Modeling Fault-Tolerant and Reliable Mobile Agent Execution in Distributed Systems

H. Hamidi and K. Mohammadi
Department of Electrical Engineering
Iran University of Science & Technology
Iran-Tehran

Abstract: The reliable execution of a mobile agent is a very important design issue in building a mobile agent system and many fault-tolerant schemes have been proposed so far. To further develop mobile agent technology, reliability mechanisms such as fault tolerance and transaction support are required. 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). To achieve fault tolerance for the agent system, especially for the agent transfer to a new host, we use Distributed Transaction Processing. 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.

We derive the Fault-Tolerant approach for Mobile Agents design which offers a user transparent fault tolerance that can be activated on request, according to the needs of the task, also discuss how transactional agent with types of commitment constraints can commit. Furthermore we propose a solution for effective agent deployment using dynamic agent domains.

Key-Words: Fault-Tolerant, Mobile Agent, Security, Network Management, Crash, Failures, Replication.

1 Introduction

A mobile agent is a computer program that acts autonomously on behalf of a user and moves through a network of heterogeneous machines[1]. Over the past years and originally triggered to a large extend by the work on Tele script [2], the field of mobile agents has attracted considerable attention, and mobile agent technology has been considered for a variety of applications such as systems and network management, mobile computing, information retrieval, and e-commerce [3,4]. 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. 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 [3,4,16]. However, on the one hand, [4] 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, 16] 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 propose 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.
2 Model

We assume an asynchronous distributed system, i.e., 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 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.](image)

2.2 Mobile Agent

A mobile agent executes on a sequence of machines, where a place \( p_i \) (0 \( i \) \( n \)) provides the logical execution environment for the agent [5]. Executing the agent at a place \( p_i \) is called a stage \( S_i \) of the agent execution [6]. 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 [6]. The sequence of places between the agent source and destination (i.e., \( p_0, p_1, ..., p_n \)) 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 2). Each stage action \( S_a \) consists of potentially multiple operations \( op_0, op_1, ..., op \). Agent \( a_i \) (0 \( i \) \( n \)) at the corresponding stage \( S_i \) represents the agent \( a \) that has executed the stage actions on places \( p_j \) (0 \( i \) \( n \)) 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). We denote the resulting agent \( a_{i+1} \). Place \( p_i \) forwards \( a_{i-1} \) to \( p_{i+1} \) (for 0 \( i \) \( n \)).

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.

2.3 Fault Model

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.

![Figure 2: Model of a mobile agent execution with three stages.](image)

Several types of faults can occur in agent environments. Here, we first describe a general fault model, and focus on those types, which are for one important in agent environments due to high occurrence probability, and for one have been addressed in related work only insufficiently.

- Node failures: The complete failure of a compute node implies the failure of all agent places and agents located on it. Node failures can be temporary or permanent.
- Failures of components of the agent system: Failures of agent places, or components of agent places become faulty, e.g. faulty communication units or incomplete agent directory. These faults can result in agent failures, or in reduced or wrong functionality of agents.
- Failures of mobile agents: Mobile agents can become faulty due to faulty computation, or other faults (e.g. node or network failures).
- Network failures: Failures of the entire communication network or of single links can lead to isolation of single nodes, or to network partitions.
- Falsification or loss of messages: These are usually caused by failures in the network or in the communication units of the agent systems, or of the underlying operating systems. Also, faulty transmission of agents during migration belongs to this type.

