FAULT-TOLERANT AND TRANSACTIONAL MOBILE AGENT EXECUTION

THÈSE N^{O} 2654 (2002)

PRÉSENTÉE À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

DANS LE DOMAINE DES SYSTÈMES RÉPARTIS

PAR

Stefan PLEISCH

ingénieur informaticien diplômé EPF
de nationalité suisse et originaire de Luzein (GR)

acceptée sur proposition du jury:

Prof. A. Schiper, directeur de thèse
Prof. K. Aberer, rapporteur
Prof. G. Alonso, rapporteur
Prof. K. Marzullo, rapporteur

Lausanne, EPFL
2002

Abstract

Mobile agents constitute a computing paradigm of a more general nature than the widely used client/server computing paradigm. A mobile agent is essentially a computer program that acts autonomously on behalf of a user and travels through a network of heterogeneous machines. However, the greater flexibility of the mobile agent paradigm compared to the client/server computing paradigm comes at additional costs. These costs include, among others, the additional complexity of developing and managing mobile agent-based applications. This additional complexity comprises such issues as reliability. Before mobile agent technology can appear at the core of tomorrow's business applications, reliability mechanisms for mobile agents must be established. In this context, *fault tolerance* and *transaction support* are mechanisms of considerable importance.

Various approaches to fault tolerance and transaction support exist. They have different strengths and weaknesses, and address different environments. Because of this variety, it is often difficult for the application programmer to choose the approach best suited to an application. This thesis introduces a classification of current approaches to fault-tolerant and transactional mobile agent execution. The classification, which focuses on algorithmic aspects, aims at structuring the field of fault-tolerant and transactional mobile agent execution and facilitates an understanding of the properties and weaknesses of particular approaches.

In a distributed system, any software or hardware component may be subject to failures. A single failing component (e.g., agent or machine) may prevent the agent from proceeding with its execution. Worse yet, the current state of the agent and even its code may be lost. We say that the agent execution is *blocked*. For the agent owner, i.e., the person or application that has configured the agent, the agent does not return. To achieve fault-tolerance, the agent owner can try to detect the failure of the agent, and upon such an event launch a new agent. However, this requires the ability to *correctly* detect the crash of the agent, i.e., to distinguish between a failed agent and an agent that is delayed by slow processors or slow communication links. Unfortunately, this cannot be achieved in systems such as the Internet. An agent owner who tries to detect the failure of the agent thus cannot prevent the case in which the agent is mistakenly assumed to have crashed. In this case, launching a new agent leads to multiple executions of the agent, i.e., to the violation of the desired *exactly-once* property of agent execution. Although this may be acceptable for certain applications (e.g., applications whose operations do not have side-effects), others clearly forbid it. In this context, launching a new agent is a form of replication. In general, replication prevents blocking, but may lead to multiple executions of the agent, i.e., to a violation of the exactly-once execution property. This thesis presents an approach that ensures the exactly-once execution property using a simple principle: the mobile agent execution is modeled as a sequence of agreement problems. This model leads to an approach based on two well-known building

blocks: consensus and reliable broadcast. We validate this approach with the implementation of FATOMAS, a Java-based FAult-TOlerant Mobile Agent System, and measure its overhead.

Transactional mobile agents execute the mobile agent as a transaction. Assume, for instance, an agent whose task is to buy an airline ticket, book a hotel room, and rent a car at the flight destination. The agent owner naturally wants all three operations to succeed or none at all. Clearly, the rental car at the destination is of no use if no flight to the destination is available. On the other hand, the airline ticket may be useless if no rental car is available. The mobile agent's operations thus need to execute *atomically*, i.e., either all of them or none at all. Execution atomicity also needs to be ensured in the event of failures of hardware or software components. The approach presented in this thesis is *non-blocking*. A non-blocking transactional mobile agent execution has the important advantage that it can make progress despite failures. In a blocking transactional mobile agent execution, by contrast, progress is only possible when the failed component has recovered. Until then, the acquired locks generally cannot be freed. As no other transactional mobile agents can acquire the lock, overall system throughput is dramatically reduced. The present approach reuses the work on fault-tolerant mobile agent execution to prevent blocking. We have implemented the proposed approach and present the evaluation results.

Version abrégée

Récemment, les agents mobiles sont devenus à la mode en tant que paradigme plus général que le traditionel paradigme client/serveur. Un agent mobile est un programme qui agit de façon autonome pour le compte d'un utilisateur et qui traverse un réseau d'ordinateurs hétérogènes. Cependant la plus grande flexibilité du paradigme, comparé au paradigme client/serveur, a un coût. Ce coût comprend entre autres une plus grande complexité du développement et de la gestion des applications. Il faut aussi compter dans cette complexité additionelle l'aspect fiabilité. Avant que la technologie des agents mobiles ne puisse s'installer au coeur des futures applications de *business*, la fiabilité doit être développée. Parmi ces mécanismes, la *tolérance aux fautes* et le *support transactionnel* sont extrêmement importants.

Actuellement il existe différentes approches pour la tolérance aux fautes et le support transactionnel, chacune ayant des avantages et des inconvénients, et résolvant des problèmes dans des domaines d'applications différents. À cause de cette variété, il est souvent difficile pour le développeur d'applications de choisir la meilleure approche. Une des contributions de cette thèse est l'introduction d'une classification des approches existantes. Cette classification, qui est centrée sur les aspects algorithmiques, tente de structurer le domaine des agents tolérants aux fautes et transactionnels, et aide à comprendre les avantages et désavantages d'approches particulières.

Dans un système distribué n'importe quel composant logiciel ou matériel peut être sujet à une défaillance. Celle-ci (par exemple, celle d'un agent ou d'une machine) peut empêcher l'agent de continuer son exécution ; plus grave encore, l'état actuel de l'agent, et même son code, peuvent être perdus. Nous disons que l'agent est *bloqué*. Le propriétaire de l'agent, c'est-à-dire la personne ou l'application qui a configuré l'agent, constate que l'agent ne revient pas. Pour être tolérant aux fautes, le propriétaire peut essayer de détecter la défaillance de l'agent, dans le but d'envoyer un nouvel agent. Par contre, ceci nécessite de pouvoir détecter *correctement* les défaillances d'un agent, et donc de pouvoir faire la distinction entre un agent défaillant et un agent qui aurait été retardé par des processeurs ou des liens de communication lents. Malheureusement, ceci n'est pas possible dans des systèmes tels que l'Internet. Le propriétaire de l'agent qui essaye de détecter la défaillance de son agent ne peut donc pas empêcher le cas où il commettrait une erreur sur l'état de l'agent. Dans ce cas, il risque d'envoyer un nouvel agent, ce qui conduit à des exécutions multiples de l'agent, et donc à une violation de la propriété d'exécution "une et une seule fois" (*exactly-once*). Bien que ceci puisse être acceptable pour certaines applications, comme par exemple les applications dont les operations n'ont pas d'effets de bord, ce n'est pas le cas de toutes les applications. Dans ce contexte, l'envoi d'un nouvel agent est une forme de réplication. La réplication empêche les blocages de l'agent, mais peut conduire à des exécutions multiples de l'agent. Cette thèse propose une approche qui assure la propriété d'exécution *exactly-once* en

utilisant un principe simple : l'exécution de l'agent mobile est modélisée comme une séquence de problèmes d'agréments. Ce modèle mène a une approche fondée sur deux blocs de bases : le consensus et la diffusion (*multicast*) fiable. Notre approche est validée grâce à FATOMAS (*FAult-TOlerant Mobile Agent System*), un système développé en Java, dont nous évaluons la performance.

Les agents mobiles transactionnels s'exécutent comme une transaction. Supposons par exemple qu'un agent doive acheter un ticket d'avion, réserver une chambre d'hôtel et louer une voiture. Le propriétaire de l'agent exige en général que les trois opérations réussissent ou aucune. En effet, louer une voiture n'a pas de sens si aucun vol n'est disponible. Par ailleurs, le ticket d'avion est inutile si aucune voiture ne peut être louée sur place. Les opérations de l'agent mobile doivent donc s'exécuter de facon *atomique*. L'atomicité de l'exécution doit être garantie même en cas de défaillance de composants logiciels ou matériels. L'approche proposée dans cette thèse est non-bloquante. Une exécution non-bloquante de l'agent transactionnel permet de continuer l'exécution malgré des défaillances. Dans une exécution bloquante, un progrès n'est possible que lorsque le composant défaillant redémarre. Entre temps, les verrous acquis ne peuvent être libérés, et comme aucun autre agent mobile transactionnel ne peut acquérir ces verrous, la performance du système est considérablement réduite. Notre approche est basée sur notre proposition dans le domaine des agents tolérants aux fautes. Nous avons implémenté l'approche proposée, et présentons ses performances.

Acknowledgments

This thesis was only possible in its current form because of the support and help of various people. My foremost thanks go to André Schiper for accepting to act as advisor while I was pursuing the PhD and for his constant support and constructive comments throughout the thesis.

The IBM Zurich Research Laboratory has provided me with the opportunity to work as a PreDoc. I am very grateful to Metin Feridun, who has acted as my mentor during the time of my PhD and has supported me in every possible way. Furthermore, I like to thank Andreas Wespi for his support and Lilli-Marie Pavka for proof reading many of my publications.

Special thanks go to the president of the jury, Prof. Martin Odersky, and to the members of the jury, Prof. Karl Aberer, Prof. Gustavo Alonso, and Prof. Keith Marzullo, for the time they have invested to examine this thesis.

I am also grateful to the members of the Distributed Systems Laboratory at EPFL, in particular, Xavier Défago for the discussions on DIV consensus and semi-passive replication and for providing the Latex sources for these algorithms to me, and Arnas Kupsys for the many discussions and the collaboration on parts of the content in Chapter 7.

I am thankful to the members of the former Distributed Systems and Network Management group at the IBM Zurich Research Lab, Anthony Bussani, Christian Hörtnagl, Jens Krause, and Sean Rooney, for collaborations on a variety of projects and the frequent discussions. Jens Krause is the developer of MOPROS, the mobile process platform used for some of the testing performed for this thesis. He has answered all my questions and has provided constant support in case of problems. Also, he has been my office mate for some time and has shared the occasional troubles of a PreDoc with me. Finally, Matthieu Verbert has helped with the French part of this thesis.

My thanks also go to the people who contributed through discussions to one or more parts of this thesis. In particular, Felix Gärtner, Rachid Guerraoui, Klaus Kursawe, Riccardo Jimenez Peris, Bettina Kemme, and Matthias Wiesmann.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

The concept of distributed client/server computing can be too limiting for certain applications. For instance, the functionality of a server is defined by its computing interface and generally cannot be adapted to the client's needs without recompiling that interface. Clearly, this limits the flexibility with which a client can use a server and forces the client to process the server data locally according to the client's needs. Hence, the *mobile code* computing paradigm was introduced. With this paradigm, not only data but also the code to act on the data is transported between the client and the server. Transporting the code makes the applications more flexible and allows the client to tailor the functionality provided by the server to its needs. Consider, for instance, a client that accesses a database. Instead of transferring all the data to the client and filtering the data on the client, the client can send the actual filter code to the database server, perform the filtering close to the data, and return only the result to the client. Furthermore, code can be transferred from the server to the client. Indeed, among the most widespread applications of the mobile code computing paradigm are *applets*[1], small pieces of Java code, that are downloaded from a Web server to the client machine and executed there. The most general variety of the mobile code computing paradigm are *mobile agents* and as such have received considerable attention. A mobile agent[2] is a computer program that acts autonomously on behalf of a user and moves through a network of heterogeneous machines.[3] With mobile agents, it is possible to bring the code close to the resources, which is not foreseen by the traditional client/server paradigm. Mobile agent technology thus has been considered for a variety of applications [CGH+95, CHK98, LO99] such as systems and network management [BPW98, GFP99], mobile computing [TST01], information retrieval [TR00], and e-commerce [MGM99]. Moreover, mobile agent technology has attracted renewed interest in the context of environments that provide/rent distributed computing resources to users (e.g., grid computing [FKT01]) [BSH02]. In such environments, resources are available on various machines, and the application is executed on any machine that is able to provide the

[1] See Sun's Java applet homepage http://java.sun.com/applets/index.html

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

[3] The mobile agent research community has not yet agreed on a common definition of mobile agents. Consequently, various definitions exist. For the purpose of this thesis, we adopt the definition given in [Mat98, Obj00]. It is beyond the scope of this thesis to provide an universally accepted definition.

requested resources. This, however, may require that the application, i.e., its state and code, be moved to the resources, where it can be executed. This is exactly the problem that mobile agent technology tries to address, although generally a mobile agent makes multiple hops and not just one.

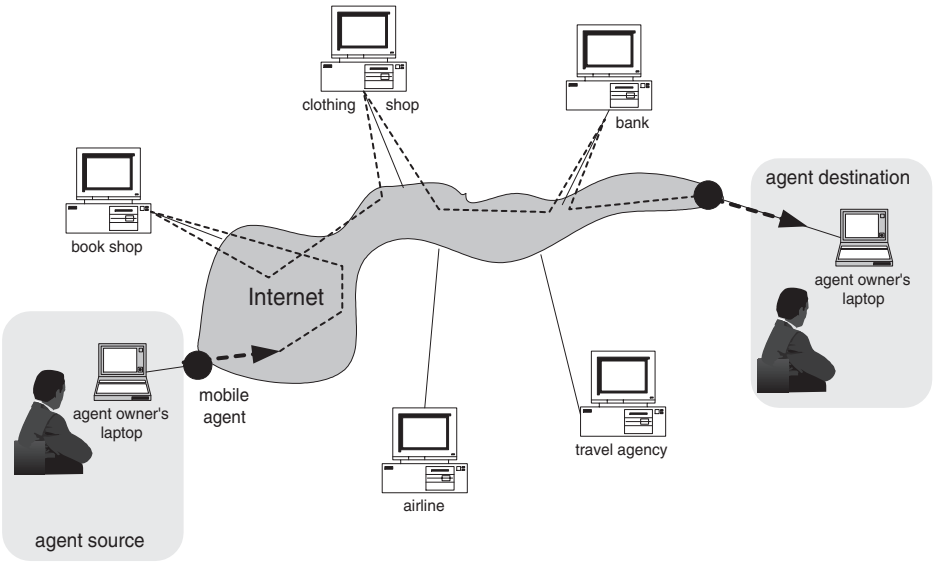

Figure 1.1: Example of an agent execution. The agent owner injects the agent into the Internet. The agent sequentially visits multiple servers and is finally collected by the user again.

The greater flexibility of the mobile agent paradigm compared to the client/server computing paradigm comes at additional costs. These costs include, among others, the additional complexity and increased difficulties of developing and managing applications in this context. The additional complexity comprises such issues as reliability. Before mobile agent technology can appear at the core of tomorrow's business applications, reliability mechanisms for mobile agents must be established. Among these reliability mechanisms, *fault tolerance* and *transaction support* are mechanisms of considerable importance and are the subject of this thesis. For example, assume a mobile agent in the context of e-commerce, whose task is to buy a particular book, a T-shirt, and to withdraw money (i.e., electronic money such as e-coins [CFN90]) from a bank account (see Figure 1.1). The agent owner (i.e., the person or application that has configured and launched the agent) creates the agent on the owner's device and configures the actions sa_1 (i.e., buying the book), sa_2 (buying the T-shirt), and sa_3 (money withdrawal). Once the agent has been launched into the network, it autonomously searches for the bookstore with the best price offer for the particular book, and for a T-shirt that corresponds to the agent owner's preferences (e.g., size and color). Having executed actions sa_1, sa_2, and sa_3 the agent delivers the results (electronic money and receipts of the book and T-shirt) to the agent owner. Unfortunately, failures can lead to undesirable results of the mobile agent execution. Indeed, any software or hardware component in a distributed system can be subject to failures. Assume, for instance, that the agent has bought the book at the book shop and currently executes on the server of a clothing shop which fails

by crashing (i.e., simply stops working). The agent execution thus cannot proceed any more. Worse yet, the current state of the agent and even its code may be lost. In this case, we say that the mobile agent execution is *blocked*. After a certain time, the agent owner finds that the agent has not returned yet. In an asynchronous system such as the Internet, it is impossible to detect correctly whether the agent has failed or is merely slow [FLP83], as no boundaries on relative processor speed and communication delay exist. In other words, the agent owner cannot correctly detect failures of the mobile agent. Nor does the agent owner generally know where exactly the agent failed and which shops it has visited, as the shops are autonomously selected by the agent. If this selection algorithm is non-deterministic (i.e., if its result cannot be reproduced) then it is difficult or even impossible to reproduce the itinerary of the mobile agent. Sending another agent to perform the same tasks may lead to multiple executions of the agent's code, i.e., a violation of the *exactly-once* execution property. Whereas this is not a problem for idempotent operations (i.e., operations that yield an equivalent result when redone an arbitrary number of times or only once [GR93]), it leads to incorrect system states if the agent contains non-idempotent operations. Assume, for instance, that the failed agent has withdrawn money from a bank account. Sending a second agent results in two money withdrawals, which is clearly undesirable to the agent owner. On the contrary, reading the balance of a bank account is an idempotent operation that can be executed multiple times without influencing the state of the bank account nor the state of the agent (unless the account balance has changed in the meantime).

Therefore, the agent owner has to wait for the agent to return. However, the agent execution can only proceed if the failed machine and the agent recover. In the meantime, the agent execution is blocked, in the worst case indefinitely (if the machine does not recover). If the agent owner is another application, then this application may be blocked as well. Blocking is thus undesirable in a mobile agent execution and mechanisms that prevent blocking are needed. The problem of blocking is a fundamental issue in fault tolerance and can be addressed by masking the occurrence of failures in a computing system to the user. The mechanism suggested for this is replication. Although replication prevents blocking, it may lead to a violation of the exactly-once execution property. For instance, sending another agent if the first does not return after a certain time is a form of replication and may lead to multiple executions. In the present thesis, we propose an approach for fault-tolerant mobile agents that prevents blocking *and* ensures that the agent is executed exactly-once. The idea is to model fault-tolerant mobile agent execution as a sequence of agreement problems. In an agreement problem [CT96], a set of processes, the so-called participants in the agreement problem, all agree on a certain result. This well-studied problem is at the basis of the present approach. The model is implemented in FATOMAS (FAult-TOlerant Mobile Agent System) and the performance evaluation of the implementation is presented.

Moreover, we show how our approach relates to existing approaches to fault-tolerant mobile agent execution. For this purpose, we propose a classification of existing approaches to facilitate an understanding of the strengths and weaknesses of these approaches. In particular, we will show that current solutions are either complex and thus difficult to prove correct [RS98, ASPZ98], or that they make limiting assumptions such as correct failure detection [JMS+99] or strict timing constraints[4] [PPG00].

[4]These strict timing constraints also enable correct failure detection.

Although fault tolerance prevents blocking of the mobile agent, it does not guarantee the success of the actions. Revisiting our example above, the desired book may be sold out and the account balance equal to 0. Hence, both operations will be unsuccessful, although no failure occurs. We say that the operations fail *semantically*. Although some operations are unsuccessful, others (i.e., purchase of a T-shirt) may have succeeded. Upon its return, the agent has acquired a T-shirt on behalf of the agent owner, but has failed to buy the book and withdraw the money. The agent owner still keeps the T-shirt. Whereas the actions in this example are independent, other applications require that either all actions be successful or none at all. Assume a mobile agent that books a flight and rents a car at the flight destination. Renting a car at the destination is of no use if no seats are available with any airline. In the latter example, there exists a dependence between the actions of the mobile agent, in that they all need to succeed or none at all. This is not something that is covered by mechanisms that prevent blocking and ensure exactly-once execution, as elaborated above. It is an additional property, called *execution atomicity*, that must be ensured. This property is crucial for any mobile agents performing dependent operations. Execution atomicity, however, is not sufficient. Rather, the mobile agent execution needs to run as a transaction [GR93], and the mobile agent is called a *transactional mobile agent*. Again, non-blocking is an important property for transactional mobile agents and we show how it can be ensured by building transactional mobile agent execution on top of FATOMAS. More specifically, we reuse the approach for fault-tolerant mobile agent execution to ensure non-blocking transactional mobile agents. To validate this approach, we have implemented a prototype system called TRANSUMA (for TRANsaction SUpport for Mobile Agents) and evaluate its performance overhead relative to FATOMAS agents.

In the rest of this chapter, we will state the major contributions of this thesis and give a brief summary of the content of the subsequent chapters.

1.2 Contributions

One contribution of this thesis is the introduction of a classification of existing approaches in fault-tolerant and transactional mobile agent execution that aims at identifying the strengths and weaknesses of these approaches in an algorithmic sense.

The major contributions of this thesis, however, can be grouped into the domains of fault-tolerant mobile agent execution and transaction support for mobile agents.

1.2.1 Fault-Tolerant Mobile Agent Execution

Model This thesis presents a model for fault-tolerant mobile agent execution, which represents fault-tolerant mobile agent execution as a sequence of agreement problems. The model ensures non-blocking agent execution and enforces the exactly-once property without relying on correct failure detection. Contrary to other models such as [RS98, ASPZ98], the presented model is much simpler and as such easier to understand.

Algorithm The present algorithm provides fault-tolerant mobile agent execution corresponding to the stated model. It consists of two building blocks: consensus and reliable broadcast. More

specifically, the algorithm is based on a variant of consensus, called DIV consensus (consensus with deferred initial value) [DSS98]. This thesis shows how DIV consensus can be applied in the context of mobile agent execution. In contrast to other approaches [JMS+99, PPG00], our algorithm does not assume reliable failure detection.

Moreover, a discussion on the case of a replicated agent executing on a set of replicated places and thus invoking a replicated service is provided. This case is similar to the more general one of a replicated client invoking a replicated server in traditional client/server computing, which we call replicated invocation. We present a novel approach that achieves the exactly-once execution property in this context. This approach, at the same time, also solves the problem of exactly-once execution and one important part of the problem of non-blocking in the context of mobile agents.

System We introduce FATOMAS, a fault-tolerant mobile agent system, which implements the above algorithm. Contrary to existing work [JMS+99, RS98, ASPZ98], FATOMAS uses the agent-dependent approach to provide fault tolerant for mobile agents. With this approach, the fault tolerance mechanisms travel with the agent. Performance evaluations show that the overhead of FATOMAS compared to that of a non-fault-tolerant agent is reasonable.

1.2.2 Non-Blocking Transaction Support for Mobile Agents

This work shows how the present approach to fault-tolerant mobile agent execution can be used to achieve non-blocking transactional mobile agent execution. The model for non-blocking transactional mobile agents is based on closed nested transactions [Mos85] and encompasses both fault-tolerant and transactional mobile agent execution. A specification for non-blocking atomic commitment [GR93] in the context of transactional mobile agents is presented.

The system that implements the approach for non-blocking transactional mobile agents is called TRANSUMA (TRANsaction SUpport for Mobile Agents). It is built on top of FATOMAS and adds transactional support to the FATOMAS agents. To our knowledge, TRANSUMA is the first system to implement non-blocking transactional mobile agents.

Finally, we extend the model of non-blocking transactional mobile agent execution to open nested transactions [WS92].

A considerable amount of the research on mobile agents is also valid in the context of the client/server computing paradigm. This is also the case for the work presented in this thesis. In this sense, we try to establish the links to traditional client/server computing whenever possible and not to reinvent solutions already available in other computing paradigms.

1.3 Thesis Roadmap

The rest of the thesis' nine chapters have the following content:

Chapter 2 This chapter first gives the background required for the understanding of the thesis. It is divided into two parts: (1) mobile agents and (2) replication. Part (1) presents the mobile code computing paradigm and its advantages and briefly introduces the most important capabilities and services of the mobile agent platforms used for FATOMAS and

TRANSUMA. In part (2), we discuss replication and briefly summarize the most important replication techniques by indicating their advantages and drawbacks.

Chapter 3 This chapter presents the model concerning the system, mobile agents, and failures. An infrastructure failure occurs if a hardware or software component in the system crashes. In contrast, a semantic failure occurs if a service is not provided although no crash may have happened. For instance, requesting a seat reservation on a full airplane leads to a semantic failure. Finally, a distinction is made between transactional and non-transactional fault-tolerant mobile agents.

Chapter 4 This chapter specifies the properties for fault-tolerant mobile agent execution, i.e., *non-blocking* and *exactly-once* execution. Blocking occurs if a single infrastructure failure (e.g., the failure of the place where the agent currently executes) prevents the mobile agent from continuing its execution. Hence, a non-blocking mobile agent execution is able to proceed despite a single failure. The exactly-once property prevents multiple executions of the agent. Multiple executions are undesirable if the operations of the agent have side-effects. For instance, withdrawing money from a bank account is an operation with side-effects. We show why a violation of the exactly-once execution property can occur and introduce a basic building block for fault-tolerant and transactional mobile agent execution, called *local transaction*.

Chapter 5 This chapter presents a survey of the current state of the art in fault-tolerant and transactional mobile agents. For this purpose, we introduce a classification based on when and by whom redundant duplicate agents are identified and discarded, i.e., how a violation to the exactly-once execution property is resolved.

Chapter 6 This chapter is the core of the thesis and presents the approach to achieve fault-tolerant mobile agent execution. This approach is non-blocking and ensures that the agent is executed exactly-once. We have validated the approach with the help of a prototype system, called FATOMAS, whose architecture and implementation is discussed. Moreover, we give the performance evaluation results of FATOMAS.

Chapter 7 In Chapter 6, we examine fault-tolerant mobile agent execution in the context of so-called hetero-places. In this chapter, the discussion is extended to replicated and independent iso-places, where the places that execute the agent replicas are also replicas among themselves. Instead of immediately solving the problem of fault-tolerant agents executing on iso-places, we investigate the more general problem of a replicated client invoking a replicated server. Finally, we discuss how this approach can be adapted to solve the problem of iso-places.

Chapter 8 In this chapter, we show how the approach to fault-tolerant mobile agent execution can also be used to achieve non-blocking transactional mobile agent execution. We first give the specification of non-blocking atomic commitment in the context of transactional mobile agents, before discussing the algorithm and presenting the prototype system, called TRANSUMA. We then evaluate the overhead of TRANSUMA and compare it to FATOMAS.

Chapter 9 This chapter concludes the thesis and identifies areas for future research.

Chapter 2

Background: Mobile Agents and Replication

The thesis combines elements of two research fields: mobile agents and replication. In this chapter, we set the ground for the rest of the thesis by giving the background on these two research fields.

2.1 Mobile Agents

We begin with the definition of a mobile agent (Section 2.1.1) and give a brief history on mobile code (Section 2.1.2). In Section 2.1.3, we present different variants of the mobile code computing paradigm. Sections 2.1.4 and 2.1.5 argue on the usefulness of mobile agent technology by pointing out some advantages and application areas for mobile agents. Then, we briefly discuss the most important standards in Section 2.1.6. Finally, Section 2.1.7 presents the mobile agent platforms used for our prototypes: Voyager and MOPROS.

2.1.1 Definition

So far, the research community has not agreed on a common definition for mobile agents. Various sets of required capacities for mobile agents have been devised, but none of these sets is commonly accepted as a minimal set to define a mobile agent. It is not the goal of the thesis to provide a universally accepted agent definition. Rather, we adopt a definition that is adequate for our purpose and that integrates the definitions given in [Mat98, Obj00]:

Definition 1 (Mobile Agent)*: A mobile agent is a computer program that acts autonomously on behalf of a user and travels through a network of heterogeneous machines.*

The term "autonomously" thereby means that we require the agent to exploit a certain degree of autonomy. For instance, the agent can decide itself on the itinerary it takes, only based on its current state.

Figure 2.1 illustrates an example agent execution spanning 4 machines (i.e., Machines 1 to 3 and again Machine 1). Here, the agent is configured and launched by the agent owner on Machine 1

(see Figure 2.1 (1)), travels to Machine 2 and accesses Server[1] 1 (see (1) and (2), respectively). After that, the agent travels to Machine 3, where it accesses Server 2 (see (3) and (4)). Finally, the agent returns to Machine 1 and presents the result to the agent owner (see (5)). Note that the agent execution can terminate on any machine, although in most applications it will generally return to the machine it was configured and launched.

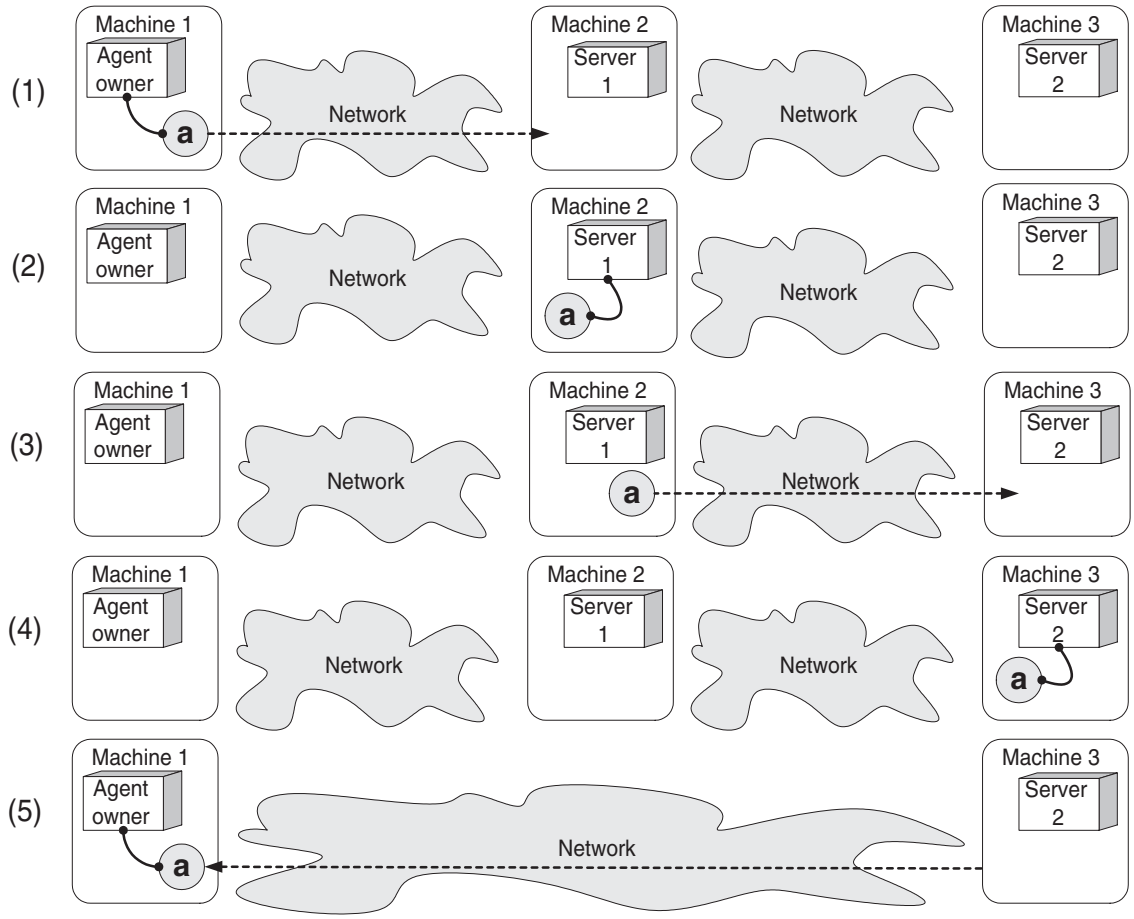

Figure 2.1: Mobile agent execution with three hops (i.e, from Machine 1 to Machine 2 to Machine 3 and back to Machine 1). The arrows denote the moving of the agent, while lines ending with dots show an interaction that takes place.

2.1.2 Brief History

The idea of sending programs to and executing them at a remote site has been explored for some time. The Postscript language can be considered a rudimentary form of this idea, as it involves

[1]By "server", we do not mean the entire machine, but the process executing on Machine 2 that provides a particular service.

sending programs to a remote processor in a printer. In 1990, General Magic launched the first commercially available mobile agent platform called Telescript [Gen95] and obtained a patent[2] for its mobile agent technology in 1997. However, Telescript had never been a commercial success, partly because it was not open to the public, and was soon abandoned. In the past few years, several mobile agent platforms such as Agent Tcl [Gra96], Aglets [Agl], Mole [SBH97], Tacoma [Tac], and Voyager [Obj99] have emerged.[3]

Although some mobile agent platforms have been based on compiled languages such as C [Tac] most of the existing platforms use interpreted languages such as Tcl, Perl, or Java. Interpreted languages have the advantage that they are highly portable and that the mobile agent system can to some extent influence the execution of the agent code. In particular, system calls built dynamically during execution can be verified and illegal instructions potentially rejected. This is generally not possible with binary code format. Platform-independent programming languages such as Java (more specifically the Java byte code) are favored in heterogeneous execution environments.

Recently, the influence of Java has grown rapidly and most of today's newly created mobile agent platforms are written in this language. Java provides a large set of standard libraries and tools that solve certain problems related to mobile agents such as serialization, remote method invocation, and the class-loading mechanism, which facilitates the migration of code [WPM99]. It is also straightforward to extend and modify the behavior of the virtual machine, for example using the security manager or the class loading facility.

2.1.3 Mobile Code Paradigm

The client/server computing paradigm is today's most prominent paradigm in distributed computing. In this computing paradigm, the server is defined as a computational entity that provides some services. The client requests the execution of these services by interacting with the server. Having executed the service, the server delivers the result back to the client. The server therefore provides the knowledge of how to handle the request as well as the required resources. The computing paradigm of mobile code generalizes this concept by performing changes along two orthogonal axes:

1. Where is the know-how of the service located?
2. Who provides the computational resources?

Depending on the choices made on the client and server sides, the following variants of mobile code computing paradigms, illustrated in Table 2.1, can be identified [FPV98]:

Remote Evaluation (REV). In the REV paradigm, component A (e.g., a process or object) sends instructions specifying how to perform a service to component B (represented by *code* in Table 2.1). These instructions can, for instance, be expressed in Java byte code. Component

[2] US patent no. 5603031: System and Method for Distributed Computation Based upon the Movement, Execution, and Interaction of Processes in a Network.

[3] See http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole/mal/mal.html for a list of mobile agent platforms.

	before the invocation		after the invocation	
	machine 1	machine 2	machine 1	machine 2
Client/Server	A	code,resource,B	A	code,resource,B
Remote Evaluation	code,A	resource,B	A	*code*,resource,B
Code on Demand	resource,A	code,B	*code*,resource, A	B
Mobile Agent	code,A	resource, B		*code*,resource,*A*, B

Table 2.1: Different variants of the mobile code computing paradigm [FPV98]. Code or computational entity transported between machines are indicated by italics. Component A accesses the services provided by component B.

B then executes the request using its own resources, and returns the result, if any, to A. Java Servlets[4] are an example of remote evaluation [Cow01].

Code on Demand (CoD). In the CoD paradigm, the resources are collocated with component A, but A lacks the knowledge of how to access and process these resources in order to obtain the desired result. Rather, it gets this information from component B (represented by *code* in Table 2.1). As soon as A has the necessary know-how (i.e., has downloaded the code from B), it can start executing. Java applets[5] fall under this variant of the mobile code paradigm.

Mobile Agent. The mobile agent computing paradigm is an extension of the REV paradigm. Whereas the latter focuses primarily on the transfer of code, the mobile agent paradigm involves the mobility of an entire computational entity, along with its code, the state, and potentially the resources required to perform the task. As developer-transparent capturing and transfer of the execution state (i.e., runtime state, program counter, and frame stacks, if applicable) requires global state models as well as functions to externalize and internalize the agent state, only few systems (e.g., Agent Tcl [Gra96], Telescript [Gen95]) support this *strong mobility* scheme. In particular, Java-based mobile agent platforms are generally unsuitable for this approach, because it is not possible to access an agent's execution stack without modifying the Java Virtual Machine.

Most systems thus settle for the *weak mobility* scheme where only the data state is transferred along with the code. Although it does not implicitly transport the execution state of the agent, the developer can explicitly store the execution state of the agent in its member attributes. The values of these member attributes are transported to the next machine. The responsibility for handling the execution state of an agent thereby resides with the developer. In contrary to REV, mobile agents can move to a sequence of machines, i.e., can make multiple hops.

The next section elaborates on the advantages of the mobile agent computing paradigm over the traditional client/server paradigm.

[4] See Sun's Java servlet homepage http://java.sun.com/products/servlet.
[5] See Sun's Java applet homepage http://java.sun.com/applets/index.html

2.1.4 Advantages of the Mobile Agent Paradigm

At the moment it is not yet clear whether mobile agent technology will establish itself as an independent computing paradigm for practical use in the long run. Chess et al. [CHK98] made an attempt to estimate the benefit of mobile agent technology. They concluded that all the problems considered as good examples for the use of mobile agents can also be solved using traditional client/server solutions. However, mobile agents allow a general solution to all such problems. Instead of having to create different, well-tailored solutions to every problem, mobile agents provide a generic solution to all these problems. The mobile agent computing paradigm has several advantages over the traditional client/server model [LO99, WPM99, CHK98]:

Communication latency and bandwidth. If the communication between two interacting components (e.g. a client and a server) involves a considerable amount of data, it may be beneficial to move the client close to the server instead of moving the data between them. The locality of the client and the server allows them to decrease the *latency* and save *bandwidth* in the communication, especially if the mobile agents act as filters and process only the data that is really useful to the agent owner. Clearly the gain in latency and bandwidth must overcome the cost of sending the agent to the server node. Fuggetta et al. introduce a model for performance measurements in [FPV98]. They state that in the client/server paradigm, the overhead of the interaction depends on the size of the database accessed by the application. On the other hand, the overhead of the mobile agent paradigm is bound to the size of the code and the data sent by the application. This leads to the conclusion that the application of the mobile agent paradigm is only justified when the database size has surpassed a given threshold.

To exploit locality it is necessary for the developer of mobile agent applications to associate the notion of locality with every server. Based on the locality information, the agent may decide to move to the server location instead of invoking the server functions remotely. This is a contradiction to the general programming model in client/server distributed systems. In such a model the client does not have to know where the server is located. This is taken care of by the communication middleware (e.g., CORBA [OMG98a], Tuxedo [ACDF96]), which transparently forwards the message to the machine that runs the server.

Off-line processing. In *mobile computing*, roaming devices such as Personal Digital Assistants (PDAs) or laptops are often disconnected from the fixed network. In addition, the emission of messages from the mobile device over a wireless communication link is expensive in terms of power consumption. Connections to the fixed network may also incur considerable financial costs. This presents an opportunity for mobile agent technology because the mobile device delegates an agent to act on its behalf and to perform the required actions. Hence, instead of sequentially invoking every server $Serv_0, Serv_1, \ldots, Serv_N$, the PDA sends the mobile agent to $Serv_0$ and eventually retrieves the agent again from $Serv_N$. This involves two accesses to fixed machines, whereas the client/server computing paradigm requires N. The next time the mobile device connects to the network it can collect the results. Such a mobile agent can be used not only to query servers but also to act as a filter that preconsolidates messages sent to the mobile device.

Reaction time. Having mobile components allows an application to exploit locality. *Reaction time* is less significant for a local agent acting on behalf of a remote monitoring device than for the device itself. In addition resources may under certain circumstances only be accessible locally. Agents are a concept to provide external service in this situation.

Asynchronous behavior by request aggregation. Instead of interacting with a server in many RPC-style communications, a client can bundle or aggregate the requests within a mobile agent. Having reached the server the agent starts interacting with the services locally. This also simplifies the recovery from communication failures, because either the entire agent, i.e., all the requests, arrive at the server or none at all. For instance, in space communication it is often appropriate, due to the communication latency, to use this type of interaction.

