Modeling Fault-Tolerant Mobile Agent Execution as a Sequence of Agreement Problems

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

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

Abstract

Fault-tolerance is fundamental to the further development of mobile agent applications. In the context of mobile agents, fault-tolerance prevents a partial or complete loss of the agent, i.e., ensures that the agent arrives at its destination. Simple approaches such as checkpointing are prone to blocking. Replication can in principle improve solutions based on checkpointing. However, existing solutions in this context either assume a perfect failure detection mechanism (which is not realistic in an environment such as the Internet), or rely on complex solutions based on leader election and distributed transactions, where only a subset of solutions prevents blocking.

This paper proposes a novel approach to fault-tolerant mobile agent execution, which is based on modeling agent execution as a sequence of agreement problems. Each agreement problem is one instance of the well-understood consensus problem. Our solution does not require a perfect failure detection mechanism, while preventing blocking and ensuring that the agent is executed exactly once.

1 Introduction

Mobile agents[1] are computer programs that act autonomously on behalf of a user and travel through a network of heterogeneous machines [9].

Failures in such an environment may lead to a partial or complete loss of the agent. For instance, a failure of the device on which the agent is currently executing causes all information about the agent not held in stable storage to be lost. In the worst case, the information about the entire agent is discarded. As no boundaries on relative processor speed and communication delays exist in asynchronous systems such as the Internet, the owner of the agent (i.e., the person who created the mobile agent) is left with the problem of uncertainty. Indeed, the agent owner cannot determine whether the agent is lost or whether its execution has only been delayed due to slow processors or communication links. This uncertainty may lead to the following situations:

- The agent owner believes that the agent has been lost, when in fact it has not been. Launching another agent may cause multiple executions of the agent code on some devices.
- The agent owner waits for the agent to finish its execution, but the agent has failed. Clearly, this is a blocking situation.

Fault-tolerant mobile agent execution removes this uncertainty and ensures that the agent eventually reaches its destination or at least notifies the agent owner of a potential problem. We first show that a simple checkpointing-based execution of an agent, even though it ensures that the agent is not lost, is prone to blocking (the agent execution might not terminate until the crashed machine recovers). Whereas adding a simple time-out-based failure detection mechanism solves the blocking problem, it also leads to the violation of the exactly-once execution property, which is fundamental to many applications (the exactly-once execution property is violated when the code of the agent is executed more than once).

Replication allows us to solve both the blocking problem and the exactly-once execution problem. The idea of replication in the context of mobile agents is not new [4, 7, 10]. However, on the one hand, [7] assumes a perfect failure detection mechanism, which is a very constraining assumption in the context of a wide area network such as the Internet. On the other hand, [4, 10] model their solution as a sequence of two problems: leader election and distributed transactions. The interference of those two problems leads to a complex and difficult-to-understand solution. We pro-

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

pose a much simpler solution, which is specified in terms of a *single* problem, the consensus problem. More specifically, our solution models a mobile agent execution as a sequence of agreement problems, where each agreement problem decides, at each stage (an agent executes in a sequence of stages) of the agent execution, on:

- the place (i.e., the logical agent execution environment running on a machine) that has executed the agent,
- the resulting agent, and
- the set of places for the next stage of the execution.

Such a model is extremely easy to understand, and at the same time addresses the issues of blocking and of exactly-once execution of agents.

The rest of the paper is structured as follows. Section 2 presents an overview of the problem. In Section 3, we specify the problem. Section 4 discusses how consensus can be used to solve the agreement problems. In Section 5, we summarize other work published in this area and compare it with our work. Finally, Section 6 concludes the paper.

2 Overview

2.1 Agent Execution

A mobile agent executes on a sequence of machines, as shown in Figure 1. A place p_i thereby provides a logical execution environment for the agent. Each machine may potentially host multiple places. Executing the agent at a place is called a *stage* S_i of the agent execution. Two stages S_i and S_{i+1} are separated by a *move* or a *spawn* operation of the agent. A move operation describes the transfer of the agent from one place to another, whereas with a spawn operation, an agent launches another (sub)agent. In an application that is comprised of mobile agents we call its principal *agent owner*, and the places where its first and last stages execute we call the agent *source* and *destination*. The agent source and the destination may be identical.

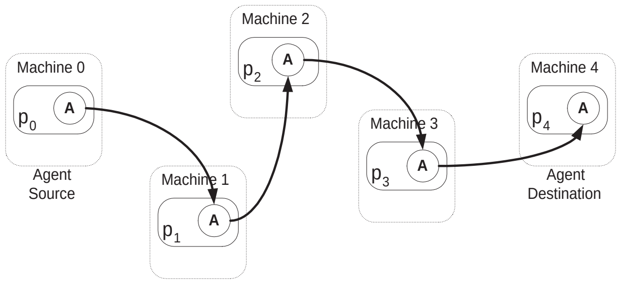

Figure 1. Example of the execution of an agent A.

