lead to blocking.

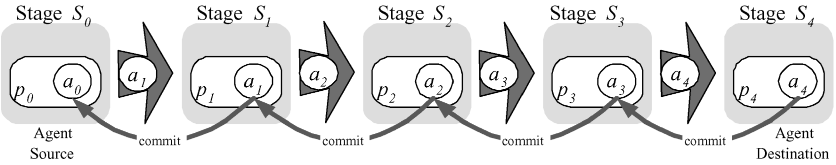

Fig. 11. Local transactions are only committed when the agent arrives at the agent destination. In this example, we assume pessimistic concurrency control.

5.1.8. Multiple/Multiple/Distributed—MMD. To circumvent the problem of blocking, both the execution of the stage action of a_i as well as the commit decision are distributed to a disjoint set of places (see Figure 10(b)). The main difference to MMC is that MMC can exploit the locality of commit decision and the execution of a_i. Moreover, MMC does not suffer from communication failures (i.e., network partitioning) between the places executing the stage action (i.e., p_i^j) and the places executing the commit protocol (i.e., p_k^j).

5.2. Commit-At-Destination Approaches

In contrast to commit-after-stage approaches, the stage actions of the mobile agent are only committed at the end of the agent execution. Whereas in commit-after-stage approaches duplicate agents are detected and discarded at each stage, duplicates continue their execution in commit-at-destination approaches. Duplicates can only be detected at a common place, where they and the original agent meet. Generally, only the agent destination is such a common place, because dynamic itineraries may be different for the original agent and among the duplicates. Hence, the agent destination is the only place where a correct decision about which agent (original or duplicates) to commit and which to discard is possible. Usually, the first arriving agent is committed, whereas the later arriving agent(s) are aborted and their stage actions undone. This allows us to ensure the exactly-once property for fault-tolerant mobile agent execution.

Until the agent has reached the agent destination, the local transactions are not committed/aborted. Rather, they are kept unterminated until the outcome of the agent execution is determined. Indeed, at the moment of executing stage action sa_i of agent a it is not clear whether a is committed or whether a duplicate agent will arrive first at the agent destination and a thus needs to be aborted. With closed local transactions (see Section 4.3), data items that are accessed by the mobile agent a are generally only accessible to another agent b when the corresponding local transaction is committed. During this time, b has to wait until the data items become available. Committing the agent's stage actions only at the agent destination makes all data accessed data items only available when agent a arrives at the agent destination. As other agents have to wait before accessing the data items until a finishes its execution, overall system throughput is seriously reduced. Moreover, this approach requires sending additional messages to all places of the itinerary to either commit or abort the stage actions once the agent has arrived at its destination (see Figure 11).

Liberating data items only when the agent has reached the agent destination and the sequential access to the data items of different stages may lead to deadlocks. Deadlocks can be handled by waiting for data items only for a limited time. If a timeout occurs, a deadlock is assumed and the agents back off. If other places with similar services are available, they may try those, or otherwise retry the same service. Although this approach does not completely rule out livelocks, their probability can be made sufficiently small.

As discussed in Section 4.3, some applications allow open local transactions. Using this approach, data items are made available to other agents immediately

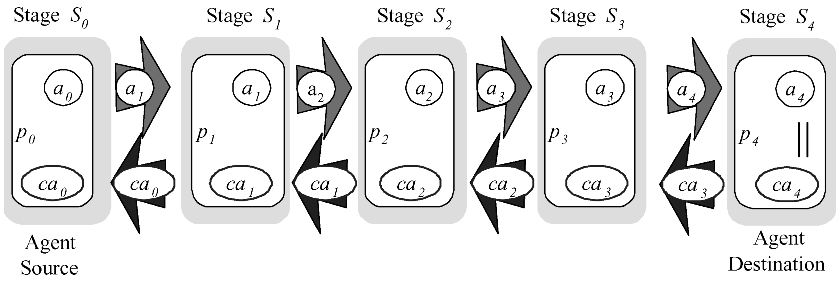

Fig. 12. At the agent destination, a compensation agent ca is created that runs the compensating transactions.

after stage execution. Revisiting the example in Figure 11, the compensating transactions are run in the inverse order of the agent execution (i.e., on $p_3, p_2, \ldots, p_0$). Indeed, executing sa_i followed by the corresponding compensating transaction may result in an agent different from a_i. Assume, for instance, that sa_i buys a book using e-cash [Strasser and Rothermel 2000]. Undoing this local transaction means to return the book and be reimbursed the amount paid, potentially less some penalty. As the change of the agent's state may also have an impact on the compensation of stage S_{i-1}, the inverse order is necessary. Note that if the agent state is the same for every stage as before executing the stage action, then compensation may run in parallel. To execute the compensating transactions, compensating agent ca is created (see Figure 12). Recall that not all applications can be compensated (see Section 4.3). Although feasible, compensation may also be unsuitable because of unacceptable run-time costs. This is especially true in environments with frequent false failure detections. Indeed, the use of compensation transactions makes an abort very expensive.

We classify the approaches similar to Section 5.1, but focus on the only two meaningful classes: SSD and MSD.

5.2.1. Single/Single/Distributed—SSD. This solution is similar to the SSD solution for commit-after-stage approaches, except that the commit decision occurs only at the agent destination for all stage actions (see Figure 11). It is blocking, but, on the other hand, also prevents duplicate agents. Clearly, the SSD commit-at-destination solution is only of theoretical interest. As the commit decision is always commit, the local transactions could be committed immediately after the stage execution (as in SSD commit-after-stage) instead of waiting until the agent reaches the agent destination. We present this solution here because it clarifies the difference between nontransactional and transactional mobile agent execution (Section 7.1).

5.2.2. Multiple/Single/Distributed—MSD. Contrary to the SSD approach, blocking is prevented by executing the agent on multiple places, if necessary. Because previous places already have a copy of the agent, they generally take over once the current place fails. More specifically, while the agent is executing on place p_i at stage S_i, its execution is monitored by the previous place p_{i-1}. In addition, p_{i-1} maintains a copy of the agent a_i. If a failure occurs at the current place p_i, place p_{i-1} launches its copy of the agent and sends it to another place p'_i (see Figure 13). Sending a_i to p'_i, however, may lead to duplicate agents, especially in the presence of unreliable failure detection. Duplicates are only detected at the agent destination, where the commit decision is made. This allows to enforce the exactly-once property to nonblocking fault-tolerant mobile agent executions. In Figure 14, duplicate agent a' is undone,

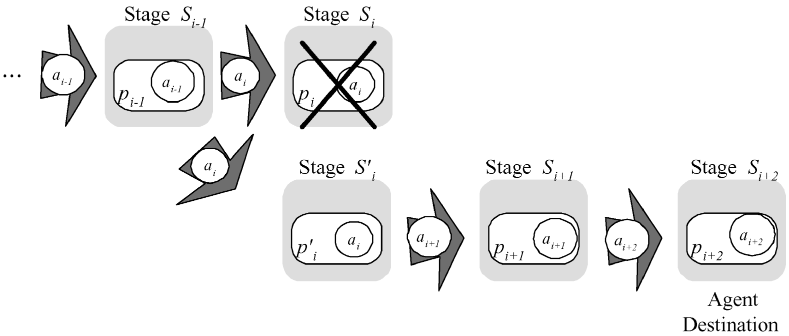

Fig. 13. Local transactions are only committed when the agent arrives at the agent destination.

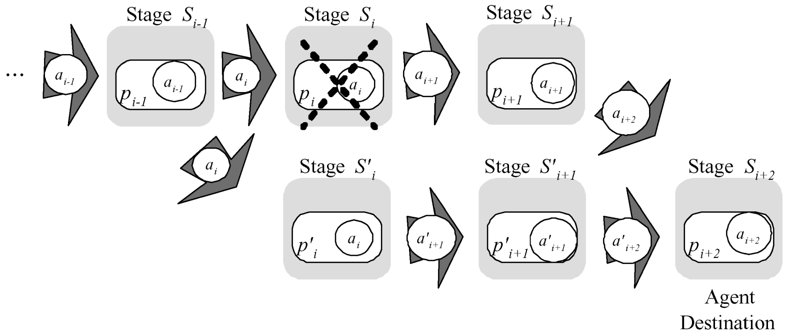

Fig. 14. Unreliable failure detection may lead to duplicate agents. They can generally only be detected at the agent destination.

that is, its stage actions on $p'_{i+2}, \ldots, p'_i$ are undone.

With open local transactions, compensating transactions are run on the places p'_{i+1} and p'_i, when it recovers. Executing compensating transactions generally modifies the state of the agent and the place. However, compensating duplicate agents must lead to the same state the agent had on the place that erroneously detected the failure. In Figure 14, the duplicate agent has been created by p_{i-1}. Compensating the duplicated agent must lead again to agent a_i. Otherwise, the state of the agent at the agent destination (a_{i+2}) is no longer valid. Compensating transactions are thus only possible if the state of the agent is not changed on p_{i-1}. Assume, for instance, that agent a_i carries $100 of electronic cash with it. If compensating the stage actions on places p'_{i+1} and p'_i costs a penalty of $20, then the state of agent a_i on place p_{i-1} is now $80. However, the agent a_{i+1} is not aware of this and thus an inconsistency arises.

The degree of fault tolerance is determined by the number of copies stored on places where the agent has previously executed. In other words, if the agent is currently executing on place p_i, places $p_{i-1}, p_{i-2}, \ldots$ may store their copy $a_i, a_{i-1}, \ldots$, respectively, of the agent. The higher this number, the more concurrent failures can be tolerated. For instance, assume that copies of the agent are stored at d predecessor places. In this case, the MSD solution is d-nonblocking. However, a high number also increases the probability of duplicate agents.

On recovery of a failed agent, we have to distinguish between two cases: the agent (1) has executed only partially on this place or (2) has executed the entire stage action and forwarded the agent to the next place. In (1) the recovering agent can abort/undo the partial execution of the stage action. Case (2) is more complex because the agent does not know whether the forwarding has succeeded and whether it is part of the successful mobile agent execution. Hence, it has to wait until it receives either a commit or an abort message. This message may arrive from its successor if the agent has reached the agent destination or from its predecessor if the forwarding to the next place has failed (e.g., from p_{i-1} in Figure 13).

5.3. Comparison

In the case of approaches with agent execution on multiple places (i.e., MSC, MSD, MMC, MMD), the most important differences between committing at the agent destination and committing at the stage execution are (1) the lifetime of duplicate agents and (2) the number of commit decisions. The lifetime is crucial as it influences the time data items are unavailable to other agents. The greater the lifetime, the longer data items remain unavailable. During this time, other mobile agents cannot access the data items and have to wait, which limits overall system throughput. Committing at the stage execution generally detects duplicates on a stage level; their lifetime is limited to a stage execution. In contrast, a commit at the agent destination generally only makes the data items available at the end of the agent execution. Clearly, this is a disadvantage of the commit-at-destination approach. At the agent destination, the modifications of one agent are committed, while all duplicate agents are detected and their effects undone. Undoing and committing agent stage actions requires that additional messages be sent to all places of the itinerary.

Using open local transactions (i.e., compensating transactions), data items can be accessed by other agents at the end of the stage execution, similar to commit-after-stage approaches. However, duplicate agents become very costly, as their stage actions have to be compensated. The longer the lifetime of a duplicate agent, the more costly its undoing becomes.

Another disadvantage of commit-at-destination approaches is the need to store copies of the agent's state as well as code at multiple locations. This requires a considerable amount of storage. Although mobile agents are generally small, a large number of them still imposes considerable storage requirements on the places. Generally, the copies of the mobile agent have to be maintained until the mobile agent execution has terminated, that is, until the commit/abort message has been received. In the commit-after-stage approach, copies of the agent are stored on the places in M_i only during the stage S_i and then discarded.

On the other hand, committing at the agent destination is more efficient with respect to the number of commit decisions. Whereas committing at the agent destination requires only one commit decision, committing at the stage execution requires $n-2$, that is, one at all places except agent source and destination. Moreover, agent replicas are only launched if a failure is (potentially erroneously) detected. In contrast, commit-after-stage approaches send multiple replica agents to the next stage although no failure may have been detected.