Dynamic adaptation. Mobile agents have the ability to adapt dynamically to changes in their environment. They can, for instance, react autonomously to balance the load in the network or move on to a replica of a current node that is failing.

Protocol encapsulation. Today's networks consist of many legacy applications. As their protocols evolve, legacy problems often occur. Mobile agents move to the remote legacy application and encapsulate its protocol. Other applications communicate with this application via an agent, using a proprietary protocol.

These advantages make mobile agents a suitable and beneficial technology for various application domains. We elaborate on this in the next section.

2.1.5 Application Domains for Mobile Agents

Whereas the potential usefulness of the mobile agent computing paradigm has been widely accepted, the mobile agent technology has not yet found its way into today's more prominent applications. The following use cases may represent domains where mobile agent technology can make an impact and an important qualitative difference [LO99, CHK98], and has been used to a certain extent:

e-commerce. The use of mobile agent technology has been proposed to provide some of the services of e-commerce. More specifically, agents are sent out into the Internet by a user and gather the required information. If necessary they can collaborate with other agents, engaging in information sharing, exchanging, or buying [MGM99]. Schemes are devised to create marketplaces where agents can deal and even take part in auctions. For this purpose an agent is equipped with the mechanisms to deliver payment for purchased goods or services on behalf of its authority.

Watchdog applications. An agent monitors a component, such as a device driver, an application, or a switch. It is able to react locally to a certain behavioral pattern of the monitored component. For instance a network management agent could monitor the network traffic until it detects traffic congestion. It then sends an email to the administrator or autonomously takes appropriate corrective actions.

Itinerant actions. Transactions may involve several nodes. If the nodes have to be visited sequentially rather than in parallel, an agent could be an alternative to performing a client/server type of call to every host. This application area is related to e-commerce, e.g., comparison shopping [DEW97, Kar00].

Information gathering. Information gathering can be left to an agent's responsibility if multiple sources have to be considered or if the sources are not exactly known beforehead. An agent might accumulate some knowledge during its itinerary that allows it to make a decision about its future itinerary.

System configuration. Agents can provide a more flexible mechanism for system configuration. In particular, the dynamic dispatching of mobile code allows reconfiguration without shutting down the whole system. Active networks [TW96], for instance, rely on instructions sent with the communication packets. Dynamic system maintenance and software updates are also supported by this mechanism.

Parallel processing. Mobile agents can clone themselves and split up the work among the clones. This allows tasks to execute in parallel and distribute processing power among different nodes. An application based on mobile agent technology is thus also more easily scalable than applications consisting of monolithic blocks.

Before mobile agent applications begin to appear on a large scale, however, the mobile agent platforms need to provide the infrastructure services to facilitate agent development. Among these are security, management of agents, fault tolerance, and transaction support. Fault tolerance and transaction support are the focus of this thesis. Moreover, aspects of agent technology have to be standardized to allow different agent systems to interoperate. Although standardization efforts are being devised or are already stable, they are not yet widely adopted.

2.1.6 Standards

Because of their relatively recent nature, today's mobile agent systems differ widely in architecture and implementation. These differences prevent interoperability and rapid development of agent technology. To promote interoperability yet still permit system diversity, the Object Management Group (OMG) [OMGb] has defined a standard for mobile agent technologies called the *Mobile Agent System Interoperability Framework* (MASIF or MAF) [Obj00, MBB+98]. This standard does not consider programming language interoperability but only interoperability between agent systems written in the same programming language by potentially different vendors. It relies on basic CORBA facilities such as naming service, life cycle service, etc. [OMG98b]. The complexity of the MASIF standard has been criticized as well as its focus on Java and its proximity to the IBM Aglets [Agl, LO98] implementation. Tham et al. [TFWR97] claim that internal APIs have been too extensively specified and that the standard is weak in terms of interoperability and conformance specification.

Besides OMG, the Foundation for Intelligent Physical Agents (FIPA) [Fip] has also standardized aspects of mobile agent technology. This standard aims at maximizing the potential interoperability between different agent platforms. It addresses a variety of domains such as *agent management*, *agent security*, and *agent mobility*.

2.1.7 Mobile Agent Platforms

In this section, we present the mobile agent platforms relevant to the thesis: Voyager, the primary choice, and MOPROS.

Voyager

Our chosen platform is ObjectSpace's Voyager v3.1 mobile agent platform [Obj99]. Voyager is entirely written in the Java programming language and integrates fundamental distributed computing with agent technology. It considers a mobile agent as a special kind of object, that simply has an additional property, namely mobility. Mobile agents can move between Voyager servers. Upon arrival on the destination, the method passed as argument to the move command is invoked and the mobile agent resumes its execution. Being a Java mobile agent platform, Voyager supports weak mobility, i.e., the user needs to explicitly save all the state to be transferred into local variables of the object/agent. Voyager uses standard Java serialization for the transfer of the agent and its state. At the destination Voyager server, the state is regenerated and can again be accessed by the mobile agent.

Voyager allows to send a Java message to a stationary or mobile agent, i.e., supports inter-agent communication. To ensure that a message reaches a moving agent, Voyager leaves *forwarders* at the former Voyager server, which then route the messages to the agent's new destination. Different modes of communication are supported:

- *(Oneway)* The message is sent and forgotten. The sender is not informed if the message does not arrive at its destination.
- *(Synchronous)* The method invocation returns an exception if the destination is not available. The sender can thus retry the method invocation at a later time. It is blocked until it receives the result of the invocation, unless an exception is raised.
- *(Asynchronous)* The method invocation returns an exception if the destination is not available. However, the sender does not block while the result of the invocation is computed. Rather, the result is available at a later point in time and can then be collected by the sender.

Voyager provides a basic directory service (or repository), which runs locally on each Voyager server, but can be remotely accessed. Our implementation makes use of this directory server; however, this is just a simplification and the same functionality could be provided by the use of agents.

MOPROS

MOPROS (MObile PROcesseS) is a proprietary experimental mobile agent platform that has been implemented in the IBM Zurich Research Laboratory to study the problem of resource control in the context of Java-based mobile agents. Its early stage of development and its focus on resource control are manifest in its not yet optimized performance. Despite the performance issues, the possibility of discussions with MOPROS' developer and the availability of the source code have made

MOPROS an ideal platform to port our prototype to. Unfortunately, MOPROS is not publically available and its development has been discontinued in the meantime.

The notion of a process is the base entity in MOPROS. When launched, a process starts executing in its `main` method. Using a particular move command, the process (i.e., the agent) can be moved to another Java Virtual Machine (JVM). Before moving the agent, the developer must explicitly store the state to be transferred with the agent. By default, MOPROS transfers the state and the code of the agent, i.e., only supports weak mobility. Upon arrival at the destination JVM, the agent resumes the execution again by restarting the `main` method. Inter-agent communication in MOPROS is based on Java Remote Method Invocation (RMI) [Sun], which is Java's distributed object model.

2.1.8 Summary

We have presented a brief overview on mobile agent technology. Today few large-scale applications are based on mobile agents. This is partly because no "killer" application exists that would extend the use of mobile agents into many computation areas. However, in some domains the influence of mobile agent technology is growing. In active networks, for instance, packets carry code which is executed in the switches and allows the switches to be configured dynamically. Another application domain can be found in mobile computing. Here, the agents act on behalf of mobile components and therefore minimize the connection time between the base station and the mobile device. Mobile agent technology is also believed to have a significant impact on e-commerce. Although the example applications used in the thesis are taken from the application domain of e-commerce, the presented approaches are not limited to e-commerce and are also applicable to the other domains.

2.2 Replication

This thesis discusses fault tolerance for mobile agents. Fault tolerance can only be achieved through some form of replication. We discuss now the relevant replication mechanisms.

Fault tolerance aims at (1) transforming an unpredictable or undesired behavior into a predictable behavior, or (2) masking such an unpredictable or undesired behavior. Item (1) is usually achieved through the use of atomic transactions [GR93], while (2) generally uses replication. Indeed, fault tolerance can only be achieved by introducing redundancy into a system. While the use of redundancy is obvious in (2), it is somewhat hidden in (1). For instance, the implementation of atomic transactions often relies on redundant information, e.g., stored in log files.

Various forms of redundancy exist, such as data or execution redundancy. With data redundancy, copies of the data are stored in different locations. If one copy becomes corrupted or is lost, another copy can be accessed. As an example, consider the backup of a file system. If the disk fails and the file system becomes unaccessible, then the backup is used to restore files and directories. Execution redundancy, in contrast, duplicates the execution of a program. In case of a failure of one replica of the program, the other(s) may still succeed. Examples can be found in astronautical applications, where often hardware is replicated as well and the programs that

are run on them [SG84].[6] In distributed computing, often the server is replicated to achieve fault tolerance. In the following, we discuss replication of the server in more detail and briefly present existing replication techniques that allow to maintain consistency among the replicas.

2.2.1 Replicating the Server

Let T denote a server and C a client that sends requests to T. Server T executes the client requests and sends the result back to C. If T fails, it is not able to handle requests from C any more until it recovers. During this time, the service provided by T is not available and the execution at C is blocked. Hence, T is a *single point of failure*. Replication introduces redundancy and thus masks the failure of a physical server T from the client. Instead of accessing T, the client accesses a replica group composed of the *server replicas*[7] $T^0, T^1, T^2, \ldots$ (see Figure 2.2), that generally are located on different physical machines. These replicas collaborate in order to execute the client request and send the result back to the client. In this context, it is generally assumed that crashed server replicas do not recover any more; rather, they may join the group again later, with a new identity (i.e., they behave like new replicas).

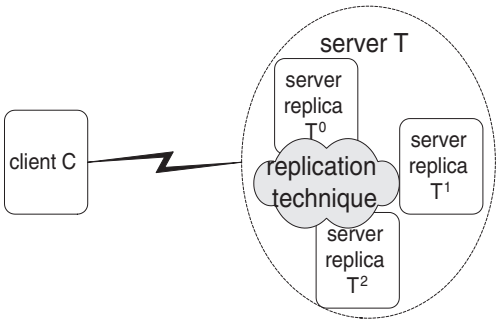

Figure 2.2: Interaction of a client with a replicated server.

2.2.2 Replica Consistency

Consistency is an important issue with replication. Indeed, the access of a client to the replica group needs to provide a result that is consistent with the specification of the service. Generally, the replicas are required to be *strictly consistent*, i.e., behave exactly as a single copy would. Assume, for instance, that client C increments data item x on replica T^1. Client D, on the other hand, multiplies the value of data item x on T^2 by "3". The replicas forward the operations to the other replicas to maintain consistency. However, delayed messages may cause inconsistencies in the value of x on the replicas. Whereas x has the value "6" on T^1 and T^2, x has the value "4" on T^3 (Figure 2.3).

[6] Moreover, different versions of a program on different hardware may be executed to prevent that, for instance, the same fault occurs on all replicas [Avi95].

[7] Throughout the thesis, we use *superscripted* indices to distinguish among replicas. Subscripted indices have a different meaning (e.g., see Section 3.2).

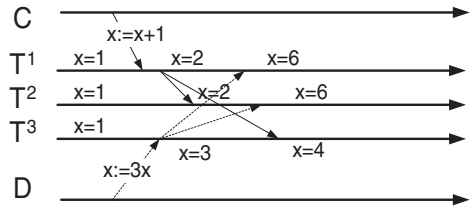

Figure 2.3: Violation of strict consistency.

To maintain consistency among the replicas, replication techniques such as *active*, *passive*, *semi-active*, and *semi-passive* replication have been proposed. In Section 2.2.3, we briefly summarize these replication techniques.

2.2.3 Replication Techniques

In this section, we briefly outline the most popular replication techniques, i.e., active and passive replication, which allow to maintain consistency among the server replicas. Moreover, we present semi-passive replication, which will be instrumental in the context of fault-tolerant mobile agents. Semi-active replication is only briefly mentioned; the reader is referred to [BHB+90, Pow91] for an in-depth discussion.

Active Replication

With active replication, also called state-machine replication [Lam78, Sch93], the client request rq is executed on all replicas of the group. Every replica returns the result to the client; the client thus receives a reply from every server replica T^i (see Figure 2.4). It keeps the first reply and discards the other replies. Active replication has two limitations: the processing redundancy leads to increased resource usage, and the need for deterministic execution of the client request. Indeed, the result of rq must only depend on the initial state of a replica and on the sequence of operations within rq. Multiple threads, for instance, generally lead to non-determinism, as the thread scheduling is not deterministic. We discuss the problem of determinism in more detail in Section 7.1. Moreover, all replicas need to execute the requests in the same order to maintain consistency. This order is ensured by a communication primitive called *total order broadcast* or *atomic broadcast* [Sch93, CT96, CM84].

Passive Replication

In contrast to active replication, in passive replication (also called *primary-backup* [BMST93]) only one replica, the *primary*, executes the client request. The update is then sent to the backup replicas T^1 and T^2 and to the client (see Figure 2.5). The backup replicas do not directly communicate with the client; rather, they only communicate with the primary. As only the primary executes the client request, deterministic execution is not needed. However, the passive replication technique needs to handle failures of the primary.

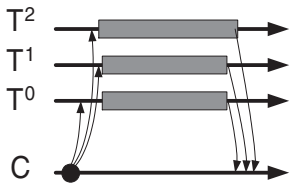

Figure 2.4: With active replication, the client request is *atomically broadcasted* (represented by the dot) to all server replicas T^i, which all execute the request and return the result.

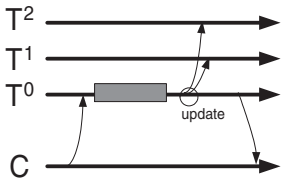

Figure 2.5: In passive replication, the client request is executed by the primary and the result is reliably sent (represented by the circle) to the backups and to the client.

For this purpose, passive replication relies on the notion of *group membership*. The group membership evolves in a sequence of views [BJ87, MS95, GS96] (see Figure 2.6). Every member has the same view and agrees on messages delivered within this view. We say that the primary *vs-casts* the updates to the backups, which *vs-deliver* them. If the primary fails, a new view is installed, in which one of the backup replicas assumes the role of the primary. Compared to active replication, the failure of the primary leads to an increased response time [SM96]. Indeed, if the primary fails, client C eventually time-outs. It then has to learn the identity of the new primary (e.g., T^1) and reissues its request. Hence, failure transparency from the perspective of C is not entirely achieved by passive replication.

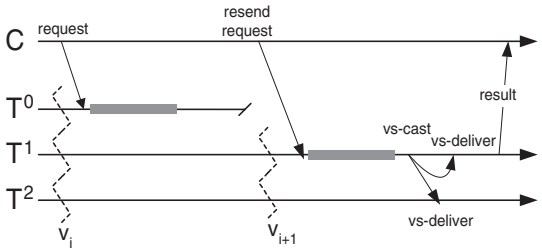

Figure 2.6: The group membership evolves in the sequence of views v_i and v_{i+1}. View v_{i+1} excludes the failed primary T^0.

Semi-Passive Replication

Similarly to passive replication, semi-passive replication [DSS98, Déf00, DS00], processes the client request on one server replica (called the *primary*), if no failures occur, and then updates the backup replicas (see Figure 2.7). As the request is handled by a single server, determinism of the server replicas is not required. In contrary to passive replication, semi-passive replication does not rely on view synchrony. Rather, the selection of the primary is based on the *rotating coordinator paradigm* [CT96] of the underlying consensus problem. The particular consensus problem defined in semi-passive replication is called *Consensus with Deferred Initial Values*[8] (DIV consensus) [DSS98, Déf00, DS00]. The decision of DIV consensus is the update value obtained by processing the client request and is eventually known to all T^j. As the client request is sent to and replies are received from all replicas, failure transparency from the perspective of the client is achieved.

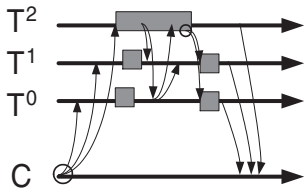

Figure 2.7: Semi-passive replication is similar to passive replication, but does not rely on view synchrony. Client C reliably sends its request to T^0, T^1, and T^2 (denoted by the circle at the message emission).

DIV Consensus: The classical consensus problem [CT96] is defined such that every process p^i (see Figure 2.8) starts with an initial value v_i, and the processes agree on a common decision value v. The participants start consensus by calling the function `propose(`v_i`)` with their initial value passed as argument. In the DIV consensus algorithm, on the other hand, a process is not required to provide its initial value when starting the consensus algorithm. Rather it can compute this value later when needed in the consensus algorithm. Instead of passing the initial value to `propose`, the participants pass a handler $\mathcal{GIV}()$ (stands for "Get Initial Value") to compute the initial value. If no failure occurs, the decision is reached by p^0 within 2 rounds of messages (see Figure 2.8 (a)). This decision is then reliably broadcasted to the other participants, i.e., p^1, p^2. DIV consensus evolves in a sequence of phases, starting in the second phase of round 1. In this phase, p^0 computes its initial value and sends it to all other participants, which either send an ACK or a NACK back in phase 3. If p^0 manages to collect a majority of ACKs, it can decide and reliably broadcasts this decision in phase 4. Figure 2.8 (b) depicts the scenario in which p^0 fails after having executed $\mathcal{GIV}()$. The other participants detect the failure of p^0, and p^1 takes over the role of p^0.

[8] In [Déf00, DS00], the name has changed to *Lazy Consensus*. In this work, we use the name originally given in [DSS98], i.e., *Consensus with Deferred Initial Value (DIV consensus)*.

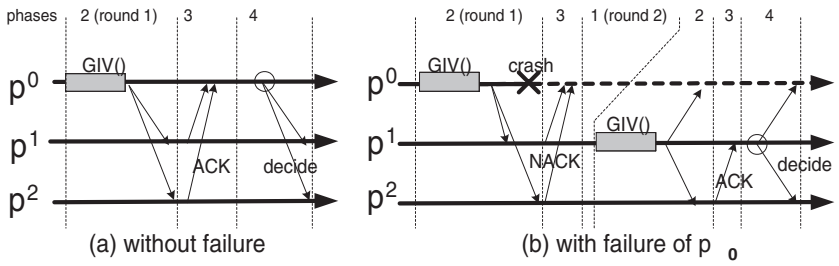

Figure 2.8: DIV consensus (a) without failure and (b) with the crash of p^0. The decision message is reliably broadcasted (denoted by a dotted circle at the message emission).

Semi-Active Replication

Semi-active replication [BHB+90, Pow91] attempts to combine the advantages of passive replication (i.e., handling of non-determinism) and active replication (fast response time despite failures, i.e., fast fail-over). One of the replicas is the *leader* and the others are called *followers*. Semi-active replication splits the processing of the replicas into deterministic and non-deterministic parts. While the deterministic parts are executed by all replicas, only the leader executes the non-deterministic parts and informs the followers. However, [Pow91] assumes a synchronous system model, i.e., relative process speeds and communication delays are bounded. This synchrony assumption guarantees that the leader is always slightly ahead in the processing compared to the followers. The approach in [Pow91] may be adapted to work also in an asynchronous system model; however, this is beyond the scope of this thesis. Hence, semi-active replication is not considered any further.

2.2.4 Summary

In this section, we have presented an introduction to replication. In the context of replication, replication techniques address the problem of replica consistency. While active replication masks failures from the client and provides fast service despite failures, it requires deterministic execution. This limitation is avoided with passive and semi-passive replication. However, failures lead to increased response time with these techniques. Passive and semi-passive replication are distinct in the way they select a new primary after the failure of the previous one. Semi-passive replication, or rather DIV consensus, is an important building block for fault-tolerant mobile agent execution (see Chapter 6). Passive and active replication are used in Chapter 7, where we discuss replicated agents executing on replicated machines.

Chapter 3

Model

In this chapter, we present the model we use throughout the thesis. More specifically, Section 3.1 gives the general system model. The model of a mobile agent is presented in Section 3.2. Failures may disrupt a mobile agent execution and Section 3.3 thus presents the failures we are addressing. Finally, Section 3.4 distinguishes between transactional and non-transactional fault-tolerant mobile agents.

3.1 System

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

3.2 Mobile Agent

A mobile agent executes on a sequence of machines (see Figure 3.1), where a *place*[1] p_i $(0 \leq i \leq n)$ provides the logical execution environment for the agent. Executing the agent at a place p_i is called a *stage* S_i of the agent execution. We call the places where the first and last stages of an agent execute (i.e., p_0 and p_n) the agent *source* and *destination*, respectively. The sequence of places $p_0, p_1, ..., p_n$ 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 3.2). 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 $p_0, \ldots, p_{i-1}$ 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 the agent have side effects).[2] Note that the agent generally does not access the state of the place

[1] Also called *landing pad* in [JMS+99], *agency* [SBS00], or *computational environment* [FPV98].

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

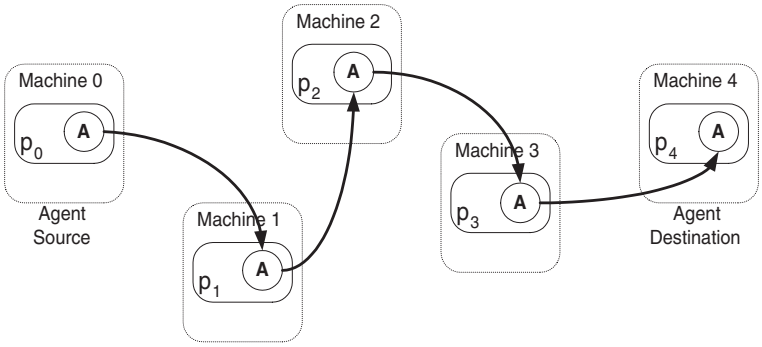

Figure 3.1: Example of the execution of an agent A.

directly. Rather, each place runs a set of services, which together define its state. For simplicity, we say that the agent "accesses the state of the place", although this access occurs through a service running on the place. Assume, for instance, that the place runs a service that sells airline tickets. The mobile agent accesses this service to buy a ticket and thus at the same time modifies the state of the place (that comprises the state of all the services running on it). We denote the resulting agent a_{i+1}. Place p_i forwards a_{i+1} to p_{i+1} (for $i < n$).

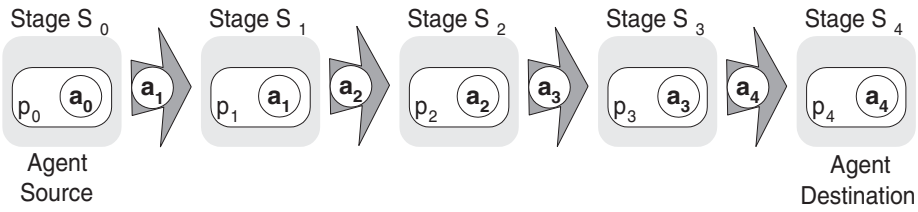

Figure 3.2: Model of a mobile agent execution with four stages.

In this thesis, we consider mobile agents that do not receive messages from other agents or applications, which modify the state of the agent. Actually, the approach presented in the thesis can generally also handle these mobile agents, but limiting the scope of the thesis in this way considerably simplifies the presentation of our approach.

3.3 Failures

3.3.1 Infrastructure Failures

Machines, places, or agents can fail and can recover later. A component (machine, place, or agent) that has failed but not yet recovered is called *down*, whereas it is *up* otherwise. We assume that

place.

components are eventually up forever[3], i.e., are *good* [ACT00]. In contrary, components that are eventually down forever or always alternate between up and down are called *bad*. The assumption of good components is only needed to achieve consistency on the level of the application that has launched the mobile agent. For instance, crashed databases accessed by the mobile agent need to properly terminate the transactions that have not yet committed or aborted. For this purpose, they need to eventually recover. A component is *correct*, if it does not crash during the execution of the agent at a particular stage. A place that crashes at stage S_i can recover and again participate in the execution of the agent at a later stage S_k $(k > i)$. In this thesis, we focus on crash failures (i.e., processes prematurely halt); malicious failures (i.e., Byzantine failures) are not discussed, with the exception of a brief reference to [Sch97] in Section 5.2.1. The crash of a machine causes any place and any agent running on this machine to crash as well (Figure 3.3 (d)). A crashing place causes the crash of any agent on this place, but this generally does not affect the machine (Figure 3.3 (c)). Similarly, a place and the machine survive the crash of an agent (Figure 3.3 (b)). 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 as relevant failures in this context.[4] Failures of machines, places, and agents are called *infrastructure failures*. In the following, when we speak about failures, we mean infrastructure failures, unless explicitly stated otherwise.

Communication links do not create or duplicate messages, but they can drop single messages. However, if a component c sends a message to another good component d an infinite number of times, then d eventually receives messages from c an infinite number of times [ACT00]. Note that a communication link that drops messages can lead to a partitioning of the network. With a partitioned network, only components in the same network partition can communicate among themselves. Communication with components in other partitions is only possible if the partitions are merged again, i.e., when the communication link stops dropping messages. In this respect, communication links that drop messages are equivalent to communication links that can fail and eventually recover. However, the distinction between communication links that fail and communication links that drop messages becomes important when we discuss blocking in Section 4.1. Blocking occurs if a failed component prevents progress in the mobile agent execution. Using this definition, a link that fails may cause blocking, however, we do not speak of blocking in the case of a link that drops messages. Reasoning about blocking in this context requires to take into account also the structure of the underlying network. Indeed, a failure of a link may not cause a partition at all (if redundant links exist), or may lead to a singleton partition (a partition with one component), or, in the worst case, partition the entire network into 2 partitions. As an in-depth discussion of these issues is outside of the scope of the thesis, our definition of blocking only addresses failures of agents, places, and machines. By allowing communication links to drop messages, we take partitioning into account in our system model. Indeed, the approach presented in the thesis can handle network partitions (i.e., allows progress in the execution) if a majority of components participating in an algorithm is in the same partition. If no such partition exists, then our approach must wait until partitions are merged such that a majority of components is again in the same partition.

[3] Actually, components only need to be up sufficiently long to finish their tasks.

[4] Johansen et al. [JMS+99] 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.

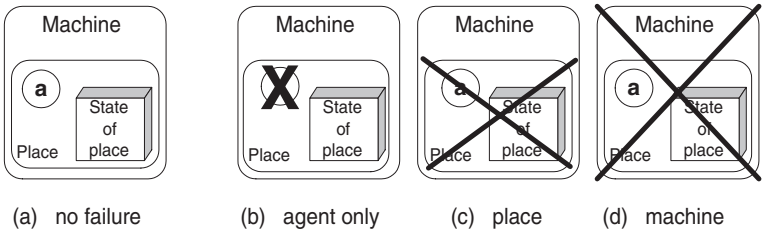

Figure 3.3: Failure model for mobile agents.

The detection of infrastructure failures is generally encapsulated into a failure detector module [CT96]. Failure detectors are defined in terms of completeness and accuracy properties. Completeness requires that failed processes are eventually suspected, while accuracy limits the number of false suspicions, i.e., processes that are suspected but have not crashed. In [CT96], Chandra and Toueg 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. Moreover, the perfect failure detector is defined in a model without recovery. A model with recovery is presented in [ACT00], which distinguishes between good and bad processes, where good processes are eventually up forever. A perfect failure detector would require to know the future. Hence, assuming unreliable failure detectors is a more realistic assumption.

The approach to fault-tolerant and non-blocking transactional mobile agent execution presented in this thesis only requires an unreliable failure detector of class $\diamond S$ [CT96], which provides *strong completeness* and *eventual weak accuracy*. Strong completeness ensures, that all crashed components are suspected, while eventual weak accuracy guarantees that eventually some correct component is not suspected any more.

3.3.2 Semantic Failures

A *semantic failure* is different from an infrastructure failure in the sense that neither machine, place, nor agent initiating the request crash. Rather, it occurs when a requested service is not delivered because of the application logic or because of service failure. 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, i.e., the request for a ticket, executes in its entirety. 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 a "semantic failure".

3.4 Transactional *vs.* Non-Transactional 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 and semantic failures. Assume, for instance, that stage action sa_i books a flight, while sa_j reserves a hotel room at the flight destination. Clearly,

the use of the hotel room is limited if no seat is available any more on any flight to the destination. Hence, sa_i and sa_j need to execute atomically, i.e., we want both to succeed. If one of the two semantically fails (e.g., because no seats are available) then the other one needs 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 semantic failures. 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 non-transactional fault-tolerant mobile agent executions can be implemented using transactions (e.g., [ASPZ98, RS98]). However, the use of transactions still does not ensure atomicity in the entire mobile agent execution.

In the following, we first focus on non-transactional mobile agents, i.e., general fault-tolerant mobile agent execution (see Chapter 6 and Chapter 7). When we refer to "fault-tolerant mobile agents", we mean non-transactional, fault-tolerant mobile agents, unless explicitly stated otherwise. Transactional mobile agents are discussed in detail in Chapter 8. A brief discussion on transactional mobile agents is already given in the survey of the current state of the art (see Section 5).

Chapter 4

The Problem of Fault-Tolerant and Transactional Mobile Agent Execution

In this chapter we specify fault-tolerant mobile agent execution in terms of two properties: non-blocking and exactly-once execution. We begin with the definition of a blocking mobile agent and show how replication can overcome blocking. However, replication may lead to multiple executions of the agent. Hence, a fault-tolerant mobile agent execution must be non-blocking and execute exactly-once. For this purpose, we identify the notion of *local transaction* as a basic building block for fault-tolerant as well as transactional mobile agent execution.

4.1 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 (see Figure 4.1). 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 [BHG87, GR93], the execution 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, or agent) recovers. In contrast, a *non-blocking* mobile agent execution can continue the execution despite a single failure. 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 slow response times and may be unacceptable for the agent owner. Hence, mobile agent executions are preferably non-blocking. Note that semantic failures do not lead to blocking in the mobile agent execution.

We use a weak definition of blocking, because any approach only needs to tolerate a single failure to achieve non-blocking. This definition is appealing because of its simplicity. Moreover, any approach able to cope with any single failure is generally already sophisticated enough to address (or can easily be extended to handle) also multiple failures in the context of mobile agents. Indeed, the approach presented in this thesis is non-blocking even in the case of multiple failures (see Section 6.3.1).

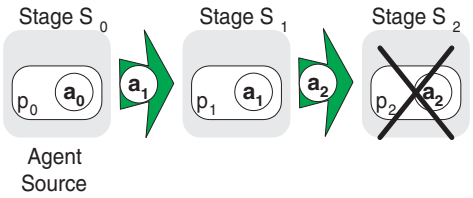

Figure 4.1: Agent execution where place p_2 crashes while executing a_2. While p_2 is down, the execution of a_2 is blocked.

4.2 Agent Replication and the Exactly-Once Execution Problem

4.2.1 Replication to Prevent Blocking

Blocking can only be overcome by introducing redundancy. Instead of sending the agent to only one place at the next stage, replicas of the agent are sent to a set of places. We denote by a_i^j the agent replica of a_i executing on place p_i^j, but omit the superscripted index if the meaning is clear from the context. Although a place may fail, another place can take over the execution to prevent blocking of the mobile agent. However, redundancy of execution may result in multiple executions of (parts of) the mobile agent. While this is not a problem for idempotent operations, it should not occur for non-idempotent operations. Assume, for instance, an agent that withdraws money from the agent owner's bank account. This is clearly a non-idempotent operation and multiple executions of this operation have the undesired effect of multiple money withdrawals. In general, non-idempotent stage actions must be executed *exactly-once* [RS98] from an application point of view. Indeed, to achieve exactly-once execution, the stage action can be executed multiple times, but all executions except one need to be undone. Hence, from the point of view of the application, it appears as if the agent had executed exactly once. In contrast to non-idempotent operations, idempotent 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.

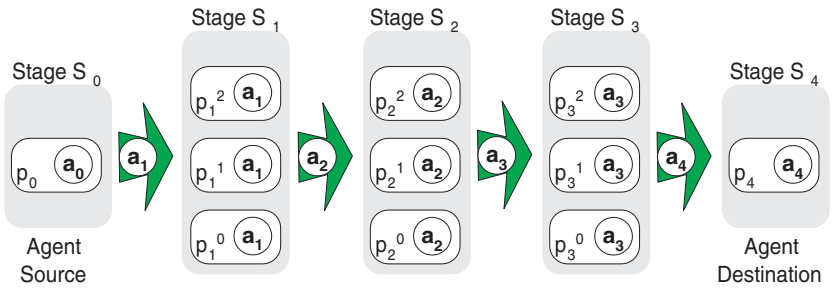

Figure 4.2: Agent execution spanning five stages $S_0, \ldots, S_4$ with two redundant places at stages S_1, S_2, and S_3 (i.e., replication degree 3).

The redundancy introduced by replication masks failures and ensures progress of the mobile agent execution. Figure 4.2 illustrates the replication approach. At stage S_i, a set of places $\mathcal{M}_i = \{p_i^0, p_i^1, p_i^2, ..\}$ executes the agent a_i. Even if place p_2^0 fails (see Figure 4.3) the agent a_2 is not lost, as the other places in $\mathcal{M}_2$ 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 only intermittently connected 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.

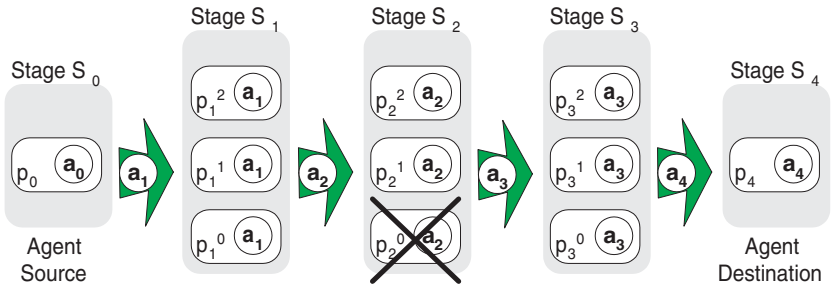

Figure 4.3: Agent execution with redundant places p_i^j ($\mathcal{M}_i = \{p_i^0, p_i^1, p_i^2\}$), where place p_2^0 fails. The redundant places mask the place failure.

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 (although this is not a requirement).

Ideally, the agent only executes on one place at a stage, unless a failure occurs. Only if this place fails or is suspected to have failed, the other places may start executing the agent. Using this lazy approach is less resource consuming than a priory executing the mobile agent on all places of a stage.

Despite of agent replication, network partitions can still prevent the progress of the agent. Moreover, catastrophic failures may still cause the loss of the entire agent. A failure of all places in $\mathcal{M}_2$ (see Figure 4.3), 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 any more. In other words, replication does not solve all problems. The non-blocking property only addresses single failures, as they cover most of the failures that occur in a realistic environment. Increasing the replication degree (i.e., the number of replica agents at a stage) generally prevents blocking even in cases in which more than a single agent fails. This is also the case for the approach presented in this thesis.

4.2.2 Properties of Places $\mathcal{M}_i$

In Section 4.2.1 we have introduced replication as a way to overcome the problem of blocking. Replication occurs on the agent level: the agent replicas execute on different places $p_i^j \in \mathcal{M}_i$ at a stage S_i. Depending on the relation among these places, we distinguish two different classes of places: *hetero-places* and *iso-places*.

Hetero-Places

Hetero-places correspond to a set $\mathcal{M}_i$ of places (see Figure 4.3) that all provide a similar service such as selling airline tickets from Zurich to New York. However, the places are provided by different airlines, e.g., Swiss Air Lines, Delta, or Lufthansa: they are not replicas of each other.

Hetero-places with witnesses[1] are a generalization of hetero-places. While hetero-places all provdfdide the particular service (i.e., airline tickets from Zurich to New York), in hetero-places with witnesses only a subset of the places provides the service. The others (i.e., the witnesses), although they can execute the agent, do not provide an airline ticket service to the agent and thus the service request of the agent can only fail (semantic failure). However, the agent is not lost and proceeds with the execution, while potentially reporting the failed ticket acquisition to the agent owner.

Iso-Places

Iso-places correspond to the traditional case of server replication: the set $\mathcal{M}_i$ consists of replica places, where all places are provided by the same company. Revisiting our airline example, all places are provided by Swiss 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., airline Swiss 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), that we call *replicated iso-places*, the places run a replication mechanism (e.g., using one of the replication techniques presented in Section 2.2.3) that ensures consistency among the place replicas. Note that executing the mobile agent a_i on two such iso-places in $\mathcal{M}_i$ at stage S_i may cause all iso-places in $\mathcal{M}_i$ to reflect the modifications twice.[2] In contrast, in (2) the agent replicas update the iso-places in $\mathcal{M}_i$. The mobile agent thus ensures consistency of the replicas. However, this requires that an instance of the agent (i.e., an agent replica) (1) executes on all place replicas and (2) should not fail as long as the service is up and running. We refer to this case as

[1] Also called *exception nodes* [SR98].

[2] Unless a mechanism (e.g., request IDs) is provided that prevents iso-places to execute the same operation twice. See Section 4.2.3 and Chapter 7 for a discussion on multiple executions of mobile agents.

independent iso-places[3]. Independent iso-places are assumed for instance in [Sch97].

Approaches to fault-tolerant mobile agent execution generally address hetero-places and hetero-places with witnesses, and replicated iso-places. In this work, we first present our solution in the context of hetero-places and hetero-places with witnesses (Section 6). The cases of replicated and independent iso-places are discussed in Chapter 7.

4.2.3 Replication and the Exactly-Once Problem

As explained in Section 4.2.1, replication allows executions to be non-blocking, but may also lead to multiple agent executions. Assume, for instance, that p_i^0 fails. Place p_i^1 starts executing a_i, which results in agent a_{i+1} and $\mathcal{M}_{i+1}$. In the meantime, p_i^0 recovers and continues the execution of a_i. If 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.

Our failure model (see Section 3.3) includes 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_i^1, modifications are (partially) applied twice (to p_i^0 and p_i^1). Replication of the agent thus leads to a violation of the exactly-once execution property of mobile agents. Consequently, the replication protocol of agents has to undo the modifications of a_i to the place p_i^0. 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.

Another source for a violation of the exactly-once execution property is given by unreliable failure detection.[4] In asynchronous systems such as the Internet, it is impossible to correctly detect failures (see Section 3.3.1). Even if a place p suspects the failure of another place q (i.e., believes that q has failed), q may not have failed in reality. Indeed, slow communication or processor speeds may have caused p to erroneously suspect q. When place p_i^1 suspects the failure of p_i^0, it starts executing a_i (see Figure 4.4). 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}, i.e., a violation to the exactly-once property.

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).
In summary, we require that a fault-tolerant mobile agent execution satisfies the following liveness and safety properties [AS85]:

Non-blocking A single failure must not prevent the termination of the agent execution *(liveness)*.

[3] In [PS00] and [PS01b], replicated iso-places are called *non-integrated iso-places*, while independent iso-places are referred to as *integrated iso-places*.

[4] This is captured by the property of *weak parsimony* in [DS00]. This property requires that if the same request is processed by two replicas p and q, then at least one of p and q is suspected by some replica.

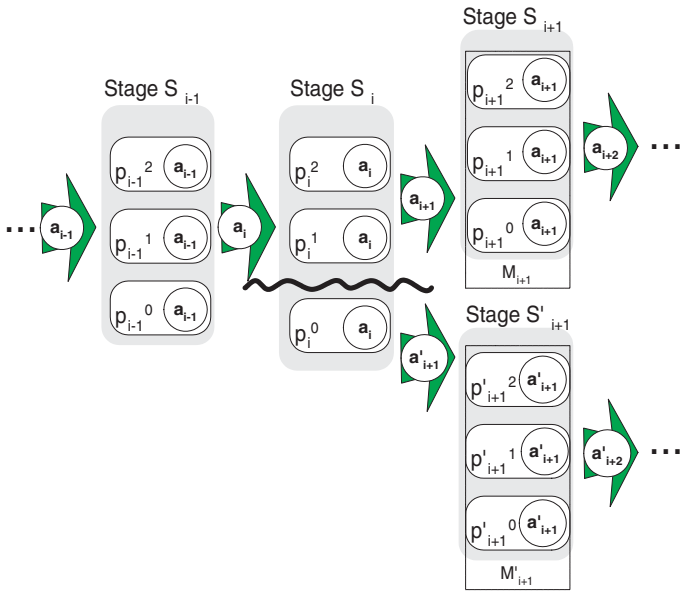

Figure 4.4: Replication potentially leads to a violation of the exactly-once property.