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

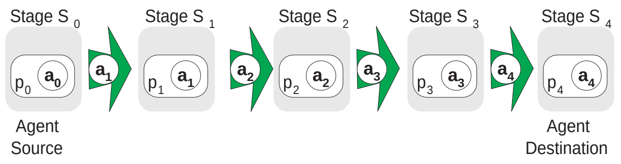

Figure 2. Logical agent execution.

The agent execution must ensure that the agent is executed exactly once. The *exactly-once property* is crucial to mobile agent execution.

2.2 The Problem of Failures

Any hardware and software component in a computing environment is potentially subject to failures. This paper addresses the following failures: crash of an agent, a place, or a machine. Clearly, the crash of a machine causes any place and any agent running on this machine to crash as well (Figure 3.d). A crashing place causes the crash of any agent on this place, but this generally does not affect the machine (Figure 3.c). Similarly, a place and the machine survive the crash of an agent (Figure 3.b). We do not consider programming errors in the code of the agent or the place as relevant failures in this sense.

2.2.1 Atomicity and Durability

As mentioned in Section 2.1, executing an agent action on a place generally results in a modification of the agent state as well as the state of the place. Because of the particular failure dependency between agent and place, a crash of the agent may leave the machine and the place in an incorrect state. Assume, for instance, that the agent transfers money from a banking service provided by the place to itself. If the agent fails at this point, the state of the place reflects the agent's operation (money retrieval), whereas the agent state (and the money itself) is potentially lost. Clearly, this leads to an incorrect place state.

Consequently, the agent actions on the place have to be undoable and adequate protocols must be devised that ensure the consistent state of machine and place even when

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

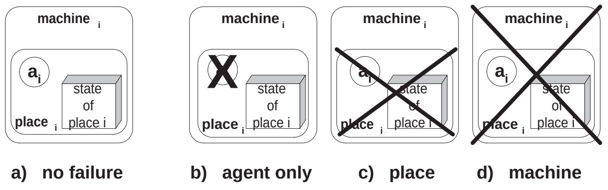

Figure 3. Failure model for mobile agents.

the agent fails. Generally, two approaches are possible: *pessimistic* and *optimistic* execution.

Pessimistic execution The pessimistic execution approach assumes that failures and incorrect failure suspicions occur with a non-negligeable frequency and thus need to be dealt with in an efficient way. Agent operations are thus executed tentatively and changes to the place and the machine made permanent (i.e., committed) only when it is ensured that the agent does not fail.

Optimistic execution On the other hand, in optimistic execution, failures and incorrect failure detection are assumed to be quite rare and modifications are immediately made permanent. Failure recovery (i.e., undoing agent actions) thus becomes a more complex and troublesome task.

Our model supports both optimistic and pessimistic execution. These issues are not discussed any further, as they are orthogonal to the main contribution of this paper.

2.2.2 Non-blocking Execution of Agents

In mobile agent execution, a single failing component (agent, place, or machine) may block or corrupt the entire mobile agent execution, or result in a partial or complete loss of the agent. Assume, for instance, that the place p_2 crashes while executing a_2 (see Figure 4). Without adequate measures, the agent a_2 is lost.

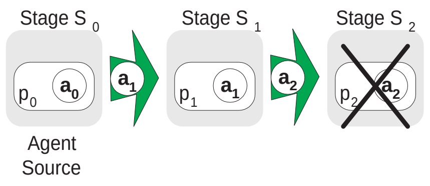

Figure 4. Logical agent execution where place p_2 crashes while executing a_2.

The loss of the agent a_2 can be prevented by checkpointing its code and its state on stable storage before starting its execution on p_2. In this case the agent a_2 survives the crash of p_2. After recovery of p_2, the execution of a_2 can proceed. However, while the place p_2 is down (i.e., p_2 has crashed but not yet recovered), the agent execution is blocked. One way to circumvent the blocking problem is the following. A copy of a_2 is kept on place p_1 (see Figure 4). If p_1 detects the crash of p_2, it resends a_2 to another place that provides the same service as p_2. This, however, requires the ability of p_1 to reliably detect failures of p_2, i.e., a reliable failure detection mechanism. Indeed, assume that p_1 erroneously detects the failure of p_2 and resends the agent to another place p'_2. We would have two instances of the agent, which violates the exactly-once execution property of agents at each stage. Obviously, in the Internet, reliable failure detection is impossible.

2.3 Introducing Replication

The discussion in the previous section has shown an approach to overcome the blocking problem. However, this approach leads to a violation of the exactly-once property, based on the impossibility of reliable failure detection.

Replication of the agent enables prevention of blocking without requiring a reliable failure detection mechanism. In other words, the failure detection mechanism is allowed to erroneously suspect failures. Adding redundancy at the stage execution masks system failures and enables the agent to continue execution despite failures, as shown in Figure 5. Instead of sending the agent to a single place p_i^0 at the next stage, the agent is forwarded to a set $\mathcal{M}_i$ of places $\{p_i^0,\ p_i^1,\ p_i^2,\ \ldots\}$. The places p_i^j thereby are either:

Hetero-places Every place p_i^j can potentially execute the agent and deliver the required services[3], but at most one of them will have executed the agent. For instance, if p_i^0 is the server of Swissair, p_i^j $(j \neq 0)$ could be the server of a Swissair partner airline that offers comparable quality and prices.