6. APPROACHES TO FAULT-TOLERANT MOBILE AGENT EXECUTION

In this section, we present a survey of existing approaches to fault-tolerant mobile agents. Each approach is classified according to the classification presented in Section 5. First, we present commit-after-stage approaches (Section 6.1). Wherever possible, we estimate the message complexity of a stage execution. More specifically, we indicate the total number of messages sent and the messages in the critical path, that is, the messages that are needed by the agent execution to continue. We do not consider messages between processes on the same machines. For these calculations, we assume that all stages support replication degree m (where applicable) and that no infrastructure failures occur. Commit-at-destination approaches are discussed in Section 6.2. The results

Table I. Classification of the Existing Approaches

	Commit-after-Stage	Commit-at-Destination
SSC	[Walsh et al. 1998] (6.1.2), [Vogler et al. 1997a, 1997b] (6.1.12)	—
MSC	[Rothermel and Strasser 1998] (6.1.11)	—
MMC	[Schneider 1997] (6.1.1), [Pleisch and Schiper 2000] (6.1.5), [Mishra and Huang 2000] (6.1.9, SG-ARP), [Johansen et al. 1999] (6.1.10)	—
SMD	[Lyu and Wong 2003] (6.1.7), [Mishra and Huang 2000] (6.1.9, UC-ARP, WC-ARP)	—
MSD	—	[Mohindra et al. 2000] (6.2.1), [Silva et al. 2000] (6.2.2)
MMD	[Pears et al. 2003] (6.1.3), [Pals et al. 2000] (6.1.4), [Silva et al. 2000] (6.1.6), [Dasgupta 2000] (6.1.8), [Assis Silva and Popescu-Zeletin 1998] (6.1.11)	—

of our classification are summarized in Table I. Table II at the end of the section summarizes the message overhead of approaches for which sufficient information was available.

Note that several mobile agent systems provide a mechanism to make mobile agents persistent. This mechanism trivially achieves some level of fault tolerance. In particular, it allows a place to recover the state and code of a mobile agent in case of a failure and to restart the agent. However, persistence on its own is not sufficient, as the modifications to the state of the place by the failed agent need to be undone, especially with non-idempotent stage actions. In this survey, we thus do not consider any further approaches that only provide persistency and instead focus on approaches that provide more elaborate mechanisms.

6.1. Commit-After-Stage Approaches

We consider the following commit-after-stage approaches: the Byzantine failures approach [Minsky et al. 1996; Schneider 1997], Concordia [Walsh et al. 1998], the exception handling approach [Pears et al. 2003], FANTOMAS [Pals et al. 2000], Fatomas [Pleisch and Schiper 2000, 2000a], Lyu and Wong's approach [Lyu and Wong 2003], MAgNET [Dasgupta 2000], Mishra and Huang's ARP family of protocols [Mishra and Huang 2000], NAP [Johansen et al. 1999], the transaction and leader-election based approaches of Rothermel and Strasser [1998] and Assis Silva and Popescu-Zeletin [1998], and Vogler et al.'s [1997a, 1997b] approach.

6.1.1. Byzantine Failures Approach. Minsky et al. [1996] and Schneider [1997] propose multiple executions of the mobile agent as a fundamental approach to provide invulnerability against Byzantine failures, more specifically against attacks from malicious hosts on the mobile agent. As such, their approach also addresses nonblocking and exactly-once in the context of crash failures. Clearly, as it is designed to address Byzantine failures, it is not very efficient if only crash failures occur. Nevertheless, we present this approach here for completeness. A more detailed discussion of this approach is given in Section 9.

To achieve fault tolerance, all places $p_i^j \in M_i$ of a stage S_i execute a_i and commit the modifications. Although an adversary may corrupt a number of agents at a stage, this approach still allows one to safely deduce the true result of the stage execution if enough uncorrupted agents are left. Hence, the exactly-once execution property is not desired in this approach. The places in M_i are independent iso-places as defined in Section 3.2.2: Schneider assumes replica places without replica update mechanisms to maintain

Table II. Message Overhead of Existing Commit-After-Stage Approaches. We Assume Replication Degree m at All Stages and No Infrastructure Failures Nor False Suspicions

	Number of Msgs in Critical Path	Total Number of Msgs	Characteristics
Assis Silva and Popescu-Zeletin's approach (6.1.11) [Assis Silva and Popescu-Zeletin 1998]	$12\lceil\frac{m+1}{2}\rceil +$ $5m - 12$	$19m - 14$	non-blocking
FANTOMAS (6.1.4) [Pals et al. 2000]	4	4	violation of exactly-once
Fatomas (6.1.5) [Pleisch and Schiper 2000]	$2\lceil\frac{m+1}{2}\rceil + 2$	$4m +$ $m\lceil\frac{m+1}{2}\rceil^a$	non-blocking
Lyu and Wong's approach (6.1.7) [Lyu and Wong 2003]	4	4	blocking
MAgNET (6.1.8) [Dasgupta 2000]	$3n - 3i + 1$	$3n - 3i + 1$	static itinerary
NAP (6.1.10) [Johansen et al. 1999]	1	$2m^a$	assumes reliable failure detection
Rothermel and Strasser's approach (6.1.11) [Rothermel and Strasser 1998]	$6\lceil\frac{m+1}{2}\rceil - 2$	$8m - 4$	blocking in the protocol implementing the commit decision
Schneider's approach [Schneider 1997] (6.1.1)	k^{2b}	m^2	Byzantine failure model
Vogler et al.'s approach (6.1.12) [Vogler et al. 1997a, 1997b]	2	2	blocking

[a] Assuming a linear strategy for reliable multicast [Hadzilacos and Toueg 1994].
[b] k is the threshold needed to reconstruct the result in a (m, k) threshold scheme.

consistency and prevent stale data. Actually, an accurate level of consistency is maintained by executing the agent on all places. Schneider's approach can be classified as an MMC approach, where the places always unilaterally decide to commit the agent replica's modifications.

Schneider proposes an (m, k) threshold scheme[12] to correctly deduce the result of the previous stage execution. Hence, at each stage, k places have to receive at least k agents of the previous stage. No additional messages are needed for the commit, as every place decides commit unilaterally. As a result, k^2 messages are in the critical path. In total, m^2 messages are sent from the places at S_i to the places at S_{i+1}.

6.1.2. The Concordia Approach. Concordia provides a framework for the development and execution of mobile agent applications [Walsh et al. 1998]. Similar to the approach in Vogler et al. [1997a] (see Section 6.1.12), the agent is forwarded to the next stage using transactional message queues. Hence, Concordia uses a SSC approach. The loss of the agent is prevented by the use of a so-called persistent store manager (PSM). The PSM allows the agent to regularly checkpoint its state to stable storage. After a failure, the agent is retrieved from stable storage and restarted. However, this scheme only guarantees exactly-once for idempotent operations. Indeed, the new execution of the agent may lead to different effects.

6.1.3. The Exception Handling Approach. Pears et al. [2003] address fault-tolerant mobile agent execution in the particular case where the agent execution has no effect on the state of the places. Such operations are idempotent with respect to the place state. To prevent the loss of the mobile agent, the agent source (called *home agent server* in Pears et al. [2003]) monitors the agent execution and upon detection of a crash sends a duplicate agent.

[12] A (n, k) threshold scheme divides a secret into n fragments, where only possession of at least k fragments will allow the secret to be restored [Shamir 1979].

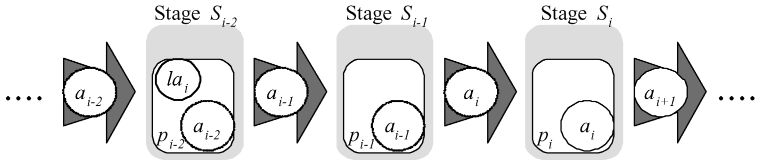

Fig. 15. Agent a_i with logger agent la_i and distance 2.

The drawback of this approach is that the agent is always executed from the beginning. As a remedy, Pears et al. adopt an approach based on a logger agent as proposed in FANTOMAS [Pals et al. 2000]. The concept of a logger agent is discussed in more detail in Section 6.1.4.

6.1.4. FANTOMAS. Pals et al. [2000] present FANTOMAS, an MMD approach that addresses transparent fault tolerance for distributed and parallel applications in cluster systems. Its fault tolerance mechanisms can be activated on request, according to the needs of the agent's task. FANTOMAS assumes only one place failure at a time. Associated with each agent is a so-called *logger agent* la, which follows the agent at distance d. For example, if the agent executes on p_i and the logger agent is on p_{i-2}, then d equals 2 (see Figure 15). The logger agent stores checkpoints of the agent with which it is associated. For this purpose, the agent periodically captures its state and sends it to the logger agent. The agent and its logger agent monitor each other and, upon a failure of one of them, the other can be restored from the information stored in the surviving one. Unless more than one place fails simultaneously, non-blocking is achieved. Hence, FANTOMAS is 1-non-blocking. Unfortunately, unreliable failure detection may lead to a violation of the exactly-once execution property. Indeed, assume that the logger agent erroneously detects the failure of the agent and recovers it. This results in two agents and thus in multiple executions of the agent's code. However, FANTOMAS addresses cluster systems, where erroneous failure suspicions can be assumed to be very rare. The problem of network partitioning is not addressed by FANTOMAS, that is, reliable communication is assumed. Indeed, network partitions would violate the exactly-once property if the network is partitioned such that the logger agent is in one partition and the agent in the other.

The approach in Pals et al. [2000] is efficient and can be dynamically switched on and off without interference of the agent owner. Moreover, the fault tolerance mechanisms are transparent to the agent owner.

FANTOMAS has a very low message complexity. Indeed, it requires that four messages be sent per stage: one message to send the updates to the logger agent and one to acknowledge its reception by the logger agent, one message to forward the logger agent, and one message to forward the agent. The number of messages in the critical path is the same as the total number of messages.

6.1.5. Fatomas. Pleisch and Schiper [2000, 2001a, 2003b] present a MMC commit-after-stage approach, called Fault-Tolerant Mobile Agent System (Fatomas). The redundancy illustrated in Figure 2 enables the mobile agent execution to proceed despite failures, that is, it 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 Pleisch and Schiper [2000, 2001a] assume a system model in which failure detection is unreliable. The solution presented in Pleisch and Schiper [2000] consists, for all agent replicas at stage S_i, of solving the *stage agreement problem*, which leads the agent replicas to agree on:

—the place that has executed the agent, called the *primary* and denoted p_i^{prim},

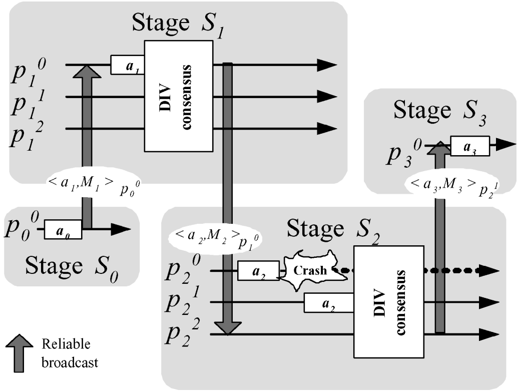

Fig. 16. Agent execution where p_2^0 fails. An erroneously suspected place p_2^0 leads to the same situation.

—the resulting agent a_{i+1}, and

—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 16 shows an example of a mobile agent execution spanning four 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 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 M_2.

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 of stage S_i, and (3) finally $\langle a_{i+1}, M_{i+1}\rangle$ is sent to the next stage.

Items (1) and (2) are performed together as part of a variant of the consensus problem, called Deferred Initial Value Consensus (DIV consensus for short) [Défago et al. 1998]. DIV consensus is the first building block of the Fatomas system. In the consensus problem, each process has an initial value at the beginning of consensus [Chandra and Toueg 1996]. Here, 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 desirable (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 (i.e., execute the agent) when requested by the DIV consensus algorithm. For example, if p_i^0 succeeds in computing its initial value and does not fail, 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.

Finally, 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 this case we have a $r-m$ communication schema: r senders have the same message to reliably broadcast to m destinations.

Fatomas is $\lfloor\frac{(m-1)}{2}\rfloor$-non-blocking. Catastrophic failures and network partitions may prevent progress of the mobile agent execution (liveness). However, the exactly-once property (safety) is always enforced.

The message complexity of Fatomas is as follows: $2\lceil\frac{m+1}{2}\rceil + 2$ messages are in the critical path of the fault-tolerant mobile agent execution at a stage. Reliable forwarding requires $\lceil\frac{m+1}{2}\rceil$ messages, which are sent concurrently in the optimal case. Overall, the stage execution and reliable forwarding of the mobile agent requires $4m + m\lceil\frac{m+1}{2}\rceil$ messages.

6.1.6. JAMES.

JAMES [Silva et al. 2000] is a system that belongs to MMD. However, it also has elements of a commit-at-destination approach and will be discussed in more detail in Section 6.2.2.

6.1.7. Lyu and Wong's Approach.

Lyu and Wong [2003] propose an SMD approach, in which all duplicates of agent a_i execute on the same place. As a consequence, failed places must eventually recover, but the mobile agent execution is blocked while the place is down. The stage actions of the agent are executed as local transactions. Checkpointing of the agent's state and logging of its operations are applied to facilitate the recovery of a failed agent, and to ensure exactly-once using rollback recovery.

The agent a_i at stage S_i is monitored by a *monitoring agent*[13] w_{i-1} executing at the place of the previous stage S_{i-1}. However, recursively, any monitoring agent w_k in turn is monitored by w_{k-1} executing on place p_{k-1} with ($k > 0$). It is assumed that the monitoring agent w_0 does not fail.