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

4.3 Basic Building Block: Local Transaction

In Section 4.2 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 the non-blocking property: the *local transaction*.

4.3.1 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 4.5). 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[5] (consistency). The effects of executing sa_i need to be durable, i.e., 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 executes entirely or not at all (atomicity). Only when sa_i has completed its execution should the results (including the modifications to the place) be visible to other agents (isolation). These four properties correspond to the specification of a transaction (ACID, see [HR83, BHG87, GR93]). Hence, sa_i needs to run as a *local transaction*. The concept of a local transaction is an important building block to fault-tolerant mobile agent execution.

[5]The resulting agent is called a_{i+1} (see Section 3.2).

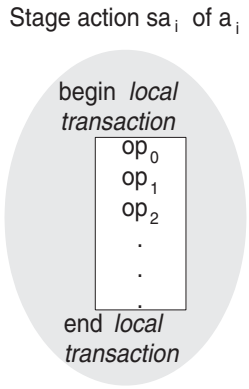

Figure 4.5: The stage action of agent a_i runs as a local transaction.

The local transaction consisting of operations $op_0, op_1, \ldots$ terminates either by a commit or an abort. If the decision is commit, the effects of executing $op_0, op_1, \ldots$ become durable, otherwise, all the modifications are undone. When and by whom this commit/abort decision is made allows to distinguish among various approaches for fault-tolerant mobile agent execution and leads to different properties of the approaches. We discuss these properties in more detail in Section 5.1.

4.3.2 Enforcing the Exactly-Once Execution Property

Executing the stage action sa_i as a local transaction allows to enforce the exactly-once property for mobile agent executions. Actually, while the mobile agent is executed exactly-once at every stage S_i, the local operations $op_0, op_1, \ldots$ of a stage action sa_i^j are executed *at-most-once*.

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, while others have not. The agent a_i as well as the place (rather, its services) are thus in an incorrect, transitory state. Executing sa_i as a local transaction prevents such inconsistent states, as the operations $op_0, op_1, \ldots$ are either executed in entirety or none at all.

Exactly-Once for the Stage S_i

In Section 4.2 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, i.e., the execution on the primary p_i^{prim}, must be committed, while the other(s) needs to be aborted. In Figure 4.6, places p_i^0 act as primaries for all stages S_i $(0 < i < n)$ and thus all commit the local transaction of the agent. This is why stage actions are executed at-most-once. Consider the example given in Figure 4.4.

Here, the execution of a_i on place p_i^0 needs to be aborted, while the execution of a_i on p_i^1 is committed. Running the stage executions on different places p_i^j, p_i^k as local transactions allows us, by controlling the commit/abort decision, to enforce the exactly-once property at the stage level.

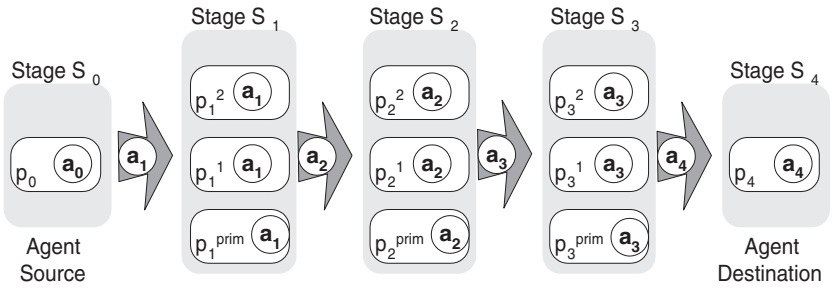

Figure 4.6: Agent execution with redundant places p_i^j in which places p_i^0 are acting as the primaries.

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 recovers after a failure.

4.3.3 Handling Isolation of (Local) Transactions

Isolation can be achieved using concurrency control techniques. Concurrency control techniques are classified into either *optimistic* or *pessimistic* approaches. In the former, operations are executed and the results are immediately visible to other agents. Undoing operations potentially requires to undo the operations of other agents that have used these results in their computations, thus resulting in cascading undo operations. In practice, this generally is avoided. In the context of transactions, optimistic techniques lead the local transaction to be immediately committed after its execution. If at a later stage, this transaction needs to be undone, a so-called *compensating transaction* [Gra81, KLS90, GMS87, RSS97] 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 cheque, or launch a rocket generally cannot be compensated. Moreover, if the local transaction launches another agent (i.e., a so-called child agent), then this agent may already have moved off. Hence, another agent needs to be sent after this child agent to compensate all its activities. This requires that the compensating child agent knows 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 also not always successful. Indeed, a slow undo message may never reach a fast moving mobile agent, causing the undo to be delayed and increasing dependencies.[6] In summary, optimistic execution is only suited for particular applications in a mobile agent environment.

Pessimistic concurrency control is based on locking, which prevents that multiple agents access the same data items, if their operations conflict. The results are only visible to other agents when the stage action commits, i.e., when the locks are released.

[6]Murphy and Picco [MP99] provide an approach to ensure eventual message delivery to an agent. However, their approach only works in an environment without failures and has a considerable cost.

4.3.4 Transactional Mobile Agent Execution

We have seen that a local transaction is the basic building block for fault-tolerant mobile agent execution. Local transactions are also important in the context of transactional mobile agents (see Section 3.4). Indeed, transactional mobile agents need to execute as a transaction, ensuring the ACID properties (i.e., atomicity, consistency, isolation, durability) of traditional transactions [HR83, BHG87, GR93] over the entire mobile agent execution. Hence, the stage operations also need to be executed as a transaction. In particular, either all of the stage operations $op_0, op_1, \ldots$ need to be executed, or none at all. This corresponds exactly to the atomicity property achieved by the local transactions. Figure 4.7 shows the case of (a) a non-transactional fault-tolerant mobile agent execution and (b) a transactional mobile agent execution.

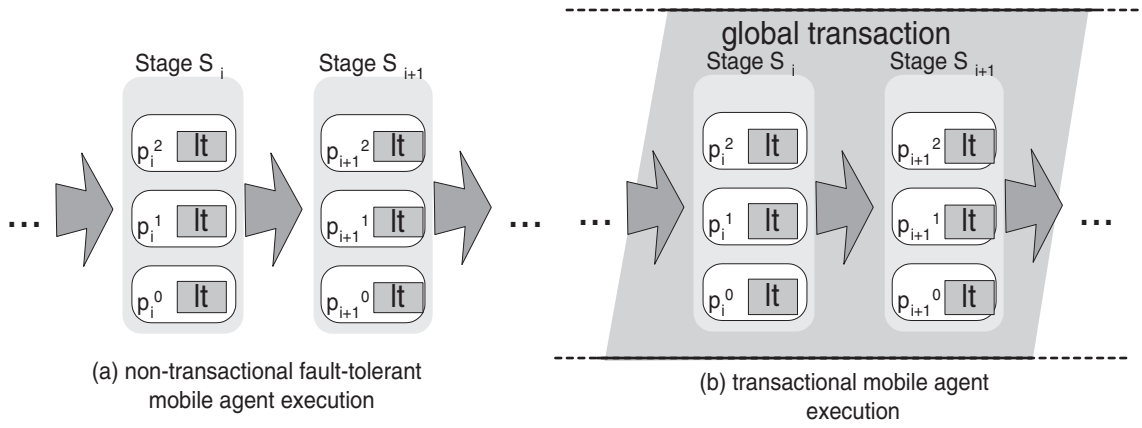

Figure 4.7: Comparison of (a) non-transactional fault-tolerant and (b) transactional mobile agent execution. Both executions rely on local transactions lt running on the places of a stage.

Chapter 5

State of the Art

In this chapter, we give a survey on the current state of the art in the fields of fault-tolerant and transactional mobile agents. First, we classify fault-tolerant mobile agent approaches according to when and by whom the commit decision for the local transaction is made (Section 5.1). Section 5.2 then discusses existing approaches for fault-tolerant mobile agent execution and relates them to our classification. In Section 5.3, we discuss approaches for transactional mobile agent execution and survey existing approaches in Section 5.4. We just provide enough information needed to understand the difference among the existing approaches. An in-depth discussion of fault-tolerant and transactional mobile agents is given in Chapters 6 and 8, respectively.

5.1 Classification of Fault-Tolerant Mobile Agent Approaches

In Section 4.3, we have identified the local transaction as the basic building block for fault-tolerant mobile agent execution (i.e., for addressing infrastructure failures)[1]. 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 to commit and which to abort the *commit decision*. In other words, the commit decision ensures exactly-once execution by committing only one local transaction at a stage, and aborting all others. It 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, i.e., 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. It is important to see that the commit decision does not ensure atomicity in the entire mobile agent execution, as required by a transactional mobile agent execution. Although in case (2), the commit decision may only happen at the agent destination, its purpose still is to ensure exactly-once at the stages, at which more than one agent replica has executed. We discuss the commit-after-stage and commit-at-destination cases in detail in Sections 5.1.1 and 5.1.2, respectively, and finally compare them in Section 5.1.3.

[1] Note that it is also the basic building block for transactional mobile agents, but this is not relevant for the following discussion.

5.1.1 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 4.3). More specifically, the commit decision allows to prevent multiple executions of the agent and thus ensures the exactly-once property. Indeed, by only committing the execution on one place of a stage and aborting all others (if there are any), it is ensured that the agent is executed only once. In this context, we need to distinguish two cases: (1) the execution of a_i on a single place (i.e., a non-replicated agent execution) and (2) on a set of places $\mathcal{M}_i$ (i.e., a replicated agent execution). Moreover, the commit decision can be made by a single place or can be distributed, i.e., 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 are denoted as follows: location of the agent execution - location of the commit decision - collocated / distributed (see Figure 5.1).

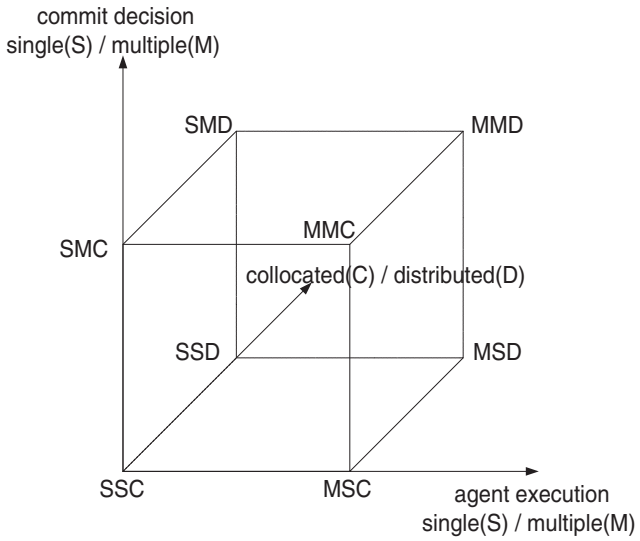

Figure 5.1: Classification of fault-tolerant mobile agent approaches along three axes: (x) location of the agent execution, (y) location of the commit decision, and (z) collocated or distributed.

In Chapter 4, we have shown how blocking can occur in the agent execution. Blocking also occurs in the commit protocol. In particular, if the commit decision is only made by a single place (i.e., solutions $x\mathrm{S}x$[2]), then there is a risk of blocking or violation of the exactly-once execution property to the mobile agent execution. For instance, in a 2PC protocol, blocking occurs if the coordinator fails at a certain point in the protocol [BHG87]. We discuss now all the 8 solutions.

[2]The character x stands for any one character in {S,M,C,D}, such that the resulting combination is among the eight solutions in Figure 5.1.

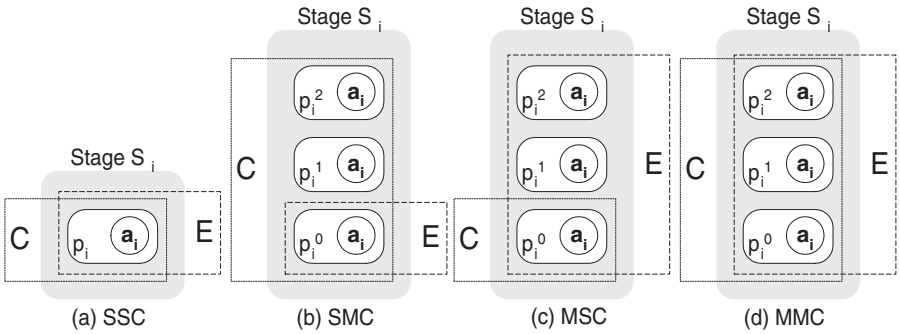

Figure 5.2: The four solutions of collocated commits. E and C specify the places involved in the agent execution and the commit decision, respectively.

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}. Figure 3.2 shows an example of an SSC approach. Both the execution of the stage action and the commit decision occur on the same place (see Figure 5.2 (a)). Actually, the outcome of the commit decision is always commit; abort is never decided, as there is no reason to abort the agent execution from the perspective of p_i.[3] As we have shown in Section 4.1, a failure of p_i causes blocking of the agent execution (see Figure 4.1). 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. Checkpointing the state and code of the agent on the current place [GR93] 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 makes use of the latest local checkpoint to recover the agent a_i.

A SSC approach is most suited in environments where failures are rare or where blocking is not a problem, either because of the nature of the application or because failed components recover fast. Moreover, the exactly-once execution property is ensured provided a suitable recovery mechanism is used.

In SSC approaches based on logging and checkpointing, the loss of the agent generally is 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 is preserved.

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, in contrast, is distributed over multiple places (see Figure 5.2 (b)). Consequently,

[3] Recall that we are not considering semantic failures here. Moreover, we assume that the local transaction does not abort spontaneously. More specifically, it can be aborted and redone during the execution of the stage action (e.g., to resolve deadlocks if two-phase locking is not used [GR93]), but when it is ready to commit, it is not aborted any more.

a failure of p_i leads to blocking of the mobile agent execution. On the other hand, the commit decision is non-blocking, as it is distributed. It seems strange to distribute the commit decision, while the execution happens on a single place and thus the SMC solution has not been discussed in literature. 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 Air Lines, which only supports a non-replicated service.

Multiple/Single/Collocated - MSC

The stage action of agent a_i is executed by multiple places, while the commit decision is made by a single place p_i^k (see Figure 5.2 (c)). Revisiting Figure 4.4, p_i^0, p_i^1, and p_i^2 execute the agent, whereas p_i^2, for instance, executes the commit protocol. In the MSC solution, the commit decision determines the place that has executed the agent, i.e., the primary p_i^{prim}. Recall from Section 4.3.2 that the primary is the place at a stage that commits the local transaction of a_i, while 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, while p_i^0 and p_i^2 must abort them (if a_i has started execution on these particular places). This allows us to prevent 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.

Multiple/Multiple/Collocated - MMC

The MMC solution is a generalization of the approaches 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 on multiple places (see Figure 5.2 (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} who commits the modifications done by the agent while all other places abort them. In other words, unless an agreement is reached, i.e., a so-called agreement problem is solved, among these places, multiple executions of the mobile agent cannot be prevented.

Single/Single/Distributed - SSD

The SSD solution corresponds to SSC, except that the execution of the agent and the commit decision are distributed (see Figure 5.3 (a)). In other words, the place that executes the agent and the place that makes the commit decision are not the same; rather, any place p_k can make the commit decision, which then needs to be communicated to p_i. This communication is prone to failures such as loss of message. 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 later that this solution is of considerable interest to transactional mobile agents (see Section 5.3).

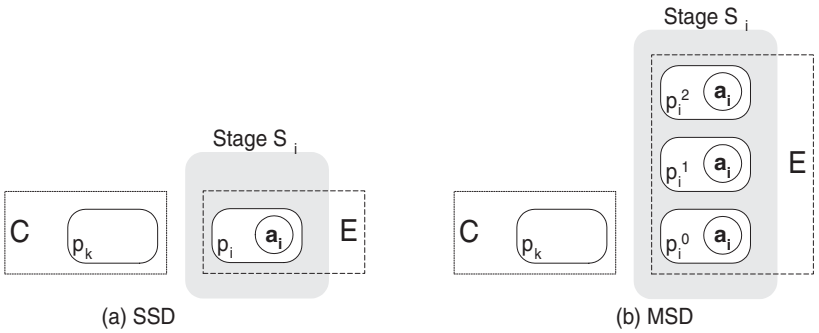

Figure 5.3: The two solutions of distributed commits by place p_k. E and C specify the places involved in the agent execution and the commit decision, respectively.

Single/Multiple/Distributed - SMD

This solution is similar to SMC and is not discussed any further. It is depicted in Figure 5.4 (a).

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 5.3 (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.

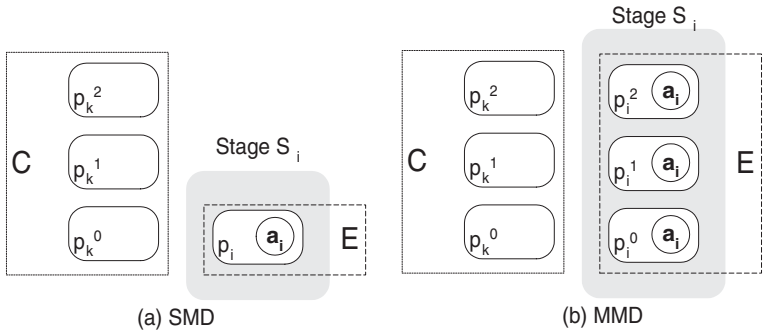

Figure 5.4: The two solutions of distributed commits by multiple places. E and C specify the places involved in the agent execution and the commit decision, respectively.

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 5.4 (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 between the places executing the stage action (i.e., p_i^j) and the places executing the commit protocol (i.e., p_k^j).

5.1.2 Commit-At-Destination Approaches

Contrary to commit-after-stage approaches, the stage actions of the mobile agent are only committed at the end of the agent execution. While in commit-after-stage approaches duplicate agents are detected and discarded at the stage execution, duplicates continue their execution in commit-at-destination approaches. Redundant 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, while the later arriving redundant agent(s) are aborted and their stage actions undone (see Figure 5.5). This allows us to ensure the exactly-once property for fault-tolerant mobile agent execution.

Using pessimistic concurrency control, while the agent has not reached the agent destination, the local transactions are not committed/aborted yet. 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 b it is not clear whether b will be committed or whether a duplicate agent will arrive first at the agent destination and b thus needs to be aborted. Data items that are accessed by the mobile agent b are thus locked until the corresponding local transaction is committed. During this time, no other agent c can access the same data items. Rather, c waits until the lock is released by b. Committing the agent's stage actions only at the agent destination requires to hold all locks until agent a arrives at the agent destination. As other agents have to wait before accessing the data items until b finishes its execution, the 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 5.5).

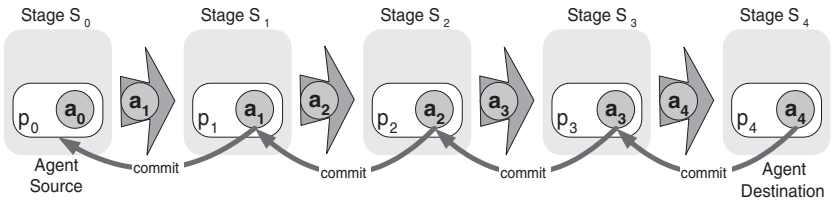

Figure 5.5: The local transactions are only committed when the agent arrives at the agent destination.

Using optimistic concurrency control, the locks on data items are immediately released after executing the local transaction. Undoing a redundant duplicate agent requires to run compensating transactions on all places this agent has visited. Recall that compensating transactions are not always possible (see Section 4.3.3).

We classify the approaches similarly to Section 5.1.1, but focus on the only two meaningful classes: SSD and MSD.

Single/Single/Distributed - SSD

This solution is similar to the SSD solution for commit-after-stage approaches, except that the commit only occurs at the agent destination, for all stage actions (see Figure 5.5). 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 helps to understand the difference between non-transactional and transactional mobile agent execution (Section 5.3.1).

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}. Additionally, p_{i-1} maintains a copy of the agent a_i. If a failure occurs at the current place p_i, p_{i-1} launches its copy of the agent and sends it to another place p_i' (see Figure 5.6). Sending a_i to p_i', however, may lead to duplicate agents, especially in the presence of unreliable failure detection. Indeed, assume that p_k erroneously detects the failure of p_{k+1}, whereas p_{k+1} actually has not failed (see Figure 5.7). In this case, two replicas of agent a_{k+1} are executed and lead to two agents a_{k+2} and a_{k+2}'. Duplicates are only detected at the agent destination, where the commit decision is made. This allows to enforce the exactly-once property.

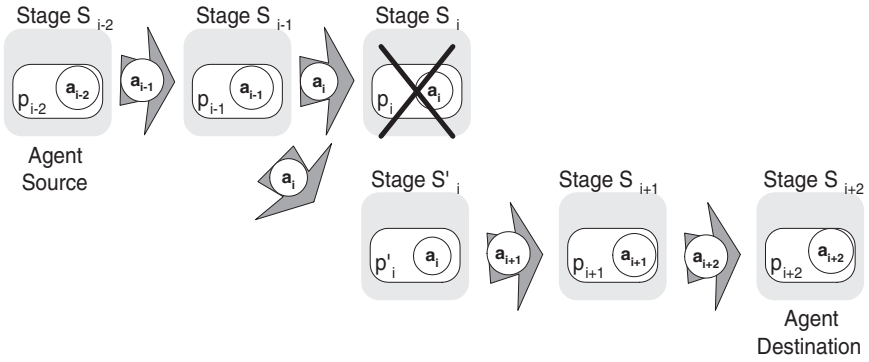

Figure 5.6: Upon detection of p_i's failure, p_{i-1} sends a_i to p_i'.

The degree of fault tolerance is determined by the number of copies stored on places where the agent previously has 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 handled. However, a high number also increases the probability of duplicate agents.

Upon recovery of a failed agent, we need to distinguish two cases: the agent has (1) only executed 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, as 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 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.

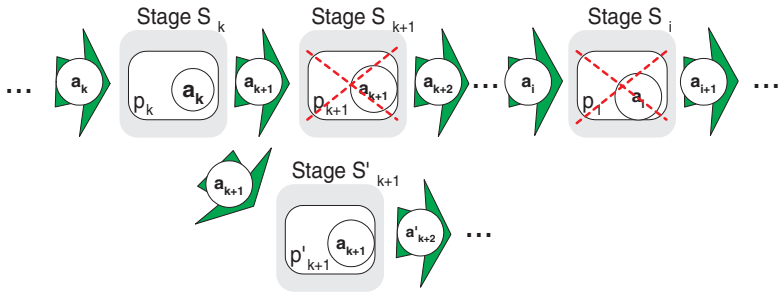

Figure 5.7: Duplicate agents caused by unreliable failure detection in the commit-at-destination approach.

An important limitation when using compensating transactions is that running the compensating transactions must lead to the same agent as when the duplication agent occurred. Assume, for instance, that the execution of agent a' needs to be undone, i.e., compensating transactions must be run on p'_{i+1} and p'_i (see Figure 5.8). A compensating transaction generally modifies the state of the place as well as the state of the agent. Having undone a' must result in the same agent a_i, as agent a reflects the state of a_i in its final state (i.e., a_{i+2} at the agent destination). A more detailed discussion is given in the context of transactional mobile agent in Section 8.8.

5.1.3 Comparison

The most important differences between committing at the agent destination and committing at the stage execution are the lifetime of duplicate agents and the number of commit decisions. The lifetime is crucial as it influences the time data items need to remain locked in the case of pessimistic concurrency control. The greater the lifetime, the longer data items remain locked. During this time, other mobile agents cannot access the data items and block, which limits the overall system throughput. If optimistic concurrency control is used, a greater lifetime leads to higher undo cost. Indeed, the more stage actions must be undone, the more compensating transactions generally must be run. 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 using pessimistic concurrency control generally needs to keep the locks on all data items until the end of the agent execution. Clearly, this is a disadvantage of the commit-at-destination approach. At

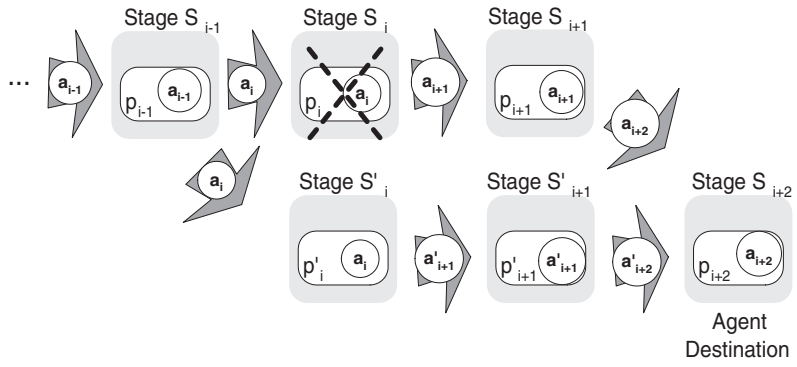

Figure 5.8: The execution of agent a' on the places p'_i and p'_{i+1} must be compensated.

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 additional messages sent to all places of the itinerary.

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. While mobile agents are of generally small size, a large number of them still leads to considerable storage requirements on the places. Generally, the copies of the mobile agent have to be maintained until the mobile agent execution has terminated, i.e., the commit/abort message has been received. In the commit-after-stage approach, copies of the agent are stored on the places in $\mathcal{M}_i$ only during the stage S_i and discarded afterwards.

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 only requires one commit decision, committing at the stage execution requires $n - 2$, i.e., one at all stages except S_0 and S_n. Moreover, agent replicas are only launched if a failure is (potentially erroneously) detected. In contrast, commit-after-stage approaches using any solution Mxx send multiple replica agents to the next stage although no failure may have been detected.

5.2 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 presented according to the classification defined in Section 5.1. First, we present commit-after-stage approaches (Section 5.2.1). Commit-at-destination approaches are discussed in Section 5.2.2. The results of our classification are summarized in Table 5.1.

5.2.1 Commit-After-Stage Approaches

We consider the following commit-after-stage approaches: Byzantine failures approach [Sch97], FANTOMAS [PPG00], FATOMAS [PS00, PS01b] (see Chapter 6), NAP [JMS+99], transaction and leader-election based approaches [RS98, ASPZ98], and Vogler's approach [VKM97a,

	commit-after-stage	commit-at-destination
SSC	Vogler's approach [VKM97a, VKM97b]	-
MSC	Rothermel's approach [RS98]	-
MMC	FATOMAS (Chapter 6), NAP [JMS+99], Schneider's approach [Sch97]	-
MSD	-	NetPebbles [MPT00], James [SBS00]
MMD	Fantomas [PPG00], Assis' approach [ASPZ98] James [SBS00]	-

Table 5.1: Classification of the existing approaches.

VKM97b].

Byzantine Failures Approach

Minsky et al. [MvRSS96] and Schneider [Sch97] proposes 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. In this context, all places $p_i^j \in \mathcal{M}_i$ of a stage S_i execute a_i and commit the modifications. Although an adversary may corrupt a number of agents at a stage, if enough uncorrupted agents are left, it still allows to safely deduce the true result of the stage execution. Hence, the exactly-once execution property is not desired in this protocol. The places in $\mathcal{M}_i$ are independent iso-places as defined in Section 4.2.2: Schneider assumes replica places without replica update mechanisms to maintain 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. We do not further discuss approaches that address Byzantine failures.

FANTOMAS

FANTOMAS [PPG00] is an MMD approach that addresses transparent fault tolerance for distributed and parallel applications in cluster systems. It's 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 that 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 5.9). The logger agent stores checkpoints of the agent it is associated with. 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. 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.

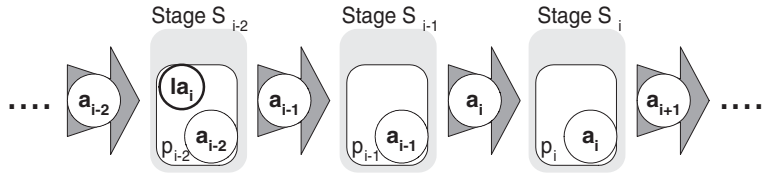

Figure 5.9: Agent a_i with logger agent la_i and distance $d = 2$.

The approach in [PPG00] 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.

FATOMAS

The approach we present in this thesis is a MMC commit-after-stage approach (see Chapter 6). The redundancy illustrated in Figure 4.3 enables the mobile agent execution to proceed despite failures, i.e., prevents blocking. However, the algorithm that prevents blocking while ensuring a consistent execution is not as easy as one might guess. This is related to the fact that we assume a system model in which failure detection is unreliable. The solution presented in Chapter 6 consists, for all agent replicas at stage S_i, to solve an *agreement problem*. In an agreement problem, the participants (in our case the agent replicas) decide on a common value. The fault-tolerant mobile agent execution then leads to a sequence of agreement problems.

JAMES

JAMES [SBS00] 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 5.2.2.

NAP

NAP [JMS+99] uses the MMC approach to fault-tolerant mobile agent execution. It assumes a fail-stop model, which implies a perfect failure detector [SM94]. 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 our approach (see Chapter 6) is required. Rather, perfect failure detectors allow to reliably detect process crashes. In particular, no process is suspected unless it has crashed, which eliminates one source for a violation of the exactly-once execution property. Unfortunately, perfect failure detectors are impossible in the Internet and therefore NAP is only applicable in systems where perfect failure detectors can be assumed. Moreover, the NAP approach does not handle link failures nor consider recovery of

places (see also Section 3.3.1). Assuming no recovery eliminates another source for a violation of exactly-once. Still, local transactions are required to ensure isolation of the stage actions. Indeed, the local transaction prevents that other agents can access partial results of the stage execution.

Transaction And Leader Election Based Approaches

The following two approaches use a model based on transactions and leader election.

Rothermel's Approach: Rothermel and Strasser's approach [RS98] 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. In [RS98], Rothermel and Strasser suggest an approach based on transactions [GR93] and leader election. Informally, the problem of leader election consists of electing a leader among a group of processes, such that there is only one leader and all processes of this group agree on this leader [GM82]. 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 $\mathcal{M}_{i+1}$. Multiple places in $\mathcal{M}_i$ 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 2-phase-commit (2PC) protocol [BHG87] to atomically commit the transactions. The 2PC protocol is known to be blocking on a single failure [BHG87]. The reader may argue that the use of a 3PC [BHG87] alleviates the blocking problem. Unfortunately, blocking stems from the combination of leader election and transactions as well as the nature of MSC, and thus cannot be prevented by the use of a 3PC.

Assis' Approach: Assis and Popescu [ASPZ98] improve Rothermel's algorithm by overcoming some of its limitations. In particular, to prevent the blocking problem in [RS98], they use a different leader election protocol and commit the local transaction using a 3-phase commit (3PC) protocol [BHG87]. However, this particular combination of leader election and transaction model may lead to a violation of the exactly-once property. Hence, [ASPZ98] relies on a so-called *distributed context database* for synchronization to prevent more than one concurrent leader and thus to enforce the exactly-once property. In summary, the commit decision is done in collaboration with the distributed context database, a leader election protocol, and the 3PC. Moreover, the distributed context database is running on the places of the stage S_i. However, to our understanding, the context database will generally be run by (an)other process(es) than the execution of the agent. Moreover, it can be implemented as a separate service. Consequently, we consider [ASPZ98] as a MMD approach.

Similar to [RS98], the approach in [ASPZ98] uses transactions and leader election to model fault-tolerant mobile agent execution. Combining the two models makes it more difficult to understand the approach and may thus give rise to additional errors. Another disadvantage of this

approach are the rather high maintainance cost for the distributed context database, which needs to be replicated (to provide fault tolerance).

Vogler's Approach

Vogler et al. [VKM97a, VKM97b] 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 (see Section 4.3) from the beginning. 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's approach is blocking, but ensures exactly-once mobile agent execution properties provided that the stage actions run as local transactions.

5.2.2 Commit-At-Destination Approaches

The most prominent example of commit-at-destination approaches is a system called NetPebbles [MPT00]. It belongs to the class of commit-at-destination MSD approaches. The James system [SBS00] can also be classified into this category, although it has some elements of the commit-after-stage MMD approach.

NetPebbles

The NetPebbles environment 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 this 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 5.6, 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 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 d. The distance is defined as the difference between the indices, i.e., $j - k$, 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 is the frequency that 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, i.e., if it stops receiving heartbeat messages. This allows NetPebbles to handle a number of concurrent failures that is equivalent to the distance value.

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, while the actions of all others must be aborted. The problem of how to commit or abort the actions of the agents is not developed in more detail and left open in [MPT00].

Using pessimistic concurrency control, data items need to remain locked 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 5.7). Due to a network partition or slow communication links, place p_k no longer receives heartbeat messages from any p_j $(k + d \geq j > k)$ 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 ago passed stage S_k and currently executes on p_i $(i > k + 2)$. Hence, locks can only be released at the agent destination.

JAMES

JAMES [SBS00], a Java-based mobile agent infrastructure, is a platform that provides a running environment for mobile agent, 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 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. This 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 [SBS00], although no explicit examples are given.

When the failure of the agent currently executing, i.e., 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 5.5). With this approach, blocking is prevented but agent duplicates may occur (see Section 5.1.2). 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 viewed as the distributed commit decision in MMD, which decides on 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.

Such a fault-tolerant, replicated lookup directory 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 need to remain consistent.

Locking of the data items that have been accessed by the agent seriously limits the overall system throughput. In addition, it 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. This problem does not seem to be addressed in [SBS00].

5.3 Transactional Mobile Agents

In this section, we present approaches, called *transactional mobile agents*, that address infrastructure and semantic failures. We start by a comparison with non-transactional, fault-tolerant mobile agents, before presenting a model for transactional mobile agents based on (open) nested transactions.

5.3.1 Commit in Non-Transactional, Fault-Tolerant Mobile Agents *vs.* Commit in Transactional Mobile Agents

In Section 5.1 we have classified non-transactional, fault-tolerant mobile agent approaches according to when and by whom the commit decision of the stage action is performed. In the context of non-transactional mobile agents, the commit decision helps to ensure the exactly-once execution property of the mobile agent. For instance, 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 $\mathcal{M}_i$ are aborted. In commit-at-destination approaches, the commit decision selects the duplicate agents arriving at the agent destination that have to be undone. Assume, for instance, that the execution of an agent a leads to two agents that arrive at the agent destination, namely the original agent a and a duplicate agent a' (see Section 5.2.2). To ensure exactly-once execution, the commit decision at the agent destination must only commit one of these agents, whereas the other is aborted. This 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 non-transactional mobile agent using the commit-at-destination approach.[4] The difference 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

[4] A similar reasoning also applies for commit-after-stage approaches.

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 whether the stage operations were semantically successful. Revisiting the first example in Section 3.4, commit-at-destination approaches commit (i.e., book a hotel room) the agent's stage operation although no flight is available. In contrast, transactional mobile agents either commit both operations or abort them both. In other words, if no flight is available, all agent operations will be aborted. While 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 semantic failures.

Because transactional mobile agent executions also address semantic failures, they have additional requirements. Indeed, a *transactional mobile agent* execution is specified in terms of the atomicity, consistency, isolation, and durability (ACID) [HR83, BHG87, GR93] properties. While in Section 4.3.1 we require ACID properties for local transactions, transactional mobile agents need to guarantee that the ACID properties encompass the entire mobile agent execution, i.e., that $sa_0, \ldots, sa_n$ run as a transaction (see Section 4.3.4).

5.3.2 Nested Transaction Model

A transactional mobile agent execution can be modeled as *nested transactions* [Mos85]. A nested transaction is a transaction that is (recursively) decomposed into *subtransactions*. Every subtransaction forms a logically related subtask. A successful subtransaction only becomes permanent, i.e., commits, if all its parent transactions commit as well. In contrast, a parent transaction can commit (provided that its parent transactions all commit) although some of its subtransactions may have failed. In a transactional mobile agent execution, the top-level transaction (i.e., the transaction that has no parent) corresponds to the entire 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 modeled by yet another level of subtransactions, which correspond to the agent replicas $a_i^0, \ldots, a_i^m$ running on the places in $\mathcal{M}_i$ and executing the set of operations $op_0, op_1, \ldots$. Note that subtransactions may be aborted, but the parent transaction still commits. Indeed, 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 with airline X, but no seat is left. We do not consider replication at the moment and assume that no failures nor false suspicions occur. Consequently, the corresponding subtransaction sa_i fails semantically and will be aborted. However, the agent may move on to the server of airline Y and attempt to book a flight with airline Y. When this subtransaction sa_{i+1} is successful, the agent continues and the top-level transaction can still commit. Note that this is fundamentally different to commit-after-stage approaches, where these servers are usually visited concurrently at the same stage. Transactional mobile agents are a simplification of nested transactions as the subtransactions never conflict and never deadlock among themselves. Indeed, the subtransactions a_i^j execute on different places and thus run in complete isolation from each other. The parent transaction (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 one executed by the primary), and aborts all others. The top-level transaction only commits if all the subtransactions sa_i, that must succeed, are ready to commit (i.e., have not aborted).

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 also for the general case, where only a subset of the stage actions needs to succeed.

5.3.3 Execution Atomicity

In this section we show how the ACID properties, in particular atomicity and durability (consistency and isolation are discussed in Section 5.3.6), can be ensured for a transactional mobile agent execution. Among the ACID properties of the top-level transaction, atomicity and durability[5] are the hardest to achieve. Atomicity encompasses semantic failures (see Section 3.4), by ensuring that either all stage actions are executed successfully or none of them. To achieve atomicity, the mobile agent execution decides whether to commit all local transactions or whether to abort all of them. As a consequence to the sequential execution of the subtransactions sa_i, the outcome of all stage actions is only known (and thus this decision only possible) at the agent destination[6]. Here, the agent destination sends a message (or another agent) that contains the commit/abort decision to all previous places (see Figure 5.5). On reception of this message, the place commits or aborts the local transaction.

If one subtransaction sa_i $(i \leq n-1)$ fails semantically, the global transaction is immediately aborted (see Figure 5.10). In particular, the local transactions $sa_{i+1}, \ldots, sa_{n-1}$ and the stage action sa_n are not yet executed. On the other hand, if no semantic failure occurs then the transactional mobile agent is committed at the agent destination.

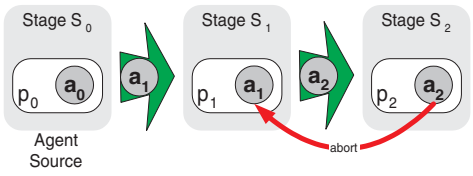

Figure 5.10: Abort is immediately communicated to all predecessor places in the transactional mobile agent execution.

5.3.4 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 exe-

[5] 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 have been violated.

[6] Actually, the decision could be made on p_{n-1} (see Section 8.1.2), as the agent source and destination contain by assumption only idempotent operations and may only be intermittently connected to the network, if it is a mobile device. For simplicity, we assume in this chapter that the agent destination does not fail and is always connected.

cution 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, while the places $p_0, \ldots, p_{i-1}$ have already previously aborted the stage actions. Clearly, this is a violation to the atomicity property. Consequently, transactional mobile agent approaches generally either are blocking or employ fault tolerance techniques (see Section 5.1) to prevent blocking.