Hetero-places with witnesses This case is a variant of hetero-places. Every place p_i^j can potentially execute the agent, but only a subset of them can potentially deliver the required services, and at most one will have executed the agent. A *witness* is a place that can host and run the agent, but cannot provide the requested services (e.g., a weather forecast service) to the agent. The execution of a_i's call to the service on a witness p_i^j thus raises an exception within the agent and forces the agent to take adequate measures to handle this case. A potential reaction of the agent could be to return to the previous place. In the meantime, the state of the agent has changed, which prevents it from retrying the same (failed) place again. This approach is preferable

[3]Every place can execute the agent as long as it offers the corresponding execution environment (e.g., a Java runtime environment for a Java agent). However, not all places can generally deliver a particular service, because this service (e.g., telling machine) may not be installed on the place.

to an approach where the agent simply backtracks with no state changes, which potentially results in the agent retrying the failed place again and again.

Iso-places Every place p_i^j can execute the agent, and after execution, all of the places will reflect the execution of the agent. The modifications of the state of one place by the agent are reflected on the other places by the replication protocol of the places. Indeed, replica p_i^j is a replica of p_i^0 (same state as p_i^0), and not a replica of the agent. It is important to note that the replication of the places p_i^j is independent of our replication protocol for mobile agents (even though our protocol may potentially take advantage of it). Iso-places are the classical case of using redundancy to achieve high server availability (i.e., fault tolerance of the server).

To improve performance, the two instances of replication could be integrated. We refer to this case as *integrated replication*. This is not discussed here[4]; we assume the non-integrated replication case in this paper.

The model in Section 3 will handle these three cases.

The group of places of a stage S_i is responsible for the fault-tolerant agent execution. To prevent a machine crash from affecting multiple places in stage S_i, each place p_i^j $(j = 0, 1, ...)$ is generally located on a different machine (even though this is not a requirement).

Redundancy is not required at the agent source and destination. At the agent source, the agent initializes and reads its configuration and thus is still under the control of the agent owner. The agent destination is the place where the agent makes its results available to the agent owner.

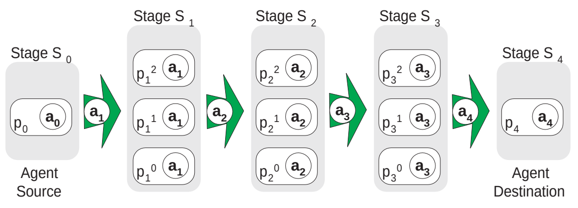

Figure 5. Example of an agent execution with three redundant places.

Revisiting the previous example (Figure 4) of a place failure, Figure 6 shows that the agent execution can now proceed at places p_2^1 and p_2^2, even though place p_2^0 failed. The agent owner thus has the guarantee that its agent will eventually terminate its execution.

[4]The mobile agent thus also ensures the consistency of the replicas. However, this requires that an instance of the agent (1) executes on all replicas, and (2) must not fail as long as the service is up and running.

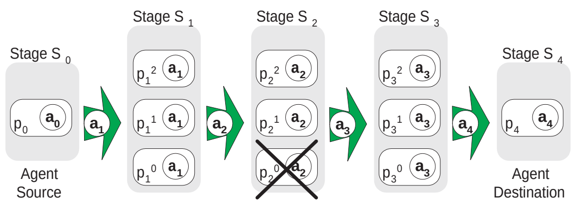

Figure 6. Agent execution with redundant places, where a place fails. The redundant places mask the place failure.

Figure 6 assumes that the entire place p_2^0 has crashed at stage S_2. However, our failure model (see Section 2.2) also identifies agent crashes. If only the agent fails, but the place survives, modifications to the place state by the failed agent survive. As the agent is then executed on place p_2^1, modifications are applied twice (to p_2^0 and p_2^1). Replication of the agent thus leads to a violation of the exactly-once execution property of mobile agents. Consequently, the replication protocol of agents has to undo the modifications of a_2 to the place p_2^0. This is fundamentally different from the traditional modeling approach for replication, where all the state is supposed to be maintained in the server replica and is lost with the crash of the replica. The computing environment thus automatically remains in a consistent state even when replicas fail.

Similar issues arise because of imperfect failure detection. Relying only on imperfect failure detection may lead to false suspicions. Given potentially false suspicions, a place p_i^j may be considered to have crashed, when, in fact, it has not. While another place is executing the agent, p_i^j already has partially or completely executed a_i. This leads to multiple executions of agent operations, which violates the exactly-once property of normal mobile agent execution. Again, the issue of undoing agent operations becomes prominent.