Upon the recovery of a failed place p_k, the monitoring agent w_k is recovered by w_{k-1}. If the agent a_i is suspected to have failed, w_{i-1} sends a probe to p_i, which, based on the information in the log, recovers the agent from the checkpoint (if the agent has really failed), or simply stops (if the agent has been erroneously suspected). Although not explicitly mentioned in their paper, Lyu and Wong [2003] assume a shared variable on place p_i to detect duplicate executions. Using this approach, they are able to handle concurrent failures of multiple monitoring agents.

In Lyu and Wong's approach, the execution of the agent at stage S_i requires four messages: one message to forward the agent a_{i+1} to the next stage, one message to send the witness w_i, and two messages to notify w_{i-1} of the arrival and departure of a_i and a_{i+1}, respectively. These four messages are also in the critical path.

6.1.8. MAgNET. The fault tolerance mechanisms in MAgNET are geared towards e-commerce applications [Dasgupta 2000]. MAgNET distinguishes between failures of agents and places, and failures of communication links and machines. While the former are handled using checkpointing (more particularly, the persistence feature of Aglets [Lange and Oshima 1998]), the latter are dealt with by using the following approach: at each stage S_i in the agent itinerary, a copy of the agent a_{i+1} is sent to all remaining places $p_{i+1}, \ldots, p_n$ ($i < n$) in the itinerary, which acknowledge the receipt of the agent. Place p_i also retains a copy of a_{i+1}. In contrast to most other approaches, MAgNET does not use predecessor places to store replicas of the agent, but the successor places. However, this limits the application of the MAgNET approach to static itineraries. Successor places that fail to send an acknowledgment to p_i are removed from the itinerary. Agent a_{i+1} is then sent again to the successor places in the updated itinerary. Upon reception of all acknowledgments from these places, the execution of the agent is started on the next place in the (updated) itinerary. Hence, MAgNET uses a MMD approach. Although not explicitly said in his paper, Dasgupta [2000] seems to assume perfect failure detection to maintain consistency. Indeed, while most duplicate agents can be prevented because of the static itinerary, the agent may still execute on a place that is removed from the itinerary because of a false suspicion.

At each stage S_i ($0 < i < n$) of the mobile agent execution, place p_i sends $n - i$ copies of the agent to the successor places and waits for $n - i$ acknowledgments. After reception of the acknowledgments, p_i sends an acknowledgment to p_{i-1} and a message to p_{i+1} to start the execution of a_{i+1}. Finally, p_{i+1} notifies all places that have received a copy of a_{i+1} that it is executing a_{i+1}. The total number of messages is $3n - 3i + 1$. The number of messages in the critical path is the same.

6.1.9. Mishra and Huang's ARP Family of Protocols. Mishra and Huang [2000] present three protocols to ensure fault-tolerant mobile agent execution: UC-ARP, WC-ARP, and SG-ARP. These protocols are based on the assumption that the agent source never fails.

In UC-ARP, which stands for *user-controlled agent recovery protocol*, the agent leaves a monitoring agent (called *watchdog* in Mishra and Huang [2000]) at each place and checkpoints its state before moving to the next place. Failures of the agent are detected by the agent owner, who contacts the latest monitoring agent in order to trigger recovery of the agent. With unreliable failure detection, duplicate agents may occur. Moreover, the consistency of the place and agent state is

[13] Note that monitoring agents are called *witness agents* in Lyu and Wong [2003]. To avoid confusion with witnesses in the context of heterogeneous places, we use the term monitoring agent in this article.

not ensured. Indeed, when the agent recovers, its state must reflect the changes its stage action has caused to the state of the place. This is also referred to as *output commit property* [Elnozahy 1992].

In WC-ARP, which stands for *watchdog-controlled agent recovery protocol*, the task of recovering the failed agent is left to the monitoring agents. This protocol suffers from the same limitations as UC-ARP. To prevent duplicate agents, the protocol may be forced to block upon a single infrastructure failure. Hence, we classify these protocols as SMD. Note that they could also be classified as MMD or commit-at-destination MSD, if duplicate agents were allowed to occur. However, no solution to the problem of duplicate agents is given in Mishra and Huang [2000]. This is why we consider these protocols as SMD.

Finally, SG-ARP (*server-group agent recovery protocol*) prevents duplicate agents and ensures place and agent state consistency. SG-ARP is an MMC approach and is based on replicated iso-places. As in the case of Fatomas (see Section 6.1.5), this requires that the iso-places agree on the place that executes the mobile agent. If (parts of) the mobile agent are executed multiple times, then consistency of the place state is no longer given. To our understanding, the iso-places use passive replication among themselves [Mishra 2001]. Hence, the agent is only executed on one replica (called the primary) and its updates to the place state are then applied to the backups. The use of passive replication suggests that SG-ARP is $\lfloor\frac{(m-1)}{2}\rfloor$-non-blocking. To prevent reexecution of a failed agent from the beginning, the mobile agent and place state is periodically checkpointed. For this purpose, all the iso-places have access to common stable storage. The execution of the mobile agent is then continued from the last checkpoint by another replica.

Mishra and Huang [2000] claim that this approach enables load balancing. However, load balancing is only possible if two agents a and b executed on iso-places p_i^j and p_i^k, respectively, do not access the same data items. Otherwise, a and b have to be executed sequentially. Executing a on p_i^j and b on p_i^k in this case requires to change primary, which is expensive.

6.1.10. NAP.

NAP [Johansen et al. 1999] uses the MMC approach to fault-tolerant mobile agent execution. It assumes a fail-stop model, which corresponds to a perfect failure detector [Sabel and Marzullo 1994]. Blocking is prevented by the nature of the MMC approach, whereas the exactly-once execution property is ensured by the assumption of perfect failure detector. Hence, no agreement as proposed in Pleisch and Schiper [2000] is required. Rather, perfect failure detectors allow the reliable detection of process crashes. In particular, no process is suspected unless it has failed, which eliminates one source for a violation of the exactly-once execution property. Still, local transactions are required in order to ensure at-most-once execution on the stage actions. Unfortunately, perfect failure detectors are impossible in the Internet and therefore NAP is only applicable in systems where perfect failure detectors can be assumed. Consequently, the NAP approach does not handle link failures nor does it consider recovery of places (see Section 2.2). The NAP approach is $(m-1)$-non-blocking, where m is the degree of replication at a stage.

The message complexity for NAP is 1 for the messages in the critical path and $2m$ for the total number of messages. However, this does not include the message overhead to implement the so-called fault-tolerant actions, that are at the basis of NAP. A fault-tolerant action consists of the action itself, say A, and an associated recovery action $\bar{A}$, which is executed exactly-once if the execution of A fails [Johansen et al. 1999].

6.1.11. Transaction and Leader-Election-Based Approaches

6.1.11.1. Rothermel and Strasser's Approach. Rothermel and Strasser's approach [Rothermel and Strasser 1998] corresponds to MSC, which is blocking. Indeed, a failure of the single commit place blocks the commit decision and thus also the mobile agent execution. The approach is based on transactions and leader election. The agent is forwarded between two consecutive stages S_i and S_{i+1} using transactional message queues. More specifically, a place p_i^j puts the agent a_{i+1} into the input message queue of p_{i+1}^k as part of a global transaction. This global transaction corresponds to the entire stage execution at S_i and encompasses (1) getting the agent a_i from the input message queue, (2) executing the agent's stage action, and (3) putting the resulting agent a_{i+1} into the message queue of the places in M_{i+1}. Multiple places in M_i potentially execute this transaction, but only the leader, elected by a leader election protocol, commits. All other places abort the agent's stage actions. Coupled with the use of local transactions this approach ensures exactly-once execution of the mobile agent, but is unfortunately vulnerable to blocking. This vulnerability is caused by the use of a 2PC protocol to atomically commit the transactions, which is known to be blocking on a single failure [Bernstein et al. 1987]. The reader may argue that the use of a 3-phase-commit (3PC) [Bernstein et al. 1987] alleviates the blocking problem. However, blocking stems from the combination of leader election and transactions and the nature of MSC and thus cannot be prevented by the use of a 3PC. Indeed, if the orchestrator that is managing the voting and is acting as resource manager fails after receiving a majority of YES votes and the 2PC or 3PC decides abort, then no new orchestrator can reach a majority of YES votes until the former orchestrator explicitly resigns. This, however, can only be done when the former orchestrator recovers from its failure, which is a blocking situation.

In Rothermel and Strasser's approach, $4(m-1)$ messages are needed for voting [Rothermel and Strasser 1998], and $2(\lceil \frac{m+1}{2} \rceil - 1)$ of them are in the critical path. The 2PC protocol involves communication primarily among local processes, but also among the transactional message queues. Indeed, sending the agent to the next stage and the 2PC protocol require $4m$ messages ($4\lceil \frac{m+1}{2} \rceil$ in the critical path). In summary, $6\lceil \frac{m+1}{2} \rceil - 2$ messages are in the critical path of a stage execution. The total number of messages generated is $8m - 4$.

6.1.11.2. Assis Silva and Popescu-Zeletin's Approach. Assis Silva and Popescu-Zeletin [1998] improve Rothermel and Strasser's algorithm by overcoming some of its limitations. In particular, to prevent the blocking problem in Rothermel and Strasser [1998], they use a different leader election protocol and commit the local transaction using a 3PC protocol [Bernstein et al. 1987]. As a consequence, Assis Silva and Popescu-Zeletin's approach achieves $\lfloor \frac{m-1}{2} \rfloor$-non-blocking. However, this particular combination of leader election and transaction model may lead to a violation of the exactly-once property. Hence, Assis Silva and Popescu-Zeletin [1998] relies on a so-called *distributed context database* to prevent more than one concurrent leader and thus to enforce the exactly-once property. In summary, the commit decision is made in collaboration with the distributed context database, a leader election protocol, and the 3PC. The distributed context database runs on the places of the stage S_i. However, to our understanding, the context database will generally be run by another process than the execution of the agent. Moreover, it can be implemented as a separate service. Consequently, we consider Assis Silva and Popescu-Zeletin [1998] a MMD approach.

Similarly to Rothermel and Strasser [1998], the approach in Assis Silva and Popescu-Zeletin [1998] uses transactions and leader election to model fault-tolerant mobile agent execution. Combining the two models makes it more difficult to understand the approach. Another disadvantage of this approach are the rather high maintenance costs for the distributed context database, which needs to be replicated (to provide fault tolerance).

According to Assis Silva [1999], writing a value into the distributed context database and committing it requires 7 $(m-1)$ messages. In addition, forwarding the agent to the next stage and running

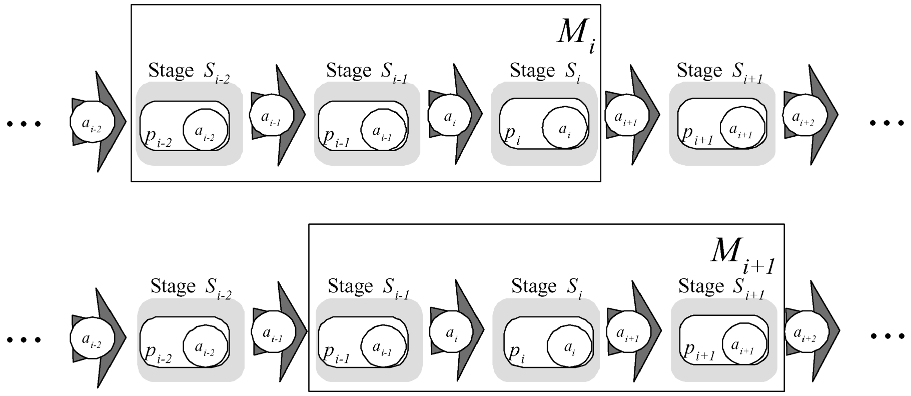

Fig. 17. Pipelined mode.

the 3PC protocol requires $5(m + m - 1)$ messages. Finally, $2(m - 1)$ messages are needed to notify the places of the current stage of the termination of the stage execution. In summary, the message complexity is $19m - 14$. To compute the number of messages in the critical path, we only consider the messages from a majority of places in the 3PC: $7(\lceil \frac{m+1}{2} \rceil - 1) + 5(m + \lceil \frac{m+1}{2} \rceil - 1)$. The termination notification messages are not in the critical path. In total, $5m + 12\lceil \frac{m+1}{2} \rceil - 12$ messages are in the critical path.

6.1.12. Vogler et al.'s Approach. Vogler et al. [1997a, 1997b] use the SSC approach. Their main focus is to ensure exactly-once semantics for the transfer of the agent between two consecutive places p_i and p_{i+1}. To achieve this, p_i starts a transaction, which encompasses sending the agent, storing the agent at p_{i+1}, initiating the agent at p_{i+1}, and deleting the copy of the agent at p_i. A 2PC protocol is used to ensure the ACID properties of this transaction. To our understanding, failures of the agent while executing the stage action at the place are not addressed. However, the fact that a copy of the agent is stored at the destination allows to recover from a place failure and redo the local transaction from the beginning (see Section 4). This corresponds to a checkpointing approach (see Section 5.1.1) where a checkpoint is taken before executing the stage action sa_i. Clearly, Vogler et al.'s approach is 1-blocking, but ensures exactly-once mobile agent execution properties provided that the stage actions run as local transactions.