5.3.5 A Simple Approach to Ensure Atomicity

The simplest approach to ensure atomicity is to reuse the SSD commit-at-destination approach (see Section 5.1.2), which is based on checkpointing. At every place, the agent's state and code is checkpointed. Upon recovery from a failure, the agent's execution is continued from the last checkpoint. The local transactions are only committed when the agent reaches the agent destination. 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 need to be committed, as we assume that it only consists of idempotent operations. The transactional mobile agent execution is immediately aborted if a semantic failure occurs that renders obsolete any further execution. For instance, if the agent owner only flies with Swiss Air Lines, but Swiss Air Lines does not have any seats available for the required destination, the agent execution can be immediately aborted.

5.3.6 Multiple Concurrent Transactional Mobile Agents

In our discussion so far we focus on enforcing the atomicity property of a transactional agent execution. In a real system, however, a transactional mobile agent does not execute in an isolated environment. Rather, it executes concurrently with other transactional mobile agents. Generally, transactional mobile agents use standard techniques [BHG87, GR93] to enforce the ACID properties: compensatable transactions [ASK97, SAE01, SR00] or locking (e.g., the approach suggested in Chapter 8) for isolation and consistency, and the local transaction infrastructure for durability.

The isolation property limits the possible level of concurrency. As a remedy, services decide themselves whether they allow concurrent access to their data. For this purpose, they design a so-called commutativity matrix [Rak94], which shows potential conflicts among operations of this service and only allow operations that do not conflict to be executed concurrently.

The use of locking and in particular the sequential acquisition of locks may lead to deadlocks. Deadlocks are resolved when transactions wait for a lock only for a limited time. After this time they assume the occurrence of a deadlock and back off. If other places with similar services are available, they may try those, or otherwise retry the same service again. Although this approach does not completely rule out livelocks, their probability may be sufficiently small.

5.3.7 Open Nested Transaction Model

With nested transactions the locks on the data items are only released at the end of the transactional mobile agent execution. As other transactional mobile agents can only access the locked data items

when the locks are released, they have to wait until the transactional mobile agent holding the locks is done. The use of compensating transactions allows to reduce the time locks must be held and thus increases overall system throughput. In this case, the stage actions of a transactional mobile agent are immediately committed and their results visible to other transactions. If the transactional mobile agent is aborted at a later stage, the stage actions are semantically undone by running the compensation transactions in the reverse order on the places that have committed the stage actions. For this purpose, the agent returns along the same itinerary and executes the compensating transaction at each stage. Assume, for instance, that the agent has committed the stage actions $sa_0, \ldots, sa_{k-1}$ on places $p_0, \ldots, p_{k-1}$ and is currently executing sa_k, which is aborted. The agent a_k thus visits the sequence of places $p_{k-1}, \ldots, p_0$ and executes the compensating transaction on each place.

However, compensating transactions are not always possible (see Section 4.3.3). Moreover, between executing the stage action sa_i and its compensating transaction, another mobile agent b can access data items modified by sa_i. Executing the compensating transaction semantically undoes the modifications performed by sa_i. Agent b may have now read an inconsistent value. Consequently, T_b also needs to be aborted, leading to cascading aborts. For this purpose, another agent or an undo message have to be sent after agent b to notify b of the abort. Unfortunately, a slow undo message or agent may never reach a fast moving mobile agent, causing the undo to be delayed and increasing dependencies. Hence, compensatable transactions work best in an environment, where compensation transactions can be run without causing cascading aborts. Compensation may also be unsuitable although it is feasible, because it has an 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 the compensation transactions must eventually commit. Consequently, failures during the compensation transactions lead to blocking. In contrast, an abort with closed nested transactions is as expensive as a commit in the sense that the message sent to all places contains the directive to abort instead of commit.

To encompass also the use of compensating transactions, we extend the model of transactional mobile agents to *open nested transactions* [WS92]. Open nested transactions can consist of *open* and *closed* subtransactions. An open subtransaction makes its results visible to other transactions as soon as its computation is terminated, independent of the outcome of its parent transaction. In contrast, a closed subtransaction isolates its updates from other transactions until its parent transaction commits and it is able to commit as well. If all subtransactions are closed subtransactions, then the case of (closed) nested transactions as presented in [Mos85] and discussed in Section 5.3.2 applies. On the other hand, the model of (extended) sagas assumes that all subtransactions are open [GMS87, GMGK$^+$91].

Depending on whether closed or open subtransactions are used, compensating transactions or transaction aborts are used. An ideal approach supports both techniques.

5.4 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.

5.4.1 Blocking Solutions

Assis and Krause [ASK97] provide a model of transactional mobile agents, that corresponds essentially to the checkpointing approach discussed in Section 5.3.5. However, stage actions can be compensatable or non-compensatable.

In [SAE01], Sher et al. 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 an inherent property of any SSD approach and [SAE01] 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, with the mediator `ANDjoin`, all parallel transactions have to arrive, or with `XORjoin`, only one has to arrive. Figure 5.11 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 Air Lines, while c_{i-1} books a flight with Delta. At stage S_i, the mediator `XORjoin` only keeps one of 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 of non-parallel transactions and mediators result in blocking of the execution. Eliminating non-parallel transactions thus prevents blocking, i.e., 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 failures occur, causes a considerable overhead.

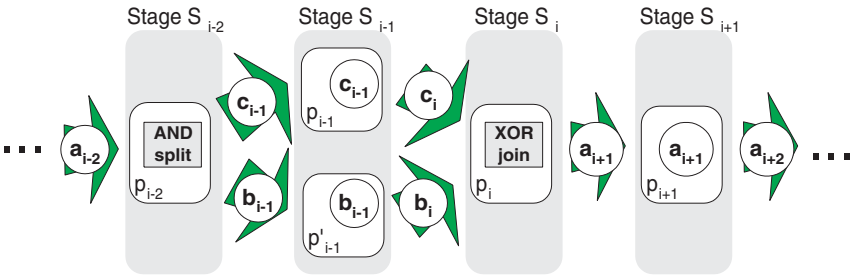

Figure 5.11: Mediators (rectangles) allow to execute parallel transactions.

In their work, Strasser and Rothermel [SR00] address the issue of partial rollbacks to a savepoint in a mobile agent execution. If a savepoint is located at S_0, then the mechanism they present ensures atomicity on the entire mobile agent execution. Their mechanism is built on top of the approach for fault-tolerant mobile agent execution given in [RS98]. Hence, the transactional mobile agent execution may block if the coordinator fails during the forward execution of the transac-

tional mobile agent (see Section 5.2.1). However, as compensating transactions are used, only the resources of a stage execution are locked. Resources of other stages are still available to other transactional mobile agents. The use of compensation transactions, as suggested in this mechanism, limits the applicability but improves the performance by releasing the locks immediately after the stage execution. The issue of closed subtransactions is not addressed in [SR00]. Moreover, 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 and thus the entire mobile agent execution may block.

5.4.2 Non-Blocking Solutions

The approach presented by Assis Silva and Popescu [ASPZ00] also builds transaction support on top of fault-tolerant mobile agent execution. For this purpose, [ASPZ00] reuses the approach in [ASPZ98]. Recall that the approach in [ASPZ98] is based on a complex model of leader election and transactions (see Section 5.2.1). Our model of fault-tolerant mobile agent execution (see Chapter 6), on the other hand, only relies on an agreement problem. This allows us to present a model for transactional mobile agent execution that integrates fault-tolerant mobile agents into a common model of (open) nested transactions. We also provide a specification of non-blocking atomic commitment in the context of transactional mobile agents.

Assis Silva and Popescu [AS99, ASPZ00] do not describe any implementation. In contrast, our approach has been implemented and quantitatively evaluated.

Finally, [ASPZ00] relies on compensatable transactions [Gra81, GMS87]. This seriously limits the applicability of [ASPZ00]. To our knowledge, the use of compensatable transactions is a consequence of the particular approach used for fault tolerance [ASPZ98]. The approach for fault tolerance is based on transactions, that have to commit before the agent execution can proceed. Consequently, if the agent execution is aborted at a later stage, compensation transactions have to run in order to undo the effects of the stage transactions. In contrast, with pessimistic concurrency control, subtransactions sa_i are only committed or aborted once the outcome of the top-level transaction is known.

The approach in [MPT00] (see Section 5.2.2) can be directly extended 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 5.4.1) running at the agent destination. The approach in [MPT00] 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 [SAE01], parallel transactions are only started if a failure is detected. Hence, if no failures occur nor are detected, no parallel transactions occur.

In this thesis, we suggest an approach, called TRANSUMA (see Chapter 8), to non-blocking transactional mobile agents based on our approach to fault-tolerant mobile agents. 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. On the primary at every stage, the agent

leaves a so-called *stationary agent* (i.e., an agent that does not move), which awaits either an abort or a commit message. Upon reception of such a message, the stationary agent either aborts or commits the operations of the local transaction according to the content of the message. The other ACID properties are achieved using the usual mechanisms [BHG87, GR93]. Moreover, we show how TRANSUMA can be extended to also address open subtransactions.

5.5 Summary

We have started this chapter with a classification of fault-tolerant mobile agent approaches. In this classification, we first distinguish between commit-after-stage approaches (stage actions are committed immediately after the stage execution) and commit-at-destination approaches (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 or not. This leads to various solutions, which have been discussed in terms of their advantages and usability.

We have also shown how commit-at-destination approaches can be extended to transactional mobile agents. Besides infrastructure failures, transactional mobile agents also address semantic failures. Finally, we have also presented a survey of transactional mobile agents.

Chapter 6

Fault Tolerance for Mobile Agents

This chapter presents our approach for fault-tolerant mobile agent execution. For simplicity, we limit our discussion to the case of hetero-places and thus first discuss the exactly-once property in the context of hetero-places (Section 6.1). Fault tolerance of mobile agents in the context of iso-places is discussed in Chapter 7. Section 6.2 is central to this chapter; it models fault-tolerant mobile agent execution as a sequence of agreement problems. In Section 6.3, we show our algorithm and present our prototype (called FATOMAS) in Section 6.4, followed by its performance evaluation.

According to the classification presented in Section 5, our approach is a commit-after-stage approach. In other words, the modifications of the mobile agent are committed immediately after the stage execution. Moreover, multiple places execute the agent and also participate in the commit, which makes our approach an MMC solution.

6.1 Exactly-Once in the Context of Hetero-Places

Hetero-places run no replication mechanism among themselves (see Section 4.2.2). Indeed, hetero-places generally are competitors that have no interest in sharing any information (i.e., communicating) among each other. The hetero-places in $\mathcal{M}_i$ need not all execute the agent request; in contrary, the exactly-once execution property of the agent request needs to be enforced. Hence, if no failure and no false suspicions occur, only one hetero-place at a stage reflects the execution of the mobile agent at this stage and only its services have been invoked. However, although no replication occurs among hetero-places, the state of the agent is replicated among the agent replicas: all agent replicas know the place that has executed the agent replica.

Failures and false suspicions can lead to the execution of agent replicas on multiple hetero-places of $\mathcal{M}_i$ (see Section 4.2.3); multiple hetero-places thus reflect the modifications of agent a_i. As the exactly-once property requires that only one place has executed the agent, the modifications on all but one place need to be undone (see Section 4.3.3). Because of the lack of interaction, hetero-places cannot prevent multiple executions of the agent request themselves; they generally have no means to detect that another hetero-place has already executed the same request. Hence, the exactly-once property needs to be ensured by the agent replicas. More specifically, the agent replicas need to guarantee that at the end of the agent execution only one agent replica has exe-

cuted the local transaction, while all others undo potential executions. In our model, all agents, places, and machines are good, i.e., eventually are up forever (see Section 3.3.1). Recovering agent replicas a_i have to retrieve the identity of the place that has executed the agent replica at stage S_i. If this identity denotes a place other than the one running the recovering agent replica, all modifications by this agent replica prior to the crash are undone.

6.1.1 Handling Witnesses

Executing an agent replica on a witness does not modify the state of the witness (see Section 4.2.2). Hence, a witness behaves similarly to a stateless server and no modification on the place state by partial executions of the agent replica need to be undone. In this respect, the exactly-once execution property of the agent replicas can be violated on witnesses. However, potential accesses to third parties such as messages to the agent owner or spawning child agents need to be undoable. Hence, failures of a witness or an agent running on a witness are thus generally handled much more efficiently than on a hetero-place.

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

We have shown in Chapter 4 that replication of agents prevents blocking without depending on reliable failure detection. However, replication may lead to a violation of the exactly-once property. Informally, to ensure exactly-once, the agent replicas of stage S_i need to agree on a place that has executed the agent at S_i. This place is called the *primary* and denoted p_i^{prim}. All places that are not the primary undo possible partial executions of the agent replica. This approach guarantees that only the primary has eventually executed the agent and reflects the modifications caused by this execution.

More formally, the places of a stage have to solve an agreement problem. We specify this agreement problem in the following section.

6.2.1 Basic Agreement Problem

Despite the differences of hetero-places and hetero-places with witnesses, we give a specification of the problem that encompasses the two cases. The idea is to model the execution of each stage S_i as an *agreement problem*. By $AgrPb_i$ we denote the agreement problem of stage S_i. The problem $AgrPb_i$ is to be solved by the agent replicas[1] a_i^j running on the places in $\mathcal{M}_i$, and the solution is the decision on which all agent replicas running on the places in $\mathcal{M}_i$ agree. We denote by dec_i the decision (i.e., the solution) of $AgrPb_i$, with the following properties:

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

[1] In the place-dependent architecture mentioned in Section 6.4.1, the agreement is actually solved by the places.

- (*Uniform integrity*) Every agent replica of stage S_i decides at most once.
- (*Termination*) Every correct agent replica of stage S_i decides eventually.

The decision dec_i is as follows for hetero-places and hetero-places with witnesses (see Section 4.2.2):

Hetero-places. The decision dec_i has three parts:

1. the single place $p_i^{prim} \in \mathcal{M}_i$, called primary (see Section 4.3.2), that has executed the agent in stage S_i,
2. the resulting agent a_{i+1}, and
3. the places $\mathcal{M}_{i+1}$ for a_{i+1}.

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

The agreement problem is fundamental to enforce the exactly-once property of an agent execution. Indeed, the decision on the primary p_i^{prim} implements the commit decision presented in Section 5.1.

6.2.2 Sequence of Agreement Problems

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

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

Note that our model of fault-tolerant mobile agent execution makes no assumption about the number of failures that can be handled. Rather, the failure resilience depends on the selected algorithms to solve this problem. In the algorithm we propose in the next section, only a minority of the agent replicas per stage is allowed to fail.

6.3 Two Building Blocks Towards 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, while (2) allows to reliably forward the agent between consecutive stages. Our approach encompasses various system models such as process recovery and unreliable communication, depending on the implementation of consensus and of the reliable forwarding of agents.

Figure 6.1 depicts a fault-tolerant mobile agent execution. The execution at stage S_i consists of (1) one (or, in case of a failure or false suspicions, multiple) place(s) executing the agent, (2) the 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}.

Stage 2 in Figure 6.1 illustrates the case of a place failure. When p_2^1 detects the failure of p_2^0, which has 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 $p_2^j \in \mathcal{M}_2$. Upon recovery, p_2^0 learns the outcome of the agreement (i.e., dec_2). If $p_2^0 = p_2^{prim}$ the modifications of a_2 on p_2^0 become permanent, otherwise (i.e., $p_2^0 \neq p_2^{prim}$) they are undone/aborted.

In the following, we present the Consensus with Deferred Initial Value (DIV consensus) as building block 1 (Section 6.3.1) as well as a protocol to reliably forward the agent to the next stage (building block 2) in Section 6.3.2.

6.3.1 Building Block 1: Solving the Agreement Problem Using DIV Consensus

DIV Consensus Problem

The consensus problem is a well-specified and studied problem in 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 [CT96][2]:

[2] Actually, this is the definition of uniform consensus. However, as shown in [Gue95], 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.

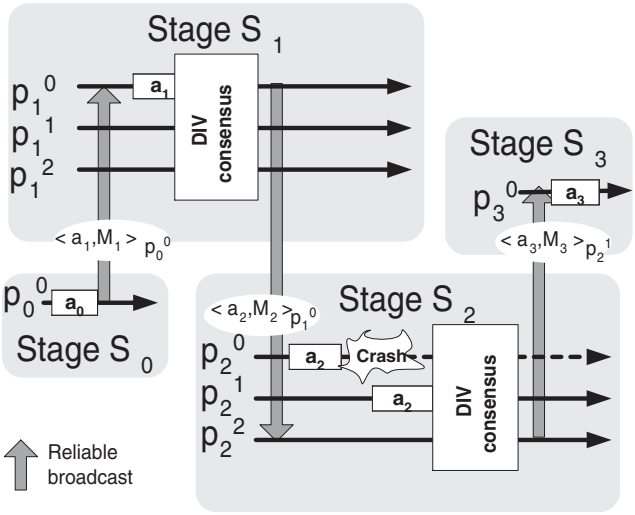

Figure 6.1: Fault-tolerant mobile agent execution with p_2^0 failing. An erroneously suspected place p_2^0 leads to the same situation. The notation $< a_{i+1}, \mathcal{M}_{i+1} >_{p_i^{prim}}$ means that p_i^{prim} has executed agent a_i (which leads to a_{i+1} and M_{i+1}).

- *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.

[CT96] solves the consensus problem with the unreliable failure detector $\diamond S$ and a majority of correct processes. DIV consensus [DSS98] modifies the consensus problem such that all processes need not have an initial value (see Section 2.2.3). 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 function (or handler) $\mathcal{GIV}()$ that allows the protocol to compute the initial value only when needed.

Applying DIV Consensus

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

Initial function $\mathcal{GIV}()$. The initial function $\mathcal{GIV}()$ 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 a_i^j running on $p_i^j \in \mathcal{M}_i$ agree on the p_i^{prim} that has executed a_i, on the new agent a_{i+1}, as well as on the places of the next stage S_{i+1}.

DIV consensus as given in [DSS98] assumes reliable communication channels. As stated in [DSS98], the algorithm can easily be extended to handle also unreliable communication channels by using an approach along the lines of [ACT97]. Moreover, it makes the assumption that a majority of a_i^j does not fail, i.e., is correct. In our system model, agents, places, and machines are good (see Section 3.3). However, this assumption is not needed for the agent replicas. In other words, the termination of the agreement does not depend on the recovery of the agents. Rather, we assume that a majority of them does not fail. When they recover, they do not participate in consensus any more. 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 (see also Section 3.3.1).

However, the protocol presented could easily be extended to also encompass recovery by using a corresponding version of consensus along the lines of [ACT00, OGS97, HMR98]. Indeed, we argue in the next section that recovery needs to be supported to a certain degree because of asynchronous agent propagation.

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

Asynchronous Agent Propagation

We have assumed (see Section 3) an asynchronous system, where there is no bound on the transmission delay of messages. This has an impact on the different instances of the agreement protocol (i.e., DIV consensus) that run at each stage S_i of an agent execution. Because of the asynchrony, the agent a_i may not arrive simultaneously at the different places p_i^j of stage S_i. Assume for instance that the agent replicas a_i^0, a_i^1, a_i^2 are sent respectively to $p_i^0, p_i^1, p_i^2 \in \mathcal{M}_i$ (see Figure 6.1), and assume that a_i^2 arrives late at p_i^2. DIV consensus may have already started executing for agent replicas a_i^0 and a_i^1 when a_i^2 arrives. The execution of DIV consensus may even have terminated when a_i^2 arrives. In theory, the late arrival of a_i^2 at p_i^2 is indistinguishable to a_i^0 and a_i^1 from the crash of a_i^2 before a_i^2 has received any messages from a_i^0 or a_i^1, followed by the recovery of a_i^2. Indeed, the late arrival is a special case of general recovery, where a_i^2 fails and recovers again

preserving the same state as before its failure, and receiving all messages sent to it in the meantime. To handle general recovery a mechanism is required that stores the actual state of a_i^j before its failure and retrieves this state again upon recovery. Moreover, DIV consensus messages need to be sent reliably in this context. In our approach, we support the latter, but not the former and thus distinguish between the late arrival of a_i^j and a crash of a_i^j followed by recovery. Indeed, late arriving agent replicas can still participate in DIV consensus, while recovered agent replicas do not. Late arrivals are handled correctly if all messages are ensured to eventually arrive at the agent replica, although the latter may not have reached the place yet.

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

Having solved the problem of executing the agent at a stage, we must address the issue of reliably forwarding the agent to the next stage. A naive approach leads to a protocol, where every place in $\mathcal{M}_i$ broadcasts the result dec_i to every place in $\mathcal{M}_{i+1}$. However, this incurs significant overhead in terms of message number as well as number of communication steps[3], depending on the protocol selected. Our approach reduces this overhead considerably. For this purpose, only a majority of the places in $\mathcal{M}_i$ broadcast to all places in $\mathcal{M}_{i+1}$. As DIV Consensus assumes that a majority of places in $\mathcal{M}_i$ do not fail, it is ensured that at least one place actually broadcasts the agent.

6.3.3 Optimization: Pipelined Mode

In our discussion so far, we have assumed that $\mathcal{M}_{i-1}$ and $\mathcal{M}_i$ are a disjoint set of places. However, this is not a requirement. On the contrary, reusing places of stage S_{i-1} as witnesses for S_i (see Section 4.2.2) improves the performance of the protocol and prevents high messaging costs [SR98]. Generalizing this idea, every stage S_i merely adds another place to $\mathcal{M}_{i-1}$, while removing the oldest from the set $\mathcal{M}_{i-1}$. More specifically, at stage S_2, agent a_2 executes on the hetero-place p_2 and on the witnesses p_1 and p_0. If no failures or false suspicions occur, the agent executes on p_2. It is then forwarded to the places in $\mathcal{M}_3$, i.e., p_1, p_2, and p_3. Place p_3 is the only place to still receive a_3, as p_2 and p_1 have been participants to the agreement at S_2 and thus already know a_3 as part of dec_2. We call this mode *pipelined* [RS98]. In the pipelined mode, forwarding costs are minimized and limited to forwarding the agent to the new place (see Figure 6.2). Note that for set $\mathcal{M}_1$ we assume the existence of one place that acts as a witness for the stage execution [4]. The execution at stage S_0 is not replicated (see Section 4.2.1) and no witnesses are needed for $\mathcal{M}_0$.

Note that generally any other place can act as witness. Indeed, Johansen et al. [JMS+99] suggest to reuse the *oldest* places in $\mathcal{M}_i$ as witnesses in $\mathcal{M}_{i+1}$, because these places have not failed for the longest period of time. To push this reasoning even further, one could suggest to have multiple dedicated witness machines in the network, that can take on the role of a witness. At any stage S_i, a set of this witness machines is in $\mathcal{M}_i$. However, this approach violates the autonomy property of a mobile agent to a certain extent and may introduce bottlenecks.

[3] 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.

[4] If p_0 is only intermittingly connected to the network, p_0 is replaced by some other witness.

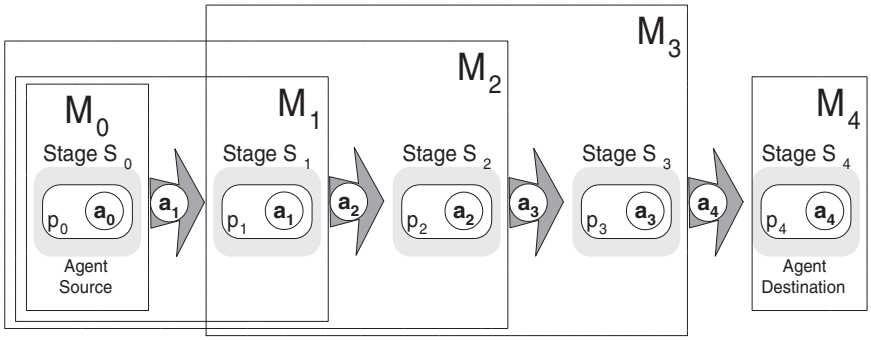

Figure 6.2: Pipelined mode without failures.

6.4 FATOMAS

In this section, we present an implementation of the algorithm introduced in Section 6.3, called FATOMAS (FAult TOlerant Mobile Agent System). We first give the architecture (Section 6.4.1), before discussing some implementation issues (Section 6.4.2). Our implementation is written in Java and builds on ObjectSpace's Voyager mobile agent platform (see Section 2.1.7). In Section 6.4.3, we show the results of the performance evaluation of FATOMAS. To validate our design, we have also implemented FATOMAS on MOPROS (see Section 2.1.7) and briefly present these results in Section 6.4.4. To improve readability in this section, method names are written with a `special font`.

6.4.1 Architecture

Isolation of the Fault Tolerance Mechanisms

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

Ideally, fault tolerance should be orthogonal to mobile agents and its mechanisms transparent to the agent owner. Unfortunately, complete transparency is difficult to achieve and the user-defined agent, i.e., the part that defines the application-specific operations of the agent, needs to interact with the fault tolerance mechanisms. While in single-agent execution, for instance, an agent just needs to specify the next place it moves to, our fault-tolerant agent execution generally[5] requires a set of destination places for the next stage ($\mathcal{M}_{i+1}$). Clearly, the agent is aware of the replication and complete transparency is no more possible. Moreover, ensuring fault tolerance adds another phase to the agent execution: the *commit/abort phase*. In Section 4.2, we mention that imperfect failure detection may lead to a violation of the exactly-once property of mobile agent execution. Solving an agreement problem prevents multiple executions of the agent by deciding

[5]Except in the particular case of the pipelined mode (see Section 6.3.3).

on a primary p_i^{prim}. Only the primary commits the operations, while all other places that have executed the agent as well must abort/undo the agent operations. Hence we need a commit/abort phase, which follows every stage operation phase. The semantics of this phase depends on the agent operations. For instance, database transactions need to be committed or aborted (or rolled back, depending on the database), while idempotent operations generally do not require any further action.

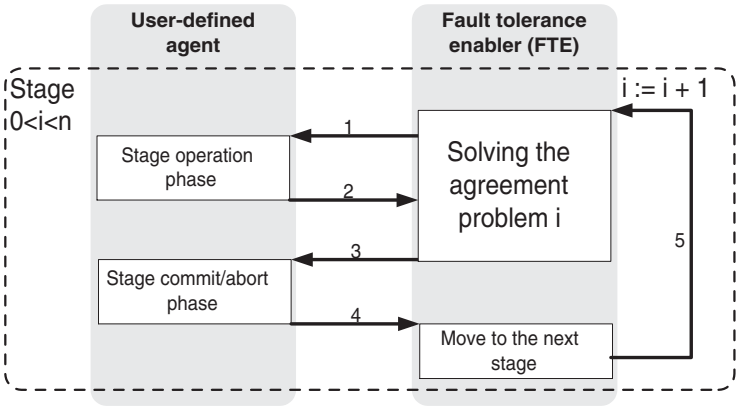

Figure 6.3: Phases of a fault-tolerant mobile agent execution and interaction with the FTE.

We propose an architecture that isolates the fault tolerance mechanisms in a component called *Fault Tolerance Enabler* (FTE). Figure 6.3 shows the flow of interaction between the FTE and the user-defined agent. This interaction occurs during the stage operation and the commit/abort phase. The FTE groups the fault tolerance mechanisms, while the user-defined agent contains the application-specific part. At each stage S_i $(0 < i < n)$, the FTE solves the agreement problem specified in Section 6.2.1. Depending on the outcome of the agreement, the operations performed in the stage operation phase (see Figure 6.3, arrows 1 and 2) are either committed or aborted (arrows 3 and 4). Finally, the FTE moves the agent to the set of places in $\mathcal{M}_{i+1}$ (arrow 5), which are computed by the user-defined agent and returned as the result of the stage operation phase (see Section 6.3).

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

Agent-Dependent Approach

In the agent-dependent approach, the FTE is integrated into the agent and travels with it. Only one instance of the FTE exists per agent, but it is replicated when the agent is replicated. The FTE is initialized by the user-defined agent at the agent source, and triggers the termination phase of the user-defined agent at the agent destination. The interaction of a user-defined agent with the FTE creates a fault-tolerant mobile agent. Hence, the replication mechanisms are transparent

to the places; the agent appears to the place as a normal agent. Consequently, existing mobile agent platforms do not need to be modified. However, we redefine the way agents are created and moved. Instead of programming the agent against the proprietary mobile agent platform API, the agent uses the functionality of the FTE-API (see Figure 6.4). The FTE then addresses issues such as fault tolerance and mobility. For instance, in ObjectSpace's Voyager mobile agent platform [Obj99], an agent moves with a call to the `move` method. Beside the destination, the `move` method also accepts the name of the method to be called upon arrival on the destination place. In our approach, the callback method `doStageOperation` in the FTE-API is invoked whenever the agent arrives at a new destination. The next destination places are returned as a result of the execution of method `doStageOperation`. These changes are straightforward and, in our opinion, simplify the notion of an agent.

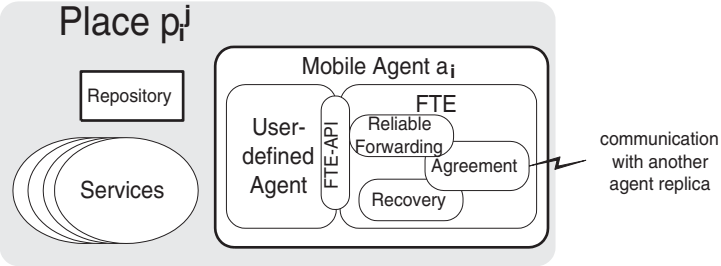

Figure 6.4: Agent-dependent approach: architecture of the fault-tolerant mobile agent framework.

Figure 6.4 shows the architecture of the agent-dependent approach. The FTE is composed of an *agreement* component (implementing consensus), a *reliable forwarding* component (responsible for the agent forwarding to the next stage), and a *recovery* component. The latter handles the recovery in case the agent fails or arrives at a place when the decision has already been made. Finally, the *repository* is a location where place specific fault tolerance information can be stored temporarily. This location is agent platform dependent, but typically corresponds to some sort of local repository, such as the Voyager directory. For convenience, we require that such repositories allow other agents at place p_i to remotely access some information at another place p_k $(k \neq i)$. If this is not the case, an agent needs to be defined that acts as a proxy between the local directory and the fault-tolerant agents.

Place-Dependent Approach

In the place-dependent approach, the FTE is provided by the mobile agent platform, e.g., [MPT00, SBS00] (see Figure 6.5). Here, fault tolerance is built into the places, and a new instance of the FTE is created and executes at every stage of the agent execution. A disadvantage of the place-dependent approach is the need to modify existing proprietary mobile agent platforms. In particular, the installed base of mobile agent platforms needs to be replaced by platforms that all use the same fault tolerance mechanisms, which is problematic. Moreover, providing the fault tolerance mechanisms locally on a place may lead to versioning problems.

On the other hand, the FTE can be reused if two agents a and b execute on a similar set of

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

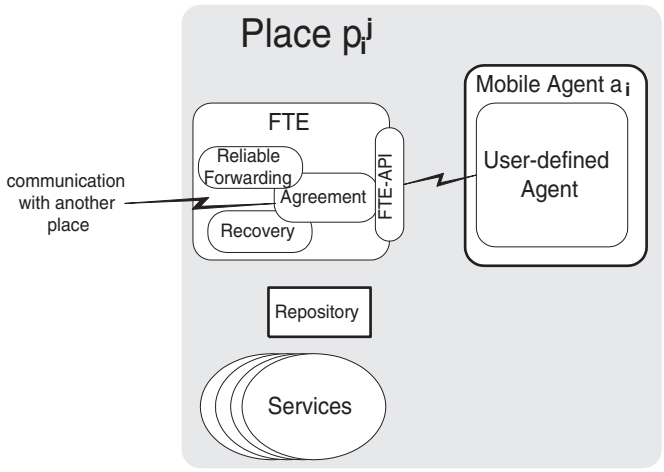

Figure 6.5: Place-dependent approach: architecture of the fault-tolerant mobile agent framework.

Moreover, the place-dependent approach can reduce the number of messages compared to the agent-dependent approach. Indeed, implementing the FTE as part of the mobile agent platform allows to integrate parts of building block 1 (i.e., DIV consensus) with building block 2 (i.e., reliable forwarding). For this purpose, we take advantage of the reliable broadcast used as part of the DIV consensus algorithm (see Section 2.2.3) 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.

Despite these advantages of the place-dependent approach, we weighted the need to modify existing agent platforms as a serious limitation and consequently have chosen the agent-dependent approach.

6.4.2 Implementation on Voyager

This section describes the implementation of FATOMAS. As indicated in the agent-dependent approach in Section 6.4.1, we build fault tolerance on top of an existing mobile agent platform, without modifying existing code. We use ObjectSpace's Voyager v3.1.2 [Obj99] as the Java mobile agent platform. In the following, we first discuss the implementation of the FTE, then explain the problem of deadlock and show a way to prevent it.

FTE

The interface of the FTE to the user-defined agent, i.e., the FTE-API (see Figure 6.4), consists of a single method, called `start`. Invoking this method causes the FTE to take over the responsibility for executing the agent. As shown in Section 6.4.1, the FTE interacts with the user-defined agents in the stage operation phase and in the commit/abort phase. For this purpose, the user-defined agent implements the callback method `doStageOperation`, which represents the stage operation phase and returns the set of destination places at the next stage. The callback methods `commit` and `abort` implement the commit/abort phase. The next subsection discusses in more details how a stage execution works.

Stage Execution: A stage execution works as follows (see Figure 6.6): On arrival on place p_i^j, the agent replica a_i^j (more specifically the FTE) immediately starts executing consensus.

The communication among the agent replicas a_i^j is currently based on the communication means of Voyager, more specifically VoyagerSpaces for broadcasts, and Voyager remote method invocation for point-to-point communication [Obj99]. Remote method invocations are synchronous calls in FATOMAS and return an exception if the communication link is broken or the receiver is not available. These exceptions are caught and the message is confided to a dedicated sender thread that keeps on resending the message until it is successfully delivered or the stage execution has terminated. Replica agents that arrive when the other replica agents are already done with the stage execution run the recovery protocol.

When the consensus algorithm decides, the FTE stores the decision value in a local repository (see Figure 6.4). Actually, only part of the decision is stored, i.e., the primary's ID p_i^{prim}. This information must be kept until all participants in a stage execution, i.e., $\mathcal{M}_i$, are aware of the result. In particular, participants that have crashed during consensus and are assumed to recover again need to learn about the primary to decide whether to commit or abort the agent's operation on their place. However, it is not necessary to forward the agent to the next stage, as the agent execution may well have terminated in the meantime. After a certain time, the decision value is discarded. Selecting a timeout value that is sufficiently large, the probability of erroneously discarding an entry becomes very small and thus negligible.

Having stored the decision value in the repository, the FTE either calls `commit` or `abort` on the user-defined agent, depending on the decision value, more specifically, on the p_i^{prim} value of the decision (see Section 6.3). Finally, the FTE forwards the agent to the next stage as described in the next section.

In our implementation, the decision value contains the entire state of the mobile agent. This state is captured using the Java serialization mechanisms [Fla97]. Actually, it is sufficient to agree only on the modifications to the state. This significantly reduces the size of messages exchanged among the agent replicas during DIV consensus. However, this requires support from the agent developer. Indeed, the user-defined agent must specify the state that has changed. Hence, fault-tolerance becomes less transparent for the agent developer. This optimization is not implemented in the actual version of FATOMAS.

Reliably Forwarding the Agent: Having solved consensus at stage S_i, the agent needs to be forwarded reliably to members M_{i+1} of the next stage. To assure reliable forwarding, each partic-

1: **Stage action of a_i^j on intermediate place p_i^j:**

2: Initialisation:
3: $decision \leftarrow \perp$ {*contains the decision of DIV Consensus*}

4: $decision = DIV\ Consensus.propose(a_i^j)$
5: $LocalRepository.a_i \leftarrow decision.primary$ {*store the primary in the local repository*}
6: **if** $primary = p_i^j$ **then**
7: commit local transaction {p_i^j *is the primary*}
8: **else**
9: abort local transaction
10: reliably multicast $decision.a_{i+1}$ to places in $decision.\mathcal{M}_{i+1}$

Figure 6.6: Pseudo code of the stage execution of the agent a_i^j at place p_i^j.

ipant of stage S_i sends a clone of the agent to the participants in $\mathcal{M}_{i+1} \setminus \mathcal{M}_i$.

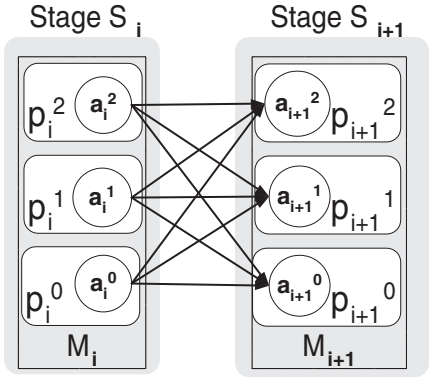

Figure 6.7: Reliably forwarding the agent from S_i to S_{i+1}.

If $\mathcal{M}_{i+1} \cap \mathcal{M}_i = \emptyset$, i.e., the places at two consecutive stages S_i and S_{i+1} form disjoint sets, the simplest solution to reliable forwarding consists in sending $|\mathcal{M}_i| * |\mathcal{M}_{i+1}|$ agents (see Figure 6.7). However, to reduce the communication overhead, we chose the following optimistic approach: the agent a_{i+1} is sent to each places in $\mathcal{M}_{i+1}$ only by the agent a_i at the place p_i^{prim}. The other agents $a_i^j \neq a_i^{prim}$ simply verify whether the agent a_{i+1} has arrived at the places in $\mathcal{M}_{i+1}$ by remotely accessing the corresponding value in the repository on the places in $\mathcal{M}_{i+1}$. If an entry for the agent a_{i+1}^j already exists, the agent a_{i+1}^j has successfully arrived, otherwise, the agent a_{i+1}^j is cloned and sent to this place. In other words, instead of a priori always sending the entire agent, a small message is sent to check the need for sending the entire agent. This approach is optimistic, since it assumes that in most cases the agents arrive at their destinations. Although the performance gain for a single agent is not great, the communication overhead is reduced for large agents, as discussed in Section 6.4.3. If an agent fails to arrive at its destination, because either (1) the sender place failed or (2) the agent was lost during transmission, agent forwarding leads to

additional latency.

Recovery: Although recovery and non-simultaneous agent arrival can be handled in the same way (see Section 6.3.1), our prototype distinguishes between the two cases: a delayed agent takes part in the running instance of consensus (except if consensus has finished already), whereas a recovering agent does not. A recovering agent a_i^k requests the decision value of the consensus, more specifically p_i^{prim}, once it is available. Based on p_i^{prim}, a_i^k either commits or aborts its stage operations and can thus recover into a consistent state.

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

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

A deadlock may occur between two different agents a_i and b_k, if they happen to share the same places. This deadlock is a consequence of the interaction between the DIV consensus algorithm and the stage action. Consider for example the two replicas a_i^0, a_i^1 of agent a_i and the two replicas b_k^0, b_k^1 of agent b_k, were a_i^0, b_k^0 share the place p_i^0 and a_i^1, b_k^1 the place p_i^1 (Figure 6.8).[6] Assume further that agent a_i performs stage action sa_i (either on p_i^0 or on p_i^1), and b_k performs stage action sa_k (either on p_i^0 or on p_i^1), and that the two stage actions sa_i and sa_k access the same data item. Accessing the data item requires locking of the data, and the data must remain locked until the decision of the consensus of stage S_i (for agent a) or S_k (agent b) is known.

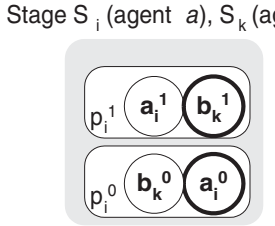

Figure 6.8: Deadlock between the agents a and b at stage S_i and S_k, respectively.