3 Fault-Tolerant Mobile Agent Execution: Specification of the Problem

We have claimed in the previous section that replication of agents allows blocking to be prevented without depending on reliable failure detection. However, to enforce the exactly-once property of mobile agent execution, the replicas have to decide on a place that has executed the agent[5]. This decision is modeled as an agreement problem.

[5]In the case of "iso-places", a violation of the exactly-once property of mobile agent execution leads to multiple executions of the agent operations on all places.

3.1 Basic Agreement Problem

Despite the differences of iso-places, hetero-places, and hetero-places with witnesses, we give a specification of the problem that encompasses the three cases. The idea is to model the execution of each stage S_i as an *agreement* problem. By $AgrPb_i$ we denote the agreement problem of stage S_i. The problem $AgrPb_i$ is to be solved by the places[6] in $\mathcal{M}_i$, and the solution is the decision on which all places in $\mathcal{M}_i$ agree. We denote by dec_i the decision (i.e., the solution) of $AgrPb_i$, with the safety properties:

- (Agreement) No two correct[7] places of stage S_i decide differently.
- (Uniform validity) If a place of stage S_i decides dec_i, then dec_i was proposed by some place of S_i and is the result of executing a_i at that place.
- (Uniform integrity) Every place of stage S_i decides at most once.

The liveness property requires that every correct place of stage S_i decides eventually.

The decision dec_i is as follows for the three cases identified in Section 2.3:

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

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

Iso-places (non integrated replication case). The decision dec_i is similar to the case with hetero-places.

Iso-places (integrated replication case) In the integrated replication case (see Section 2.3) the decision value is different: (1) the new state for all the places in $\mathcal{M}_i$, (2) the resulting agent a_{i+1}, and (3) the places $\mathcal{M}_{i+1}$ for a_{i+1}. Because of this difference in the integrated replication case, the agent must know beforehead whether it deals with iso-places or hetero-places.

In the following we no longer distinguish among the four cases.

The agreement problem is fundamental to enforce the exactly-once property of an agent execution. Assume, for instance, that the participants of a stage are separated by a network partition or slow communication (represented by the wavy line in Figure 7), such as p_2^0 from p_2^1 and p_2^2. If the places participating in a stage execution disagree on the outcome of the stage execution, a subset of these places (i.e., p_2^0) could decide on a different outcome (a'_3, S'_3) than the rest (a_3, S_3). As the action a_2 has thus been executed twice, resulting in two different agents a_3 and a'_3, the exactly-once property is violated.

[6]In the agent-based approach mentioned in Section 4.1.2, the agreement is actually solved by the agent itself.

[7]A place is called *correct* if it does not fail.

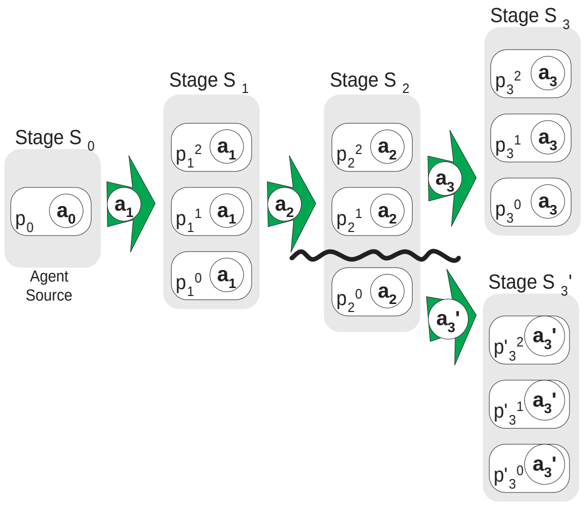

Figure 7. Disagreeing stage agents potentially lead to a violation of the exactly-once property.

3.2 Sequence of Agreement Problems

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

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

results are presented to the agent owner or to another designated destination.

4 Modeling Fault-Tolerant Mobile Agent Execution as a Sequence of Consensus Problems

The previous section has shown that fault-tolerant mobile agent execution can be expressed as a sequence of agreement problems. In this section we show how the agreement problem is solved using consensus and identify which mechanisms are required to reliably forward the agent. Our approach encompasses various system models such as process recovery or unreliable communication, depending on the implementation of consensus and of the reliable forwarding of agents.

Figure 8 depicts a mobile agent execution without failures. The execution at stage S_i consists of (1) one (or, in case of a failure, multiple) place(s) executing the agent, (2) the places in $\mathcal{M}_i$ reaching an agreement on the computation result, and (3) the reliable forwarding of the result a_{i+1} to the next stage S_{i+1}. The computational result contains the new agent a_{i+1} and the set of places executing the agent at stage S_{i+1}, as well as the place p_i^{prim} that has executed the agent. Note that the latter relates to stage S_i, whereas the former two results provide information about the next stage S_{i+1}.

Figure 9 illustrates the case of a place failure in stage 2. When p_2^1 detects the failure of p_2^0, which attempted an execution of a_2 first, it executes the agent and tries to impose its computation as the decision value of the agreement protocol to all $p_2^j \in \mathcal{M}_2$.