Especially in the intended scenario of parallel applications, node failures and their consequences are important. Such consequences are loss of agents, and loss of node specific resources. In general, each agent has to fulfill a specific task to contribute to the parallel application, and thus, agent failures must be treated. In contrast, in applications where a large number of agents are sent out to search and process information in a network, the loss of one or several mobile agents might be acceptable [2,3].
Machines, places, or agents can fail and recover later. A component that has failed but not yet recovered is called down; otherwise, it is up. If it is eventually permanently up, it is called good [7]. In this paper, we focus on crash failures (i.e., processes prematurely halt). Benign and malicious failures (i.e., Byzantine failures) are not discussed. A failing place causes the failure of all agent running on it. Similarly, a failing machine causes all places and agents on this machine to fail as well. We do not consider deterministic, repetitive programming errors (i.e., programming errors that occur on all agent replicas or places) in the code or the place as relevant failures in this context. Finally, a link failure causes the loss of the messages or agents currently in transmission on this link and may lead to network partitioning. We assume that link failures (and network partitions) are not permanent. The failure of a component (i.e., agent, place, machine, or communication link) can lead to blocking in the mobile agent execution. Assume, for instance, that place P1 fails while executing a1 (Fig. 4). While P1 is down, the execution of the mobile agent cannot proceed, i.e., it is blocked. Blocking occurs if a single failure prevents the execution from proceeding. In contrast, and execution is non-blocking if it can proceed despite a single failure, the blocked mobile agent execution can only continue when the failed component recovers. This requires that recovery mechanism be in place, which allows the failed component to be recovered. If no recovery mechanism exists, then the agents state and, potentially, even its code may be lost. In the following, we assume that such a recovery mechanism exists (e.g., based on logging [8]). Replication prevents blocking. Instead of sending the agent to one place at the next stage, agent replicas are sent to a set $M_i$ of places $p_i^0, p_i^1, \ldots$ (Fig. 4). We denote by $a_i^j$ the agent replica of $a_i$ executing on place $p_i^j$. But will omit the superscripted index if the meaning is clear from the context. Although a place may crash (i.e., stage 1 in Fig. 4), the agent execution does not block. Indeed, $p_i^1$ can take over the execution of $a_i$ and thus prevent blocking. Note that the execution at stages $S_0$ and $S_2$ is not replicated as the agent is under the control of the user. Moreover, the agent is only configured at the agent source and presents the results to the agent owner at the agent destination. Hence, replication is not needed at these stages. Despite agent replication, network partitions can still prevent the progress of the agent. Indeed, if the network is partitioned such that all places currently executing the agent at stage $S_i$ are in one partition and the places of stage $S_{i+1}$ are in another partition, the agent cannot proceed with its execution. Generally (especially in the Internet), multiple routing paths are possible for a message to arrive at its destination. Therefore, a link failure may not always lead to network partitioning. In the following, we assume that a single link failure merely partitions one place from the rest of the network. Clearly, this is a simplification, but it allows us to define blocking concisely. Indeed, in the approach presented in this article, progress in the agent execution is possible in a network partition that contains a majority of places. If no such partition exists, the execution is temporally interrupted until a majority partition is established again. Moreover, catastrophic failures may still cause the loss of the entire agent. A failure of all places in $M_1$ (Fig. 4), for instance, is such a catastrophic failure (assuming no recovery mechanism is in place). As no copy of $a_1$ is available anymore, the agent execution can no longer proceed. In other words, replication does not solve all problems. The definition of non-blocking merely addresses single failures per stage as they cover most of the failures that occur in a realistic environment. In the next section, we classify the places in $Mi$ into iso-places and hetero-places according to their properties [9].

3 Replication and the Exactly-Once Property

Replication allows us to prevent blocking. However, it can also lead to a violation of the exactly-once execution property. Indeed, the exactly-once property and non-blocking are closely related. Assume, for instance, that place $p_i^0$ fails after having partially executed agent $a_i$ (see Fig. 5). After some time, $p_i^1$ detects the failure of $p_i^0$ and takes over the execution of $a_i$. The agent $a_i$ has now (partially) executed multiple times. Consequently, upon recovery, place $p_i^0$ needs to undo the modifications performed by agent $a_i$; the issues if "only an agent replica fails, at not the place in this case, modifications by the failed agent to the place state survive. As the agent is then executed on place $p_i^1$, modifications are applied twice (to $p_i^0$ and $p_i^1$). Replication of the agent thus leads to a violation of the exactly-once execution property of mobile agents. Consequently, the replication protocol of agents has to undo the modifications if $a_i$ to the place $p_i^0$.

Another source for the violation of the exactly-once execution property is unreliable failure detection. Indeed, in an asynchronous system such as the Internet, no boundaries exist on communication delays or on relative process speeds. Hence, it is impossible to detect failures reliably [10]. Assume, for instance, that $p_i^1$ suspects $p_i^0$ has failed, when, in fact, $p_i^0$ has not (see Fig. 3). This may lead to two agents $a_{i+1}$ and $a_2$, which are potentially sent to different places for the next stages. Clearly, this is a violation of the exactly-once execution property. In summary, a violation of the exactly-once execution property can occur 1) in the agent replicas and 2) at the places (or rather, the services running on the places). Clearly, both instances are related in that a violation of the exactly-once execution property at the places is a consequence of multiple executions of the agent (e.g., $a_i$ on $p_i^0$ and $a_i$ on $p_i^1$).