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

- Because of the distributed consensus algorithm, we can have $Thr(a_i^0) \rightarrow Thr(a_i^1)$, where $\rightarrow$ stands for "waits-for".[7]
- For the same reason, we can have $Thr(b_k^1) \rightarrow Thr(b_k^0)$.

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

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

- Because of the data locking, we can have $Thr(b_k^0) \rightarrow Thr(a_i^0)$ (if $Thr(a_i^0)$ locks the data item of place p_i^0 before $Thr(b_k^0)$).
- For the same reason, we can have $Thr(a_i^1) \rightarrow Thr(b_k^1)$.

We have here a cycle in the wait-for-graph, i.e., a deadlock, as shown in Figure 6.9 (a). Deadlocks can be prevented by having two threads per agent a_i^j: one thread, denoted by $ConsThr(a_i^j)$, for executing the consensus on place p_i^j, and another thread, denoted by $OpThr(a_i^j)$ for executing the stage action on place p_i^j. The above wait-for dependencies become:

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

Obviously the cycle in the wait-for-graph, and the deadlock, have disappeared. The modifications to DIV consensus are shown in Algorithm 1 (Appendix A.1). While this approach prevents deadlocks, timed locks[8] in data accesses allow us to detect them. The advantage of timed locks is that they only require one thread for executing consensus on p_i^j and performing stage action sa on p_i^j.

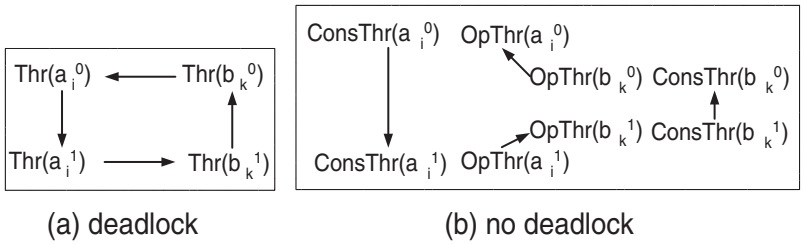

Figure 6.9: Wait-for-graphs for (a) the single thread and (b) the double thread case.

6.4.3 Performance Measurements on Voyager

This section evaluates the performance of FATOMAS on Voyager. Our evaluation uses the example agent presented in the next section. The results first show the costs of the replication mechanisms. In a second evaluation suite we identify the influence of the size of the agent. For this purpose, we compare the costs of a replicated agent with the costs of a single, non-replicated agent. Finally, we show the benefits of some optimization techniques, such as the pipelined mode and the agent forwarding optimization presented in Section 6.4.2.

Example: A Fault-Tolerant Mobile Agent Accessing Counters

To measure the performance of FATOMAS, we use a simple service running on every place: a counter. Accesses to this counter are performed as local transactions, via the three methods:

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

`increment` (to increment the value of the counter), `commit` (to commit the modifications), and `abort` (if the modifications need to be undone). A call to method `increment` locks the counter; the lock is only released after a call to either `commit` or `abort`.

Figure 6.10 shows a simplified version of our agent, which increments the value of the counter at each stage of its execution. It implements the methods defined in the interface `IFtAgent`: method `doStageOperation` corresponds to the stage operation phase in Figure 6.3, methods `commit` and `abort` implement the commit/abort phase.

```
public class ReplAgent
             implements IFtAgent, Serializable {
   public ReplAgent (int myId,
                     boolean initialAgent) {
    if (initialAgent) {
        FTE fte = new FTE() ;
        fte.start(_myId,
                  this,
                  _itinerary.elementAt(1)) ;
    } // if
  } // <init> ()

  public String[] doStageOperation(int stage) {
    if (stage+1 == _itinerary.size()) {
      // Agent destination: report to user
      _testControler.report() ;
      return IFtAgent.NO_NEXT_STAGES ;
    } else {
      // Lookup the counter
      _counter = (ICounter)Namespace.lookup(counterName) ;
      _counter.increment() ;
      return (String[])_itinerary.elementAt(stage+1) ;
    } // if
  } // doStageOperation ()

  public void commit (int stage) {
    if (stage+1 < _itinerary.size()) {
      _counter.commit() ;
    } // if
  } // commit ()

  public void abort (int stage) {
    if (_counter != null) {
      _counter.abort() ;
    } // if
  } // abort ()
} // END CLASS ReplAgent
```

Figure 6.10: A user-defined agent that increments the value of a counter at each stage.

Our test consists in sending a number of agents that increment the value of the counter at each stage of the execution. Each agent starts at the agent source and returns to the agent source, which allows to measure its round-trip time. Between two agents, the places are not restarted. Conse-

quently, the first agent needs considerably longer for its execution, as all classes need to be loaded into the cache of the virtual machines. Consecutive agents benefit from already cached classes and thus execute much faster. We do not consider the first agent execution in our measurement results. For a fair comparison, we used the same approach for the single agent case (no replication).

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

Experimental Setup

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

Each stage S_i $(0 < i < n)$ is composed of three places. In the experiment with two intermediate stages S_x and S_y, each intermediate stage uses three AIX machines, while another machine hosts the agent source and destination. If the number of stages exceeds four, S_x corresponds to all odd intermediate stages, i.e., S_{2k+1} $(2k + 1 \leq n - 1)$ and S_y to the even intermediate stages S_{2k} $(2k \leq n - 1)$.

Costs of Replication

We measure two aspects of the replication costs: (1) the overhead of the replication mechanism, by considering replication degree 1, and (2) the costs of replication degree 3. The replication degree denotes the number of places at a stage and is an indicator of the number of failures the algorithm tolerates at a stage. Because of the assumption for our consensus algorithm, i.e., a majority of agent replicas do not fail (see Section 6.3.1), replication degree 3 handles one failure, and replication degree 5 would handle 2 failures. The results of these measurements are given in Table 6.1. They represent the arithmetic average of 10 runs, with the highest and lowest values discarded to eliminate outliers. The coefficient of variations is in most cases much lower than 5%. However, for very few results, it went up to 15%. As a mobile agent execution combines agent forwarding and consensus, minor variations on network load and load on the AIX machines have a considerable influence on the execution time of a mobile agent.

The first line in Table 6.1 shows the costs for a single agent, i.e., a traditional Voyager agent, that performs exactly the same task as the replicated agent. The single FTE-agent (line 2 in

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

Table 6.1: Costs of replication degree 1 and 3 in milliseconds compared to the single agent.

Table 6.1) uses the replicated agent's code to execute in a single agent mode. Compared with the previous line, the second line shows the overhead of the replication mechanism (increased agent state adding to the communication costs, increased computing time). The results show that the replication mechanisms add about a 30% overhead compared to a single agent. The overhead is lower (18%) in the case of three stages, as no communication between intermediate stages occurs.

A replicated agent that is able to tolerate one failure at a stage is three to four times more expensive than a single agent (line 3). The increase in the agent execution time is mainly caused by the additional communication costs of agent forwarding and consensus. Indeed, consider for instance the single agent execution on 4 stages: there are here 3 messages in the critical path. On the other hand, with replication there are 11 messages in the critical path in the most favorable scenario. We suspect Voyager communication to be rather inefficient. Nevertheless, the overhead of the fault tolerance mechanisms seems reasonable, considering the guarantees the fault tolerance mechanisms provide: non-blocking and exactly-once mobile agent execution. Moreover, in our experiment the execution time of the agent's stage action is less than 5ms and therefore not significant. Clearly, the larger the execution time of the agent's stage action is, the smaller the ratio of the overhead between the single agent and the replicated agent becomes.

Finally, the last line shows the execution costs when the coordinator fails. For this purpose, we force agent replica a_i^1 to crash in exactly the situation presented in Figure 6.1 (stage 2). The main part of the costs stems from the selected timeout value in the failure detection mechanisms for consensus (timeout = 10000ms). In other words, the failure detection mechanism[9] suspects agent replica a_i^1, if it has received no message from a_i^1 for a certain time, that corresponds to this timeout value. A more aggressive timeout value allows to considerably speed up the agent execution, but increases the risk of false suspicions.

Table 6.2 indicates an upper bound on the costs that can be attributed to consensus. It shows the costs of instantiating the consensus objects and of running the consensus algorithm. The test measures the costs of consensus at the intermediate stage of a mobile agent execution with three stages. The agent source sends the agent sequentially to the three places of the intermediate stage. As in agent forwarding between intermediate stages one can assume some degree of concurrency these results show an upper bound value. They highlight a significant difference of the costs among the agent replicas, caused by asynchronous agent arrival as discussed in Section 6.3.1 and by the intrinsic properties of our consensus algorithm. Indeed, agent replica a_i^1 is the first coordinator and also generally arrives first on its place in $\mathcal{M}_i$. As shown in Table 6.2 it also decides first (if no failures or suspicions arise). Hence, its performance is better than the performance of a_i^0 and a_i^2.

The results confirm the obvious expectation that the size of the agent has an impact on the

[9] We have chosen a very basic implementation of a failure detector. See [CTA02] for a more elaborate discussion.

costs of consensus. We discuss this impact in more detail in the next subsection.

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

Table 6.2: Costs of consensus in milliseconds for a replicated FTE-agent of degree 3.

A particular case arises for small agents (first line in Table 6.2). Because of asynchronous agent propagation, a_i^1 and a_i^2 solve consensus and are forwarded to the next stage before a_i^0 establishes the communication with them. Therefore, a_i^0 uses the recovery mechanisms to learn about the result of consensus. This explains the high value for a_i^0, which is mainly caused by timeouts. These timeouts encompass the timeout for the failure detection mechanism and the timeout in the recovery mechanism, that checks for a value in the repositories of p_i^1 and p_i^2.

Influence of the Size of the Agent

As already indicated in the previous section, the size of the agent has a considerable impact on the performance of the fault-tolerant mobile agent execution. To measure this impact, the agent carries a Byte array of variable length, that is used to increase the size of the agent state. As the results in Figure 6.11 show, the execution time of the agent increases linearly with increasing size of the agent. Compared to the single agent, the slope of the curve for the replicated agent is steeper. Table 6.2 indicates the part of the costs that can be attributed to consensus. For instance, with the agent size of 11440 bytes, consensus needs 3.2 seconds at each intermediate stage (see Table 6.2). Figure 6.11 indicates that the costs for the entire agent execution is 13.4 seconds. From this we conclude that the communication overhead is about 7 seconds.

Optimizations

We present now two optimizations: pipelined mode and forwarding optimization.

Pipelined Mode: We have briefly introduced the pipelined mode in Section 6.3.3. It results in a reduced number of messages (i.e., forwarded agents), as the agent only needs to be forwarded to one new place of the next stage. This reduced number of messages does not entirely show in the performance gain, because our approach waits only for the reception of the first message/agent. Reducing the number of messages, however, has a great impact on the underlying communication infrastructure.

Nevertheless, Figures 6.12 and 6.13 show that the pipelined mode has a lower execution time than the normal replicated agent. While Figure 6.12 shows the performance gain with an increasing number of stages, Figure 6.13 indicates that the performance gain increases with increasing agent size, as one would expect. In this test, we have used an agent that visits 8 stages, including agent source and destination.

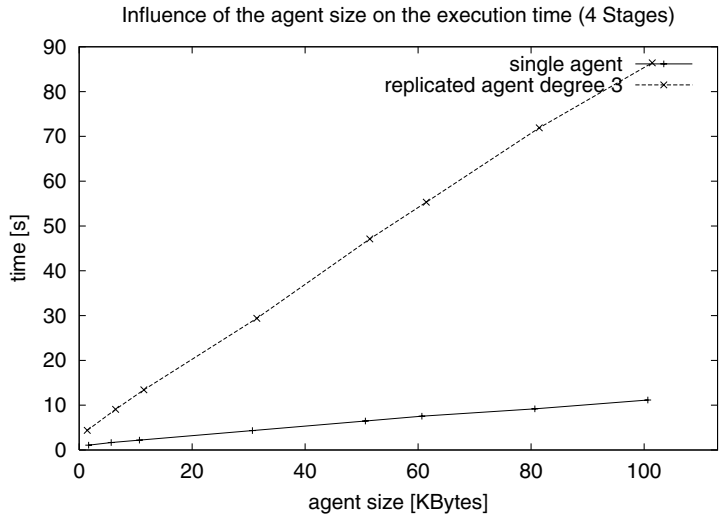

Figure 6.11: The costs of single and replicated agent execution with increasing agent size (4 stages).

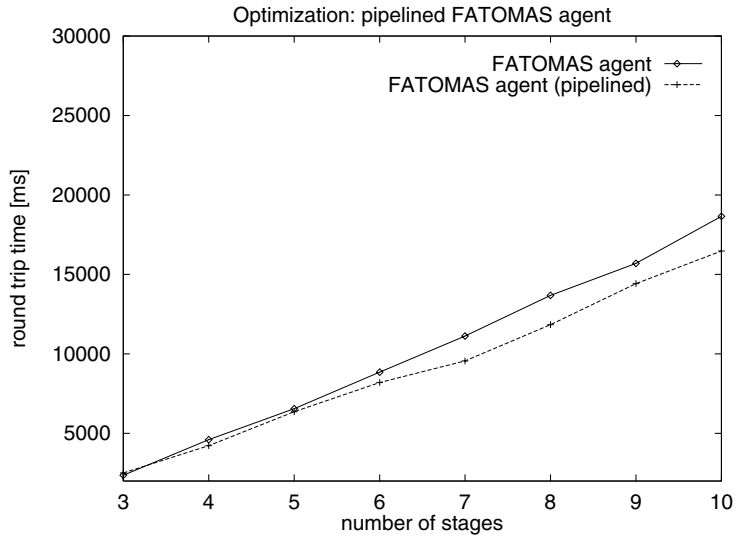

Figure 6.12: Performance of the pipelined mode with increasing number of stages.

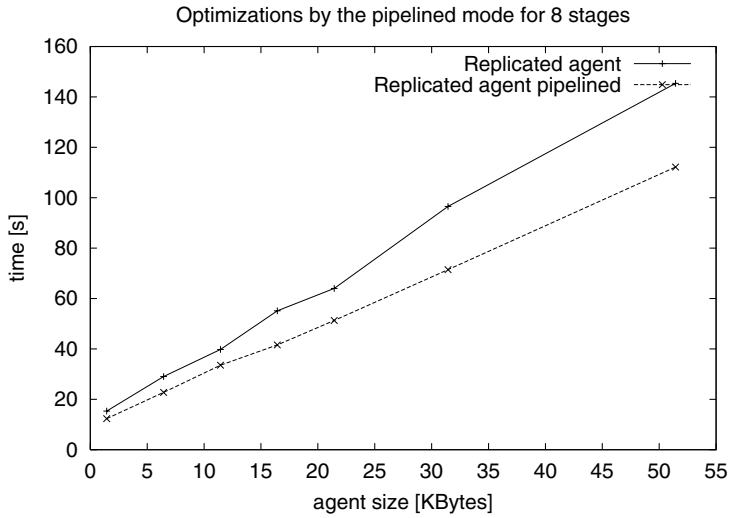

Figure 6.13: Performance gain with the pipelined mode for 8 stages with increasing agent size.

Forwarding Optimization: Similar to the pipelined mode, this optimization addresses also the communication overhead (Section 6.4.2). Although the number of bytes transfered is reduced with this approach, in particular for large agents, the performance gain for the agent execution itself is small (see Figure 6.14), as our algorithm waits only for the arrival of the first copy of the agent.

6.4.4 Implementation and Performance Measurements on MOPROS

To validate our architecture, we have ported FATOMAS to MOPROS (see Section 2.1.7). Porting FATOMAS to MOPROS is easy and shows that no modifications are needed to the mobile agent platform and has thus verified our claim of Section 6.4.1. We start with a brief description of our FATOMAS implementation on MOPROS, before presenting our performance evaluation.

FATOMAS on MOPROS

In contrary to our implementation on Voyager, the Java classes are not locally available on the places any more. Rather, they are loaded from a central location. Consequently, the performance of the two implementations is not really comparable. The purpose of the implementation on MOPROS was to validate our architecture and our claim that porting FATOMAS to another mobile agent platform is straightforward.

The execution of the fault-tolerant mobile agent is not as transparent as in Voyager any more. Upon arrival on a new place, MOPROS always calls the `main` method in the user-defined agent. Consequently, it is the user-defined agent's responsibility to call the corresponding method in the FTE. Recall that Voyager allows us to specify the function to be called upon arrival on a new place. This allows the FTE to take over the execution of the agent (see Section 6.4.2). In MOPROS, the user-defined agent must implement a dispatcher within its main method, that dispatches the thread of control to the corresponding function in the FTE.

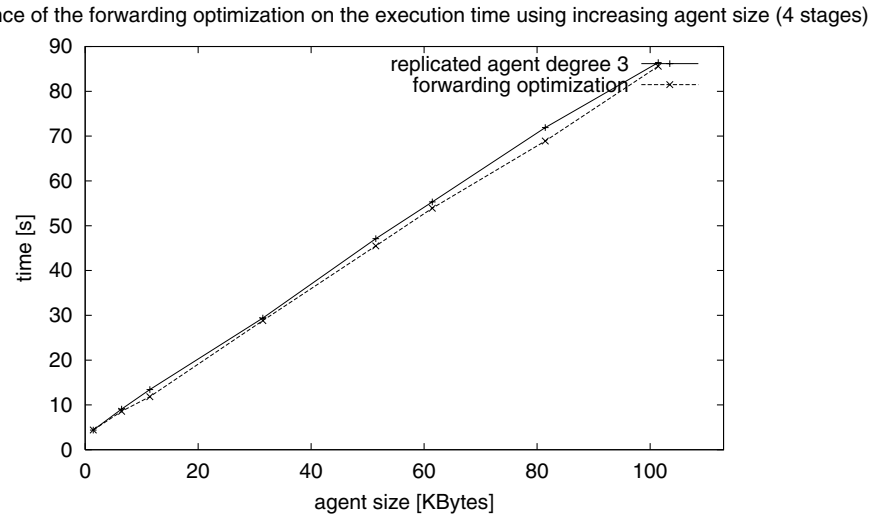

Figure 6.14: The costs of a replicated agent execution with forwarding optimization for increasing agent sizes (4 stages).

As MOPROS is targeted towards Linux, we have used this platform to measure the performance of the FATOMAS implementation on MOPROS.

Performance Measurements

Our prototype implements an example agent similar to the one presented in Section 6.4.3. All classes of the agent are loaded remotely, through a central web server.

Our performance tests are run on three PL300 machines (Pentium II 400MHz, 128MByte RAM), one Thinkpad T21 (Pentium III 800MHz, 400MByte RAM), and one machine with a 1GHz processor and 768MByte of RAM. All of them are running Linux. The machines are connected by 100Mbps Ethernet. These machines are grouped into two sets of three machines each, with machine m participating in both sets and acting as the agent source and destination. The machines are arranged within the set such that the agreement is generally reached among the places that run on the machines other than m.[10] This allows us to include the communication overhead among distributed machines in our measurements.

The results of the performance evaluation are shown in Figure 6.15. They represent the average for ten runs, whereby the highest and lowest results have been ignored to eliminate outliers. The coefficient of variation is generally below 5%. Recall that MOPROS is a prototype system in a very early stage of development and thus its performance is not optimized. However, it allows us to validate FATOMAS' architecture and to test our approach on a system that provides support for remotely loaded class files.

The results show the overhead of the replication mechanism compared to the execution of a single agent. For illustration, we also show the overhead of a single agent whose Java classes

[10]Only a majority of places in $\mathcal{M}_i$ is needed to reach an agreement. Depending on the order in $\mathcal{M}_i$, this majority can be achieved without m (see Section 6.3.1).

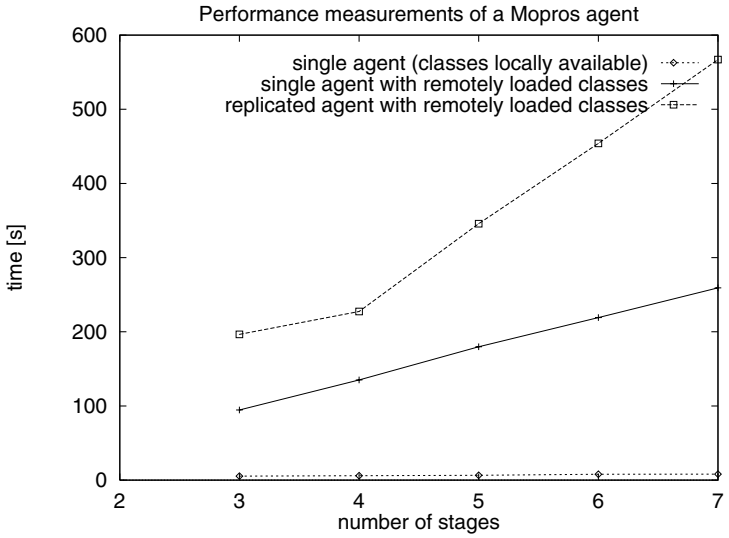

Figure 6.15: The costs of single and replicated agent execution with increasing number of stages.

are locally available on the places. The performance difference between this agent and the single agent with remotely loaded classes thus stems from the overhead of remote class loading. This overhead indicates an upper bound, as each class is loaded separately. Assembling all classes into a Jar file allows to load them all at the same time and is thus more efficient. In the execution of a replicated agent with 5 stages, Figure 6.15 shows a significant increase of the overhead. This increase is caused by the agent's return to the machines of predecessor stages, while these machines still have not completely cleaned up all the leftovers from previous stage executions. The cleaning-up at stage S_i $(i \geq 0)$ thus interferes with the execution of the agent at stage S_{i+2w}, where $w = 0, 1, 2, 3, \ldots$ and $(n \geq i + 2w)$.

6.5 Child Agents

So far, we have not discussed the case of spawning a child agent. A child agent is spawn when agent a_i creates a new agent during its stage execution. In other words, a_i becomes an agent owner. This case is more difficult to handle than the normal case of a linear mobile agent execution.

6.5.1 Spawning Child Agents

The agent a_i at stage S_i can spawn a new agent b, which causes two agents to move off stage S_i^a: a_{i+1} resulting from executing a_i on S_i^a and b_1 (see Figure 6.16). If a_i crashes, then all its modifications have to be undone. In particular, the spawn agent b has to be terminated as well and its operations undone. If b_1 has been immediately sent off, undoing its operations is not a simple task. Indeed, the undoing message may trace agent b_1 forever, never catching up with it (see Section 4.3.3). Therefore, b can only start execution when the current stage S_i^a has decided on the result. This requires that b_1 and its set of executing places $q_1^j \in \mathcal{M}_1^b$ are part of the decision value $dec_i =< a_{i+1}, \mathcal{M}_{i+1}^a, < b_1, \mathcal{M}_1^b >>_{p_i^{prim}}$. The places in $\mathcal{M}_1^b$ only receive b_1 when the

decision is reliably broadcast to the concerned places. At this point, the execution of the spawn child agent can proceed. Consequently, only when the places in $\mathcal{M}^a_{i+1}$ receive dec_i, they send off the agent b_1.

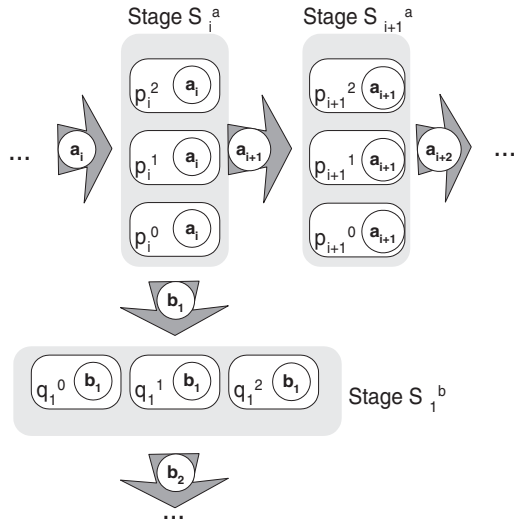

Figure 6.16: Agent a_i spawns a new agent b_1 at stage S^a_i.

6.5.2 Agent Coordination

In the previous section, we have discussed the case of a spawn agent. In this section, we address the issue of coordinating two agents. For instance, the agent b spawn by agent a may rejoin agent a again at a later stage in a's execution, say S^a_k. As agent a is replicated at stage S^a_k, a replica of agent b needs to run on every place in $\mathcal{M}^a_k$ as well. This way, it is ensured that no replica agent a^j_k waits forever for the arrival of agent b. Clearly, agents a and b need to agree on their meeting point, i.e., $\mathcal{M}^a_k$.

6.6 Summary

In this chapter, we have presented our approach to fault-tolerant mobile agent execution. In contrast to existing approaches, which are either blocking [RS98, VKM97a], assume reliable failure detection [JMS+99, PPG00], or are based on complex models of transactions and leader election [ASPZ98, RS98], we model fault-tolerant mobile agent execution as a sequence of agreement problems. In other words, at each stage an agreement problem is solved among the agent replicas at this stage. Together with replication, these agreements achieve the non-blocking and exactly-once property specified for fault-tolerant mobile agent execution. Our approach uses two well-studied and proven building blocks: consensus and reliable broadcast. To validate our results, we have implemented our approach in a system called FATOMAS and evaluated its performance.

Chapter 7

Fault-Tolerant Mobile Agent Execution In The Context of Iso-Places

In Section 4.2.2 we have briefly introduced replicated and independent iso-places, but in Chapter 6 we have limited the discussion to hetero-places and hetero-places with witnesses. In this chapter, we present fault-tolerant mobile agent execution in the context of iso-places. In particular, we identify the modifications that are required to our approach in order to also handle iso-places. We start with an overview on the notion of *determinism* in the execution (Section 7.1); execution determinism is an important property in the context of iso-places. In Section 7.2, we discuss the problem of locality and transparency of an agent executing on a replicated iso-place. Similarly to hetero-places, ensuring non-blocking and the exactly-once property is not easy. To show how the exactly-once property can be achieved in the context of iso-places, we first address a related, but more general problem: the problem of a replicated client invoking (by sending a request to) a replicated server (Section 7.3). This latter problem, which we call the problem of *replicated invocation*, is of interest also in traditional distributed client/server systems. We show that solving this problem in the context of traditional client/server systems solves at the same time the problem of replicated iso-places in the context of mobile agents (Section 7.4). Section 7.5 presents the case of independent iso-places, which are a special case of replicated iso-places. Finally, Section 7.6 discusses the combinations of hetero-places and iso-places that are meaningful in the set of places $\mathcal{M}_i$.

7.1 Deterministic Execution

The execution of the agent replicas a_i^j running on the places in $\mathcal{M}_i$ is *deterministic*, if all a_i^j in $\mathcal{M}_i$, given the same input and initial state, execute the same steps and return the same results. Moreover, these results must be reproducible. Multiple threads, for instance, lead to non-deterministic execution as the thread scheduling is somewhat arbitrary. Further sources for non-determinism are, for instance [Pol94]:

- Non-deterministic system calls (local time reading, randomized numbers)
- Asynchronous events (software interrupts, system exceptions)

- Non-deterministic program constructs. For instance, the `select` statement in Ada [Bar95] arbitrarily chooses among a set of pending service calls (see Figure 7.1) and is thus a source of non-determinism.

The same input to the agent replicas can only be achieved if requests of the agent replicas to the places (or rather to the services running on the places) are also executed deterministically on the places. Hence, in addition to the deterministic execution of the agent we implicitly also require deterministic execution of the places. Indeed, assume that agent replicas a_i^j and a_i^k both send request rq to p_i^j and p_i^k, respectively. Both invocations need to produce the same result to prevent different outcomes of the executions of a_i^j and a_i^k. Clearly, the requirement of deterministic execution limits the applicability of agent replication.

```
task my_task is
   entry a(...);
   entry b(...);
   entry c(...);
end;

task body my_task is
begin
   select
      accept a(...) do ... end;
   or
      accept b(...) do ... end;
   or
      accept c(...) do ... end;
   end select;
end my_task;
```

Figure 7.1: In Ada, a task can be used to provide a number of alternative services, e.g., `a(...)`, `b(...)`, and `c(...)`. Requests to this services are queued until the task has the resources to service them. The choice of the queue from which the next request is serviced is made *arbitrarily* by the `select` statement [Bar95].

Note that deterministic execution of the replicas can be enforced: The simplest approach consists of not using any source of non-determinism at all. For instance, calls to the local clock and to random number generators need to be avoided. Although multiple threads may be possible, Narasimhan [Nar99] proposes an approach that ensures that only a single logical thread of control is active at a time and that the thread scheduling is the same for all replicas. However, it may be impossible to eliminate all sources of non-determinism and to enforce determinism in the execution of the client replicas. For instance, enforcing deterministic thread scheduling generally requires access to the underlying platform, which is not usually given. Hence, we need to address the issue of non-deterministic agent replicas. Moreover, in the context of iso-places, we have to consider the issue of exactly-once execution.

7.2 Replicated Iso-Places

In contrary to hetero-places, replicated iso-places run a replication mechanism among themselves (see Figure 7.2). Consequently, executing the replicated agent on replicated iso-places leads to two levels of replication: replication running (1) among the iso-places[1] and (2) among the agent replicas. The replication technique used for (2) is based on DIV consensus (i.e., is similar to semi-passive replication, see Section 6.3), while (1) can use either active, passive, or semi-passive replication (see Section 2.2.3).

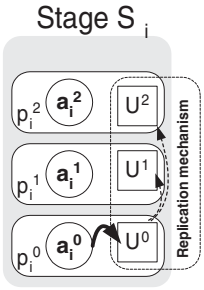

Figure 7.2: Agent a_i at stage S_i executes on replicated iso-places. U^j represents the replica of the service that is running on place p_i^j and provides access to the state of p_i^j (see also Section 3.2). In this example, the iso-places use passive replication.

Note that agent replicas need not run on all replicated iso-places; rather, they can only run on a subset of these places. Indeed, the replication mechanism among replicated iso-places ensures that all iso-places will eventually reflect the changes, whether they run a replica agent or not. Revisiting the example in Section 4.2.2, although Swiss Air Lines may provide five replicated iso-places, $\mathcal{M}_i$ may only contain three of them.

One advantage of agent mobility is that the agent can move to the places hosting the services and access these services locally. With replicated iso-places, local access is not always given any more, although the agent replica a_i^j is collocated with U^j on place p_i^j. Assume, for instance, that the replicated iso-places use passive replication (see Figure 7.2). In the passive replication technique, all accesses to the service go to the *pr_primary*, e.g., U^0. Actually, the pr_primary is usually called *primary* [BMST93]. However, to avoid confusion with the primary used in agent replication (see Section 6.2.1) we use the term pr_primary, whereby "pr" stands for "place replication". To exploit locality, it is desirable that the primary and the pr_primary are collocated, e.g., on p_i^0 in Figure 7.2. However, even if the agent replica on primary p_i^1 executes, it still accesses the pr_primary U^0 running on p_i^0 in the case of passively replicated iso-places (unless U^0 has failed). Hence, locality is not given any more in this example. Similarly, in the case of actively (semi-passively) replicated iso-places, the requests are atomically (reliably) broadcast to all replicated iso-places. Consequently, local accesses are not always given any more and the agent needs to know the replication technique used on the places. To isolate the replication technique used by

[1] Strictly speaking, the replication mechanism occurs among the services running on the replicated iso-places (i.e., U^j in Figure 7.2). For simplicity, we do not make this distinction in our discussion (see also Section 3.2).

the replicated iso-places from the agent replicas (i.e., to achieve transparency) and to provide the illusion of local accesses to the agent, the agent replicas access the service on the iso-place through a local proxy [Maz96, MGG95a, MGG95b]. This proxy then forwards the request to the service. With passively replicated iso-places, for instance, the proxy forwards the request to the pr_primary. Although local accesses to the service may not be given any more, replicated iso-places would generally be located in the same LAN, or have fast communication links among themselves in order to minimize the time to update the other replicas. Hence, communication latency is small even if locality (in the sense that primary and pr_primary are collocated) is not given.

Similarly to the case of hetero-places, ensuring exactly-once is not easy. Indeed, a violation to the exactly-once execution property can occur in the case of false suspicions or failures of the primary (see Section 4.2.3).

7.3 The General Case: the Problem of Exactly-Once in Replicated Invocation in a Traditional Client/Server System

Interestingly, ensuring exactly-once in the context of mobile agents is related to the more general problem of ensuring exactly-once in the context of a replicated client invoking a replicated server in traditional distributed client/server systems. Indeed, assume that replicated agent a_i executes on replicated iso-places p_i^j. More specifically, a_i^j executes on p_i^j and accesses local services on p_i^j. Clearly, this corresponds to the situation in client/server distributed computing, where a replicated client R accesses a replicated server T. We call this latter interaction a *replicated invocation*. Replicated invocation is the general interaction paradigm that encompasses the case of replicated agents executing on replicated iso-places. In this respect, replicated invocation is an important issue for our work on fault-tolerant mobile agents. In this section, we study the more general problem of replicated invocations and show how the exactly-once property can be achieved. As a consequence, we do not consider mobile agents in this section; rather, we consider the traditional client/server computing paradigm. However, based on the results of this section, we then show how exactly-once can be achieved with replicated mobile agents executing on replicated iso-places in Section 7.4.

Most work on replication considers the case of a non-replicated client that invokes (i.e., sends a request to) a replicated server. This corresponds to the invocation between client C and server R, consisting of replicas R^0, R^1, and R^2, in Figure 7.3. However, to service C's request, server R in a component-based (or object-based such as CORBA [OMGa, OH98]) system often invokes another server T. In a fault-tolerant configuration, both R and T may be replicated, leading to a replicated server invoking a service on another replicated server (see Figure 7.3). To simplify our discussion, we assume reliable communication between R and T, i.e., no messages are altered, garbled or lost. In the following, we focus on servers R and T, and denote R as *client* and T as *server*. Furthermore, when we refer to R (respectively T) we implicitly mean the client R (respectively server T) consisting of replicas R^0, R^1, and R^2 (respectively T^0, T^1, and T^2), unless explicitly stated otherwise. If T uses optimistic (or pessimistic) concurrency control (see Section 4.3.3), we say that T is an *optimistic* (*pessimistic*) *server*. Finally, we denote by $|R|$ the replication degree of R.

The problem of replicated invocations has already been addressed in various contexts. Most

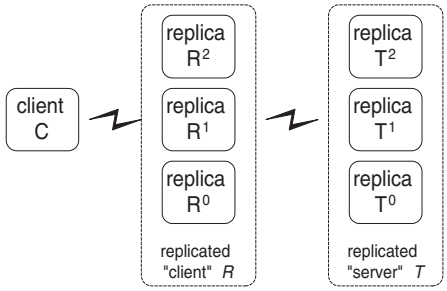

Figure 7.3: Client C invokes replicated server R that, in turn, invokes another replicated server T. The invocation between R and T involves three replicas that call three other replicas and thus is more complex than the invocation between C and R.

of this work, however, assumes deterministic execution of the client replicas [Maz96, Nar99] or explicitly enforces determinism [Nar99]. Enforcing determinism is particularly important in real-time applications [Pol94]; real-time guarantees cannot be ensured in the presence of non-determinism. In [FG00b, FG01b], Frølund and Guerraoui propose a correctness criterion for exactly-once in the context of replication, that addresses also non-determinism in the execution and external side-effects such as requests to other servers. Furthermore, they introduce a replication technique, called *asynchronous replication* [FG00a, FG01a]. This replication technique is targeted towards the classical three-tier architecture with slim clients, stateless application servers, and databases [LD98]. In contrary, our approach is more general in that it also addresses statefull components. Indeed, our approach does not make a clear distinction between client and server. Rather, any client can at the same time act as a server for another client. Having statefull components requires that our replication protocol ensures that all replicas are updated. Moreover, our approach, which is based on passive or semi-passive replication, does not require two runs of consensus per client request; rather, we only require one run of consensus. Termination of the client request in the sense of [FG01b] is ensured even with failure detectors of class $\diamond S$ [CT96], whereas [FG00b] needs a failure detector from the stronger class $\diamond P$. On the other hand, our response time is generally higher, as consistency of the replica state needs to be maintained.

In the following, we first address the issue of exactly-once in replicated invocation (Section 7.3.1). We distinguish between deterministic (Section 7.3.2) and non-deterministic execution of the client replicas (Section 7.3.3), both supported by passive and semi-passive replication. The reader is referred to [Nar99, Maz96] for a discussion on active replication (i.e., solely deterministic execution).

7.3.1 Exactly-Once Execution

The exactly-once property is violated if C's request rq leads to a state of R and T, that reflects the execution of rq multiple times. To be fault-tolerant, R and T can execute the request multiple times, but must have exactly-once semantics relative to their environment [FG00b, FG01b]. Similar to the violation of the exactly-once property in the case of replicated mobile agents (see Section 4.2.3), the violation of exactly-once is caused by unreliable failure detection and partial

execution of requests on the replicas. The execution on multiple replicas R^j may lead to multiple invocations to server T and thus potentially to multiple executions on T. To return into a consistent state, the redundant executions of the request on the replicas of R need to be undone (see Section 4.3.3).

Exactly-once execution within R is enforced by the replication technique used by R, i.e., passive or semi-passive replication in our case. More specifically, the replicas R^j agree on the primary, i.e., the replica that has executed the request. The other replicas, that have (partially) executed the request, need to undo the modifications. This may also require to undo modifications to other servers, such as T. Unfortunately, this is more difficult, especially if some replicas of R have failed. The replication technique used by T together with an undo mechanism enforces exactly-once within T.[2] Duplicate messages sent from replicated server T to replicated client R can easily be discarded by R, as they all have the same ID. This is ensured by the replication technique used in T and is independent of deterministic or non-deterministic execution of T. In the following, we investigate mechanisms to prevent redundant invocations to other servers such as T and, if they still occur, to undo the modifications caused by these invocations. For this purpose, we need to distinguish between deterministic and non-deterministic execution of R^j.

7.3.2 Deterministic Execution of Client R Achieves Exactly-Once

Generally, redundant invocations of T are prevented by attaching an identifier (ID) to every request. Multiple requests to server T can easily be detected by T if these requests are exact duplicates, i.e., use identical request IDs. Indeed, the use of request IDs allows T to detect whether the same request has already been processed [MGG95b, Nar99]. If this is the case, T simply returns the previously computed result; otherwise, the client's request is executed. However, this requires that the client replicas use identical request IDs when generating the same request. Assume, for instance, that client replica R^0 sends request rq_0 to server replica T^0 (see Figure 7.4, step (1)), that executes rq_0. Because of the replication mechanism among the replicas of T, all other replicas T^j reflect the result of this execution (Figure 7.4, step (2)). Before the result of rq_0 is communicated to client replica R^0, R^0 fails. Another client replica (e.g., R^1) takes over the execution and sends request rq_1 to server replica T^0. If the request IDs of rq_0 and rq_1 are equal, then server replica T^0 detects the duplicate request and simply returns the previously computed result. Even if the client request rq_1 is directed to T^1 (step (3) in Figure 7.4), it is identified as a duplicate request, as T^1 will have been updated in the meantime. Identical request IDs on different client replicas are in general only possible if the client replicas execute *deterministically* (see Section 7.1).

Unfortunately, deterministic execution is not always given. In the next section, we therefore discuss the case of non-deterministic execution of R^j. We show that in this case achieving exactly-once is more difficult and requires undoing the execution of requests on the servers.

7.3.3 Non-Deterministic Client R

Assume, that client replicas R^j are executing non-deterministically. Non-deterministic execution of R^j is supported by passive or semi-passive replication (see Section 2.2.3). Executing two

[2] Note that with active replication, *all* T^j execute R^j's request. Exactly-once in this context requires that all T^j execute the request *exactly-once*, leading to $|T|$ executions of the request.