In our discussion so far, we assumed that $\mathcal{M}_{i-1}$ and $\mathcal{M}_i$ are a disjoint set of places. However, this is not a requirement. On the contrary, reusing places of stage S_{i-1} as witnesses[8] for S_i (see Section 2.3, case 2) improves the performance of the protocol and prevents high messaging costs. At a limit, every stage S_i merely adds another place to $\mathcal{M}_{i-1}$, while removing the oldest from the set $\mathcal{M}_{i-1}$. In this mode, forwarding costs are minimized and limited to forwarding the agent to the new place (see Figure 10). We call this mode *pipelined*.

In the following sections, we present the Consensus with Deferred Initial Value (DIV consensus) as the means to solve the successive agreement problem (Section 4.1) as well as a protocol to reliably forward the agent to the next stage (Section 4.2).

[8]One can also imagine particular scenarios that cover case 1 of Section 2.3. For instance, if an agent is to visit sequentially the servers of Swissair, Lufthansa, Delta, etc., we would rather deal with case 1.

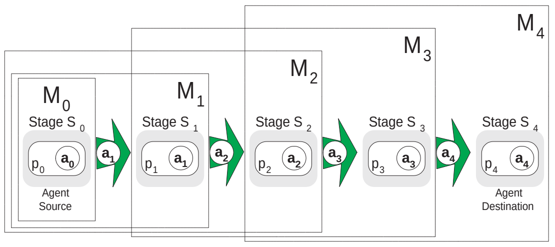

Figure 10. Pipelined mode without failures.

4.1 Solving the Agreement Using DIV Consensus

4.1.1 DIV Consensus Problem

The consensus problem is a well-specified and studied problem in fault-tolerant distributed systems research. It is defined in terms of the primitive $propose(v)$. Every process p_k in a set of processes Ω calls this primitive with an initial value v_k as an argument. Informally, the consensus allows an agreement on a certain value to be reached among the correct processes in Ω. Formally, consensus is specified as follows [3][9]:

- *Termination*. Every correct process in Ω eventually decides some value.
- *Uniform integrity*. Every process in Ω decides at most once.
- *Uniform agreement*. No two processes in Ω decide differently.
- *Uniform validity*. If a process in Ω decides some value v, then v was proposed by some process in Ω.

Reference [3] solves the consensus problem with the unreliable failure detector $\diamond S$ and a majority of correct processes. DIV consensus[10] [5] modifies the consensus problem such that all processes need not have an initial value. The initial value is computed during the execution of the consensus algorithm, whenever needed. Specifically, in the absence of failures, only one process computes the initial value. For this purpose, the participants do not invoke the consensus by passing their initial value as an argument. Rather, they pass a handler $\mathcal{H}(x)$ that allows the protocol to compute the initial value only when needed.

4.1.2 Applying DIV Consensus

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

[9]Actually, this is the definition of uniform consensus. However, as shown in [6], in some system models, each algorithm that solves consensus also solves uniform consensus. For this reason, we do not make a distinction between consensus and uniform consensus.
[10]DIV = Deferred Initial Value.

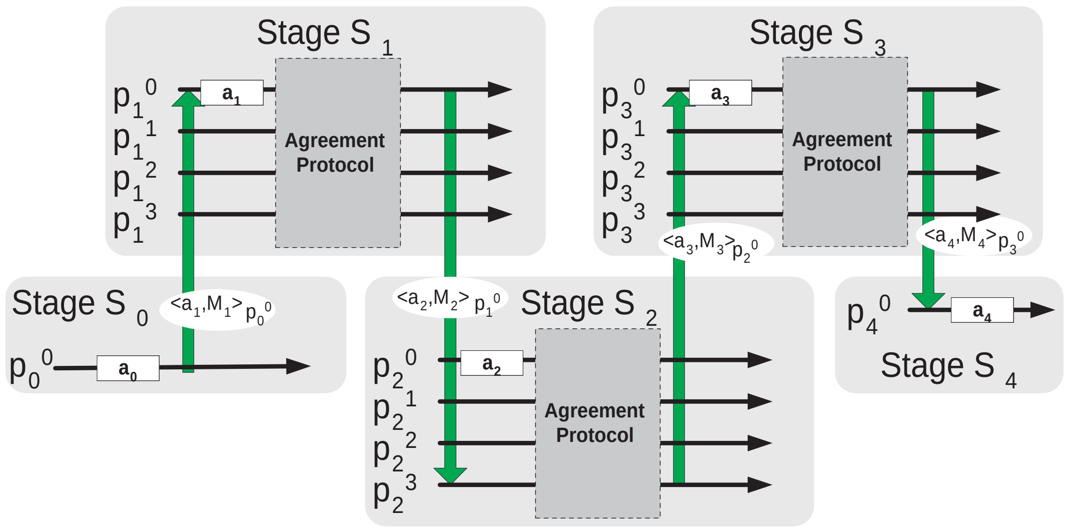

Figure 8. Agent execution without failures. The notation $< a_{i+1}, \mathcal{M}_{i+1} >_{p_i^{prim}}$ means that p_i^{prim} has executed agent a_i (which leads to a_{i+1} and M_{i+1}).