Vogler et al.'s approach needs one message to send the agent and one message to acknowledge its receipt.

6.1.13. Pipelined Mode. One drawback of the approaches where multiple places handle the execution of the stage action (i.e., MSC, MSD, MMC, MMD) is the need for multiple places M_i at each stage S_i. This also adds an overhead to the communication between consecutive stages. Reusing places of previous stages for the current stage execution improves the performance and prevents high messaging costs [Strasser and Rothermel 1998; Johansen et al. 1999; Pleisch and Schiper 2000]. Figure 17 illustrates the pipelined mode with replication degree 3. At stage S_i, only place p_i is given as next destination, while p_{i-1} and p_{i-2} are reused. Usually, p_{i-2} and p_{i-1} are witnesses (see Section 3.2.2) to the execution on p_i. However, iso-places and hetero-places are also supported by the pipelined mode, although their practical use is limited.

6.2. Commit-At-Destination Approaches

The most prominent example of commit-at-destination approaches is a system called NetPebbles [Mohindra et al. 2000].

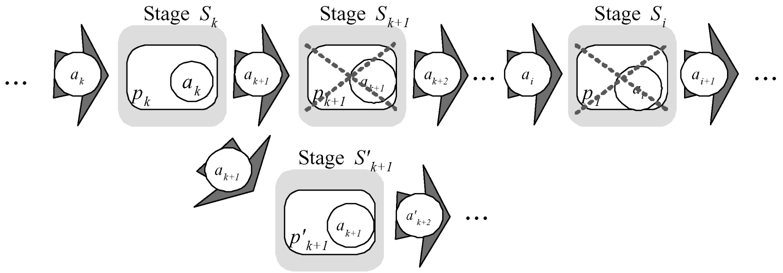

Fig. 18. Duplicate agents caused by unreliable failure detection in the commit-at-destination approach.

It belongs to the class of commit-at-destination MSD approaches. The JAMES system [Silva et al. 2000] can also be classified into this category, although it has some elements of the commit-after-stage MMD approach.

6.2.1. NetPebbles. The NetPebbles environment [Mohindra et al. 2000] defines an agent as a script that moves among places. This script contains the code to be executed at any place. Fault tolerance is based on the observation that choices exist in the task to execute (i.e., the stage actions) as well as in the location where to execute a task (i.e., the itinerary). Based on these choices, the script can route around failures of both the network and the places. Fault tolerance is achieved by the following mechanism in NetPebbles: As shown in Figure 13, place p_{i-1} keeps a copy of a_i. When it detects a failure of agent a_i on place p_i, this copy is sent to another place p'_i. Monitoring the current agent execution a_i by place p_{i-1} of the previous stage allows NetPebbles to tolerate any number of sequentially occurring failures to $p_i, p'_i, p''_i, \ldots$. Indeed, assume that p'_i also fails. The failure of p'_i is eventually detected by p_{i-1} and a copy of a_i is also sent to another place p''_i. However, a simultaneous failure of p_i and p_{i-1} results in the loss of the agent and thus in a blocking execution. NetPebbles overcomes this problem by setting up a monitoring scheme where places of previous stages monitor their successor places. Every place sends heartbeat messages to the previous places within a certain distance. This distance is defined as the difference between the indices, that is, $j - k$, of two places p_k and p_j, $(j > k)$. The heartbeat frequency decreases with increasing distance. In other words, the greater the difference between j and k, the lower the frequency p_j uses to send heartbeats to p_k. Place p_k sends the agent a_{k+1} to another place p'_{k+1} if and only if it suspects that all successor places have failed, that is, if it stops receiving heartbeat messages. This allows NetPebbles to handle a number of concurrent failures equivalent to the distance value. In other words, NetPebbles is $(j - k)$-non-blocking.

As the places within this distance do not solve any agreement problem, they cannot prevent agent duplicates. NetPebbles, however, assumes that the agent destination is the same as the agent source. Hence, all surviving duplicate agents (including the original agent) eventually arrive at the agent destination. At this point, the first arriving agent (either the original or any duplicate) is committed, whereas the actions of all others have to be aborted. The problem of how to commit or abort the actions of the duplicate agents is left open in Mohindra et al. [2000].

Using closed local transactions, data items are not available to any other agent until the end of the agent execution, even if the agent does not fail. Indeed, assume that the agent executes at stage S_i (see Figure 18). Owing to a network partition or slow communication links, place p_k no longer receives heartbeat messages from any p_j $(j > k)$ and thus suspects the failure of all successor places. It sends a copy of the agent a_{k+1} to a place p'_{k+1}, resulting in a duplicate agent a'_{k+2}, although the original agent execution has long passed stage S_k and is currently executing on p_i ($i > k + 2$). Hence, data items can only be liberated at the agent destination.

6.2.2. JAMES. JAMES [Silva et al. 2000], a Java-based mobile agent infrastructure, is a platform that provides a running environment for mobile agents, with enhanced support for network management. An agency of the JAMES platform corresponds to a place in our model. JAMES defines agent managers, which act as agent source and allow them to manage and monitor running agents. It provides fault tolerance support for mobile agents, but does not ensure exactly-once agent execution. Rather, it uses at-most-once or at-least-once execution semantics. These semantics are weaker than the exactly-once property (exactly-once stage action is equivalent to a stage action that is executed at-least-once *and* at-most-once). In addition, the mobile agent either executes on all places of its itinerary (called *atomic*) or on the maximum possible (*best-effort*). The occurrence of duplicate agents is justified for certain execution semantics, such as best-effort agent execution and at-least-once execution of the agent's stage actions. These execution semantics seem to address the aspects of network management considered in Silva et al. [2000], although no explicit examples are given.

When the failure of the agent currently executing, that is, a_i, is detected, the place with the most recent copy of the agent starts executing the agent. This place is elected using an election protocol and generally defaults to the predecessor place p_{i-1} (see Figure 13). With this approach, blocking is prevented but agent duplicates may occur (see Section 5.2). In JAMES, a fault-tolerant lookup directory prevents agent duplicates that are not caused by network partitions. Network partitions may disrupt the communication between places and the lookup directory and thus either cause blocking or duplicate agents. The lookup directory is replicated and provides exclusive access to its methods. Every agent a_i, once it has executed the stage action, inserts a corresponding entry into the lookup directory. If such an entry exists already, then another agent has already executed the actions of this stage and the current agent rolls back its stage actions and commits suicide. Otherwise a_i sets the corresponding entry in the lookup directory to reflect the fact that the stage action of a_i has terminated. The replicated, fault-tolerant lookup directory can be seen as the distributed commit decision in MMD, which decides which agent to commit (i.e., $a_i, a'_i, \ldots$). This, together with the execution of a_i on potentially multiple places (i.e., $p_i, p'_i, \ldots$), shows that JAMES has some elements of a commit-after-stage MMD approach.

However, the lookup directory is not sufficient to ensure the exactly-once semantics. Assume that the agent currently executes on place p_{i+1}. Hence, the corresponding entry in the lookup directory indicates that the execution of a_i has finished. Assume further that p_i and p_{i+1} are suspected by the previous places. A run of the election protocol identifies p_{i-1} as the place with the latest available state of the agent (i.e., a_{i-1}). To our understanding, the entry in the lookup directory is of limited use in such a case. When p_{i-1} sends agent a_i to p'_i, the place p'_i has two choices: (1) take into consideration the information in the lookup directory and discard a_i, with the risk of blocking if suspecting p_i and p_{i+1} was accurate, or (2) ignore the status entry and execute a_i. The latter choice leads to a duplicate agent if p_i and p_{i+1} have been erroneously suspected. Duplicate agents can generally only be detected at the agent destination. This problem does not seem to be addressed in Silva et al. [2000].

The fault-tolerant, replicated lookup directory used in Silva et al. [2000] violates to some extent the autonomy assumption of mobile agent execution. Moreover, frequent updates to the lookup directory, such as in JAMES, are costly, as all replicas have to remain consistent.

7. TRANSACTIONAL MOBILE AGENTS

In this section, we present approaches for *transactional mobile agents* that address infrastructure and unfavorable outcomes and ensure atomicity of the entire mobile agent execution. Recall, that a transactional mobile agent is an agent whose stage actions execute atomically, that is, either all or none at all (see Section 2.4). We start with a comparison with nontransactional mobile agents, before presenting a model for transactional mobile agents based on open/closed nested transactions.

7.1. Commit in Nontransactional Mobile Agents vs. Commit in Transactional Mobile Agents

In Section 5, we have classified nontransactional mobile agent approaches according to when and by whom the commit decision of the stage action is performed. In the context of nontransactional mobile agents, the commit decision helps to ensure the exactly-once execution property of the mobile agent. Indeed, only the stage action on the primary of stage S_i is committed in the commit-after-stage approach, while the stage actions on other places in M_i are aborted. In the commit-at-destination approach, the commit decision leads to the selection of the duplicate agents arriving at the agent destination that have to be undone. This undo/abort occurs although the agent may have successfully executed at all stages. In contrast, transactional mobile agents use the commit to ensure atomicity in the execution of one mobile agent.

Consider, for instance, a nontransactional mobile agent using the commit-at-destination approach.[14] The diff erence between a commit-at-destination approach and a transactional mobile agent is best shown in the case where no failures and no false suspicions occur. In this context, a commit-at-destination approach always successfully executes the agent and commits the agent's stage operations. On the other hand, even with no failures and no false suspicions, transactional mobile agents might decide to abort the agent's stage operations; the success of the agent execution does not depend exclusively on the fact that the agent has reached the agent destination. Rather, it also depends on whether the stage operations were semantically successful. Revisiting the first example in Section 2.4, commit-at-destination approaches commit the agent's stage operation (i.e., book a hotel room and rent a car) although no flight is available. In contrast, transactional mobile agents either commit all three operations or abort them all. In other words, if no flight is available, all agent operations will be aborted. Whereas a commit-at-destination approach eventually commits, transactional mobile agents can also abort. This is because commit-at-destination approaches only need the commit to prevent agent duplicates, whereas transactional mobile agents use it to address unfavorable outcomes.

Because transactional mobile agent executions also address unfavorable outcomes, they have additional requirements. Indeed, a *transactional mobile agent* execution is specified in terms of the ACID properties. While in Section 4 we require ACID properties for local transactions, transactional mobile agents have to guarantee that the ACID properties encompass the entire mobile agent execution, that is, that the sequence $sa_0, \dots, sa_n$ runs transactionally.

7.2. Open Nested Transaction Model

A transactional mobile agent execution can be modelled as *open nested transactions* [Weikum and Schek 1992]. An open nested transaction is a transaction that is (recursively) decomposed into *subtransactions*. Every subtransaction forms a logically related subtask and can be either *open* or *closed*. An open subtransaction makes its results visible to other transactions as soon as its computation has successfully terminated, independent of the outcome of its parent transaction. In

[14] A similar reasoning also applies to commit-after-stage approaches.

contrast, a successful closed subtransaction only makes its updates visible to other transactions, that is, commits, if its parent transaction commits. The case when all subtransactions are closed subtransactions corresponds to the (closed) nested transactions of Moss [1985].[15] Contrary to flat transactions (i.e., nonnested transactions), in open nested transactions, a parent transaction can commit (provided that its parent transactions all commit in the case of closed subtransactions) although some of its subtransactions may not have been successful. In other words, some subtransactions may be aborted, but the parent transaction still commits.

In a transactional mobile agent execution, the top-level transaction (i.e., the transaction that has no parent) corresponds to the entire transactional mobile agent execution. The first level of subtransactions is composed of the stage actions sa_i. If replication is applied, each stage action, in turn, can be modelled by yet another level of subtransactions, which correspond to the agent replicas $a_i^0, \ldots, a_i^m$ running on the places in M_i and executing the set of operations $op_0, op_1, \ldots$. Transactional mobile agents are a simplification of general open nested transactions as the subtransactions generally neither conflict nor deadlock among themselves. Indeed, the subtransactions a_i^j execute on different places and thus run in complete isolation from each other. To further improve the level of concurrency, the services running on the places decide themselves whether to allow concurrent access to their data. For this purpose, each service has a so-called commutativity matrix [Rakotonirainy 1994], which shows potential conflicts among operations of this service and allows operations that do not conflict to be executed concurrently. The parent transaction of subtransaction sa_i^j (i.e., sa_i) only commits if exactly one of its subtransactions has committed. More specifically, it issues a commit only to one of its subtransactions (i.e., the primary sa_i^{prim}), and aborts all others. The top-level transaction only commits if all the subtransactions sa_i that must succeed are ready to commit (in the case of a closed subtransaction) or have already committed (open subtransaction).