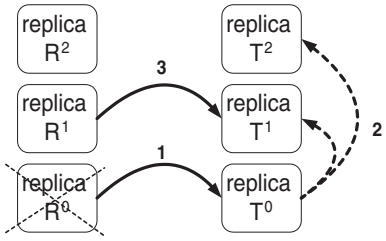

Figure 7.4: Client replica R^0 fails before updating the other client replicas, but after sending a request to T^0 (1). Hence, R^1 takes over the execution and accesses T^1. If T is passively replicated, R^1 also accesses T^0 unless T^0 has been excluded from the view in the meantime (3). Assume that T^0 has already updated the other replicas using the passive replication technique (2). Because the R^j execute deterministically, T^1 identifies the duplicate request and simply returns the previously computed result. A similar example can be constructed using semi-passive replication for R.

replicas R^k and R^l may lead to different requests, in the sense that the requests either

1. have a different content,
2. are sent to different servers (e.g., see Figure 7.5), or
3. one of the requests is not sent at all, or
4. any combination of (1), (2), or (3).

In the context of passive and semi-passive replication, two replicas are executed if the primary replica fails or is suspected and another replica (a backup replica in case of passive replication) takes over the execution. Non-identical requests may lead to an inconsistent state. Indeed, non-identical requests from two replicas R^k and R^l cannot be distinguished by T from the requests of two different clients V and W. Hence, in the following, we focus on how to maintain consistency despite redundant requests.

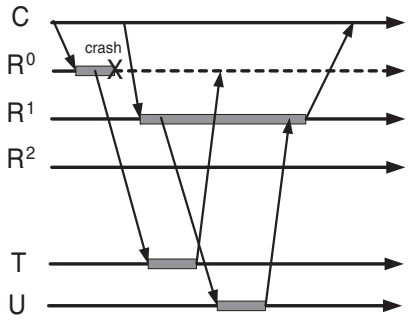

Figure 7.5: Replica R^1 accesses another server than R^0 after detecting the failure of R^0. In this example, R uses passive replication.

Actually, the fact that T is replicated has no influence on the particular exactly-once problem we address. Clearly, a failure to non-replicated server T renders its services unavailable but the replication of T does not lead to a violation of the exactly-once property (see Section 7.3.1). For simplicity, but without loss of generality, server T is thus represented as a non-replicated server in the figures.

To restore the consistent state of the system, the modifications caused by redundant (duplicate) requests have to be undone. However, undoing the request of a failed replica is not easy; it requires knowledge about the particular request. Generally, only the sender of the request has this knowledge. In this context, we need to distinguish between optimistic and pessimistic server T. If T is an optimistic server, request rq is undone by sending *undo request* $\overline{rq}$. Similarly to compensating transactions (see Section 4.3.3), $\overline{rq}$ semantically undoes the modifications caused by rq. Assume, for instance, that rq reserves a ticket on a flight, then $\overline{rq}$ simply cancels this reservation. Note that in this case, an undo message is only needed in case of duplicate requests. Moreover, the sequence of requests $rq_x, rq_y, \overline{rq}_x$ is a valid sequence and is semantically equivalent to the sequence that consists only of rq_y. On the other hand, if T is a pessimistic server, unterminated transactions on T need to be terminated, either by committing or aborting them. Hence, although no duplicate requests may have occurred, an additional `COMMIT` message is needed. Figure 7.6 shows a scenario without failures or false suspicions, in which R^0 commits its request. Duplicate requests lead to duplicate transactions on T, which are aborted by sending an `ABORT` message. We call the `COMMIT` and `ABORT` messages *termination requests*. Moreover, we assume that the execution of termination requests is idempotent.

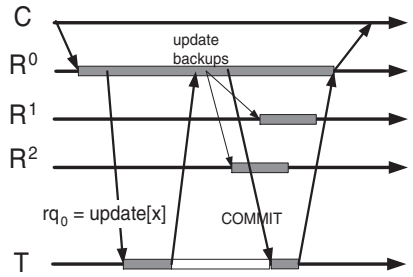

Figure 7.6: Pessimistic execution on the server T. R uses the passive replication technique. The request $update[x]$ updates data item x on T.

Generally, two approaches can be identified according to who sends these undo or termination requests. Indeed, assume that replica R^k has send duplicate request rq:

1. Replica R^k itself sends $\overline{rq}$ or `COMMIT/ABORT` in the case of optimistic or pessimistic T, respectively.

2. Another replica R^l $(l \neq k)$ sends this undo or termination request.

The first approach (Section 7.3.4) assumes eventual recovery of R^k, while the second makes no such assumption and is presented in Section 7.3.5.

7.3.4 Non-Deterministic Client R - With Recovery

Assume that all failed replicas eventually recover. Hence, failed replica R^k eventually recovers again. Upon recovery, R^k finds out (e.g., by using a logging mechanism [GR93]) (1) whether it has sent any unterminated requests before failing, and if this is the case, (2) whether the result of these requests have been accepted by the other replicas. Item (2) is important in the context of passive and semi-passive replication, as the other replicas may have adopted the result of a failed former primary (see Section 2.2.3). Indeed, R^k may have already communicated the result to other replicas before failing. Based on this information and depending on the isolation strategy of T, R^k takes the actions presented in Table 7.1.

	request rq adopted by R^j $(j \neq k)$	request rq not adopted by R^j $(j \neq k)$
optimistic server T	-	$\overline{rq}$
pessimistic server T	`COMMIT` message	`ABORT` message

Table 7.1: Type of message sent upon recovery of failed primary R^k, assuming that R^k has previously sent a request to T.

Note that pessimistic server T must offer the possibility to terminate transactions from another process than the one that has started the transaction. To our knowledge, this is not generally implemented in today's databases. This must especially be ensured in the case no proper handshake occurs between these two processes. Indeed, assume that R^k accesses T, fails, and eventually recovers. Generally, R^k will recover with a different process ID. Furthermore, it may be desirable to terminate the transaction from another client replica, as discussed in more detail in the following section.

Assuming FIFO order on the communication channels[3], undo and termination requests are ignored by T if no corresponding request has been received previously. This behavior is easily implemented by keeping track of the list of request IDs. If no such support is provided by T, then the actual request is sent together with the termination or undo request. Server T ignores the request if it has already been processed. With this approach, the undo or termination request is never executed without prior execution of the corresponding request. If FIFO order is not implemented by the communication channels, or if the sender of the request is different from the sender of the corresponding undo or termination request (see Section 7.3.5), the server needs to keep track of all unsolicited undo or termination requests. Storing these undo or termination requests allows to reuse them in case the actual request arrives at the server later.

Moreover, an optimistic server needs to support the undoing mechanism to a certain degree. Assume, for instance, that client replica R^0 sends the request for a ticket to T. Server T can either grant the request and return a (virtual) ticket, or reject it. If R^0 has failed in the meantime, no other replica knows how to properly undo the request. Indeed, cancelling the ticket is different from undoing the rejected ticket request; in the latter, no undo action is needed at all. Unless the server sends the result back to all client replicas, they do not know how to properly undo the request. To reduce the number of messages between R and T, T stores a history of requests and the corresponding results. Upon reception of an undo or termination request, T undoes the corresponding request based on the information in the history.

[3]This can be implemented along the lines of [BCBT96].

Potential of Blocking With Pessimistic Server T

Pessimistic execution on T may lead to blocking in the execution of the client replicas R^j. Indeed, pessimistic execution makes modifications on T available to other clients only if they are committed (see Section 4.3.3). More specifically, after sending a request to T, replica R^k must terminate the transaction this request has started by either an abort or commit. Blocking occurs if another replica R^l (or another client) attempts to access the same data items before the failed replica R^k has terminated its open transaction. Assume, for instance, that R^0 fails immediately after sending request rq_0 (see Figure 7.7). Eventually, another replica, e.g., R^1, takes over the execution and sends request rq_1 to T. If rq_1 attempts to acquire the same locks as rq_0 already holds, blocking of R^1 occurs. Worse yet, eventually, R^2 suspects R^1 and sends its request, which may also block. Hence, the entire client R is blocked. Blocking of R is undesirable, as it acts itself as a server for other applications (see Figure 7.3).

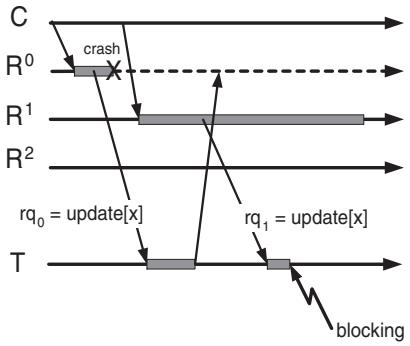

Figure 7.7: Pessimistic execution on the server T may lead to blocking.

Blocking can be resolved if T is allowed to spontaneously abort the transaction caused by rq_0, i.e., unilaterally decide the outcome. However, this may lead to inconsistencies. Assume that R^0 is very slow, has invoked T, has updated the other replicas R^j, and then fails. Hence, the state of R reflects the fact that the service on T has been invoked, but T itself does not. Clearly, this is inconsistent, and must be avoided.

Another approach consists of a timeout mechanism in the replicas R_j. After having sent a request to S, they wait for a certain time and then send an abort unless they have received a reply in the meantime. Revisiting the above example, R_1 and R_2 simply timeout and proceed with their execution, potentially reporting the failed request to the user. However, this approach has the drawback that server S is no longer available, despite its replicated nature. Although blocking in the clients of S is prevented, S basically becomes unavailable with this approach.

As a consequence, T will remain blocked until R^0 recovers and undoes its request. This is unacceptable in any practical system. Hence, in the case of blocking, recovery does not entirely solve the problem. In other words, recovery prevents infinite blocking, i.e., ensures that eventually the blocking situation is resolved. However, blocking is undesirable and a mechanism needs to be devised that resolves the blocking situation without depending on the recovery of a failed replica. This mechanism is similar to the one proposed in a model without recovery and is discussed in the

following section.

7.3.5 Non-Deterministic Client R - No Recovery

Without recovery, the failed primary cannot itself undo its modifications to T; rather, these modifications need to be undone by some other replica in R, preferably the new primary. This requires that *undo information*, which allows to undo a particular request, is made available to the other replicas of R prior to invoking T. For this purpose, the undo information is reliably sent to the other replicas. Actually, the primary has to make sure that a sufficiently large set of replicas has received the undo message, before sending the request to T. For this purpose, it can either use (1) a n^2 reliable broadcast [HT94] to the replicas in R, or (2) an approach with a higher latency, but less messages. In approach (2), the primary R^0 sends the undo message to the replicas in R until it gets an acknowledgment from a subset of them. For instance, using failure detector $\diamond S$ [CT96] a majority of processes need to receive the undo information. If R^0 fails, then all other replicas have to agree whether its results are accepted or whether the new primary redoes the computation.

Note that the replica's processing must not depend on the arrival of the undo information, as this may lead to blocking if the primary fails before sending the undo information. Rather, they must be able to process the undo information at any time. For instance, the replicas must also handle delayed undo information. Trivially, if the primary fails before sending the undo information to the other replicas, the state in R and T is still consistent and no undo on T is required.

In the following, we first discuss the case of a passively replicated client R, before addressing semi-passive replication.

Passively Replicated Client

In this section, we address the issue of ensuring exactly-once with a passively replicated client R. Recall that failed replicas do not recover and R^j execute non-deterministically. Ensuring exactly-once in this context is not easy; it requires that all redundant invocations to the server(s) are undone. Consequently, all replicas R^j must agree on the invocation that is not undone. In passive replication based on group membership and view synchronous broadcast (vs-cast) (see Section 2.2.3), an update is accepted by R^j if it is vs-delivered. If this is the case, then any potential invocation to server T must not be undone. In contrast, the effects of any other requests that are sent to T but do not lead to an adopted update on R^j must be undone. In the following, we first discuss the case of optimistic server T, before addressing the case of pessimistic server T.

Optimistic Server T: With an optimistic server T, an undo request $\overline{rq}$ is sent to T in order to undo request rq. Executing $\overline{rq}$ on T semantically undoes all the effects of the prior execution of rq.

Consider first the case where no undo request is required (see Figure 7.8). In this case, the primary (i.e., R^0) executes C's request, which requires the sending of a request to T, vs-casts the update, and crashes. The backups (i.e., R^1 and R^2) vs-deliver the update before the new view v_{i+1} is installed and thus adopt it. When C resends its request, the new primary R^1 simply returns the result previously computed by R^0.

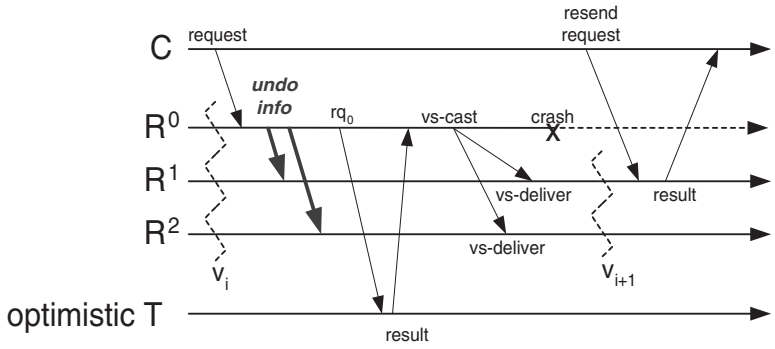

Figure 7.8: The primary R^0 fails after invoking optimistic server T and updating the backups R^1 and R^2.

However, if R^0 fails and its update is not vs-delivered, the effects caused by rq_0 on T need to be undone (see Figure 7.9). The undo is performed by sending $\overline{rq}_0$ to T. Eventually, C resends its request to the new primary R^1, which recomputes the result. Note that the order of $\overline{rq}_0$ and rq_1 is not significant; rather, $\overline{rq}_0$ can be send at any time after the new view v_{i+1} is installed. This is a consequence of the properties of the undo request (see Section 7.3.3).

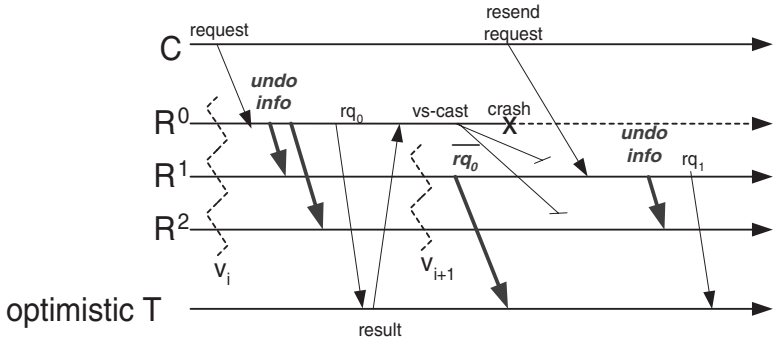

Figure 7.9: The primary R^0 fails after invoking optimistic server T and updating the backups R^1 and R^2, but the updates are not vs-delivered.

A particular case arises if R^0 fails after having sent the undo information to the backup replicas, but before sending the request to T. As the backup replicas have received the undo information, they will use this undo information to send an undo request to T. Assume that $\overline{rq}_0$ arrives at T without the previous rq_0. T must handle this case: if rq_0 has not been received, then $\overline{rq}_0$ is not executed, but stored to be reused in case rq_0 eventually arrives (if it does at all). Note that such early undo messages, i.e., rq_0 arrives at T after $\overline{rq}_0$, occur even if R^0 fails after sending rq_0. Indeed, rq_0 may take longer than $\overline{rq}_0$ sent from another replica. If T does not provide support for handling early undo messages, the undo message can comprise rq_0 and $\overline{rq}_0$ (see Section 7.3.4). If rq_0 has already been executed on T, it will be ignored and only $\overline{rq}_0$ is executed; otherwise, first

rq_0 and then $\overline{rq}_0$ are executed.

Pessimistic Server T: If T executes pessimistically, a termination message is always required. Indeed, assume that the primary R^0 fails after vs-casting the update to the backups, which vs-deliver it (see Figure 7.10). To terminate the transaction on T, a `COMMIT` message is sent by the new primary R^1 to T. Note that after R^0 has sent rq_0 and received the result from T, it sends a prepare message to T. T replies affirmative to this message if it is ready to commit. Actually, this message is not really needed, as we assume that the databases do not spontaneously abort transactions. However, legacy databases may still require it.

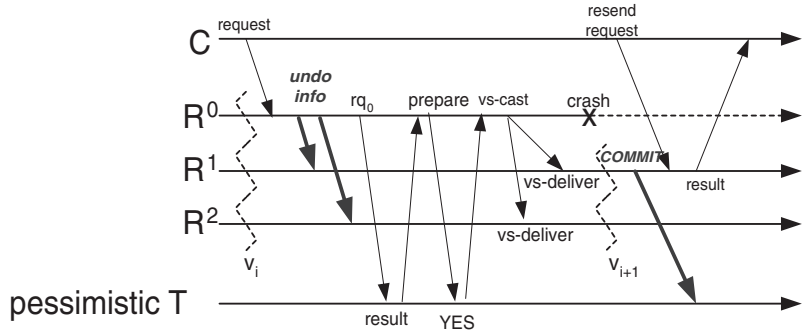

Figure 7.10: The primary R^0 fails after invoking pessimistic server T and updating the backups R^1 and R^2.

In contrary, an `ABORT` message is sent to T by R^1 if R^0's update is not vs-delivered (see Figure 7.11). When C resends its request, this request is executed by R^1.

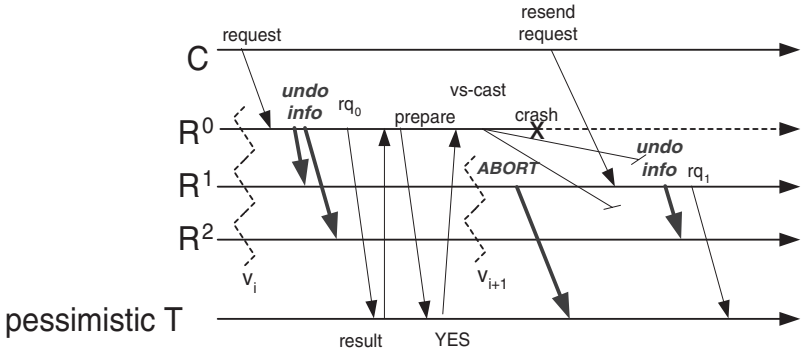

Figure 7.11: The primary R^0 fails after invoking pessimistic server T and updating the backups R^1 and R^2, but the update is not vs-delivered.

Depending on the broadcast strategy for the undo information, the undo information may arrive late at the backup replicas (see Figure 7.12). If R^1's request rq_1 is sent before the undo message of R^0 has arrived and rq_1 accesses the same data items as rq_0, rq_1 may have to wait until rq_0 releases

its locks. These locks will be released as soon as the ABORT message arrives at T. Hence, waiting for locks does not require the recovery of R^0 and blocking does not occur.

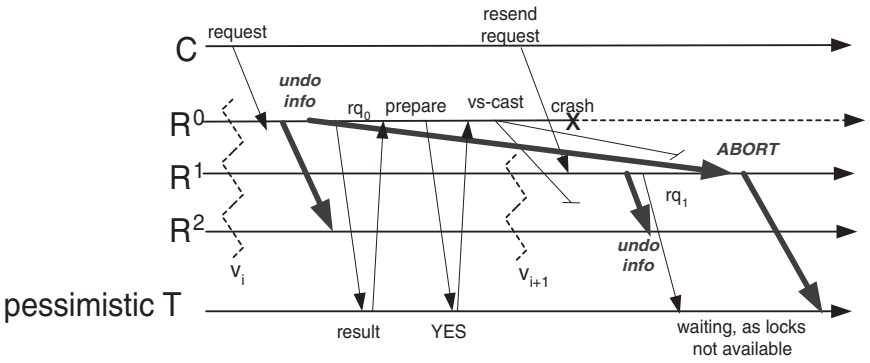

Figure 7.12: The primary R^0 fails after invoking pessimistic server T and updating the backups, but the update is not vs-delivered. As the undo message is delayed, the request rq_1 from R_1 waits until the locks of rq_0 are released.

Consider again the particular case of R^0's failure after sending the undo information, but before sending the request to T. In the case of pessimistic server T, no special mechanisms are needed. Indeed, termination messages not related to an actual request are (stored and eventually) simply ignored by T.

Semi-Passively Replicated Clients

In the previous section, we have addressed exactly-once execution with a passively replicated client. In this section, we discuss the case of a semi-passively replicated client R, again with non-deterministic execution of R^j and no recovery. In particular, we indicate the minor differences to the case of passive replication and show that the same mechanisms can be used to ensure exactly-once.

With semi-passive replication, agreement on the update is achieved by running DIV consensus among the replicas R^j. At the end of DIV consensus, all R^j agree on this update (see Section 2.2.3). However, to ensure exactly-once, they now also need to agree on the primary, i.e., the process having executed the function $\mathcal{GIV}()$ that has lead to the current update value. Using this information, the processes that are not primary can undo their modifications, while the primary commits them in the case of pessimistic server T.

Similarly to the case of passively replicated R, blocking can still occur with a pessimistic server T. Assume, for instance, that replica R^0 fails after having sent request rq_0 to T, but before it manages to send the new estimate (i.e., the result of the execution of $\mathcal{GIV}()$) to the other replicas in phase 2 (see Figure 7.13 and also Line 27 in Algorithm 2, Appendix A.2). Eventually, R^1 detects the failure of R^0 and itself sends request rq_1. However, if T is a pessimistic server, rq_1 waits until the locks held by rq_0 are released. Hence, rq_0 needs to be aborted before rq_1 is executed. To prevent infinite blocking, the termination message needs to be sent by another replica R^j, preferably the new primary. Again, we apply the same idea as in passive replication and reliably

send undo information to the other replicas prior to sending a request to T (see Figure 7.13). Hence, when the new primary is in phase 2, has passed the wait statement and the estimate is $\perp$, the coordinator starts executing the function $\mathcal{GIV}()$. Before executing $\mathcal{GIV}()$, however, it needs to abort previous executions in order to avoid blocking (Line 23 in Algorithm 2).

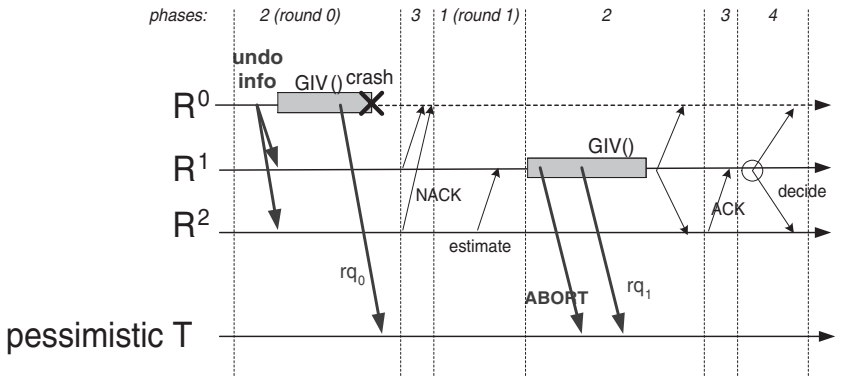

Figure 7.13: R_0 fails after invoking pessimistic server T, but before reaching an agreement on the results with the other replicas R^j. Eventually, $R^j (j \neq 0)$ detect the failure of R^0 and another replica (e.g., R^1) executes $giv()$. However, R^1 first needs to abort the transaction caused by rq_0 on T.

Actually, the function $\mathcal{GIV}()$ is only evaluated more than once, if a majority of processes has *not* received the suggestion message of the process that has executed $\mathcal{GIV}()$. If at least one process receives the suggestion and participates in the next round, then this estimate is also accepted as the estimate of the next round and thus the function $\mathcal{GIV}()$ is not executed any more. Interestingly, the outcome of DIV consensus could be a value, that has been computed by a process that has crashed in the meantime.

7.3.6 Practical Considerations

The approach presented for replicated invocations in the case of non-deterministic execution has two limitations, that are, however, not caused by our approach but are rather a consequence of the replicated invocation problem:

- The first drawback is that pessimistic servers are not allowed to spontaneously abort unterminated transactions. Hence, a transaction can be left unterminated by the client for a long time, thus effectively preventing other clients from accessing the resources. As the server cannot terminate transactions any more, it has to rely on the clients to behave accordingly and terminate all transactions they have started.

- Pessimistic server T needs to support aborts/commit of a transaction by another process than the one that has started the transaction (see Section 7.3.4). To our knowledge, although a mechanism to pass on the responsibility for a transaction to another process is foreseen in the XA Specification [ISO96] for distributed transaction processing, this mechanism seems

not to encompass the situation where processes fail; rather, in this case, the unterminated transaction is simply aborted.

7.3.7 Summary on Replicated Invocations

Table 7.2 summarizes Sections 7.3.3 to 7.3.5. It assumes that R^k has sent a request to server T before failing. The simplest situation occurs in a model with recovery and optimistic T. In this case, the modifications are undone by the failed replica R^k upon recovery. A pessimistic server T, on the other hand, always requires that the modifications are undone by other replicas to ensure consistency and to prevent blocking. Finally, in a model without recovery, R^j needs to undo the request in the case of optimistic server T.

non-deterministic execution of R	pessimistic T	optimistic T
no recovery	*blocking, inconsistency* R^j $(j \neq k)$	*inconsistency* R^j $(j \neq k)$
recovery	*blocking, inconsistency* R^j $(j \neq k)$ to prevent blocking, R^k (and R^j) to resolve inconsistencies	*inconsistency* R^k

Table 7.2: Summary of the problems (in italics) that occur in replicated invocation with a non-deterministic client R and of which replica(s) send(s) the undo or termination request.

7.4 Exactly-Once in the Context of Replicated Iso-Places

In the previous section, we have discussed the case of replicated invocation. A replicated agent executing on replicated iso-places is a special case of replicated invocation. Here, the agent acts as the client and accesses the server running on the replicated iso-place. Thereby, the stage execution of the mobile agent corresponds to a semi-passively replicated client. Indeed, similarly to semi-passive replication, agent replication also uses DIV consensus to achieve an agreement. In this section, we thus present the particularities of agent execution with respect to replicated invocation with a semi-passively replicated client and show how exactly-once is achieved in this context.

7.4.1 Exactly-Once and Determinism

In the context of agent replication and iso-places, determinism as defined in Section 7.1 is not sufficient to ensure the property of exactly-once. Indeed, a fundamental difference between replicated invocations and agent replication on replicated iso-places is the fact that in the former client replicas R^j all start from the same initial state and always maintain the same state after executing the same requests. With mobile agent replication, this is different. Indeed, another mobile agent b may execute on replicated iso-place p_i^k before a_i^k, but not on p_i^l, where a_i^l executes. Consequently, when the replicated iso-places execute the mobile agent replicas, they do not start from the same initial state and thus may not return the same results to the requests of the agent replicas any more. However, the agent replicas still have the same initial state. The replicated iso-places

abstraction assumes that all replicated iso-places are exact replicas. Hence, the state of replicated iso-places can only be accessed by the agent replicas through a replicated server; identical initial state is always ensured and the case of agent replication is thus similar to the case of replicated invocations.

As shown in Section 4.2.3, replication can lead to multiple executions of the agent's code and thus multiple generations of agent requests to the place. However, multiple executions of the agent replica's requests on the place can easily be prevented if the agent replicas execute deterministically. Indeed, the use of request IDs allows replicated iso-places to detect whether the same request has already been processed. If this is the case, then iso-places simply return the result; otherwise, the agent replica's request is executed. Similar to replicated invocation (see Section 7.3), this requires that the agent replicas use identical request IDs when executing the same request. Figure 7.14 shows the example with agent replicas corresponding to the one illustrated in Figure 7.4. Here, the agent replica executing on p_i^0 sends request rq_0 to the service U^0 on p_i^0 (see Figure 7.14 (b), step (1)). The service U^0 executes rq_0 and, because of the replication mechanism among the replicated iso-places, all other iso-places reflect the result of this execution (step (2)). Before the result of rq_0 is communicated to the agent replica, p_i^0 fails. Another agent replica (e.g., the one executing on p_i^1) takes over the execution and sends request rq_1 to service U^1 (step (3)). If the request IDs of rq_0 and rq_1 are identical, then the service on p_i^1 detects the duplicate request and simply returns the previously computed result. Recall that identical request IDs on different agent replicas are generally only possible if the agent replicas execute deterministically (see Section 7.3.2).

With deterministic execution of the agent replicas the stage agreement (see Section 6.2.1) may not be needed any more. Indeed, if the agent *only* performs invocations to replicated iso-places and receives no messages from other agents or applications (as assumed in Section 3.2), then no agreement is required among the agent replicas; rather, the exactly-once execution is ensured by the replicated iso-places and the execution determinism.

In contrast, if the agent accepts messages from other agents or applications, then the agent replicas generally need to agree on the messages that are processed and the order in which these messages are processed. Although the agent replicas may execute deterministically, a different order of processed messages may lead to a different result in the execution of the replica. Hence, the agent replicas either have to (1) agree on the order of processing the messages, or (2) support non-deterministic execution. Case (2) is discussed in Section 7.4.2, while (1) can be achieved if messages are atomically broadcast (total order) to the agent replicas. In the following, we will not further consider agents that receive messages.

Note that no deadlocks as discussed in Section 6.4.2 occur in this case. Recall that these deadlocks are caused by the combination of agreement problem and pessimistic server execution. As no agreement is needed for deterministically executing agent replicas on replicated iso-places, the cause of deadlocks has disappeared. The reason for this is that the replicated iso-places (or rather the replicated servers running on them) implicitly process all requests from agent replicas in the same order. However, this requires that the replicated iso-places hold back any duplicate request to a not yet processed request until the result to the request has been computed.

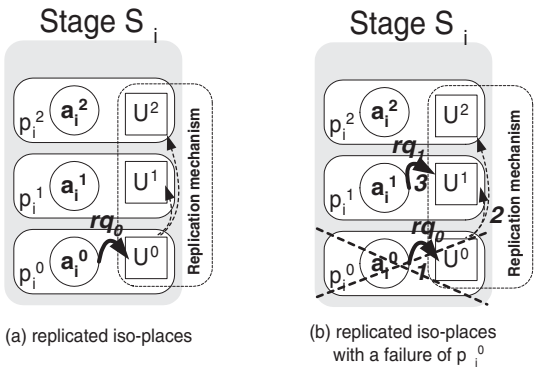

Figure 7.14: Agent replicas execute on replicated iso-places. U^j represents the replica of the service that is running on place p_i^j and provides access to the state of p_i^j (see Section 3.2). The failure of p_i^0 in the case of replicated iso-places is shown in (b).

7.4.2 The Problem of Non-Determinism

Non-determinism in the execution of the agent replicas may lead to inconsistent system states (see Section 7.3.3). Indeed, assume that the request IDs of rq_0 and rq_1 are not equal (see Figure 7.14 (b)). Although p_i^1 already reflects the execution of rq_0 on place p_i^0 (through the replication mechanism among the replicated iso-places), it still executes rq_1. Because of the different request IDs, p_i^1 considers rq_1 as a new request. Actually, a_i^0 and a_i^1 act like two different clients from the view of the iso-places, and not as the replicas they are. This leads to a violation of the exactly-once execution property, as the iso-places reflect the result of executing the agent request twice.

As our model assumes that agent replicas are good (i.e., are eventually always up), the case of non-deterministic a_i^j corresponds to the second row in Table 7.2. Hence, ensuring exactly-once is possible under the assumptions presented in Section 7.3.4. Despite eventual recovery, we have shown that the use of pessimistic servers may lead to blocking. Consequently, we use the approach to semi-passive replication proposed in Section 7.3.5 to prevent blocking in the agent replicas and the replicated iso-places. The modified DIV consensus algorithm is given in Appendix A.2.

7.4.3 Agreement

With the exception of the blocking problem, non-deterministic agent replicas are similar to the case of hetero-places. As a consequence, the decision dec_i in the basic agreement problem for replicated iso-places is similar to the one presented in Section 6.2.1:

1. the primary place $p_i^{prim} \in \mathcal{M}_i$, that has executed the agent at stage S_i
2. the resulting agent a_{i+1}
3. the places $\mathcal{M}_{i+1}$ for a_{i+1}

Because of this similarity, FATOMAS (see Section 6.4) also handles the execution of the agent on replicated iso-places. However, it does not yet implement the undo mechanism given in Algorithm 2 (Appendix A.2).

In the following section, we discuss the case of independent iso-places, which is a special case of replicated iso-places.

7.5 Exactly-Once in the Context of Independent Iso-Places

Independent iso-places can be viewed as a special case of replicated iso-places, where the two levels of replication are integrated into one. More specifically, although independent iso-places are exact replicas, no replication mechanism runs among them (see Section 4.2.2). Instead, the agent replicas ensure that all places in $\mathcal{M}_i$ learn about the agent replica's request and about the new state of the replica iso-place (see Figure 7.15). This can be done by either

1. executing the agent on all places, or
2. executing the agent on one place and sending the state update information of the place to the other agent replicas, which then update their local iso-place.

In approach (1), agreement (i.e., DIV consensus) between the agent replicas is not required. However, approach (1) requires deterministic execution of the agent replicas. Indeed, to ensure that all iso-places p_i^j have the same state after the execution of a_i, the agent replicas a_i^j need to execute the same sequence of steps. Clearly, the execution of the agent replicas on all places p_i^j leads to a higher computation overhead. Moreover, the replicas of two mobile agents a_i and b_i need to be executed in the same order on all independent iso-places p_i^j, i.e., the execution of agent replicas needs to be totally ordered. Hence, reliable broadcast is not sufficient to forward the agent between two stages; rather, the agent is atomically broadcasted (total order) to $\mathcal{M}_{i+1}$ by the places in $\mathcal{M}_i$.

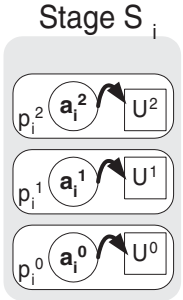

Figure 7.15: Agent replicas execute on independent iso-places. U^j represents the replica of the service that is running on place p_i^j and provides access to the state of p_i^j (see Section 3.2).

Similarly, approach (2) also requires atomic broadcast to forward the agent between two consecutive stages. However, it is more generic in that it also addresses non-deterministic execution

of the agent replicas. For this purpose, the basic agreement problem (see Section 6.2.1) is used to decide on the iso-place that has executed the agent replica. Consequently, approach (1) has a lower computation overhead if no failures and false suspicions occur. Indeed, in this case only one agent replica executes and sends the state update information to the other agent replicas. However, this state update information might be large or difficult to obtain. In Section 6.2.1, we have given the specification of fault-tolerant mobile agent execution in the context of hetero-places. With independent iso-places, the decision value dec_i at stage S_i is different:

1. the state update information for all the places in $\mathcal{M}_i$
2. the resulting agent a_{i+1}
3. the places $\mathcal{M}_{i+1}$ for a_{i+1}

Instead of the primary p_i^{prim} the decision now contains the new state of the place replicas in $\mathcal{M}_i$. The place replicas then install this new state and thus all of them are in a consistent state. The difference in the decision value requires that the mobile agent knows beforehand whether it is executing on independent iso-places or not. Hence, the execution on the places of a stage is not completely transparent for the agent any more. Recall that replicated iso-places and hetero-places can be handled by an identical mechanism.

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

Note that with independent iso-places, the exactly-once property of agent execution is, in a strict sense, not required any more; rather, all agent replicas are executed, i.e., we have multiple executions of agent a_i at stage S_i. However, we still need to ensure that agent replica a_i^j executes exactly-once. In other words, the exactly-once property now refers to the execution of replica a_i^j, and not to the execution of agent a_i as with hetero-places and replicated iso-places.

7.6 Composing $\mathcal{M}_i$ out of Hetero-Places and Iso-Places

In this section, we discuss the case of $\mathcal{M}_i$ that is at the same time composed of hetero-places and iso-places. We argue, that it makes no sense to compose $\mathcal{M}_i$ such that it consists of independent iso-places and any places from hetero-places or replicated iso-places (Section 7.6.1). While our discussion seems to be of only theoretical interest, we show that it may have an interest in a practical setting as well (Section 7.6.2). Table 7.3 summarizes the discussion in the form of a compatibility matrix.

7.6.1 $\mathcal{M}_i$ with a Subset of Independent Iso-Places

A set of places containing independent iso-places and any replicated iso-place(s) or hetero-place(s) is generally impossible. Indeed, recall that in the case of independent iso-places, the decision

place properties	hetero-places (no witnesses)	witnesses	replicated iso-places	independent iso-places
hetero-places (no witnesses)	√	√	√	-
witnesses	√	√	√	√
replicated iso-places	√	√	√	-
independent iso-places	-	√	-	√

Table 7.3: Compatibility matrix of hetero-places, replicated and independent iso-places with respect to $\mathcal{M}_i$.

value of the agreement at the stage is different. More specifically, it contains the new state of the independent iso-places. Hence, if $\mathcal{M}_i$ contains at least one independent iso-place, then the agent replica running on this place needs to be executed. Otherwise, the new state of the independent iso-place cannot be extracted from the information available in the decision value. This makes it generally impossible to compose $\mathcal{M}_i$ such that it contains independent iso-places with any other places. A particular case arises if the places that are not independent iso-places are all witnesses. If the agent executes on a witness, the state of independent iso-places does not change.

7.6.2 $\mathcal{M}_i$ With a Subset of Replicated Iso-Places and Hetero-Places (with Witnesses)

A non-uniform $\mathcal{M}_i$ only makes sense if it is composed of replicated iso-places and hetero-places. For instance, $\mathcal{M}_i$ may contain three replicated iso-places and two witnesses. Note that $\mathcal{M}_i$ needs to contain an odd number of places; an even number actually increases the probability of blocking among the agent replicas [RS98]. However, among the odd number of places in $\mathcal{M}_i$ may be any number (possibly also a pair number) of replicated iso-places. Figure 7.16 shows the example with only one replicated iso-place p_i^2. However, p_i^2 may be part of a larger set of replicated iso-places. Consequently, when a_i^2 executes on p_i^2, the entire set including the replicated iso-places not in $\mathcal{M}_i$ (i.e., q and r) are updated and reflect the modifications.

7.7 Summary

In this chapter, we have discussed the case of a replicated agent executing on iso-places. First, we have introduced the more general problem of replicated invocations and presented a solution to this case. We have then shown how the solution can be adapted to agent replication. A summary of this discussion is given in Table 7.4. Here, we identify the building blocks required to achieve fault-tolerant mobile agent execution in the context of independent and replicated iso-places. Table 7.4 also presents the case of hetero-places. Note that deterministic execution of agents and places is not meaningful with hetero-places, as hetero-places are generally provided by different companies and thus produce different results (see Chapter 6).

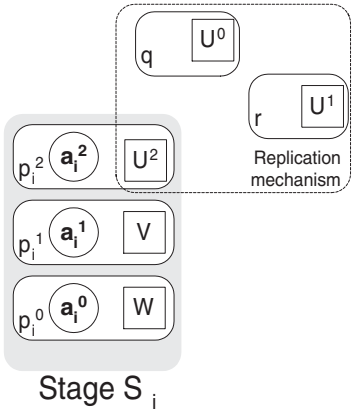

Figure 7.16: The places in $\mathcal{M}_i$ consist of replicated iso-place p_i^2 running U^2 and hetero-places (or witnesses) p_i^0 and p_i^1, running W and V respectively. The places running the other replicas of U^2 (i.e., q and r) are not part of $\mathcal{M}_i$.

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

Table 7.4: Summary of the required building blocks needed to achieve fault-tolerant mobile agent execution. The building blocks are given with respect to the place properties and determinism in the agent/place execution. The case of hetero-places with deterministic execution of the agent/place is not meaningful, as no replication mechanism runs among hetero-places.