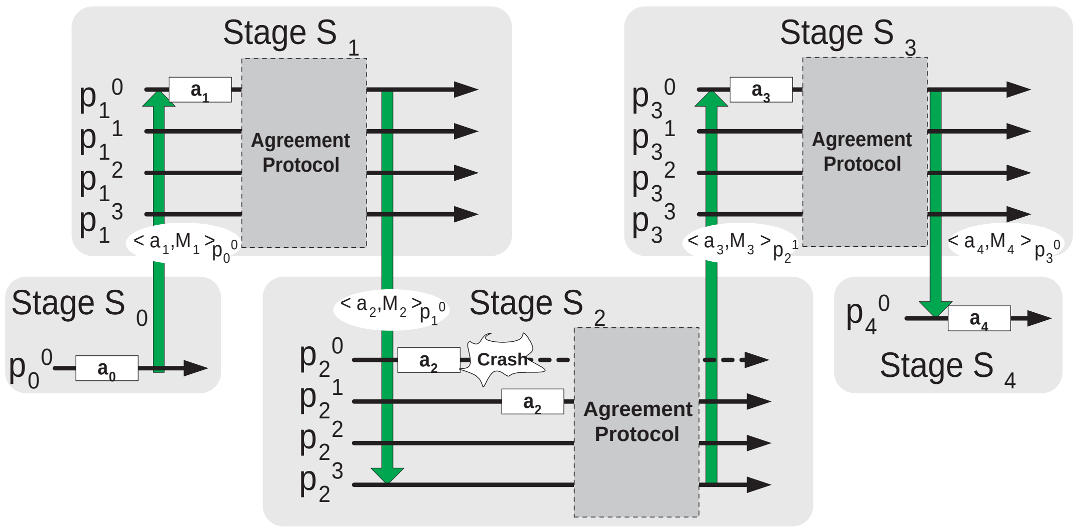

Figure 9. Agent execution with p_2^0 failing. An erroneously suspected place p_2^0 leads to the same situation.

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

Decision value dec. The execution of consensus decides on the tuple $dec_i =< a_{i+1}, \mathcal{M}_{i+1} >_{p_i^{prim}}$, where:

- p_i^{prim} is the primary of the current stage execution
- a_{i+1} is the resulting agent
- $\mathcal{M}_{i+1}$ is the set of places for stage S_{i+1}

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

From an implementation point of view, two approaches for solving consensus are possible: *place-based* or *agent-based*. In the place-based approach, the places implement the consensus protocol and only the primary place executes the agent (i.e., Ω is identical to $\mathcal{M}_i$). This approach requires a modification to all the places potentially hosting fault-tolerant mobile agents. On the other hand, the fault tolerance mechanisms are transparent to the agent developer, with the exception of the set of destination places $\mathcal{M}_i$ for every stage.

The agent-based approach requires the agent to implement the consensus protocol (i.e., Ω corresponds to the set of copies of a_i at the places $p_i^j \in \mathcal{M}_i$). Contrary to the place-based approach, a copy of the agent is run on each $p_i^j \in \mathcal{M}_i$. The agents reach an agreement among themselves and adopt the result. Clearly, the size of the agents increases because it also contains the code for the agreement. However, fault tolerance is transparent to the places and therefore does not require modifications to existing places. In the following, we focus on the place-based approach. The same reasoning, however, also applies for the agent-based approach.

In our discussion so far, we have not addressed the recovery from failures of machines, places, or agents. The protocol presented can easily be extended to also encompass recovery by using a corresponding version of consensus [1].

4.2 Reliably Forwarding the Agent Between S_i and S_{i+1}

Having solved the problem of executing the agent at a stage, we must address the issue of reliably forwarding the agent to the next stage. A naive approach (which we call *independent approach*) leads to a protocol, where every place in $\mathcal{M}_i$ multicasts the result dec_i to every place in $\mathcal{M}_{i+1}$. However, this incurs significant overhead in terms of message number as well as number of communication steps[11], depending on the protocol selected. Our approach (called the *integrated approach*) reduces this overhead considerably. For this purpose, we take advantage of the reliable broadcast used as part of the DIV consensus algorithm to send the decision dec_i to all participants of the consensus. Instead of reliably broadcasting dec_i only to $p_i^j \in \mathcal{M}_i$, we broadcast it to $\mathcal{M}_i \cup \mathcal{M}_{i+1}$. This ensures that the agent a_{i+1} is not lost. The changes to the DIV Consensus algorithm are small.

4.3 Handling Spawn Agents

So far, we have not discussed the case of spawning a child agent. This case is more difficult to handle than the normal case. We briefly outline an approach to this case.