If a service request fails on one place, the subtransaction sa_i can be aborted and retried on another place without aborting the top-level transaction. Assume, for instance, that an agent a_i attempts to book a flight on Swiss International Air Lines, but no seat is left. We do not consider replication at the moment and assume that no failures or false suspicions occur. Consequently, the corresponding subtransaction sa_i has an unfavorable outcome and is aborted. However, the agent may move on to the Lufthansa server and attempt to book a flight with Lufthansa. If this subtransaction sa_{i+1} is successful, the agent continues and the top-level transaction can still commit. More generally, the agent may perform a partial rollback to p_i that involves several stages (e.g., p_{i+1}, p_{i+2}), and then continue the execution along an alternative itinerary (e.g., $p_k, p_{k+1}, \ldots$) [Strasser and Rothermel 2000]. Note that this is fundamentally different from commit-after-stage approaches, where these servers are usually visited concurrently at the same stage. To simplify our discussion, we assume that all stage actions sa_i of a transactional mobile agent execution must succeed for the transaction to commit. Our observations remain valid for the general case, in which only a subset of the stage actions needs to succeed.

Mobile agents can be composed of open or closed subtransactions. A mobile agent execution that is composed of closed nested transactions suffers from the drawback that results of the stage actions are only visible to other agents at the end of the mobile agent execution. For long-living agents, this has a negative impact on the performance, as the agent has to wait until other agents have terminated their execution. In contrast, open nested transactions do not suffer from this problem and are thus more suited to long-living

[15] See also Chrysanthis and Ramamritham [1994] for a formal description of open and closed nested transactions.

agents. However, to ensure atomicity, committed subtransactions may need to be compensated. Unfortunately, not all stage actions can easily be compensated (see Section 4.3), or can only be compensated at a considerable cost. Indeed, between executing the stage action sa_i and its compensating transaction, another agent b can access data items modified by sa_i. Executing the compensation transaction semantically undoes the modifications performed by sa_i. Agent b may have now read an inconsistent value. Consequently, b needs to be aborted, leading to cascading aborts. Hence, compensatable transactions work best in an environment where compensating transactions can be run without causing cascading aborts. This is the case for a large number of applications (e.g., flight reservations).

Compensation may also be unsuitable although feasible, because of unacceptable run-time costs. This is especially true in environments with frequent aborts. The use of compensation transactions makes an abort very expensive. Moreover, all compensation transactions must eventually commit. Consequently, failures during the compensation transactions lead to blocking. In contrast, an abort with closed nested transactions is not more expensive than a commit in the sense that the message sent to all places contains the directive to abort instead of commit.

Hence, an ideal approach to transactional mobile agent execution supports both closed and open subtransactions. We distinguish between blocking and nonblocking approaches, but also indicate whether the approach supports closed nested transactions or open nested transactions. In the following, we discuss atomicity in more detail in the context of transactional mobile agents.

7.3. Execution Atomicity

In this section, we show how the ACID properties, in particular atomicity and durability (consistency and isolation are discussed in Section 4.3), can be ensured for a transactional mobile agent execution. Among the ACID properties of the top-level transaction, atomicity and durability[16] are the hardest to achieve. Atomicity encompasses unfavorable outcomes (see Section 2.4) by ensuring that either all stage actions are executed successfully or none of them. Note that in the case of open nested transactions the atomicity property is achieved if the modifications of all stage actions are reflected [Weikum and Schek 1992] or none at all. The latter case occurs if the modifications have been compensated for. Hence, the atomicity property is achieved on a more abstract level than in the case of closed subtransactions. To achieve atomicity, the agent a can decide at every stage S_i whether to continue or abort the transactional mobile agent execution, denoted T_a. If one subtransaction sa_i $(i < n)$ has an unfavorable outcome, that is, not executed successfully, T_a is immediately aborted (see Figure 19). Otherwise, a_i continues the execution of T_a. Commit can only be decided at the agent destination,[17] when all the sa_j $(0 < j \leq n)$ have been successfully executed. For both commit or abort, a_i reliably multicasts the decision to all p_k $(k < i)$ (in the case of closed subtransactions) or sends a compensating agent in the case of abort (open subtransactions). Similar

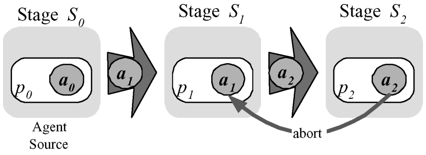

Fig. 19. Abort is immediately communicated to all predecessor places in the transactional mobile agent execution.

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

[17] Actually, the decision can be made on p_{n-1} [Pleisch and Schiper 2001b], as the agent destination generally contains only idempotent operations and may only be intermittently connected to the network if it is a mobile device. For simplicity, we thus assume in the following that the agent destination does not fail and is always connected.

to the case of commit-at-destination approaches (see Section 5.2), the compensating agent itinerary generally is the inverse itinerary of the corresponding agent a.

7.4. Addressing Infrastructure Failures

Infrastructure failures do not lead to the abort of the mobile agent execution, but may cause blocking (while the place currently executing the mobile agent fails, the transactional mobile agent execution is blocked). Aborting the transaction if infrastructure failures occur (i.e., on p_i) may lead to a violation of the atomicity property. Assume that the previous place p_{i-1} monitors the execution of the agent on place p_i. Incorrect failure detection may cause p_{i-1} to suspect p_i and thus to abort the transactional mobile agent T_a. However, a_i continues executing on p_i and is forwarded to p_{i+1}. If the agent destination decides commit, then all places $p_i, \ldots, p_n$ commit the agent's stage actions, whereas the places $p_0, \ldots, p_{i-1}$ have already previously aborted the stage actions. Clearly, this is a violation of the atomicity property. Consequently, transactional mobile agent approaches are generally either blocking or employ fault tolerance techniques (see Section 5) to prevent blocking.

7.5. A Simple Approach to Ensure Atomicity

The simplest approach to ensure atomicity is to reuse the SSD commit-at-destination approach (see Section 5.2.1), which is based on checkpointing. At every place, the agent's stage and code is checkpointed using standard checkpointing approaches [Gray and Reuter 1993]. Upon recovery from a failure, the agent's execution is continued from the previous checkpoint. The local transactions are only committed when the agent reaches the agent destination, i.e., it uses the model of closed nested transactions [Moss 1985]. Messages are sent to all previous places $p_1, \ldots, p_{n-1}$ to commit the local transactions. Note that the stage action sa_0 does not have to be committed, as we assume that they are outside the scope of the transactional mobile agent execution. The transactional mobile agent execution is immediately aborted if an unfavorable outcome occurs that renders any further execution obsolete. For instance, if the agent owner only flies with Swiss International Air Lines, but Swiss International Air Lines does not have any seats available for the required destination, the agent execution can be immediately aborted.

8. APPROACHES TO TRANSACTIONAL MOBILE AGENTS

In this section, we present a survey of approaches to transactional mobile agent executions. We classify the approaches into blocking and non-blocking solutions.

8.1. Blocking Solutions

8.1.1. Assis Silva and Krause's Approach. Assis Silva and Krause [1997] provide a model of transactional mobile agents that corresponds essentially to the checkpointing approach discussed in Section 7.5. However, their model assumes open subtransactions.

8.1.2. Sher et al.'s Approach. Sher et al. [2001] present an approach to transactional mobile agents. It is based on the commit-at-destination SSD approach and ensures the ACID properties on the entire mobile agent execution. However, blocking is inherent in any SSD approach and Sher et al. [2001] suffers from this problem. The probability of blocking is relaxed by allowing parallel transactions to run over different parts of the itinerary that are combined again using so-called *mediators*, which govern how the parallel transactions are processed further. For instance, the mediator *ANDjoin* dictates that all parallel transactions have to arrive, whereas with *XORjoin*, only one has to arrive. Figure 20 depicts the example of an *XORjoin* mediator. The transactional mobile agent execution of a splits into two parallel transactions represented by agents b and c. For instance, b_{i-1} tries to book a flight with Swiss, while c_{i-1} books a flight with Delta Airlines. At stage S_i, the mediator *XORjoin* keeps only one of

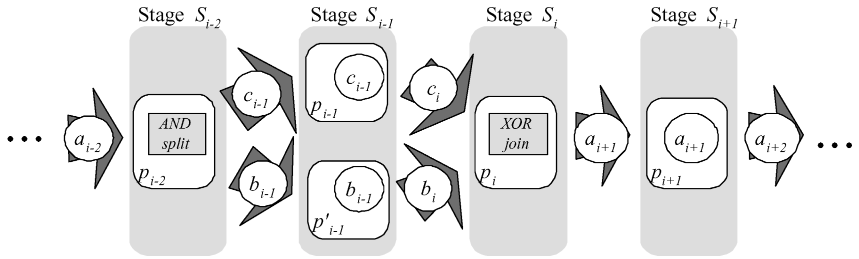

Fig. 20. Mediators (rectangles) allow the execution of parallel transactions.

the subtransactions (represented by c_i and b_i), while the other is aborted. The agent a then continues to reserve a hotel room at stage S_{i+1}. The places that run a join mediator (i.e., p_i) must be visited by the partial mobile agents executing in parallel. This generally limits the itinerary to a (partially) static itinerary. Moreover, failures to non-parallel transactions and mediators result in blocking of the execution. Eliminating nonparallel transactions thus prevents blocking, that is, a split mediator resides at the agent source and a join mediator at the agent destination. The entire mobile agent execution then runs as parallel transactions. However, executing parallel transactions from which only one is committed at the end, even if no failure occurs, causes considerable overhead.

8.1.3. Strasser and Rothermel's Approach. Strasser and Rothermel [2000] address the issue of partial rollbacks to a savepoint in a mobile agent execution. If this savepoint is located at S_0, then their approach ensures atomicity on the entire mobile agent execution. Strasser and Rothermel's approach is based on the protocol in Rothermel and Strasser [1998]. Hence, the transactional mobile agent execution may block if the coordinator of the 2PC fails during the forward execution of the transactional mobile agent (see Section 6.1.11). As open subtransactions and thus compensating transactions are used, only the resources of a stage execution are unavailable; resources of other stages are still available to other transactional mobile agents. The use of compensation transactions limits the applicability but improves the performance by making the accessed resources available again immediately after the stage execution. However, compensating transactions may not be suitable in an environment with frequent aborts of transactional mobile agents, that is, with frequent unfavorable outcomes. The issue of closed subtransactions is not addressed in Strasser and Rothermel [2000].

Blocking may occur in the execution of the compensating transactions. As a consequence, partial rollback involving the rollback of more than the current stage execution may block, thus blocking the entire mobile agent execution.

8.2. Nonblocking Solutions

8.2.1. Extending NetPebbles. The approach in Mohindra et al. [2000] (see Section 6.2) could be extended directly into a transactional mobile agent approach, although this is not done by the authors. This can be done by having a *XORJoin* mediator (see Section 8.1) running at the agent destination. The approach in Mohindra et al. [2000] uses the MSD commit-at-destination solution, and thus is non-blocking unless the agent destination fails. Instead of a priori executing parallel transactions to avoid blocking [Sher et al. 2001], parallel transactions are started only if a failure is detected. Hence, if no failures occur or are detected, no parallel transactions occur.

8.2.2. Assis Silva and Popescu-Zeletin's Approach. The approach presented by Assis Silva and Popescu-Zeletin [2000] and Assis Silva [1999] also builds transaction support on top of fault-tolerant mobile agent execution. For this purpose, Assis Silva and Popescu-Zeletin [2000] reuses the approach in Assis Silva and Popescu-Zeletin [1998]. Assis Silva and Popescu-Zeletin's approach uses the model of open nested transactions and thus relies on compensatable transactions. In Assis Silva [1999], the author mentions that his approach also supports closed subtransactions, although no details are given.

8.2.3. TranSuMa. Pleisch and Schiper [2003, 2003a] suggest an approach, called TranSuMa, to nonblocking transactional mobile agents based on their approach to fault-tolerant mobile agents [Pleisch and Schiper 2000]. This approach can support open nested transactions that consist of both open and closed subtransactions [Pleisch 2002]. In the case of closed subtransactions, instead of immediately committing the operations on the primary of stage S_i the local transaction is kept unterminated. If a subsequent stage action sa_k aborts, all the predecessing local transactions $sa_j (j < k)$ are also aborted. On the other hand, if all stage actions succeed, then the local transactions are only committed when the agent reaches the destination. All other places $p_i^j \neq p_i^{prim}$ immediately abort the local transactions they may have (partially) executed. On the primary, the agent leaves a so-called *stationary agent*, which awaits either an abort or a commit message. On reception of such a message, it either aborts or commits the operations of the local transaction corresponding to the message. The other ACID properties are achieved using the usual mechanisms.

9. MOBILE AGENT EXECUTION UNDER A BYZANTINE FAILURES MODEL

In the previous sections, we have only considered crash failures, that is, machines, places, and agents simply stop executing. With crash failures, the principle issues that need to be resolved are blocking and exactly-once execution. In this section, we generalize the failure model to also accommodate Byzantine failures. More specifically, failing components behave arbitrarily and can, for instance, send arbitrary message to other components. Byzantine failures thus introduce additional issues to the mobile agent execution that need to be resolved. Moreover, malicious places can tamper with the code or internal state of the agent and thus modify the agent behavior. Consequently, the agent may behave differently than originally defined by the agent owner, which may lead to a (potentially financial) disadvantage for the agent owner.