Chapter 8

Transactional Mobile Agents

In this chapter, we present an approach that achieves non-blocking transactional mobile agent execution. We first assume that the transactional mobile agents rely on pessimistic concurrency control and start with a discussion on the problem of execution atomicity in Section 8.1. Section 8.2 then generalizes the discussion to non-linear itineraries and Section 8.3 explains the difficulties of handling failures. After presenting the specification of non-blocking atomic commitment in the context of transactional mobile agents in Section 8.4, we give an algorithm that complies with this specification (see Section 8.5). Section 8.6 briefly discusses our approach in an environment with multiple concurrent transactional mobile agents. In Section 8.7, we describe our prototype implementation, called TRANSUMA (TRANsaction SUpport for Mobile Agents), and evaluate its performance with respect to FATOMAS. The last section of this chapter is devoted to open nested transactions. More specifically, it shows how our approach can use also optimistic concurrency control by relying on compensating transactions.

8.1 The Problem of Execution Atomicity

An *atomic* mobile agent execution ensures that either all stage operations succeed or none at all (see Section 3.4). Assume, for instance, a mobile agent that books a flight to New York, books a hotel room there, and rents a car. In this context, the use of the hotel room and the car in New York is limited if no flight to New York is available any more. On the other hand, the flight is not of great use if neither a hotel room nor a rental car are available. This example illustrates that either all three operations (i.e., flight ticket purchase, hotel room booking, and car rental) need to succeed or none at all, i.e., the operations have to be executed *atomically*. Execution atomicity ensures that all operations execute as an atomic action, i.e., either in their entirety or none at all. Both infrastructure and semantic failures may lead to a violation of the atomicity property. In the following, we focus on semantic failures only; infrastructure failures are discussed in Section 8.3.

8.1.1 Traditional Distributed Transactional Systems: Background

In traditional distributed transactional systems, *atomic commitment* protocols such as 2PC and 3PC [BHG87, GR93] address the issues of execution atomicity. In the 2PC, for instance, a des-

ignated coordinator (also called transaction manager) queries all the participants in the execution of the distributed transaction (called the resource managers) on the state of the corresponding operations. The participants return either a `YES-VOTE` or a `NO-VOTE`, depending on whether their operations have succeeded or not. If all operations have been successful, i.e., all returned votes are `YES-VOTE`s, the coordinator decides `COMMIT`, otherwise `ABORT`. The decision is propagated to the participants, which then either commit or abort their operations.

8.1.2 Transactional Mobile Agent Execution

The operations of the participants of traditional distributed transactions can run in parallel. In a transactional mobile agent execution T_a, the operations of mobile agent a are executed *sequentially* in a sequence of stages S_i $(0 \leq i \leq n)$. The execution of agent a at stage S_i depends on the outcome of the previous stage S_{i-1} and implies that agent a has successfully executed on all previous stages S_j $(j < i)$, i.e., all a_j have implicitly issued a `YES-VOTE`. Consequently, the vote of a_i unilaterally determines whether the agent execution is continued (in case of a `YES-VOTE`) or aborted (`NO-VOTE`); the agent a_i only returns a `NO-VOTE` if a semantic failure has occurred.[1] Actually, the transaction T_a spans only over stages $S_1, \ldots, S_{n-1}$: the agent source p_0 and destination p_n are executing the agent outside of the transaction context T_a. On these places, the interaction with the agent owner (i.e., initialization of the agent and presentation of the results) takes place and a transaction context is not needed. Moreover, mobile users are often disconnected from the network and hence stage S_n may be temporarily unreachable. Consequently, executing sa_n within the context of T_a may lead to blocking of the mobile agent execution until the mobile user reconnects to the network. Unless compensating transactions can be used, T_a maintains its locks on data items while it is blocked, thus reducing overall system throughput. Terminating transaction T_a already at stage S_{n-1} prevents blocking due to disconnections of mobile devices. The agent a_n is then kept at place p_{n-1} until p_n reconnects and is able to collect the result. At stage S_{n-1}, the agent a_{n-1} unilaterally decides either `COMMIT` (if the execution of a_{n-1} has succeeded), or `ABORT` (in case of a semantic failure).

In the case of a dynamic itinerary, any place p_i may become the final place of T_a (i.e., p_{n-1}), based on the outcome of the execution of a_i. In this case, the vote of a_i immediately becomes the outcome of the transaction T_a. In other words, a_i unilaterally decides the outcome of the transaction. This is different from traditional distributed transactions. Moreover, in traditional distributed transactions, a participant can only unilaterally *abort* a transaction, namely by issuing a `NO-VOTE`. In contrast, a_i can also unilaterally decide to continue T_a, in addition to the abort decision.

To summarize, the outcome of T_a at S_i solely depends on the result of the stage operations of a_i on S_i and on the value of i:

- At $\{S_i | i < n-1\}$: the agent a_i casts either a `YES-VOTE` or `NO-VOTE`. A `NO-VOTE` immediately results in an abort of the transactional mobile agent execution T_a (see Figure 5.10). A `NO-VOTE` is cast when the stage action operations semantically fail. Successful operations lead to a `YES-VOTE` and allow the agent to proceed with the transaction execution. Only the `ABORT` decision is communicated to all the participants p_j $(0 < j < i)$.

[1] Remember that we are only addressing semantic failures at this point.

- At S_{n-1}: the agent a_{n-1} decides either `ABORT` or `COMMIT`, depending on whether the operations at stage S_{n-1} have failed or succeeded. A successful execution of the operations at S_{n-1} implies that all operations of the agent a have successfully executed (i.e., voted `YES-VOTE`) and the transaction is thus ready to commit. Figure 8.1 illustrates a successful transactional mobile agent execution, where place p_3 decides `COMMIT`.

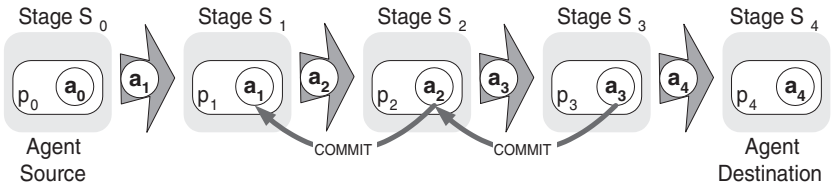

Figure 8.1: An agent execution that commits.

8.2 Generalization of the Itinerary

8.2.1 Itinerary Choices

A transactional mobile agent execution permits itinerary choices. For example, the agent owner may specify that the agent should sequentially visit car rental companies Herz and Avis. As soon as it receives the desired car from any one of them, the agent terminates. Failed requests for a car rental are ignored and the execution on this place locally aborted. The commitment only spans places that have successfully executed the service requests. Figure 8.2 illustrates the example, where the agent first books a flight from Swiss Air Lines and then attempts to rent a car from Herz. As Herz does not have any more rental cars available, agent a (i.e., a_3) moves to Avis' server. The stage actions of a_2 on p_2 can be locally aborted without aborting the entire transactional mobile agent execution. The outcome of the mobile agent execution, i.e., the decision `COMMIT` or `ABORT`, will not be sent to p_2.

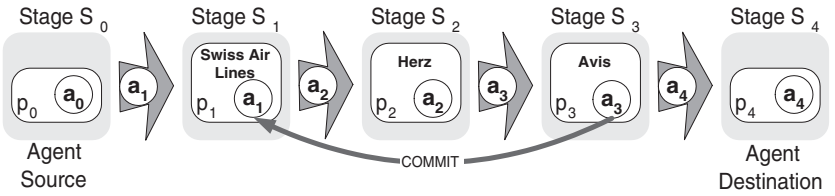

Figure 8.2: A committing agent execution with choices. Herz does not have a rental car available and thus is not part of the global commitment.

8.2.2 Generalization To Non-Linear Itineraries

So far, we have only considered mobile agent executions with linear itineraries (see (a) in Figure 8.3). An agent, however, can spawn transactional child agents[2], which lead to non-linear itineraries: itineraries that terminate in a single place (see Figure 8.3 (b)), and itineraries that terminate in several endpoints (see Figure 8.3 (c)). Case (b) contains itineraries where all parent and child agents meet again on a common place (e.g., at p_4 in Figure 8.3 (b)). Assume, for instance, that transaction of agent T_b acquires clothes, while T_b' buys books. Both transactional agents can run in parallel, and their results are collected at place p_4.[3] In contrast, parent and child agents finish the execution at different agent destinations in (c) (e.g., at p_5, p_4', and p_4''). An example of (c) is a transaction that reconfigures routers in different subnets. At the occurrence of a subnet, a new transactional child agent is spawn recursively.

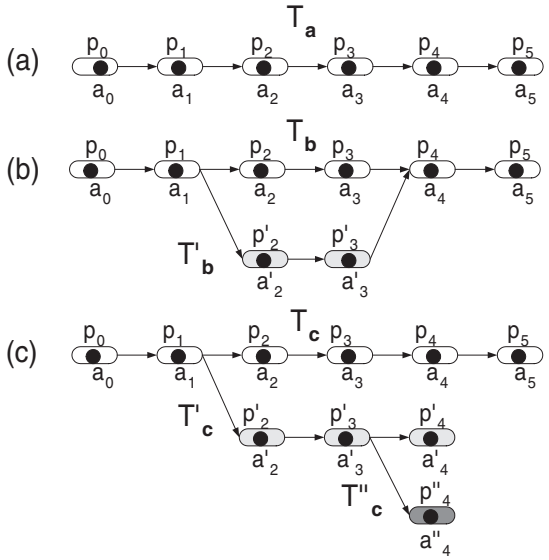

Figure 8.3: Classification of mobile agent executions.

Atomicity is more difficult to ensure in cases (b) and (c) than in case (a). Assume, for instance, that transactional child agent T_b' aborts on p_3'. Transactional agent T_b continues the execution until it reaches p_4 and there waits for T_b', which never arrives. The simplest approach is to wait for a certain time and then abort. However, this may lead to a prematurely unnecessary abort if T_b' is just slow (but has not aborted). Another approach is to abort T_b' immediately, but still forward the agent to p_4, where it notifies T_b of the abort. T_b thus always waits for the agent to arrive. This

[2]Note that in Section 6.5.1, we have discussed the problem of spawning child agents in the context of replicated mobile agents. Here, we consider a non-replicated agent, as we assume that no infrastructure failures occur.

[3]Note that the approach depicted in Figure 4.2 is different from case (b) in Figure 8.3. While the former uses agent replication to provide fault tolerance, the latter starts a new child transaction T_b'. Child transaction T_b' is generally not identical to T_b; rather it executes different operations. Sher et al. [SAE01] use this approach. See Chapter 5 for an in-depth comparison with our approach.

has the drawback that additional communications are required, but the transactional mobile agent never prematurely aborts.

Case (c) is handled by applying the following transformation: instead of terminating the transactional child agents on p'_4 and p''_4, these agents report back to their parent agent on place p_5. This transformation converts case (c) into case (b) and allows to reuse the approaches discussed before.

For simplicity, we only address linear itineraries in the rest of this thesis. However, all presented concepts can easily be extended to itineraries of classes (b) and (c).

8.3 The Problem of Infrastructure Failures

So far, we have not considered infrastructure failures. Failing components in the system may lead to blocking or to a violation of the atomic execution of the transactional mobile agent. Figure 8.4 illustrates a crash at stage S_3. In an asynchronous system, where no bounds on communication delays nor on relative processor speed exist, p_0, p_1, and p_2 are left with the uncertainty of whether p_3 has actually failed or is just slow [FLP83]. In addition, it is impossible for p_0, p_1, and p_2 to detect the exact point where p_3 has failed in its execution. More specifically, p_0, p_1, and p_2 cannot detect whether p_3 has succeeded in forwarding the agent to the next stage or not. Assume, for instance, that agent a_i (i.e., a_3 in Figure 8.4) issues a `YES-VOTE` but then crashes. If S_i has not succeeded to forward the agent to S_{i+1}, the agent execution is blocked. During this time, all locks acquired by transaction T_a at the previous places p_j $(j < i)$ remain with T_a and another transaction T_b has to wait. This dramatically reduces overall system throughput.

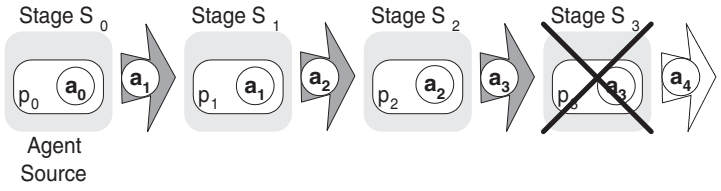

Figure 8.4: An agent execution crashes at stage S_3

To prevent blocking, another place such as p_{i-1} could monitor place p_i. If it detects the failure of p_i, it could then issue a `NO-VOTE`, which causes the transaction to abort. However, unreliable failure detection potentially leads to a violation of the atomicity property. Indeed, assume that p_j detects the failure of a_i. Place p_j thus assumes the responsibility for the decision and decides to abort transaction T_a. However, because of unreliable failure detection, p_j may erroneously suspect p_i. Actually, even if p_i has failed, it may have succeeded in forwarding the agent to p_{i+1}, resulting in potentially conflicting decisions on the outcome of the transaction: while p_j decides to abort the transaction, p_{i+1} may decide `COMMIT` if $a_{i+1} = a_{n-1}$, or cast a `YES-VOTE` otherwise. This conflicting outcome clearly violates the atomic execution property of the transaction's operations, as certain operations are aborted, whereas others are committed or may be committed later.

The approach we advocate uses replication to prevent blocking of the transactional mobile agent execution. At any time, the places know that the agent is still progressing and thus it is worth

to wait for the result. This alleviates the need to monitor the execution and prevents potentially conflicting outcomes to the atomic execution of the transactional mobile agents.

8.4 Specification

In this section, we specify the properties of the transactional mobile agent execution T_a associated with a mobile agent a. The entire execution T_a is specified in terms of the ACID properties [GR93]:

- (*Atomicity*) The stage executions of T_a are executed atomically, i.e., all of them or none are executed.
- (*Consistency*) A correct execution of T_a on a consistent state of the system (encompassing the places, the services running on them, and the agents) must result in another consistent system state.
- (*Isolation*) Updates of a stage execution of T_a on a place p_i are not visible to another transactional mobile agent T_b until T_a has committed in its entirety.
- (*Durability*) Committed changes by T_a are reflected in the system and are not lost any more.

Specifying the transactional mobile agent T_a in terms of the ACID properties implies that the sequence of stage actions $sa_1, \ldots, sa_{n-1}$ is executed as a transaction. Every stage action is itself composed of a set of operations $op_0, op_1, \ldots$ and has to run as a transaction as well. This transaction is called *local transaction* (see Section 4.3.1). Consequently, T_a can be modeled as *nested transactions*[4] [Mos85]. Recall that a nested transaction is a transaction that is (recursively) decomposed into subtransactions (see Section 5.3.2). In a transactional mobile agent execution, the top-level transaction (i.e., the transaction that has no parent) corresponds to the entire mobile agent execution T_a. Stage actions sa_i compose the first level of subtransactions, which are executed sequentially. According to the definition of nested transactions, these subtransactions may be aborted, but the parent transaction can still commit. In other words, if a service request fails on one place, the subtransaction sa_i can be aborted and retried as sa_k $(k > i)$ at another place. There is no need to abort the top-level transaction, which corresponds to our itinerary choices (see Section 8.2.1).

In the context of transactional mobile agents, consistency is ensured by the application composed of the mobile agent and the services running on the places. Isolation is discussed in Section 8.6. The properties we are mainly concerned with are atomicity and durability[5]. To ensure the atomicity property, all the places participating in the execution of the transactional mobile agent T_a need to solve an instance of the atomic commitment (AC) problem. We first give a blocking specification of the atomic commitment (AC) problem (Section 8.4.2). Informally, T_a commits if all stage actions sa_i $(0 < i < n)$ have executed successfully. Non-blocking atomic commitment

[4] See also [CR94] for a formal description of nested transactions.

[5] The distinction whether atomicity or durability has been violated is difficult for the reason already mentioned in Footnote 5 on page 53.

(NB-AC)[6] is considered in Section 8.4.3. Before giving the specifications of AC and NB-AC, we highlight an important difference between AC and NB-AC (see Section 8.4.1).

8.4.1 Correct and Faulty Machines, Places, or Agents

In our model, we assume that machines, places, and agents are good, i.e., are eventually always up (see Section 3.3). This assumption is instrumental in the atomic commitment problem (Section 8.4.2), as it allows the termination property to require that *every* place eventually decides. In other words, AC can only terminate if failed processes eventually recover. AC thus requires a model *with* recovery. For the decision, it might be necessary to wait for the recovery of a crashed place. This shows the blocking aspect.

On the contrary, the termination property of the non-blocking atomic commitment problem (Section 8.4.3) does not require recovery. Hence, an algorithm implementing NB-AC does not wait for the recovery of failed agents, places, or machines to reach a decision. For this reason, the termination property requires only correct places to decide, although other places might be down. This shows the non-blocking aspect.

8.4.2 Atomic Commitment Problem for Transactional Mobile Agents

The atomic commitment (AC) problem related to the execution of the transactional mobile agent T_a (assuming correct machines, places, and agents) is defined by the following properties:[7]

- (*Uniform agreement*) If two places p_i and p_j participating in the execution of T_a decide, they decide the same value (commit or abort).
- (*Uniform validity*) Place p_i $(0 < i \leq n-1)$ can decide `ABORT`. Place p_{n-1} decides either `ABORT` or `COMMIT`. Decision `COMMIT` is only possible after successful execution of the agent up to stage S_{n-1} and successful execution of stage action sa_{n-1}; otherwise the decision of p_{n-1} is `ABORT`.
- (*Uniform Integrity*) Every place p_j $(0 < j \leq i)$ decides at most once.
- (*Uniform*[8] *Termination*) Every place eventually decides.

It should be noted that deciding `ABORT` because of infrastructure failures is not admissible, as it potentially causes a violation of the atomicity property. Such a violation occurs if the agent has already moved to the next stage while the execution at the previous stages is aborted. Clearly, this could result in conflicting outcomes of the transaction (see Section 8.3). Consequently, an infrastructure failure blocks the transactional mobile agent execution. However, blocking is undesirable, as it dramatically limits overall system throughput (see Section 8.3).

[6] A specification for NB-AC in the context of traditional distributed systems (i.e., without mobile agents) is given for instance in [GHM+00].

[7] For simplicity, this specification is given with respect to the primary places participating in the agent execution. We will see in Section 8.5.1, that the AC actually runs among the so-called stage termination agents.

[8] Remember that all machines, places, and agents are assumed to be good.

8.4.3 Non-Blocking Atomic Commitment Problem for Transactional Mobile Agents

Blocking in the AC occurs if a decision on commit or abort is impossible until a failed place recovers. In the transactional mobile agent execution, this decision is made unilaterally by the agent (see Section 8.1.2). Indeed, the agent can decide whether to continue T_a (on p_i $(0 < i < n-1)$), commit (p_{n-1}), or abort (p_i $(0 < i \leq n-1)$). If p_i fails before the agent a_i has decided, then blocking occurs. At the same time, progress of the transactional mobile agent execution T_a is interrupted until p_i recovers (see Section 8.3). Blocking in the AC is thus a consequence of blocking in the mobile agent execution. Hence, non-blocking atomic commitment can be achieved by ensuring the non-blocking property in the execution of the mobile agent. In other words, it is sufficient to ensure that the decision in the AC algorithm is always possible.

Non-blocking adds another level of subtransactions to a transactional mobile agent execution. To prevent blocking, the agent at stage S_i is not executed on one place, but replicas of the agent are potentially executed on multiple places p_i^j. Consequently, subtransaction sa_i, in turn, can be modeled by yet another level of subtransactions sa_i^j, which correspond to the agent replicas $a_i^0, \ldots, a_i^m$ running on places $p_i^0, \ldots, p_i^m$ and executing the set of operations $op_0, op_1, \ldots$ (see Figure 8.5). Of the subtransactions sa_i^j at stage S_i, only one, called sa_i^{prim} and executing on the primary p_i^{prim}, is allowed to commit (if all its parent transactions commit): all others have to abort. This way it is ensured that the stage action sa_i is not executed multiple times. We recall that $\mathcal{M}_i$ is the set of places $p_i^0, \ldots, p_i^m$.

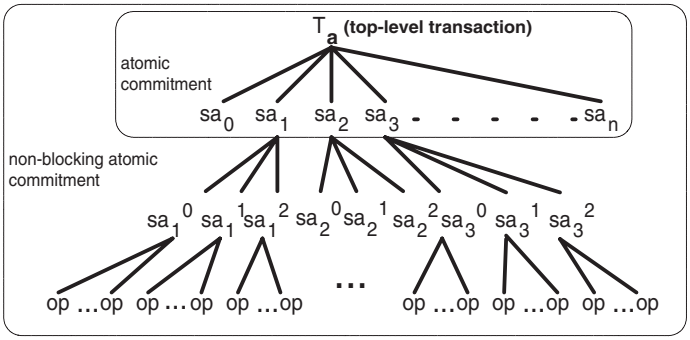

Figure 8.5: Scope of the AC and NB-AC specifications. NB-AC adds an additional layer of subtransactions sa_i^j, that execute the operations $op_0, op_1, \ldots$. In AC, subtransaction sa_i directly executes these operations without subtransactions sa_i^j.

The non-blocking atomic commitment (NB-AC) problem in the context of transactional mobile agents addresses correct and faulty machines, places, or agents and consists of two levels of agreement problems: (1) the *stage agreement problem*, and (2) the *global agreement problem*. Agreement problem (2) corresponds to the atomic commitment problem of Section 8.4.2: it ensures atomicity in the top-level transaction. On the other hand, (1) specifies the agreement problem among the places that execute the stage actions sa_i^j $(0 \leq j \leq m)$, where they decide which subtransaction may potentially commit. We begin by specifying the stage agreement problem (1) at each stage S_i:

Stage agreement problem:

- (*Uniform agreement*) No two places $p_i^j \in \mathcal{M}_i$ at stage S_i decide on a different primary p_i^{prim}.
- (*Validity*) The decision value p_i^{prim} is in the set $\mathcal{M}_i$ and p_i^{prim} has executed stage action sa_i (more specifically, sa_i^{prim}).
- (*Uniform integrity*) Every place p_i^j of stage S_i decides at most once.
- (*Termination*) Every correct place p_i^j of stage S_i eventually decides.

The decision on p_i^{prim} in the stage level agreement causes all other places $p_i^j \neq p_i^{prim}$ to abort the subtransactions sa_i^j. Consequently, the decision on p_i^{prim} is implicitly also a decision ABORT for all places $p_i^j \neq p_i^{prim}$. However, this is only related to the decision about the primary, not on ABORT or COMMIT. Indeed, subtransactions $sa_i^j \neq sa_i^{prim}$ abort, while sa_i^{prim} only aborts if its parent transaction aborts. However, this decision is again part of another agreement problem that is to be solved and is only taken by a_{n-1} at the end of the agent execution, and specified in the global agreement problem as follows (compare with Section 8.4.2):

Global agreement problem:

- (*Uniform agreement*) No two primaries p_i^{prim} and p_k^{prim} participating in the execution of T_a decide differently.
- (*Uniform validity*) Primary p_i^{prim} $(0 < i \leq n-1)$ can decide ABORT. Primary p_{n-1}^{prim} decides either ABORT or COMMIT. Decision COMMIT is a consequence of successfully executing the agent up to stage S_{n-1} and successfully executing sa_{n-1} at stage S_{n-1}. In all the other cases the decision is ABORT. Place $p_i^j \neq p_i^{prim}$ always decides ABORT (see stage agreement problem).
- (*Uniform Integrity*) Every place decides at most once.
- (*Termination*) Every correct place eventually decides.

An infrastructure failure does not allow to immediately decide ABORT. Rather, infrastructure failures cause the agent to execute on another place at the same stage. If this place provides the same service, the agent execution can proceed. Otherwise, a semantic failure occurs that, contrary to infrastructure failures, immediately results in an ABORT decision. Assume, for instance, that the agent a_i at stage S_i is entrusted with buying an airline ticket from Zurich to New York. Assume further that it executes on place p_i^j, that sells such tickets. A failure of p_i^j does not immediately abort subtransaction sa_i. Rather, a_i can be executed on another place p_i^k $(k \neq j)$ at stage S_i. If p_i^k provides the same service as p_i^j, i.e., also sells the same airline tickets, then sa_i succeeds, T_a can proceed and no reason for abort is given (see Section 8.2.1). In other words, infrastructure failures are masked by the redundancy of the agent at a stage (see the specification of the stage agreement problem).

8.5 Non-Blocking Transactional Mobile Agents

In this section, we show how our work in fault-tolerant mobile agent execution (see Chapter 6) can help to provide non-blocking transactional mobile agents. Indeed, fault tolerance in the context of mobile agents prevents that the execution of a mobile agent blocks because of the failure of a single component (e.g., an agent, place, or machine). Hence, it solves a problem similar to the stage agreement problem of NB-AC (see Section 8.4.3). However, fault-tolerant mobile agent execution generally only addresses infrastructure failures: semantic failures are not handled. In other words, it does not solve the global agreement problem of NB-AC. Revisiting the example in Section 3.4, fault-tolerant mobile agent execution prevents that the agent fails, but allows it to book a hotel room although no seat is available on a flight to New York. Hence, the approach in Chapter 6 is not sufficient to ensure atomicity of a mobile agent execution. Indeed, both infrastructure as well as semantic failures need to be covered. However, building on top of the approach in Chapter 6 allows us to easily solve the stage agreement and the global agreement problem (see Section 8.4.3) and provide non-blocking transactional mobile agent execution.

Informally, we execute the mobile agent in a non-blocking manner until it reaches place p_{n-1}. At p_{n-1}, an atomic commitment protocol is launched, which terminates (i.e., commits or aborts) the local transactions on the primaries. This protocol is also non-blocking. Eventually, all primaries must learn the outcome of the decision. Although some places may have failed in the NB-AC, the agent owner can continue using the result delivered by the mobile agent.

8.5.1 Solving the Stage Agreement Problem: From Fault-Tolerant to Transactional Mobile Agent Execution

An important property of our fault tolerance algorithm is non-blocking. This property is also desired for transactional mobile agent executions. Indeed, a transactional agent execution that blocks because of a failure on place p_i has probably acquired a large amount of locks on previous places p_j $(j < i)$. Holding these locks prevents other agents from accessing these data items and thus dramatically reduces overall system throughput. Non-blocking transactional mobile agent execution does not suffer from this problem. Progress is assured even in case of failures. This ensures that locks are released earlier and overall system throughput improves. Consequently, we reuse our work on non-blocking fault-tolerant mobile agents to prevent blocking in a transactional mobile agent execution. More specifically, we use the approach described in Chapter 6 to solve the stage agreement problem and thus build transactional mobile agents on top of it. The solution presented in Chapter 6 consists, for all agent replicas at stage S_i, to agree on (1) the place p_i^{prim} that has executed the agent, (2) the resulting agent a_{i+1}, and (3) the set of places of the next stage $\mathcal{M}_{i+1}$. In the context of fault-tolerant mobile agent execution, (1), (2), and (3) are important to prevent multiple executions of the agent, i.e. ensure the exactly-once property. All the places that have potentially started executing a_i, except p_i^{prim}, abort. Only p_i^{prim} commits the modifications of a_i. This corresponds to the stage actions sa_i^j in Figure 8.5. However, in fault-tolerant mobile agent execution, sa_i decides unilaterally which subtransaction to commit. More specifically, the decision is taken independently of the parent transaction, as no such transaction exists. Actually, the agreement on item (1) corresponds to the stage agreement problem of NB-AC and is reused by NB-AC. On the other hand, (2) and (3) are not relevant for NB-AC. In summary, we use our work

on fault-tolerant mobile agent execution to solve the stage agreement problem of the NB-AC.

8.5.2 Solving the Global Agreement Problem

To solve NB-AC (see Section 8.4.3), the modifications of a_i on the primary $p_i^{prim} \neq p_{n-1}^{prim}$ (i.e., the subtransaction sa_i^{prim}) are not immediately committed after the stage execution. In other words, stage action $sa_i^{prim} \neq sa_{n-1}^{prim}$ cannot unilaterally decide `COMMIT`. This is fundamentally different from the fault-tolerant mobile agent execution approach. The decision to `COMMIT` rather depends on the outcome of the top-level transaction (i.e., the result of the global agreement problem). Figure 8.6 shows an example transactional mobile agent that is aborted, although no infrastructure failures occur. Abort occurs, because the execution of a_2^0 has semantically failed on p_2^0. In contrary, the transactional mobile agent execution in Figure 8.7 is committed despite the failure of a_2^0. Here, a commit occurs because a has successfully executed all the stage actions a_i $(i = 0, \ldots, 2)$.

To terminate a pending transactional mobile agent execution, each primary place runs a stationary (i.e., not mobile) *stage action termination (SAT)* agent (see Figure 8.8). While agent a_{i+1} moves to p_{i+1}, the SAT agent sat_i waits for the outcome of the entire transactional agent execution, either (1) a commit message from a_{n-1} or (2) an abort message from a_j $(1 \leq j \leq n-1)$. Upon reception of an abort message, sat_i aborts the pending transaction sa_i^{prim}, otherwise commits it. Hence, SAT agent sat_i can be viewed as the transaction manager [GR93] of the local transaction represented by stage action sa_i. While the outcome of the transaction is undetermined, all data items accessed by a_i on place p_i^{prim} (i.e., the place that has executed sa_i) remain locked and are not accessible by other agents.

To improve the performance, the place p_i itself can offer the SAT service. The agent a_i registers subtransaction sa_i^{prim} that needs to be aborted or committed, and receives an ID. Using this ID, the agent can later contact the SAT service on p_i and initiate either a commit or an abort on the transaction. This service approach prevents the overhead of instantiating a SAT agent.

8.5.3 Terminating T_a

During its execution, agent a maintains a *SAT list* of all the SAT agents that it needs to contact in order to commit or abort the transaction. At every primary place p_i^{prim} ready to commit, a new entry is appended to this list. Unless the agent execution has failed on a previous stage, the execution at stage S_{n-1} decides whether to commit or abort the agent transaction. This decision is based on the outcome of the execution of a_{n-1} on place p_{n-1}^{prim}. If successful, the decision is `COMMIT`, otherwise `ABORT`. It is then communicated to $sat_i (1 \leq i \leq n-1)$, based on the SAT list. It is important that this decision eventually arrives at all destinations. Indeed, a destination sat_i that does not receive the decision message does not learn the outcome of the transaction T_a and still retains all locks on the data items. Hence, the decision message is distributed using a reliable broadcast mechanism [BG00] that ensures the eventual arrival of the message at all destinations. All correct places in $\mathcal{M}_{n-1}$ participate in the reliable broadcast to prevent that a failure of p_{n-1}^{prim} causes the loss of the decision message. Figure 8.8 depicts a successful example transactional mobile agent execution with 5 stages.

Clearly, our approach requires that aborts can be done by another process than the one that has started the transaction, even in the face of failures (see also Section 7.3.6).

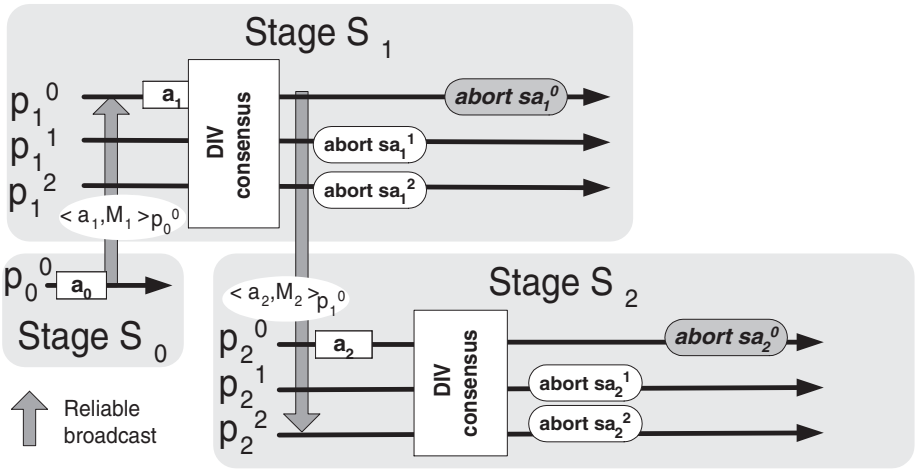

Figure 8.6: Transactional mobile agent execution, in which the top-level transaction is aborted at stage S_2. The local transactions on $p_i^0 = p_i^{prim}$ are only aborted (represented in italic font and with a shaded box) when the abort decision for the top-level transaction is known.

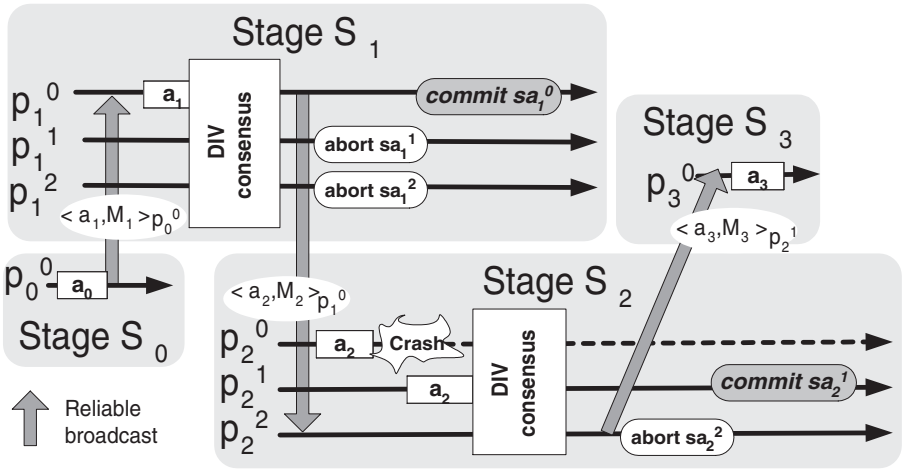

Figure 8.7: Transactional mobile agent that commits. Aborts on non-primary places are executed immediately, while primary places p_1^0 and p_2^1 only commit after a_2 has successfully executed on p_2^1.

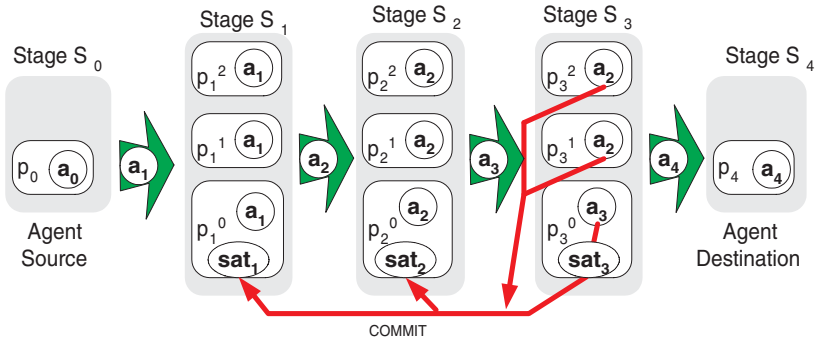

Figure 8.8: Committing the transactional execution of a mobile agent with 5 stages. The commit message is reliably broadcasted.

8.6 Multiple Concurrent Transactions

In our discussion so far we have focused on enforcing the atomicity property of a transactional mobile agent execution. In a real system, however, a transactional mobile agent does not execute in an isolated environment. Rather, it executes concurrently with other transactional mobile agents. Multiple transactions accessing concurrently the same data items may lead to a violation of the isolation property. Isolation is enforced locally by locking all accessed data items until the outcome of the transactional mobile agent execution, i.e., `COMMIT` or `ABORT`, is determined. Clearly, the isolation property limits the possible level of concurrency. As a remedy, services decide themselves whether they allow concurrent access to their data. For this purpose, they design a so-called commutativity matrix [Rak94], which shows potential conflicts among operations of this service and only allow operations that do not conflict to be executed concurrently.

Isolation also needs to be ensured on a global level, i.e., among places. Here, isolation is more difficult to achieve. One approach is to require that the stage actions on different places p_i and $p_j (j \neq i)$ are independent with respect to the execution order of stage actions. This approach is also used in Sagas, in which Garcia-Molina and Salem [GMS87] define the notion of a *saga*. A saga is a long lived transaction that can be broken up into a collection of subtransactions. This collection of subtransaction can be interleaved in any way with other transactions. Revisiting the example agent execution in Section 3.4, two transactional mobile agents T_a and T_b can execute the airline ticket purchase and the hotel room booking in any order, i.e., not necessarily in a serializable order, without violating the isolation property. Consequently, because the two services are independent, no global serializibility is required to preserve isolation. This improves the concurrency of the transactional mobile agent and hence overall system performance. Clearly, service independence is a property of the application as well as the services. In the Internet, independence among services of different service providers is usually given. We use the standard approaches [GR93] to enforce the other ACID properties.

8.6.1 The Problem of Deadlocks

With concurrent transactional mobile agents using pessimistic concurrency control, deadlocks may occur.

Example Deadlock With Three Transactional Mobile Agents

Assume, for instance, the example execution presented in Figure 8.9, which shows a deadlock among three transactional mobile agent executions T_a, T_b, and T_c. For simplicity, we assume transactional mobile agents whose execution may block (i.e., they do not use FATOMAS to provide non-blocking mobile agent execution). In this example, T_a has executed on p_1, p_2 and attempts to access a resource on p_3, whose lock is held by T_b. After acquiring the lock of the resource on p_3, T_b executes on p_4 and p_5. At the latter, it also waits for the lock of a resource to be released by T_c. T_c, in turn, is waiting on the lock of a resource acquired by T_a on p_3. Hence, a deadlock has occured among T_a, T_b, and T_c. This deadlock can only be resolved if one of the transactional mobile agents aborts or at least backs off, i.e., locally aborts the execution of the latest stage action (see Section 8.2.1).

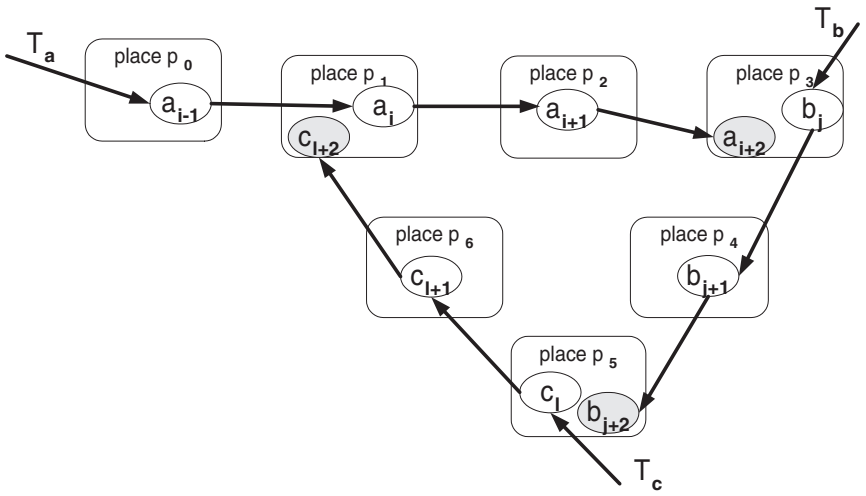

Figure 8.9: Deadlock among three transactional mobile agents: a_{i+2} waits for b_j to release its locks, b_{j+2} for c_l, and c_{l+2} for a_i.

In contrary to general nested transactions, where deadlocks within nested transactions can occur and need to be detected [Ruk91], deadlocks within a transactional mobile agent do not occur. Indeed, subtransactions sa_i generally execute strictly sequentially and on the local place.