The agent a_i at stage S_i can spawn a new agent b_{i+1}, which causes two agents to move off stage S_i, a_{i+1} resulting from executing a_i on S_i and b_{i+1} (see Figure 11). If a_i crashes, then all its modifications have to be undone. In particular, the spawn agent b_{i+1} has to be terminated as well and its operations undone. If b_{i+1} has been immediately sent off, undoing its operations is not a simple task. Indeed, the undoing message may trace agent b_{i+1} forever, never really catching up with it[12]. Therefore, b_{i+1} can only start execution when the current stage S_i has decided on the result. This requires that b_{i+1} and its set of executing places $q_i^j \in \mathcal{M}_{i+1}^b$ be part of the decision value $dec_i =< a_{i+1}, \mathcal{M}_{i+1}, < b_{i+1}, \mathcal{M}_{i+1}^b >>_{p_i^{prim}}$. The places in $\mathcal{M}_{i+1}^b$ only receive b_{i+1} when the decision is reliably broadcast to the concerned places. At this point, the decision is stable and the execution of the spawn agent can proceed.

5 Related Work

Fault tolerance for mobile agents has been an active field of research. Various work relies on replication to enable fault-tolerant mobile agent execution [4, 7, 10, 12]. We start by briefly discussing the first two references in Section 5.1. A more detailed comparison with our work and the latter two references is in Section 5.2.

5.1 Perfect Failure Detection and Byzantine Failure Model

One approach to fault-tolerant mobile agent execution has been suggested by Johansen et al. [7]. It assumes a fail-stop model, which corresponds to a perfect failure

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

[12] Murphy et al. [8] provide a solution for reliable message delivery to mobile agent, however, only in an environment without failures and at a considerable cost.

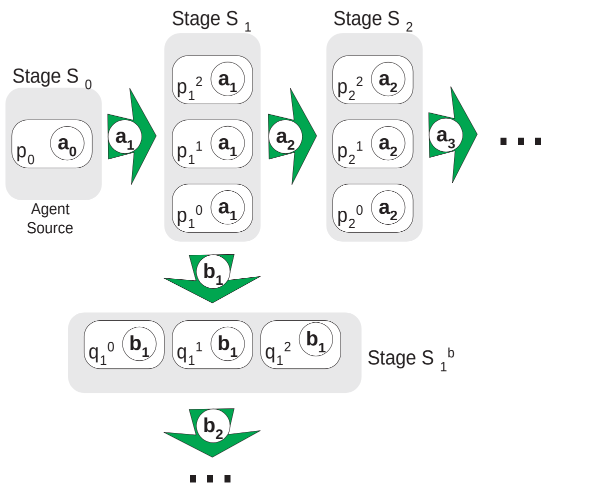

Figure 11. Agent a_1 spawns a new agent b_1 at stage S_1.

detector [11]. Our algorithm, on the other hand, tolerates incorrect failure detection and thus is of more general use. For example, [7] cannot handle partitions in the network. Rather, using an adequate algorithm for consensus, our solution is able to cope with network partitions.

Work in a Byzantine failure model has been considered by Schneider [12]. Schneider's protocol is based on the assumption that multiple replicas of a place exist in the system. Copies of the agent are executed on each replica, producing a deterministic result. A secret is associated with each copy and allows the valid outcome of a stage execution to be determined. As the Byzantine failure handling is based on the execution of multiple instances of the agent, exactly-once execution is not an issue and consequently not enforced by the protocol.

5.2 Transaction-Based Solutions

The solutions described in [4] and [10] are based on transactions. From a formal point of view, the Atomic Commitment problem (in the context of transactions) requires a perfect failure detector [6]. However, *weak* Atomic Commitment[13] does not require a perfect failure detector. In this context, a solution based on transactions is adequate in an environment such as the Internet, and it is worth comparing transaction-based solutions with our solution in more detail.

Rothermel et al. [10] model fault-tolerant and exactly-once mobile agent execution as a sequence of two problems: *leader election* (called voting protocol in [10]) and *distributed transactions*. Communication between consecutive stages S_i and S_{i+1} is based on transactional message queues, shown as shaded rectangles in Figure 12. At each stage, a place retrieves the agent from its input queue, executes the agent, and places the resulting agent in the input queues of the next stage's places as one transaction (illustrated by the dotted line). A place p_i^j can only commit the distributed transaction when it is elected by the places in $\mathcal{M}_i$, i.e., when it receives a majority of votes. Rothermel uses a 2-phase commit protocol [2] to commit the transactions, the election protocol thereby acting as a resource manager to the transaction manager.

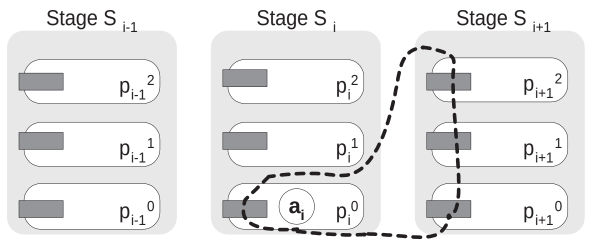

Figure 12. Rothermel's protocol. Shaded rectangles represent transactional message queues, whereas the dotted line indicates the borders of a stage transaction.