Figure 5: Replication potentially leads to a violation of the exactly-once property.
4 The Building Blocks for Fault-Tolerant Mobile Agents

The previous section has shown that fault-tolerant mobile agent execution can be expressed as a sequence of agreement problems. In this section, we identify two building blocks for fault-tolerant mobile agent execution:

1) consensus and 2) reliable broadcast. Building block 1) is used to solve the agreement problem at stage \( S_i \), whereas 2) allows the agent to be forwarded reliably between consecutive stages. Our approach encompasses various system models such as process recovery, depending on the implementation of consensus and of the reliable forwarding of agents.

Fig. 6 depicts a fault-tolerant mobile agent execution of the execution at stage \( S_i \) consists of 1) one (or, in case of a failure or false suspicions, multiple) places executing the agent, 2) the agent replicas running on the places in \( M_i \) 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 place executing the agent at stage \( S_{i+1} \) (i.e., \( M_{i+1} \)), as well as the places \( p_i^{prim} \) that has executed the agent.

Note that the latter relates to stage \( S_i \), where as the former two results provide information about the next stage \( S_{i+1} \).

Stage 2 in Fig. 6 illustrates the case of a place failure. When \( a_2^1 \) detects the failure of \( a_2^0 \), it starts executing and tries to impose its computation as the decision value of the agreement protocol to all \( p_2^j \in M_2 \). Upon recovery, \( a_2^0 \) learns the outcome of the agreement (i.e., \( dec_i \)). If \( p_2^0 = p_2^{prim} \), the modification of \( a_2^0 \) on \( p_2^0 \) become permanent; otherwise (i.e., \( p_2^0 \neq p_2^{prim} \)), they are undone / aborted.

The Consensus problem is a well-defined 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 \( \Omega \) 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 process in \( \Omega \). This value, called decision value, is an element of the set of initial values \( v_k \). The formal specification of the Consensus problem is given in [11].

The algorithm in [11] solves the Consensus problem with the unreliable failure detector \( S^S \) and a majority of correct processes. DIV consensus [12] 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 \( H(x) \) that allows the protocol to compute the initial value only when needed.

4.1 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 transformation:

**Initial handler** \( H(x) \)

The initial handler \( H(x) \), passed as 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 only once.

**Decision value** \( dec \)

The execution of DIV consensus decides on the tuple \( dec = \{ a_{i+1}, M_{i+1} \} \).

DIV consensus ensures that all \( a_i \) running on \( p_i^j \in M_i \) agree on the \( p_i^{prim} \) that has executed \( a_i \), on the new agent \( a_{i+1} \), as well as on the place of the next stage \( S_{i+1} \).

The version of DIV consensus presented in [12] assumes reliable communication channels. As stated in [12], the algorithm can easily be using an approach along the lines of [13]. As long as the network is partition in such away that one partition contains a majority places of a stage the execution is not blocked. Extended to handle unreliable communication channels as well by using an approach along the lines of [13]. As long as the network is partitioned in such away that one partition contains a majority of places of a stage, the execution is not blocked. More over, it makes the assumption that a majority of \( a_i \) dose not fail, i.e., is correct. In our system model, agents are good. However, the termination of the argument does not depend on the recovery of the agents. Rather, we assume that a majority of them does not fail while DIV consensus executes. When they recover, they no longer participate in consensus. But, they undo their modifications (if needed) to ensure exactly-once. Assuming good agents maintains consistency from the point of view of the agent owner (or application) who has launched the mobile agent by bringing all accessed places to a consistent state.

However, the protocol presented could easily be extended to also encompass recovery by using a corresponding version of consensus along the lines of [14]. Indeed, we argue in the next section that recovery needs to be supported to a certain degree because of asynchronous agent propagation.