In this survey, we focus on fault tolerance in the mobile agent execution and thus we are concerned with attacks of malicious components on a mobile agent. Such attacks can be initiated by malicious places or other malicious agents. As this survey considers single agent executions (see Section 2.1), we do not address attacks by malicious agents here. Indeed, these attacks can be prevented using mechanisms also applied in traditional distributed systems. Such mechanisms are, for instance, authentication, which aims at making sure a client (or agent) is who he claims to be, and authorization, which grants access to resources to an authenticated client (agent). Authentication and authorization are also important mechanisms to prevent attacks on a place by a malicious agent. Such attacks are addressed by most of the existing mobile agent platforms (e.g., Gray [1996], Karjoth et al. [1997], Peine [1998], Gray et al. [1998], and Herrmann and Zapf [2000]), but are outside the scope of this survey. The interested reader is also referred to the work in Fritzinger and Mueller [1996], Lee and Necula [1997], Necula [1997], and Rubin and Greer [1998] for further examples of how to cope with such attacks. Moreover, we do not consider the problem of denial-of-service attacks. Finally, we assume that the agent is protected while it is in transit between two authenticated places, which can be achieved using standard mechanisms (e.g., Secure Socket Layer [Freier et al. 1996]).

Note that in a traditional client/server system the security issues are well known and adequate solutions exist [Pfleeger and Cooper 1997]. The clients and servers are generally grouped into an administrative domain with a set of registered users. In addition, the client is executed on the user's machine. Since client and server run in the same administrative domain, a user trusts a server or can at least prove that it has received incorrect results. The security in a mobile code environment cannot rely on this trust relationship between the server and an agent because they are generally not part of the same administrative domain.

9.1. The Problem of Protecting the Agent from Malicious Places

The problem of protecting the agent from malicious places is caused by fact that the code of the agent is executed in an untrusted environment [Chess et al. 1995; Farmer et al. 1996a; Minsky et al. 1996]. Without protection of the agent, the execution environment (i.e., the place) can alter or destroy the agent's code and the data accumulated during its itinerary. Moreover, the agent may contain confidential data such as credit card information, which should only be accessible by the place if the agent intends to perform a purchase. For instance, consider an agent that books the cheapest flight to New York. After visiting several airline places it arrives at a malicious place of a airline X. Airline X accesses the prior offers the agent has collected and becomes aware that its price offer is not the lowest. As a consequence, it is tempted to modify or simply delete the collected prices of the other airline's offers such that its price becomes the lowest. It is also already an unfair advantage for a server to be able to read the current state of the mobile agent. This would allow it to learn about the offers of other airlines and thus adapt its pricing policy accordingly. Hence, the agent's code and data needs to be protected.

In order to protect a mobile agent, the following properties are required:

—*Privacy:* A place can only access the data that the agent wants to divulge.

—*Integrity:* The data and the code of the mobile agent are protected from tampering, or, at least, tampering can be detected.

Integrity for the immutable parts of the mobile agent, including its code, (or rather a hash of it) can be easily verified by using digital signatures (e.g., RSA public keys [Rivest et al. 1978]) [Chess et al. 1995]. Indeed, most existing mobile agent systems provide this kind of protection. Privacy for the immutable data of the agent requires a public key infrastructure, whereby the private parts of the agent are encrypted with the public key of target place. However, this limits the agent execution to (partially) static itineraries.

In the following, we first present replication-based approaches. Here, we also revisit the approach of Minsky et al. [1996] and Schneider [1997] (see Section 6.1.1), which we call *privilege-based approach* and which achieves non-blocking mobile agent execution even with Byzantine failures. Then, we show concepts to protect the state of the mobile agent from a malicious host, which generally are orthogonal to the approaches for fault-tolerant mobile agent execution discussed in Section 5.

9.2. Replication-Based Approaches

In this section, we present two approaches that attempt to provide Byzantine fault-tolerant mobile agent execution using replication.

9.2.1. Privilege-Based Approach. We have already shown how the approach in Minsky et al. [1996] and Schneider [1997] achieves fault tolerance in the context of crash failures in Section 6.1.1. However, Minsky et al. and Schneider also consider Byzantine failures, more specifically attacks from malicious hosts on the mobile agent. Using the MMC approach, all places $p_i^j \in M_i$ of a stage S_i execute agent a_i. Although an adversary may corrupt a number of agents at a stage, this approach still allows a place at stage S_{i+1} to safely deduce the true result of the agent execution at stage S_i if enough uncorrupted agents are left.

To protect the mobile agent from malicious hosts the agent is sent to all places of the next stage S_{i+1} by every place of stage S_i. Every place at stage S_{i+1} takes as input the majority of the inputs that it receives from stage i. At this point, the places in M_{i+1} have to know the places in M_i; otherwise, malicious places could simply produce a majority of bogus agents and send them to the places in M_{i+1}. The agents thus have to carry a *privilege*. Corrupted agents can then be identified and deleted. In Schneider [1997], two protocols are proposed to implement such privileges; both require at each stage that a majority of places in M_i are nonfaulty.

—*(Shared secret)*. These protocols ensure that only the source and the destination can learn a secret. Suppose we have a system where each stage has $2k - 1$ places. Each stage i thus divides the secret into $2k-1$ fragments using a $(2k-1, k)$ threshold scheme. Each fragment is then sent to a different place of M_{i+1}. To reconstruct the secret, a node needs at least k fragments of it.

—*(Authentication chains)*. In this scheme all agents carry unforgeable certificates describing their itineraries. Whereas only a place p can construct the certificate, if nonfaulty, any place can check its validity. Every place uses sender authentication to reject corrupted agents and selects any agent for which it received equivalent replicas from a majority of the places of the previous stage.

The privilege-based approaches prevent malicious places from inserting faulty results into an agent's execution. However, additional mechanisms, such as encryption, have to be integrated to ensure agent privacy.

Moreover, these approaches make the rather strong assumption that the replicated places fail independently. While this assumption is adequate if only crash failures occur, it is much stronger with Byzantine failures. Clearly, if all servers are exact replicas, then a successful attack on one of the servers may easily also be successful on the replicas of these server. Hence, an attacker may compromise all replicas of a particular server.

9.2.2. Comparing the Results of Two Agent Replicas. Yee [1999] uses agent replication to address the particular case of a single malicious place in an agent's itinerary. Assume, for instance, that mobile agent a is to find the minimal airfare for a particular flight. Hence, agent a visits the sequence of places $p_0, p_1, \ldots, p_n = p_0$. A replica agent a' is created, which visits the places in the inverse order, that is, $p_n, p_{n-1}, \ldots, p_0$. By comparing the results of these two agent replicas, the agent owner can determine the true minimal airfare, unless the malicious place is the one offering the minimal airfare. In this case, Yee's approach only achieves second best pricing. With this approach, violations to the integrity of the mobile agent can be detected. Clearly, this approach only works for a very limited set of applications and assumes only one malicious place in an agent's itinerary. Moreover, the itinerary must be statically known when the agent is instantiated.

9.3. Tamper-Proof Environments

A solution for protecting both the privacy and the integrity of the agent is based on a *tamper-proof environment* (TPE) [Wilhelm et al. 1999; Yee 1999]. Indeed, the TPE actively prevents tampering with the mobile agent, while most of the other approaches only allow to detect a violation to the agent's integrity. It can be seen as a hardware blackbox, which provides a well-defined, restricted interface to the outside environment as well as an execution environment for agents inside the blackbox. The restricted interface does not allow the place to inspect or tamper with an agent's code and data inside the blackbox from the outside. The agent has to trust a TPE, or rather the TPE's manufacturer, which is usually not the same company as the service provider of the place. During transmission, the agent is encrypted with the public key of the TPE and therefore cannot be accessed by the place itself. Rather, it is forwarded to the TPE, which decrypts it using its private key and finally starts to execute the agent code. The interactions between the agent and the local environment or other agents are also handled by the TPE.

Security is based on the agent owner's trust towards the TPE manufacturer. The latter is believed not to provide malicious TPEs because of its business interests. Clearly, the TPE manufacturer would be out of business instantly if somebody could prove that it delivered a malicious TPE or that its TPE is not secure from tampering (e.g., by malicious agents executed by it).

As the use of TPEs incurs considerable organizational and financial overhead we believe that its application will be limited to particularly security-sensitive domains (e.g., banks, stock market). To reduce the size of the TPE only the crucial parts of the agent, such as the certificates or the keys, are kept in the TPE, instead of accommodating the entire agent. Smartcards are an example of such a resource-constrained but low-cost TPE [Fünfrocken 1999]. Such TPE's are also used in Karjoth [2000] in the context of distributed marketplaces.

To avoid the cost of TPE's, the approach in Algesheimer et al. [2001] replaces the TPE by a generic secure computation service that is accessed by all places. The secure computation service can execute small parts of an agent application while maintaining privacy. For larger applications, the associated costs become prohibitive.

9.4. Achieving Forward Integrity of Partial Results

Several approaches address the issue of protecting the data that the agent has accumulated prior to executing on place p_i, that is, the results of its execution on places $p_0, \ldots, p_{i-1}$. This property is called *forward integrity*. To prevent malicious hosts from modifying earlier results, a so-called *chain* [Karjoth et al. 1998] needs to be established. Using this chain, the tampering of prior results can be detected. However, these approaches generally have the limitation that cooperating malicious agents or a malicious agent that is visited at least twice by the mobile agent can truncate the chain of results. Indeed, assume that agent a visits the places $p_0, p_1, \ldots, p_k, p_i, \ldots, p_j, p_k, \ldots$ and that p_k is malicious. Place p_k could remove all the results that have been contributed by the places p_i to p_j.

9.4.1. Yee's Approach. Yee [1999] was the first to devise protocols for achieving forward integrity in mobile agent executions. His protocol is based on hashing and digital signatures. However, the protocol cannot prevent a malicious place from modifying its prior offer, or the offer of a colluding malicious place. Moreover, Yee's protocol needs to have a priori knowledge about the length of the itinerary in order to detect removed results.

9.4.2. KAG Family of Protocols. Karjoth et al. [1998] improve the approaches in Yee [1999] in such a way that disallows a place to modify its own earlier offer later again. They suggest several algorithms for protecting the results of free-roaming agents. These algorithms allow to detect the tampering of the agent's results. However, two cooperating malicious hosts can still remove the results of the places between them from the agent without being detected.

Whereas one class of algorithms is based on a *per-server digital signature* scheme, another uses *hash chains*. In the per-server digital signature scheme the result of a stage's computation is encrypted using the public key of the agent authority. Then, place p_i signs this encrypted result and forwards it to p_{i+1}. In order to detect the removal of results, some random number is encrypted along with the result of the computation. In addition, a hash value over the result at p_{i-1} and the ID of p_{i+1} chains the different stages together. Changing the order of encryption and signing allows the identity of the previous places to be hidden. This scheme relies on the availability of a public-key infrastructure.

In contrast to the digital signature schemes, the hash chain algorithms in Karjoth et al. [1998] do not rely on a public key infrastructure. They are based on forwarding a value in addition to the encrypted result to the next place. This value can be a hash value containing the computation's result, a random number, the previous hash value to build the chain, and the ID of p_{i+1}.

9.4.3. Approach in the Ajanta Mobile Agent System. Ajanta is a Java-based mobile agent system [Karnik and Tripathi 1999]. It provides abstractions such as read-only agent state and append-only logs. The read-only agent state corresponds to the immutable part of the agent state and can be protected using digital signatures (see Section 9.1).

The append-only log allows a place to add new objects and ensures that any subsequent modifications to these objects can be detected by the agent owner. The implementation of append-only logs uses an encrypted checksum. The initial value of this checksum is based on some random number, generated and kept secret on the agent source. On every place p_i on which a new object is appended, the new checksum consists of the concatenation of the following elements, encrypted with the public key of the agent source:

(1) the previous checksum,

(2) the signature of the new object, and

(3) the ID of the signer of the new object.

9.4.4. Multi-Hops Protocol. Corradi et al. [1999] propose a protocol similar to the append-only log. As the chaining relation, they use a message integrity code, which contains the previous message integrity code, the data modified at the current place, an additional nonce, and the identity of the next place.

9.4.5. Roth's Attacks. Roth [2001a] has devised an attack that breaks the protocols in Karjoth et al. [1998], Karnik and Tripathi [1999], and Corradi et al. [1999]. The basic idea is for a malicious node to abuse a legitimate node as an oracle that decrypts, signs, or computes protocol data on behalf of the malicious host. To improve the robustness of these protocols, Roth introduces the notion of *mobile agent kernel* that identifies a particular instance of a mobile agent [Roth 2001b]. Using the mobile agent kernel, the abuse of nonmalicious places as oracles can be prevented.