Modeling fault-tolerant mobile agent execution based on two different, interfering problems (i.e., leader election and distributed transactions) leads to a more complex solution than ours. In addition, understanding the weaknesses of such a solution is difficult and tedious. Our solution, however, is specified in terms of a *single* problem, the consensus problem, an intensively studied problem with well-understood solutions.

In Rothermel's model, the execution of the agent as well as the forwarding of the agent from stage S_i to S_{i+1} run as a transaction. Our model, in contrast, clearly decouples the mechanisms that provide fault tolerance from the execution properties of the agent operations. In particular, the agent operations do not need to run as a transaction. If they do, they have their own transaction manager.

Moreover, to our understanding, Rothermel's algorithm may block on a single place failure. Indeed, once the election protocol has elected a leader to commit the transaction, another leader can only be elected if the current leader explicitly resigns. Assume that the leader fails immediately after its election, but before committing the transaction. As the leader can no longer explicitly resign and thus no other leader is able to get elected and commit its transaction, the mobile agent execution is effectively blocked. The use of a 3-phase commit protocol instead of the (blocking) 2-phase

[13] In the Weak Atomic Commitment problem, if one of the data managers is suspected, the transaction may abort even if all data managers are correct and have voted *yes*, see [6].

commit proposed in [10] does not prevent this blocking problem.

Assis et al. [4] improve Rothermel's algorithm by overcoming some of its limitations. In particular, to prevent the blocking problem in [10], they use a different leader election protocol and commit the stage transaction using a 3-phase commit protocol [2]. However, this particular combination of leader election and transaction model may lead to a violation of the exactly-once property. Therefore, [4] relies on a so-called *distributed context database* for synchronization to prevent more than one concurrent leader and thus enforce the exactly-once property. This replicated database achieves fault tolerance, but makes the protocol even more complex as well as difficult to understand and to prove correct. It also creates a strong interdependence among the different places. By comparison, our algorithm only relies on message passing between the various places and does not incur the cost of having to set up this database.

6 Conclusion

In this paper, we show that simple approaches such as checkpointing prevent the loss of the agent, but are prone to blocking. Extending this approaches leads to solutions that can handle blocking, but may violate the exactly-once property of mobile agent execution. Replication allows us to address the issues of fault tolerance and blocking, while, using adequate agreement algorithms, not violating the exactly-once property of agent execution. We thus model fault-tolerant mobile agent execution as a sequence of agreement problems. Contrary to [7], our solution does not require a reliable failure detection mechanism. All the places involved in the execution of the agent a_i at stage S_i have to agree on the new agent, the set of places of the next stage S_{i+1}, as well as on the place that has executed a_i. We propose using consensus to solve the agreement problem at stage S_i. Consensus is a well defined and proved problem and thus renders our solution much simpler than those proposed so far [4, 10].

A prototype of the model is currently being developed. It will allow one to compute the cost of the fault tolerance mechanisms and to measure the performance of our approach.

References

[1] M. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. Technical Report TR 98-1676, Cornell University, Apr. 1998.

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

[3] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J ACM*, 43(2):225–267, 1996. A preliminary version appeared in Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing, pp 325-340, ACM Press, August 1991.

[4] F. de Assis Silva and R. Popescu-Zeletin. An approach for providing mobile agent fault tolerance. In K. Rothermel and F. Hohl, editors, *Mobile Agents, Proceedings of the Second International Workshop, MA'98*, LNCS 1477, pages 14–25. Springer Verlag, Sept. 1998.

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

[6] R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG-9)*, LNCS 972, pages 87–100, Le Mont-St-Michel, France, Sept. 1995. Springer-Verlag.

[7] D. Johansen, K. Marzullo, F. B. Schneider, K. Jacobsen, and D. Zagorodnov. NAP: Practical fault-tolerance for itinerant computations. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS'99)*, Austin, Texas, USA, June 1999.

[8] A. Murphy and G. Picco. Reliable communication for highly mobile agents. In *Proc. of the 1st Int. Symposium on Agent Systems and Applications and 3rd Int. Symposium on Mobile Agents (ASA/MA 99), Palm Springs, CA, USA*, pages 141–150. IEEE, Oct. 1999.

[9] Object Management Group. *Mobile Agent System Interoperability Facilities Specification, OMG TC Document orbos/97-10-05*, Nov. 1997. http://www.omg.org.

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

[11] L. Sabel and K. Marzullo. Simulating fail-stop in asynchronous distributed systems. In *Proceedings of the 13th Symposium on Reliable Distributed Systems (SRDS), Dana Point, CA, USA*, pages 138–147, Oct. 1994.

[12] F. Schneider. Towards fault-tolerant and secure agentry. In *Proceedings of the 11th International Workshop on Distributed Algorithms, Saarbrücken, Germany*, Sept. 1997. Invited paper.