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

4.1.1 Asynchronous Agent Propagation

We have assumed an asynchronous system where there is no bound on the transmission delay of messages. This has an impact on the different instances of the argument protocol (i.e., DIV consensus) that run at each stage \( S_i \) of an agent execution. Because of the asynchrony, the agent \( a_i \), may not arrive simultaneously at the different places \( p_i^j \) of stage \( S_i \). Assume, for instance, that the agent replicas \( a_i^0, a_i^1, a_i^2 \) are sent respectively to \( p_i^0, p_i^1, p_i^2 \in M_i \) (see Fig. 6) and assume that \( a_i^2 \) arrives late at \( p_i^2 \). DIV consensus may have already
started executing for agent replicas \( a^0 \) and \( a^1 \) when \( a^2 \) arrives. The execution of DIV consensus may even have terminated when \( a^2 \) arrives. The late arrival of \( a^2 \) is indistinguishable to \( a^0 \) and \( a^1 \) from the crash of \( a^2 \) followed by the recovery of \( a^2 \). Agent \( a^2 \) thereby always uses a model of recovery, where no partial state survives a crash. In summary, the asynchrony assumption thus forces us indirectly to support the recovery of agents after a crash.

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 naïve approach leads to a protocol where every place in \( M_i \) broadcasts the result \( dec \) to every place in \( M_{i+1} \). As DIV consensus assumes that a majority of places in \( M_i \) do not fail, it is ensured that at least one place actually sends the agent.

Figure 6: Agent execution with a failing.

5 Optimization: Pipelined Mode

In our discussion so far, we have assumed that \( M_{i-1} \) and \( M_i \) are disjoint sets of places. However, this is not a requirement [15],[16]. On the contrary, reusing places of stage \( S_{i-1} \) as witnesses for \( S_i \) improves the performance of the protocol and prevents high messaging costs; the pipelined mode assumes hetero-places with witnesses. At a limit, every stage \( S_i \) merely adds another place to \( M_{i-1} \), while removing the oldest from the set \( M_{i-1} \). In this mode, forwarding costs are minimized and limited to forwarding the agent to the new place (see fig.7).

Figure 7: Model of the pipelined mode.

We call this mode pipelined. Note that, for set \( S \), we assume the existence of a place that acts as a witness for the stage execution (not displayed in fig.7).

The execution at stage \( S_0 \) (or \( S_4 \)) is not replicated and no witnesses are needed for \( M_0(M_4) \).

5.1 Influence of the size of the Agent

The size of the agent has a considerable impact on the performance of the fault-tolerant mobile agent execution. To measure this impact, the agent carries a Byte array of variable length used to increase the size of the agent. As the results in fig. 8 show, the execution time of the agent increases linearly with increasing size of the agent. Compared to the single agent, the slope of the curve for the replicated agent is steeper.

5.2 Optimization: Pipelined Mode

We have introduced the pipelined mode in section 5. It results in a reduced number of messages (i.e., forwarded agents) as the agent only needs to be forwarded to one new place of the next stage. This reduced number of messages does not entirely show up in the performance gain because our algorithm waits only for the reception of the first message. Reducing the number of messages, however, has a great impact on the underlying communication infrastructure. Nevertheless, fig. 9 shows that the pipelined mode has a lower execution time than the normal replicated agent.

5.3 Discussion

In [17], Silva et al. measure the performance overhead of their approach, called James, and also compare it to a partial implementation of the approach in [16] that is, similarly to our approach, based on replication. For an agent of size of about 1 Kbyte, they measure an overhead of 20 percent for James, while the approach in [16] introduces an overhead of 300 percent. This latter overhead is comparable to the overhead introduced by FATOMS (see Table 1). However, the approach in [16] is blocking, whereas our approach is not, and Silva et al. mention having made only a partial implementation. James uses an approach with different characteristics which we call the commit-at destinations approach and it is not clear to us what exactly has been taken into account in their performance measurements (i.e., overhead of fault-tolerant lookup directory and of locking).

Hence, it is difficult to compare the performance of the two approaches. Moreover, our measurements are more detailed as we also isolate the cost of consensus at a stage.
Figure 8: Costs of single and replicated agent execution with increasing agent size

Figure 9: Performance gain with the pipelined mode for eight stages.

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 agree problems. Our solution does not require a reliable failure detection mechanism. All the places involved in the execution of the agent at stage Si have to agree on the new agent, the set of places of the next stage Si+i, as well as on the place that has executed it. We propose using consensus to solve the agreement problem at stage Si. 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