9.5. Encrypted Functions

Sander and Tschudin [1998] suggest computing with *encrypted functions* to achieve software-based agent privacy and integrity. Assume that an agent knows how to compute f and requires $f(x)$ from a service located at place p_i, but wants to keep f a secret. The agent owner therefore transforms (encrypts) f to some other function $E(f)$ that hides f and may also produce encrypted output data. $P(E(f))$ describes the program that implements $E(f)$ and is sent to p_i. Therefore, p_i only learns about $P(E(f))$, which it applies to its input data x. The result $P(E(f))(x)$ is sent back to the agent, which decrypts it and obtains $f(x)$.

While in Sander and Tschudin [1998] only polynomial and rational functions are supported, Sander et al. [1999] and Cachin et al. [2000] extend this support to noninteractive evaluation of functions that can be represented by circuits of logarithmic depth and arbitrary functions, that can be mapped to polynomial-size circuits, respectively.

9.6. Cryptographic Traces

Vigna [1998a] proposes cryptographic traces to detect illegal modifications of an agent's code, state, and execution. During the execution of the agent, data, called *traces*, is collected that allows the agent owner to verify the execution of the agent at the agent destination.

While Vigna's approach can detect certain misbehavings of places, it cannot detect all of them. For instance, it cannot detect whether additional side-effects have occurred at a place, as it only takes the state of the agent into account. Assume, for instance, that the agent moves money from account A to account B. The state of the agent may not reflect this transfer. Also, a malicious place may fabricate a trace that fits the state of the agent resulting from a completely different execution.

Moreover, the approach relies on a trusted third party and does not offer protection against disclosure of data.

9.7. State Appraisal

Farmer et al. [1996a] propose a mechanism called *state appraisal* to verify the integrity of the mobile agent's state. State appraisal functions are defined at the agent source and allow to verify invariants in the agent's state. Clearly, the applicability of state appraisal is limited, as the invalid state needs to be foreseen at the agent source. In general, it is not possible to indicate the precise state the agent will eventually have, in particular with dynamic itineraries.

9.8. Code and State Obfuscation

Hohl [1997] proposes an approach that is based on code and state obfuscation. The mobile agent's code and state is obfuscated in a way that makes it difficult and time consuming for a place to find out about private data of the mobile agent, while still allowing the agent to execute properly. To limit the time a place has to try to find out about the agent's private data, each agent is given a time-to-live. Once its time-to-live is expired, the agent is not accepted any more by a legitimate place.

Unfortunately, it is not always easy to predict an appropriate time-to-live that both gives the agent time to execute and limits the time a malicious place has to break the agents obfuscation mechanism. On one hand, a small time-to-live may prevent the agent from finishing its execution, even with the absence of malicious places. On the other hand, obfuscation can still be broken if sufficient time is available. Also, if a malicious place wants to read the state of the agent, it can still do it, as it can keep the agent as long as it wants, thereby ignoring the time-to-live parameter. As a consequence, privacy cannot be ensured with this approach. However, a place can detect tampering when the tampering has lead to the expiration of the time to live.

9.9. Agent Security Based on the Itinerary or the Use of Child Agents

A certain degree of security can be achieved by carefully choosing the itinerary of a mobile agent. Indeed, if the agent periodically returns to a trusted place, the damage done by malicious places can be limited. Also, child agents can be used to visit untrusted places. In Gypsy [Jazayeri and Lugmayr 2000], a company runs a number of secure places in the network. The original agent, called *supervisor agent*, only visits these trusted places. If it needs to visit an untrusted place, it sends a child agent, called *worker*, which executes on the untrusted place and sends its results back to the supervisor agent. Using this approach limits the damage that can be done by a malicious place.

10. CONCLUSION

In this article, we have presented a comprehensive survey on fault-tolerant and transactional mobile agent execution. The survey has discussed these properties with respect to crash failures of machines, places, and agents and Byzantine failures of places. To our knowledge, our work is the first to present a comprehensive survey on fault-tolerant and transactional mobile agent execution. Although other surveys exists, they do not consider the latest work or only address security issues. Note also that several of the concepts presented in the survey have similarities with techniques used in traditional client/server computing (without mobile agents). However, an in-depth discussion of similarities and differences cannot be provided within the scope of the survey.

In the context of non-transactional mobile agents, we have presented a novel classification of fault-tolerant mobile agent approaches. We have first distinguished between commit-after-stage approaches, where the stage actions are committed immediately after the stage execution, and commit-at-destination approaches, where the stage actions are only committed at the agent destination. Within these approaches, we have further classified solutions according to the following characteristics: (1) whether the agent is executed by a single or multiple places, (2) at which point in time the modifications of the mobile agent are committed, and (3) whether commit decision and stage execution are collocated. This leads to various solutions, which have been discussed in terms of their advantages and usability. Moreover, our classification allows the discussion of strengths and limitations of these solutions solely based on the characteristics (1) to (3). Approaches that replicate the mobile agent (i.e., approaches MSC, MMC, MSD, MMD) generally have higher resource costs and a greater message overhead. Moreover, approaches based on iso-places require the service provider to provide replicated places. However, they have the clear advantage to be non-blocking. In contrast, SSC approaches are less complex but may block upon the failure of a place.

Besides infrastructure failures, transactional mobile agents also address unfavorable outcomes of the agent execution at a place. We have presented a survey of transactional mobile agents. A general solution to transactional mobile agents should support open and closed nested transactions. Generally, blocking is less of a problem in the case of open nested transaction. Indeed, it only increases potential dependencies with other mobile agents. In contrast, blocking in the case of closed nested transactions seriously limits overall systems throughput and affects also places that may not be directly involved in the blocking.

The article also surveys approaches that protect the mobile agent from malicious hosts. While some of the approaches build on concepts that are also applied to crash failures (e.g., replication), most of the approaches use concepts that are orthogonal to the ones used for crash failures. Protecting mobile agents from malicious places is a difficult problem [Farmer et al. 1996b]. The only comprehensive security is provided by TPEs. TPEs ensure both privacy and integrity. However, TPEs require additional hardware and also impose an organizational overhead. The other approaches provide partial mechanisms that address particular security aspects. Encrypted functions seem to be another possible approach, but they are too limited to be of general use. All other approaches make limiting assumptions or only address agent integrity. However, these approaches may be adequate for applications and in environments that comply with the assumptions underlying the approaches. This shows that protecting the agent from attacks is an issue that still is not adequately solved and is still subject to ongoing research. It is not clear at this point, whether software-based comprehensive security can be provided at all [Chess et al. 1995; Farmer et al. 1996b]. As a consequence, nearly all existing mobile agent systems only address the protection of places and mobile agents against malicious mobile agents, and ensure integrity of the immutable parts of a mobile agent.

The survey shows that the research on fault tolerance and security in the context of mobile agents has made substantial progress in recent years. Many different approaches have been devised, with different strengths and weaknesses. These approaches vary in terms of message overhead and resource consumption. As applications have different requirements to fault tolerance, transactional execution, and security, the approaches best suited for a given application need to be carefully chosen. Some applications can trade blocking against reduced message overhead and resource consumption, and thus achieve better performance. Others have severe requirements with respect to non-blocking and are ready to accept a performance overhead. It is the application developer's responsibility to choose the appropriate approach. This survey can help him with this task.

Despite recent advances, much work still needs to be done. Both fault-tolerant non-transactional and transactional mobile agent systems have to prove their value in real applications. Not all of the presented approaches have been implemented yet and have been quantitatively evaluated in a practical setting. Quantitative evaluations, however, are needed in order to further understand the advantages and limitations of the various approaches and to complement/confirm our results. Moreover, as some approaches are very complex and their correct application requires extensive theoretical knowledge and experience, they are not easily used by application developers. Hence, good and self-explaining user interfaces and well-defined frameworks need to be provided to facilitate the utilization of these approaches and to limit the potential of mistakes.

Although recently mobile agent technology has also found its way outside the academic community, its applications are still not very wide-spread. With more research into fault-tolerant, transactional, and secure mobile agent execution, the development of mobile agent-based applications can be furthered.

ACKNOWLEDGMENTS

The authors thank Ajay Mohindra for his explanations about NetPebbles. Flávio Assis Silva and Markus Strasser have been very helpful in computing the message complexity of their approaches. Robert Gray has answered our questions regarding AgentTcl and D'agents. The authors are also grateful to Metin Feridun and Liba Svobodova for their insightful comments on an earlier version of this document, and to the anonymous reviewers for their comments that helped improve the text. We especially thank Fred Schneider for his suggestions regarding this survey.

REFERENCES

ALGESHEIMER, J., CACHIN, C., CAMENISCH, J., AND KARJOTH, G. 2001. Cryptographic security for mobile code. In *Proc. of Symposium on Security and Privacy*. Oakland, CA.

ALPERN, B. AND SCHNEIDER, F. 1985. Defining liveness. *Information Processing Letters 21*.

ASSIS SILVA, F. 1999. A transaction model based on mobile agents. Ph.D. dissertation, Informatik, Technische Universität Berlin.

ASSIS SILVA, F. AND KRAUSE, S. 1997. A distributed transaction model based on mobile agents. In *Proc. of the 1st Int. Workshop on Mobile Agents (MA'97)*, K. Rothermel and R. Popescu-Zeletin, Eds. LNCS 1219. Springer-Verlag, 198–209.

ASSIS SILVA, F. AND POPESCU-ZELETIN, R. 1998. An approach for providing mobile agent fault tolerance. In *Proc. of the 2nd Int. Workshop on Mobile Agents (MA'98)*, K. Rothermel and F. Hohl, Eds. LNCS 1477. Springer-Verlag, New York, 14–25.

ASSIS SILVA, F. AND POPESCU-ZELETIN, R. 2000. Mobile agent-based transactions in open environments. *IEICE Trans. Commun. E83-B*, 5 (May).

BERNSTEIN, P., HADZILACOS, V., AND GOODMAN, N. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Mass.

BIESZCZAD, A., PAGUREK, B., AND WHITE, T. 1998. Mobile agents for network management. *IEEE Commun. Surv*.

CACHIN, C., CAMENISCH, J., KILLIAN, J., AND MÜLLER, J. 2000. One-round secure computation and secure autonomous mobile agents. In *Proc. of the 27th Int. Colloquium on Automata, Languages and Programming (ICALP)*, U. Montanari, J. Rolim, and E. Welzl, Eds. LNCS 1853. Springer-Verlag, New York, 512 – 523.

CHANDRA, T. AND TOUEG, S. 1996. Unreliable failure detectors for reliable distributed systems. *J ACM 43*, 2 (Mar.), 225–267.

CHESS, D., GROSOF, B., HARRISON, C., LEVINE, D., PARRIS, C., AND TSUDIK, G. 1995. Itinerant agents for mobile computing. *IEEE Pers. Commun. Syst. 2*, 5 (Oct.), 34–49.

CHESS, D., HARRISON, C., AND KERSHENBAUM, A. 1998. Mobile agents: Are they a good idea? In *Mobile Agents and Security*, G. Vigna, Ed. LCNS 1419. Springer-Verlag, New York, 25–47.

CHRYSANTHIS, P. AND RAMAMRITHAM, K. 1994. Synthesis of extended transaction models using ACTA. *ACM Trans. Datab. Syst. (TODS) 19*, 3, 450–491.

CORRADI, A., MONTANARI, R., AND STEFANELLI, C. 1999. Mobile agents protection in the Internet environment. In *Proc. of the 23rd Int. Computer Software and Applications Conference (COMPSAC'99)*. 80–85.

DASGUPTA, P. 2000. Fault tolerance in MAgNET: A mobile agent e-commerce system. In *Proc. of the 6th Int. Conference on Internet Computing* (Las Vegas, NV). 733–739.

DÉFAGO, X., SCHIPER, A., AND SERGENT, N. 1998. Semi-passive replication. In *Proc. of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98)* (West Lafayette, Ind.). 43–50.

ELNOZAHY, E. N. AND ZWAENEPOEL, W. 1992. Manetho: transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *IEEE Trans. Comput. 41*, 5 (May), 526–531.

FARMER, W., GUTTMAN, J., AND SWARUP, V. 1996a. Security for mobile agents: Authentication and state appraisal. In *Proc. of the 4th European Symposium on Research in Computer Security* (Rome, Italy). 118–130.

FARMER, W., GUTTMAN, J., AND SWARUP, V. 1996b. Security for mobile agents: issues and requirements. In *Proc. of the 19th National Information Systems Security Conference*, S. Wakid and J. Davis, Eds. Vol. 2. (Baltimore, MD). 591–597.

FREIER, A., KARLTON, P., AND KOCHER, P. 1996. The SSL protocol—version 3.0 (internet draft). Tech. Rep.

FRITZINGER, J. AND MUELLER, M. 1996. *Java Security*. Sun Microsystems.

FÜNFROCKEN, S. 1999. Protecting mobile web-commerce agents with smartcards. In *Proc. of 1st Int. Conference on Agent Systems and Applications/Mobile Agents (ASAMA'99)* (Palm Springs, CA). 90–102.

GARCIA-MOLINA, H. AND SALEM, K. 1987. Sagas. In *Proc. of ACM SIGMOD Int. Conference on Management of Data and Symposium on Principles of Database Systems* (San Francisco, CA). 249–259.

GRAY, J. 1981. The transaction concept: virtues and limitations. In *Proc. of Int. Conference on Very Large Databases* (Cannes, France). 144–154.

GRAY, J. AND REUTER, A. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA.

GRAY, R. 1996. Agent TCL: A flexible and secure mobile-agent system. In *Forth Annual Usenix Tcl/Tk Workshop*.

GRAY, R., KOTZ, D., CYBENKO, G., AND RUS, D. 1998. D'agents: Security in a multiple-language, mobile agent system. See Vigna [1998b].

HADZILACOS, V. AND TOUEG, S. 1994. A modular approach to fault-tolerant broadcasts and related problems. Tech. Rep. TR94-1425, CS, University of Toronto; CS, Cornell University. May.

HÄRDER, T. AND REUTER, A. 1983. Principles of transaction oriented database recovery—a taxonomy. *ACM Comput. Surv. 15*, 4 (Dec.), 289–317.

HERRMANN, K. AND ZAPF, M. 2000. AMETAS: on security. http://www.ametas.de.

HOHL, F. 1997. An approach to solve the problem of malicious hosts in mobile agent systems. Tech. Rep., University of Stuttgart, Germany.

JAZAYERI, M. AND LUGMAYR, W. 2000. Gypsy: A component-based mobile agent system. In *Proc. of the 8th Euromicro Workshop on Parallel and Distributed Processing (PDP2000)* (Rhodos, Greece).

JOHANSEN, D., MARZULLO, K., SCHNEIDER, F. B., JACOBSEN, K., AND ZAGORODNOV, D. 1999. NAP: Practical fault-tolerance for itinerant computations. In *Proc. of 19th IEEE Int. Conference on Distributed Computing Systems (ICDCS'99)*, M. G. Gouda, Ed. (Austin, Tex.). 180–189.

KARJOTH, G. 2000. Secure mobile agent-based merchant brokering in distributed marketplaces. In *Proc. of 2nd Int. Symposium on Agent Systems and Applications and 4th INt. Symposium on Mobile Agents (ASAMA'00)* (Zurich, Switzerland). LNCS 1882. Springer-Verlag, New York, 44–56.

KARJOTH, G., ASOKAN, N., AND GÜLCÜ, C. 1998. Protecting the computation results of free-roaming agents. In *Proc. of the 2nd Int. Workshop on Mobile Agents (MA'98)* (Stuttgart, Germany), K. Rothermel and F. Hohl, Eds. LNCS 1477. Springer-Verlag, New York, 195–207.

KARJOTH, G., LANGE, D., AND OSHIMA, M. 1997. A security model for Aglets. *IEEE Internet Comput.*, 68–77.

KARNIK, N. AND TRIPATHI, A. 1999. Security in the Ajanta mobile agent system. Tech. Rep. TR-5-99, University of Minnesota, Minneapolis, Minn. May.

KORTH, H., LEVY, E., AND SILBERSCHATZ, A. 1990. A formal approach to recovery by compensating transactions. In *Proc. of 16th Int. Conference on Very Large Data Bases* (Brisbane, Queensland, Australia). D. McLeod, R. Sacks-Davis, and H.-J. Schek, Eds. Morgan-Kaufmann, San Mateo, Calif., 95–106.

LANGE, D. AND OSHIMA, M. 1998. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, Reading, Mass.

LANGE, D. AND OSHIMA, M. 1999. Seven good reasons for mobile agents. *Commun. ACM 42*, 3 (Mar.), 88–89.

LEE, P. AND NECULA, G. 1997. Research on proof-carrying code for mobile-code security. In *DARPA Workshop on Foundations for Secure Mobile Code*.

LYU, M. AND WONG, T. 2003. A progressive fault tolerant mechanism in mobile agent systems. In *Proc. of the 7th World Multiconference on Systemics, Cybernetics and Informatics (SCI'03)*. Vol. IX. Orlando, Florida, 299–306.

MAES, P., GUTTMAN, R., AND MOUKAS, A. 1999. Agents that buy and sell. *Communications of the ACM 42*, 3 (Mar.), 81–91.

MATTERN, F. 1998. Mobile agenten. *it+ti—Informationstechnik und Technische Informatik* 4, 12–17. In German.

MINSKY, Y., VAN RENESSE, R., SCHNEIDER, F., AND STOLLER, S. 1996. Cryptographic support for fault-tolerant distributed computing. In *Proc. of 7th European Workshop on ACM SIGOPS* (Connemara, Ireland). ACM, New York, 109–114.

MISHRA, S. 2001. Agent fault tolerance using group communication. In *Proc. of Int. Conference on Parallel and Distributed Processing Techniques and Application (PDPTA)* (Las Vegas, NV).

MISHRA, S. AND HUANG, Y. 2000. Fault tolerance in agent-based computing systems. In *Proc. of 13th*

ISCA Int. Conference on Parallel and Distributed Computing Systems (Las Vegas, NV).

Mohindra, A., Purakayastha, A., and Thati, P. 2000. Exploiting non-determinism for reliability of mobile agent systems. In *Proc. of Int. Conference on Dependable Systems and Networks (DSN'00)* (New York). 144–153.

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

Murphy, A. and Picco, G. 1999. Reliable communication for highly mobile agents. In *Proc. of 1st Int. Conference on Agent Systems and Applications / Mobile Agents (ASAMA'99)* (Palm Springs, CA). IEEE Computer Society Press, Los Alamitos, Calif., 141–150.

Necula, G. C. 1997. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France). ACM, New York, 106–119.

Object Management Group (OMG) 2000. *Mobile Agent System Interoperability Facilities Specification, OMG Document formal / 00-02-01*. Object Management Group (OMG). http://www.omg.org.

Pals, H., Petri, S., and Grewe, C. 2000. FANTOMAS—Fault tolerance for mobile agents in clusters. In *Proc. of IPDPS 2000 Workshop*, J. Rolim, Ed. LNCS 1800. Springer-Verlag, New York, 1236–1247.

Pears, S., Xu, J., and Boldyreff, C. 2003. Mobile agent fault tolerance for information retrieval applications: An exception handling approach. In *Proc. of the 6th Int. Symposium on Autonomous Decentralized Systems (ISADS'03)* (Pisa, Italy). IEEE Computer Society Press, Los Alamitos, Calif.

Peine, H. 1998. Security concepts and implementation in the ARA mobile agent system. In *Proc. of the 7th IEEE Int. Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE '98)*. 236–242.

Pfleeger, C. and Cooper, D. 1997. Security and privacy: Promising advances. *IEEE Softw*.

Pleisch, S. 2002. Fault-tolerant and transactional mobile agent execution. Ph.D. dissertation. École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland. Number 2654.

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

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

Pleisch, S. and Schiper, A. 2001b. TranSuMA: Non-blocking transaction support for mobile agent execution. Tech. Rep. rz 3386, IBM Research.

Pleisch, S. and Schiper, A. 2002. Non-blocking transactional mobile agent execution. In *Proc. of 22nd IEEE Int. Conference on Distributed Computing Systems (ICDCS'02)* (Vienna, Austria). 443–444.

Pleisch, S. and Schiper, A. 2003a. Execution atomicity for non-blocking transactional mobile agents. In *Proc. of Int. Conference on Parallel and Distributed Computing and Systems (PDCS'03)* (Marina del Rey, CA).

Pleisch, S. and Schiper, A. 2003b. Fault-tolerant mobile agent execution. *IEEE Trans. Comput. 52*, 2 (Feb.), 209–222.

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

Rivest, R., Shamir, A., and Adleman, L. 1978. A method for obtaining digital signatures and public-key crypto-systems. *Commun. ACM 21*, 2.

Roth, V. 2001a. On the robustness of some cryptographic protocols for mobile agent protection. In *Proc. of 5th Conference on Mobile Agents (MA'01)* (Atlanta, GA). LNCS 2240. Springer-Verlag, New York, 1– 14.

Roth, V. 2001b. Programming satan's agents. In *Proc. of the 1st Int. Workshop on Secure Mobile Multi-Agent Systems* (Montreal, Ont., Canada).

Rothermel, K. and Strasser, M. 1998. 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'98)* (West Lafayette, Indi.). 100–108.

Rubin, A. and Greer, D. E. 1998. Mobile code security. *IEEE Internet Comput.*

Sabel, L. and Marzullo, K. 1994. Simulating fail-stop in asynchronous distributed systems. In *Proc. of the 13th IEEE Symposium on Reliable Distributed Systems (SRDS'94)*. Dana Point, CA, 138–147.

Sander, T. and Tschudin, C. 1998. Protecting mobile agents against malicious hosts. In *Mobile Agents and Security*. LNCS 1419. Springer-Verlag, New York, 44–60.

Sander, T., Young, A., and Yung, M. 1999. Non-interactive cryptocomputing for NC. In *Proc. of the 40th Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society Press, Los Alamitos, Calif.

Schneider, F. 1997. Towards fault-tolerant and secure agentry. In *Proc. of the 11th Int. Workshop on Distributed Algorithms, Saarbrücken, Germany*. Invited paper.

Shamir, A. 1979. How to share a secret. *Commun. ACM 22*, 11 (Nov.), 612–613.

Sher, R., Aridor, Y., and Etzion, O. 2001. Mobile transactional agents. In *Proc. of 21st IEEE Int.*

Conference on Distributed Computing Systems (ICDCS'01) (Phoenix, AZ). 73–80.

SILVA, L., BATISTA, V., AND SILVA, J. 2000. Fault-tolerant execution of mobile agents. In *Proc. of Int. Conference on Dependable Systems and Networks (DSN'00)* (New York). 135–143.

STRASSER, M. AND ROTHERMEL, K. 1998. Reliability concepts for mobile agents. *Int. J. Coop. Inf. Syst. 7*, 4, 355–382.

STRASSER, M. AND ROTHERMEL, K. 2000. System mechanisms for partial rollback of mobile agent execution. In *Proc. of 20th IEEE Int. Conference on Distributed Computing Systems (ICDCS'00)* (Taipei, Taiwan). 20–28.

TAKASHIO, K., SOEDA, G., AND TOKUDA, H. 2001. A mobile agent framework for follow-me applications in ubiquitous computing environment. In *Proc of Int. Workshop on Smart Appliances and Wearable Computing (IWSAWC'01)*. 202–207.

THEILMANN, W. AND ROTHERMEL, K. 2000. Optimizing the dissemination of mobile agents for distributed information filtering. *IEEE Concurr.* 53–61.

VIGNA, G. 1998a. *Cryptographic Traces for Mobile Agents*, 137–153. In *LNCS 1419* Vigna [1998b].

VIGNA, G. 1998b. *Mobile Agents and Security*. LNCS 1419. Springer-Verlag, Berlin, Germany.

VOGLER, H., KUNKELMANN, T., AND MOSCHGATH, M.-L. 1997a. An approach for mobile agent security and fault tolerance using distributed transactions. In *Proc. of Int. Conference on Parallel and Distributed Systems (ICPADS'97)* (Seoul, Korea). IEEE Computer Society Press, Los Alamitos, Calif.

VOGLER, H., KUNKELMANN, T., AND MOSCHGATH, M.-L. 1997b. Distributed transaction processing as a reliability concept for mobile agents. In *Proc. 6th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'97)* (Tunis, Tunisia). IEEE Computer Society Press, Los Alamitos, Calif.

WALSH, T., PACIOREK, N., AND WONG, D. 1998. Security and reliability in concordia. In *Proc. of the 31st Hawaii Int. Conference on System Sciences (HICSS'98)*. Vol. 7. 44–53.

WEIKUM, G. AND SCHEK, H.-J. 1992. Concepts and applications of multilevel transactions and open nested transactions. In *Database Transaction Models for Advanced Applications*, A. Elmagarmid, Ed. Morgan-Kaufmann, San Mateo, CA, 515–553.

WEIKUM, G. AND VOSSENS, G. 2002. *Transaction Information Systems: Theory, Algorithms, and the Practice*. Morgan-Kaufmann, San Mateo, CA.

WHITE, J. 1995. Telescript technology: An introduction to the language. Tech. Rep., General Magic Inc.

WHITE, J. 1996. Mobile agents white paper. Tech. Rep., General Magic Inc.

WILHELM, U., STAAMANN, S., AND BUTTYÀN, L. 1999. Introducing trusted third parties to the mobile agent paradigm. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*. LNCS 1603. Springer-Verlag, New York, 471– 491.

YEE, B. 1999. A sanctuary for mobile agents. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*. LNCS 1603. Springer Verlag, New York, 261–273.

Received September 2001; revised November 2002, May 2004; accepted August 2004