Timeout-Based Deadlock Detection Algorithm

The timeout-based deadlock detection approach [GR93] is generally best suited in the context of transactional mobile agents. In the timeout-based algorithm, the transactional mobile agent

attempts to acquire a particular lock. If the lock is not granted after a certain time, the transactional mobile agent assumes that a deadlock has occurred and backs off. Unfortunately, the timeout-based approach may lead to erroneously detected deadlocks which cause unnecessary aborts. However, correct deadlock detection is generally impossible in an environment with heterogeneous databases. With heterogeneous databases, one cannot assume that all the databases make information about their concurrency control available to the public [BGMS92].[9] Consequently, a deadlock detection algorithm [Kna87, LM93, dMFnG$^+$99] may still erroneously detect a deadlock, although no deadlock has happened in the system. Assume, for instance, that transaction T_1 accesses data item d_1, while T_2 waits to access d_1. In the meantime, T_3 accesses d_2 and so blocks T_4. Without knowing the details of the concurrency control, an outsider does not know whether T_2 waits for T_1 or T_3. Consequently, deadlocks involving T_2 and T_3 may be erroneously detected, when, in fact, no deadlock involving these two transactions has occurred. This may cause unnecessary aborts of transactions.

Existing deadlock detection algorithms [Kna87, LM93, dMFnG$^+$99] usually build a so-called wait-for graph. The wait-for graph represents (sub)transactions as vertices. An edge between two (sub)transactions represents a dependency between these two (sub)transactions. Even if we assume that databases make information about the concurrency control mechanisms available, the wait-for graph for transactional mobile agents involved in e-commerce in the Internet becomes of a size that is not manageable any more. Actually, detecting a deadlock becomes even more complex in the case of non-blocking transactional mobile agents. Indeed, each replica agent a_i^j at stage S_i needs to start deadlock detection. Only if all of them detect a deadlock, then the transactional mobile agent execution is involved in a deadlock.

Interestingly, replication decreases the probability of a deadlock among transactional mobile agents. Indeed, a deadlock only occurs if all replica agents a_i^0, a_i^1, and a_i^2 of T_a at stage S_i are involved in a deadlock. If one replica agent is not involved in a deadlock, then this replica agent can execute and the transactional mobile agent execution can proceed. Actually, to achieve a deadlock among transactional mobile agents with replication degree 3 when no failures occur, a minimum of four transactional mobile agents are needed. To see this, consider the transactional mobile agents T_a, T_b, T_c, and T_d in Figure 8.10. Assume that the transactional mobile agents all access the same set of places $\mathcal{M}_w, \mathcal{M}_x, \mathcal{M}_y, \mathcal{M}_z$, but in a different order. Each $\mathcal{M}_i$ consists of three places. Then, a transactional mobile agent execution can only proceed the execution on a set of places, if less than three other transactional mobile agents have acquired the locks on these places.

Clearly, for certain applications in well-defined environments, other deadlock detection algorithms [Kna87] may be suitable. But generally, the timeout-based approach has the advantage of its simplicity and ease of implementation.

8.7 TRANSUMA

This section introduces TRANSUMA (TRANsaction SUpport for Mobile Agents), a prototype system that implements the approach developed in Section 8.5. We first present the architecture

[9] To our knowledge, there is currently no widely accepted standard available.

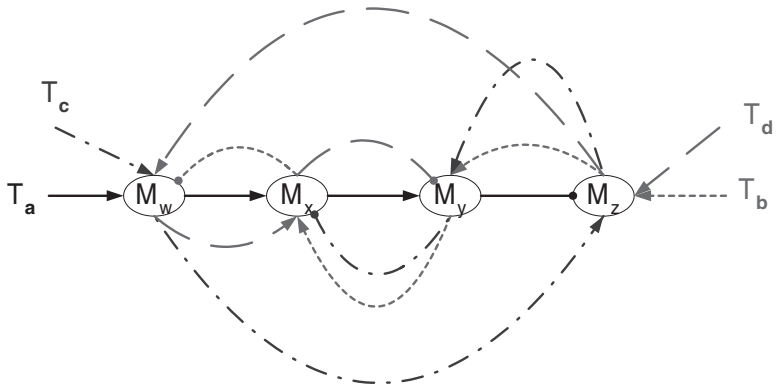

Figure 8.10: Deadlock occurring among four non-blocking transactional mobile agents. The itinerary of the transactional mobile agents is represented with arrows. For instance, T_d visits $\mathcal{M}_z$, $\mathcal{M}_w$, and $\mathcal{M}_x$ and participates in the deadlock at $\mathcal{M}_y$. A dot at the line end indicates that the transactional mobile agent execution cannot proceed at this stage.

of TRANSUMA (Section 8.7.1), before discussing performance evaluation results (Section 8.7.2). To improve readability in this section, method names are written with a `special font`.

8.7.1 Architecture

TRANSUMA is based on the agent-dependent approach (see Section 6.4.1), in which the transaction support mechanisms travel with the mobile agent. This has the important advantage, that underlying mobile agent platforms do not need to be modified. On the other hand, the agent-based approach results in an increased communication overhead because of a larger agent size.

TRANSUMA is based on FATOMAS, the fault-tolerant mobile agent system presented in Section 6.4. This dependency is also reflected in the architecture of TRANSUMA (see Figure 8.11). Here, a mobile agent a_i is composed of a TRANSUMA user-defined agent, i.e., the agent developed by the agent owner, and the *transaction support module (TSM)*. The TSM provides the mechanisms for transactional mobile agent execution. It is based on the *fault tolerance enabler* (FTE) of FATOMAS . Indeed, from the viewpoint of the FTE, the TSM and the TRANSUMA user-defined agent are just another FATOMAS user-defined agent. This has the advantage that the FTE can be reused without modifications. The TSM adds the SAT list to the decision value, which enables the implementation of the reliable broadcast to terminate the transactional mobile agent execution.

The TRANSUMA user-defined agent either interacts with the TSM (through the `TSM-API`) or with the services local to the place. The latter act as resource managers (RM) [GR93] (e.g., exporting an XA interface [ISO96] with the modifications presented in Section 7.3.6). The `TSM-API` provides functions to begin a subtransaction (i.e., the local transaction), to abort the subtransaction (i.e., `abort`), and to commit (i.e., `commitCurrentTransaction`) or abort (i.e., `abortCurrentTransaction`) the current transactional mobile agent execution. From the view point of the user-defined agent, the TSM assumes the role of a transaction manager. However, the TSM does not really act as

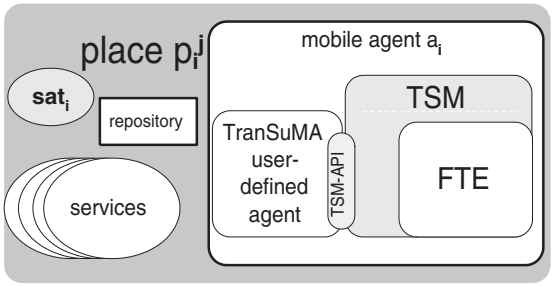

Figure 8.11: Architecture of TRANSUMA .

the transaction manager; rather, it forwards the calls of the user-defined agent to sat_i, which then sends the commit or abort to the local RM(s).

Typically, a stage action sa_i of the user-defined agent in a_i^j consists of (1) a call to begin the local transaction and (2) potentially multiple requests to local services, and (3) the end of stage. The stage may end with a call to `abortCurrentTransaction`, upon which the entire transactional mobile agent execution is terminated. The method `abortCurrentTransaction` is called, for instance, if a service request has terminated unsuccessfully and hence the transactional mobile agent execution cannot succeed any more. Note that the unsuccessful service request may not trigger the abort of the entire transactional mobile agent execution (see Section 8.2.1); rather, only the service requests of this stage action are aborted. This is achieved by calling the method `abort` in the `TSM-API`.

A call to method `commitCurrentTransaction` (or `abortCurrentTransaction`) triggers the commit (abort) of all unterminated subtransactions of the current transactional mobile agent. As shown in Section 8.5.3, transaction termination is achieved using reliable broadcast. We use a reliable broadcast with linear message cost [SGS84], which has the advantage of a lower number of messages compared to other strategies. Hence, the TSM first commits (aborts) the local transaction and then contacts sat_{i-1}. The SAT agent sat_{i-1} recursively does the same, i.e., commits (aborts) the local transaction and contacts sat_{i-2}.

8.7.2 Performance Evaluation

To measure the performance of TRANSUMA, we have implemented a prototype based on ObjectSpace's Voyager platform [Obj99]. Each place provides the simple counter service that has already been used for the performance evaluation of FATOMAS and that offers the method `increment` to increment the value of the counter, in addition to the standard methods to commit and abort/rollback the modifications.

Our performance tests consist in sending a number of agents that atomically increment a set of counters, one at each stage S_i. Each agent starts at the agent source and returns to the agent source (i.e., the agent source is identical to the agent destination). This allows to measure the round trip time of the agent. Between two agents, the places are not restarted. Consequently, the first agent needs considerably longer for its execution, as all classes need to be loaded into the cache of the virtual machines. Consecutive agents benefit from already cached classes and thus execute much

faster. We do not consider the first agent execution in our measurement results. Again, we assume that the Java class files are locally available on each place (see also Section 6.4.3).

The test environment is the same as for FATOMAS and the machines are also arranged in the same way. Our results represent the arithmetic average of 10 runs, with the highest and lowest values discarded to eliminate outliers. The coefficient of variations is in most cases lower than 5%. However, for few results, it went up to 15% because of variations in the network and machine loads.

We measure the costs of TRANSUMA compared to FATOMAS . The results in Figure 8.12 show that TRANSUMA adds an overhead of 6 to 20% compared to a FATOMAS agent. This overhead is caused by the transaction support mechanisms such as the communication with the local SAT agent and the commitment when the agent has reached stage S_{n-1}. Results are similar for pipelined (see Section 6.3.3) FATOMAS and TRANSUMA agents.

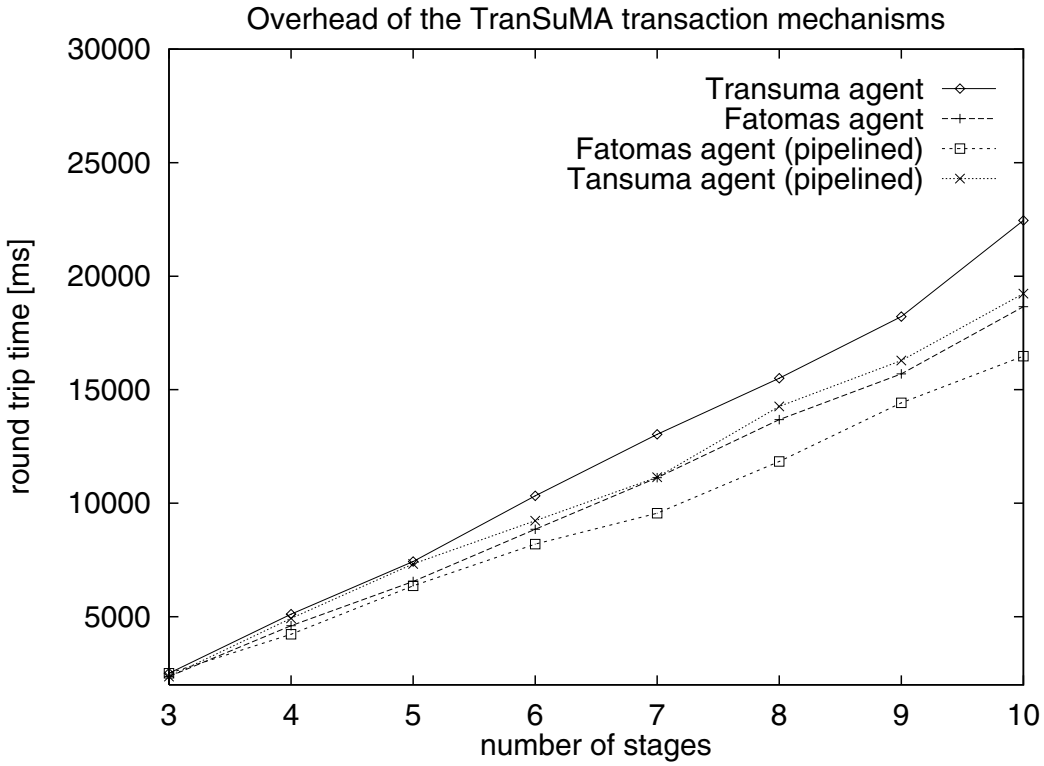

Figure 8.12: Round trip time [ms] of a TRANSUMA agent compared to a FATOMAS agent for itineraries between 3 and 10 stages.

Note that the costs for an abort of a transactional mobile agent execution at S_{n-1} are the same as for commit. Indeed, the only difference between abort and commit is the content of the message reliably broadcasted to all SAT agents.

Clearly, the relative overhead decreases if the execution time of the stage action increases. Indeed, the time needed to increment the counter is negligible. If this time becomes more important, the relative overhead of TRANSUMA compared to FATOMAS decreases considerably.

Infrastructure failures during the execution of stage action sa_i have a negative impact on the

performance of the transactional mobile agent. However, they mainly influence the performance of FATOMAS (see Section 6.4.3). With respect to the overhead of the transaction support mechanisms, they result in a greater number of messages in the reliable broadcast mechanism. Moreover, the roundtrip time as measured in Figure 8.12 may increase. Indeed, with the linear strategy for reliable broadcast, the senders wait for acknowledgments from the receivers. Unless this acknowledgment arrives within a certain time, they assume a failure and take adequate actions. This may increase the roundtrip time.

8.8 Supporting Compensating Transactions

In this chapter so far, we have considered the model of nested transactions presented in [Mos85]. This also corresponds to a model of *closed* nested transactions [WS92]. In this section, we relax the model to also accommodate compensating transactions. The use of compensating transactions has the advantage that the corresponding stage action can be immediately committed, without waiting for the outcome of the transactional mobile agent execution. In case the transactional mobile agent execution is aborted, the stage action is compensated by executing the compensating transaction. However, compensating transactions are not always feasible (see Sections 4.3.3 and 5.3.7). Extending the model of closed nested transactions to also incorporate compensating transactions leads to the model of *open nested transactions* [WS92].

This section is inspired by the work in [SR00, AS99, ASPZ00]. The advantage of our approach is that it can handle both compensatable and non-compensatable transactions at the same time, i.e., pessimistic and optimistic concurrency control. In contrary, Strasser and Rothermel [SR00] and [ASPZ00] assume compensatable transactions. In [AS99], Assis briefly mentions that it is possible to extend his approach to incorporate non-compensatable transactions, but no detailed discussion is provided.

Similarly to Sagas [GMS87], we assume that the sequence of stage actions of a transactional mobile agent can be interleaved in any way with other transactional mobile agents (see Section 8.6).

A large body of related work exists in traditional client/server computing. Closest to the work of this section, together with Sagas, is the ConTract model proposed by Reuter et al. [RSS97]. This model provides a conceptual framework for the reliable execution of long-lived computations in a distributed environment. However, the scripts (i.e., the control flow descriptions) are not mobile.

8.8.1 Open Nested Transactions

Open nested transactions [WS92] are a generalization of (closed) nested transactions as presented in [Mos85]. They allow subtransactions to be *open*, i.e., to commit prematurely without knowing the outcome of the top-level transaction. At this point, the modifications of the subtransaction are visible to other mobile agents.[10] If the top-level transaction is aborted, then open transactions are compensated by running compensation transactions (see Section 4.3.3). Recall that in closed nested transactions, subtransactions only commit if the parent transaction commits (see

[10]Note that in the context of open nested transactions, the isolation property in the specification in Section 8.4 needs to be relaxed. Indeed, intermediate results also become visible [RSS97].

Section 8.4). Hence, the results of the subtransactions are not visible to other transactions until the entire transaction commits, i.e., the parent transaction keeps the locks of its subtransactions.

Note that in the case of open nested transactions the atomicity property is achieved if the modifications of all stage actions are reflected [WS92] 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 as in the case of closed nested transactions.

8.8.2 Transactional Mobile Agents Modeled As Open Nested Transactions

A transactional mobile agent can be modeled as open nested transactions: the top-level transaction corresponds to the transactional mobile agent execution and the first level of subtransactions is again composed of the stage actions sa_i (see also Figure 8.5). They, in turn, are composed of subtransactions that run on all the places in $\mathcal{M}_i$, i.e., sa_i^j. The subtransactions sa_i^{prim} that are running on the primary place can be either closed or open. The subtransactions sa_i^j $(j \neq prim)$ are immediately aborted at the stage execution and it is thus not important whether they are open or closed. The first level of subtransactions (i.e., the stage actions sa_i) inherit the property of their subtransaction. In other words, if the subtransaction is open, they are also open, otherwise, they are closed subtransactions.

Note that with open nested transactions, the open subtransactions optimistically decide commit. Based on the outcome of the entire transactional mobile agent execution, the open subtransactions then make the "real" decision. Considering that the specifications of AC and NB-AC reflect this "real" decision, they are still valid in the case of open nested transactions. However, whereas a commit decision requires no further activities, the abort decision is more complex to handle and involves running compensation transactions.

In the following, we show how a transactional mobile agent execution can be compensated. For simplicity, we first assume that all stage actions sa_i^{prim} are open, before considering the general case where subtransactions can be either open or closed.

Supporting Only Open Subtransactions

We reuse the approaches presented in [SR00, AS99] to achieve a compensation of transactions in case the top-level transaction T_a is aborted. For simplicity, we assume that all stage actions can be compensated, i.e., we assume the model of sagas [GMS87, GMGK$^+$91]. Hence, no deadlocks among transactional mobile agents occur. As indicated in [SR00], compensating a transactional mobile agent involves compensating the transactions that have lead to modifications in the state of the place *and* in the agent state. The reason for this is that it is not always possible to simply return to the state the agent had before executing the stage action. Indeed, assume a transaction that buys a book from a book shop and pays with e-coins [SR00]. The compensating transaction returns the book and receives the money in return. However, it is very unlikely that it will again receive exactly the same e-coins; rather, it will receive the same amount of e-coins (unless some compensation fee is deduced). Hence, the state of the agent before executing stage action sa_i, and after executing stage action sa_i and its compensating transaction csa_i is not identical. This state is called *weakly reversible* [SR00]. In contrary, *strongly reversible* state contains data that can be compensated by retrieving its state before having executed the stage action from the log.

Weakly reversible agent state requires that compensating transactions have to be defined on both the agent state and the place state. In contrary, strongly reversible agent state only requires that the compensating transaction acts on the place state. The agent state corresponds to the state before executing the corresponding stage action [SR00] and can be restored from the log, to which it is written before executing the stage action. The agent state may consist of strongly and weakly reversible parts.

Clearly, the compensating transactions can only run on the primaries p_i^{prim} of the transactional mobile agent execution. If the places visited by agent a are all independent and agent a only consists of strongly reversible state, we can use the mechanisms presented in Section 8.5.3. Indeed, instead of aborting the subtransactions, the SAT agents sat_i run a compensation transaction csa_i. The independence between the places causes the order to be irrelevant.

Generally, however, the compensating transactions csa_i must be run in the inverse sequence of their corresponding stage actions sa_i (see Figure 8.13). For this purpose, the primary place that decides abort launches a compensating mobile agent ca_2, that travels back along the sequence of primaries of transactional mobile agent a (i.e., $p_2^{prim}, p_1^{prim}, p_0$) and runs the corresponding compensating stage action at each place. Note that $ca_2 = a_3$ because the last stage action sa_3 is aborted immediately and not committed. Hence, it does not need to be compensated. The compensating stage action modifies both the agent stage and the state of the place.

The compensating mobile agent ca needs to run successfully to its completion. In other words, it must be ensured that all compensating transactions execute successfully. This requires, that the compensating mobile agent survives failures of the place. Hence, the state and code of the agent need to be logged. Checkpoints are made first when the agent arrives at a new place; only then is the execution of the compensating transaction started. If the place fails before the compensating transaction is committed, the compensating transaction is restarted when the place recovers. The compensating agent ca_i is reliably forwarded to the next stage S_{i-1} by repeatedly sending it to S_{i-1} until it is checkpointed at S_{i-1} and an acknowledgment is received at S_i.

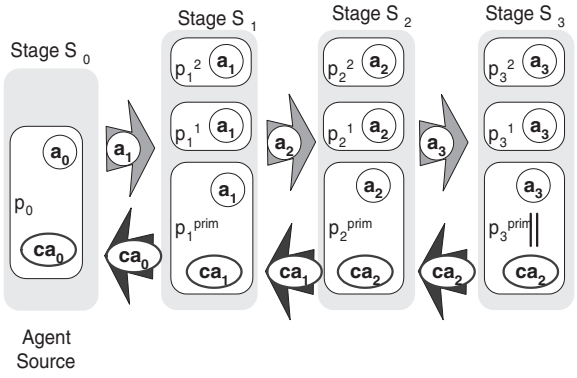

Figure 8.13: Execution of the compensating agent ca.

Clearly, the execution of the compensating agent is prone to blocking. Indeed, assume that primary p_k^{prim} has failed while or before executing ca_k. Before the stage actions on p_l^{prim}, $(l < k)$ can be compensated, the compensating agent needs to wait until p_k^{prim} recovers. If blocking

occurs in the execution of ca, the agent owner will only learn about the unsuccessful outcome of the transactional mobile agent when ca will have recovered and finished its execution. However, the outcome of the transactional mobile agent is known at the stage S_k that decides the abort. Hence, the places in $\mathcal{M}_k$ reliably send the abort information to the agent owner. At this point, the agent owner knows the outcome of the transactional mobile agent execution and can launch a new transactional mobile agent, if needed. Note that the agent a_0 at stage S_0 may also change after running the compensating agent. The agent owner needs to be aware of this when sending another (duplicate) agent a.

Supporting Open And Closed Subtransactions

A transactional mobile agent that supports both open and closed subtransactions may need to run compensating transactions for the agent state even on places where it has executed a closed subtransaction. Indeed, assume that agent a_i has executed a closed subtransaction at stage S_i. If a_i's state consists of weakly reversible state and has been changed by running stage action sa_k and the corresponding compensating stage action csa_k $(k > i)$, the state of ca_{i-1} is different from the state of a_i. Hence, it is not sufficient to simply retrieve agent a_i from the log file; rather, the compensating stage action needs to be run on ca_i to obtain ca_{i-1}, that corresponds to a_i. However, the unterminated closed subtransactions on the place can simply be aborted.

Aborting the closed subtransactions only when the compensating agent runs on the corresponding places results in poor performance. Indeed, as the execution of the compensating agent may block, the unterminated closed transactions keep their locks and no other mobile agents can access the locked resources. The execution of the compensating agent can only continue once the failed place has recovered again. As a remedy, the closed subtransactions are terminated as described in Section 8.5.3. But the compensating agent may still need to run on these places to compensate the state of the agent.

We have shown that modeling transactional mobile agents as open nested transactions is more generic than using the model of closed nested transactions. However, this model results in an increased complexity and difficulties for the developer to create mobile agents. Indeed, the developer needs to be aware and foresee all the compensating transactions along the execution of the transactional mobile agent. Moreover, deadlocks may again occur among transactional mobile agents with open and closed subtransactions.

8.9 Summary

In this section, we have introduced the problem of transactional mobile agent execution. As this problem is similar to transactional execution in the client/server paradigm, we have reused nested transactions to model transactional mobile agents. Because the approach in [SR00] builds on the approach in [RS98], it suffers from the same blocking problem (see Section 5.2.1). Based on our approach to fault-tolerant mobile agent execution, we show how non-blocking transactional mobile agent execution can be achieved. To our knowledge, we are the first to provide a specification of non-blocking atomic commitment in the context of mobile agents, that ensures termination of the atomic commitment even in the case of failures. Similarily to fault-tolerant mobile agent

execution, we have validated our approach in a prototype system called TRANSUMA. Our performance analysis shows that the overhead is reasonable compared to a non-transactional FATOMAS agent. Finally, we have shown that our approach can also handle compensating transactions.

Chapter 9

Conclusion

This chapter assesses the research performed in the context of the thesis and presents issues for further research.

9.1 Research Assessment

This work has addressed the following issues in the context of mobile agents: (1) how to achieve fault tolerance and (2) how to achieve execution atomicity in general transactional mobile agents. Both issues are related because non-blocking execution is crucial for both. We have given a classification of existing approaches and have seen how the present approaches are positioned within this classification. This classification facilitates an understanding of the strengths and drawbacks of existing approaches from an algorithmic point of view. In the following, we first highlight the contributions of our approach for fault-tolerant mobile agent execution, before assessing our work on transactional mobile agents.

9.1.1 Fault-Tolerant Mobile Agent Execution

To achieve fault tolerance in the context of mobile agents, we first have specified fault-tolerant mobile agent execution in terms of two properties: non-blocking and exactly-once execution. A mobile agent execution is blocking if a failure of either agent, place, or machine prevents the agent from continuing in its execution. The exactly-once problem is related to non-blocking in the sense that solutions to the latter (i.e., replication) may lead to multiple executions of the mobile agent. Hence, a solution to fault-tolerant mobile agent execution needs to ensure both the non-blocking and exactly-once properties. In contrast to existing approaches, which are either blocking [RS98], use a very complex model of transactions and leader election [ASPZ98, RS98], or assume reliable failure detection [JMS+99, PPG00], we model fault-tolerant mobile agent execution as a sequence of agreement problems. At each stage of the execution, the agent replicas solve an agreement problem. Thereby, replication overcomes the blocking problem, whereas the agreement ensures the exactly-once property.

We have identified two basic building blocks to implement the present approach: (1) DIV consensus and (2) reliable broadcast. Building block (1) solves the agreement problem at a stage of

the mobile agent execution. Building block (2), on the other hand, ensures that the mobile agent is reliably forwarded to the next stage. By selecting an adequate implementation for these building blocks consistent with the system model, the present approach is also applicable to other system models.

We have implemented a system called FATOMAS and measured its performance. We have seen that the fault tolerance mechanisms incurs an overhead. Consequently, fault-tolerant mobile agents should only be used for critical applications with non-idempotent operations, in which inconsistent state is to be avoided. Such applications generally include financial operations (e.g., money withdrawal or acquisition of goods). Agents that execute only idempotent operations do not need to use the fault tolerance mechanisms proposed in the thesis. Instead, they can send another agent when the previous agent is suspected to have failed (even if this suspicion is wrong).

Clearly, the performance of FATOMAS can be improved by tuning its parameters, but also by improving the performance of the underlying mobile agent platform.

Although the current research was performed in the context of mobile agents, some results are valid in the context of the traditional client/server paradigm. In particular, this thesis argues that the case of a replicated agent executing on replicated iso-places is a special case of replicated invocation (i.e., a replicated client invoking a replicated server) in traditional client/server computing. Ensuring non-blocking and the exactly-once execution property in this context is a difficult problem and its solution depends on execution determinism/non-determinism and the applied concurrency control technique. While this problem is solved if the client replicas execute deterministically [Maz96, Nar99], this thesis proposes a solution in the case of non-deterministic client replicas. This solution is based on the idea of sharing sufficient undo information to enable the replicas to undo multiple executions caused by the failure of a replica. At the same time, it also prevents blocking and ensures exactly-once in the context of replicated iso-places.

9.1.2 Transactional Mobile Agent Execution

We have shown that the presented approach to fault-tolerant mobile agent execution can be used to achieve non-blocking transactional mobile agent execution. The non-blocking property is very important if pessimistic concurrency control is applied. Indeed, blocked transactional mobile agents keep all the locks on accessed data items and thus prevent other transactional mobile agents to access these data items until the failed agent recovers. In this context, the thesis has started with a discussion on the reasons for a violation of the atomicity property within a single transactional mobile agent. Then, it models a transactional mobile agent as a simple form of nested transactions. The specification for non-blocking atomic commitment in the context of transactional mobile agents is novel, to our knowledge, and shows that the non-blocking property of atomic commitment is related to the non-blocking property of the mobile agent execution. More specifically, non-blocking atomic commitment can only be achieved with non-blocking mobile agent execution. The reason for this is that the decision is made unilaterally by the agent itself. We have implemented a prototype system, called TRANSUMA, using the agent-dependent approach, and have measured its performance.

The model of closed nested transactions has the inconvenience that the locks on data items are only released when the outcome of the transactional mobile agent execution is known. For certain

applications, it may be possible to release these locks earlier, i.e., use optimistic concurrency control based on compensating transactions. This thesis also shows how our approach can be extended to handle compensating transactions. As compensating transactions may block, the agent owner must be reliably notified about the outcome of the transactional mobile agent execution.

9.2 Open Research Issues

The following open issues, among others, remain in the area of fault-tolerant and transactional mobile agent execution and need to be further investigated:

Inter-agent communication: The communication between two mobile agents is called inter-agent communication. Inter-agent communication between fault-tolerant or transactional mobile agents becomes more difficult than the communication between two agents that are not fault-tolerant. Indeed, instead of communicating with one copy of the receiving agent, the sending agent needs to send the message to all replicas of the receiving agent. This requires that a directory service be set up, which allows replicas and their location to be identified [MLC98]. Although this problem appears to have similarities with replicated invocations (see Chapter 7), it needs to be studied in more detail.

Performance Measurements: The focus on the performance measurements in this thesis has been on a single mobile agent execution. Additional performance measurements are required to determine the throughput of a system with multiple concurrently running fault-tolerant or transactional mobile agents. In particular, it would be useful to achieve an understanding on the probability for deadlock and the cost of resulting aborts in the context of transactional mobile agents.

Performance Improvement: In order to become a viable product, the two prototypes presented in the thesis need to be improved considerably. In particular, their communication mechanisms need to be made more efficient. Moreover, they need to be ported to a mobile agent platform that is still supported and where development is going on, such as the open-source platform Aglets [Agl, LO98]. Indeed, a first attempt to port FATOMAS to Aglets has been made, but was stopped because of a bug in Aglets, which prevented its use for FATOMAS. However, this bug may have been corrected in later releases of Aglets. In general, the mobile agent platforms used here do not appear to have yet reached the same degree of maturity as client/server middleware generally has.

Deadlock Detection: An interesting field for further research is deadlock detection in the context of non-blocking transactional mobile agents as presented in Section 8.5. In particular, one could investigate whether and under what conditions and assumptions a deadlock detection mechanism may be more suitable than the timeout-based approach.

Reliable Multicast: In the present work, we have used reliable multicast as a building block for fault-tolerant and transactional mobile agent execution. In practical agent based applications, it may be useful to specify reliable multicast in such a way that the specification gives

certain guarantees about when and to whom a message is delivered. For instance, if a process does not fail for a certain period of time, it will receive the message. Clearly, this requires adding timing assumptions to the model. Which timing assumptions are the most realistic and adequate in this context is an interesting avenue for further research.

Unified Model for Fault-Tolerant and Transactional Mobile Agents: In the present work, we use agreement problems and the notion of a transaction to model fault-tolerant and transactional mobile agent execution. These are important concepts usually used in different fields of research. Indeed, transactions are important in the context of databases, whereas agreement problems are a fundamental building block in the field of replication. Exploring the potential for unification of these two concepts in the context of mobile agents could lead to a single model. Having such a model would considerably simplify the presentation of the problem of fault-tolerant and transactional mobile agent execution. Moreover, this research could lead to a model that is also important in a larger context and could help bring the fields of transactions and replication closer.

Bibliography

[ACDF96] J.M. Andrade, M. Carges, T. Dwyer, and S. Felts. *The Tuxedo System: Software for Constructing and Managing Distributed Business Applications*. Addison-Wesley, Reading, Massachusetts, USA, September 1996.

[ACT97] M.K. Aguilera, W. Chen, and S. Toueg. Quiescent reliable communication and quiescent consensus in partitionable networks. Technical Report TR 97-1632, Cornell University, June 1997.

[ACT00] M.K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, 2000.

[Agl] Aglets. http://www.aglets.org.

[AS85] B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21, October 1985.

[AS99] F.M. Assis Silva. *A Transaction Model based on Mobile Agents*. PhD thesis, Informatik, Technische Universität Berlin, June 1999.

[ASK97] F.M. Assis Silva and S. Krause. A distributed transaction model based on mobile agents. In Kurt Rothermel and R. Popescu-Zeletin, editors, *Mobile Agents, Proc. of the 1st Int. Workshop, MA'97*, LNCS 1219, pages 198–209. Springer Verlag, April 1997.

[ASPZ98] F.M. Assis Silva and R. Popescu-Zeletin. An approach for providing mobile agent fault tolerance. In Kurt Rothermel and F. Hohl, editors, *Proc. of the 2nd Int. Workshop on Mobile Agents (MA'98)*, LNCS 1477, pages 14–25. Springer Verlag, September 1998.

[ASPZ00] F.M. Assis Silva and R. Popescu-Zeletin. Mobile agent-based transactions in open environments. *IEICE Trans. Commun.*, E83-B(5), May 2000.

[Avi95] A.A. Avižienis. The methodology of n-version programming. In M.R.Lyu, editor, *Software Fault Tolerance*, pages 23–46. John Wiley, New York, USA, 1995.

[Bar95] J. Barnes. *Programming in Ada95*. Addison-Wesley, Reading, Massachusetts, USA, 1995.

[BCBT96] A. Basu, B. Charron-Bost, and S. Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In O. Babaoglu and K. Marzullo, editors, *Proceedings of the 10th International Workshop WDAG'96, Bologna, Italy*, LNCS 1151, pages 105–122. Springer Verlag, October 1996.

[BG00] R. Boichat and R. Guerraoui. Reliable broadcast in the crash-recovery model. In *Proc. of 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, Nuremberg, Germany, October 2000.

[BGMS92] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *VLDB Journal*, 1:181–539, October 1992.

[BHB+90] P.A. Barrett, A.M. Hilborne, P.G. Bond, D.T. Seaton, P. Veríssimo, L. Rodrigues, and N.A. Speirs. The delta-4 XPA extra performance architecture. In *Proc. of 20th Int. Symposium on Fault-Tolerant Computing Systems (FTCS'90)*, pages 481–488, Newcastle, UK, 1990.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts, USA, 1987.

[BJ87] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proc. of 11th ACM Symposium on Operating Systems Principles*, pages 123–138, 1987.

[BMST93] N. Budhirja, K. Marzullo, F.B. Schneider, and S. Toueg. *The primary-backup approach*, pages 199–216. In Mullender [Mul93], 2nd edition, 1993.

[BPW98] A. Bieszczad, B. Pagurek, and T. White. Mobile agents for network management. *IEEE Communications Surveys*, September 1998.

[BSH02] W. Binder, G. Di Marzo Serugendo, and J. Hulaas. Towards a secure and efficient model for grid computing using mobile code. In *Proc. of 8th ECOOP Workshop on Mobile Object Systems*, Málaga, Spain, June 2002.

[CFN90] D. Chaum, A. Fiat, and M. Naor. Untraceable electronic cash. In Shafi Goldwasser, editor, *Advances in Cryptology — CRYPTO '88*, LNCS 403, pages 319–327. Springer Verlag, 1990.

[CGH+95] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, and G. Tsudik. Itinerant agents for mobile computing. *IEEE Personal Communication Systems*, 2(5):34–49, October 1995.

[CHK98] D. Chess, C.G. Harrison, and A. Kershenbaum. Mobile agents: Are they a good idea? In G. Vigna, editor, *Mobile Agents and Security*, LCNS 1419, pages 25–47. Springer Verlag, 1998.

[CM84] Jo-Mei Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems (TOCS)*, 2(3):251–273, 1984.

[Cow01] D. Coward. *Java Servlet Specification – Version 2.3*. Sun Microsystems, August 2001.

[CR94] P.K. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems (TODS)*, 19(3):450–491, 1994.

[CT96] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J ACM*, 43(2):225–267, March 1996.

[CTA02] W. Chen, S. Toueg, and K. Aguilera. On the quality of service of failure detectors. *Transactions on Computers*, 51(1):13–32, January 2002.

[Déf00] Xavier Défago. *Agreement-Related Problems: From Semi-Passive Replication to Totally Ordered Broadcast*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, August 2000. Number 2229.

[DEW97] R.B. Doorenbos, O. Etzioni, and D.S. Weld. A scalable comparison-shopping agent for the world-wide web. In *In Proc. of 1st Int. Conference on Autonomous Agents (Agents'97)*, Marina del Rey, California, February 1997.

[dMFnG+99] J.R. González de Mendívil, F. Fariña, J.R. Garitagoitia, C.F. Alastruey, and J.M. Bernabeu-Auban. A distributed deadlock resolution algorithm for the AND model. *IEEE Transactions on Parallel and Distributed Systems*, 10(5), May 1999.

[DS00] X. Défago and A. Schiper. Semi-passive replication and Lazy Consensus. Technical Report DSC/2000/027, École Polytechnique Fédérale de Lausanne, Switzerland, May 2000.

[DSS98] X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proc. of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98)*, pages 43–50, West Lafayette, Indiana, October 1998.

[FG00a] S. Frølund and R. Guerraoui. Implementing e-transactions with asynchronous replication. In *Proc. of Int. Conference on Dependable Systems and Networks (DSN'00)*, pages 449–458, New York, 2000.

[FG00b] S. Frølund and R. Guerraoui. X-ability: a theory of replication. In *Proc. of the 19th ACM Symposium on Principles of Distributed Computing (PODC'00)*, pages 229–237, Portland, Oregon, 2000.

[FG01a] S. Frølund and R. Guerraoui. Implementing e-transactions with asynchronous replication. *IEEE Transactions on Parallel and Distributed Systems*, 12(2):133–146, February 2001.

[FG01b] S. Frølund and R. Guerraoui. X-ability: a theory of replication. *Distributed Computing*, 14(4):231–249, December 2001.

[Fip] Foundation for intelligent physical agents (FIPA). http://www.fipa.org.

[FKT01] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Journal of Supercomputer Applications*, 2001.

[Fla97] D. Flanagan. *Java in a Nutshell*. O'Reilly, Sebastopol, CA, USA, 2nd edition, May 1997.

[FLP83] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. In *Proc. of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 1–7, Atlanta, Georgia, March 1983.

[FPV98] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[Gen95] General Magic. *The Telescript Language Reference*, 1995. General Magic, Sunnyvale CA.

[GFP99] T. Gschwind, M. Feridun, and S. Pleisch. ADK – building mobile agents for network and systems management from reusable components. In *Proc. of 1st Int. Conference on Agent Systems and Applications/Mobile Agents (ASAMA'99)*, Palm Springs, CA, USA, October 1999.

[GHM+00] R. Guerraoui, M. Hurfin, A. Mostefaoui, R. Oliveira, M. Raynal, and A. Schiper. Consensus in asynchronous distributed systems: A concise guided tour. In S. Shrivastava S. Krakowiak, editor, *Advances in Distributed Systems*, LNCS 1752, pages 33–47. Spinger Verlag, 2000.

[GM82] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, 31(1), January 1982.

[GMGK+91] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem. Coordinating activities through extended sagas: A summery. In *Proc. of 36th Computer Society International Conference (Compcon)*, pages 568–573, 1991.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *Proc. of ACM SIGMOD Int. Conference on Management of Data and Symposium on Principles of Database Systems*, pages 249–259, San Francisco, CA, 1987.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, USA, 1993.

[Gra81] J. Gray. The transaction concept: virtues and limitations. In *In Proc. of Int. Conference on Very Large Databases*, pages 144–154, Cannes, France, 1981.

[Gra96] R.S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In *Forth Annual Usenix Tcl/Tk Workshop*, 1996.

[GS96] R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. In *Reliable Software Technologies - Ada-Europe'96*, LNCS 1088, pages 38–57. Springer-Verlag, June 1996.

[Gue95] 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, September 1995. Springer Verlag.

[HMR98] M. Hurfin, A. Moustefaoui, and M. Raynal. Consensus in asynchronous systems where processes can crash and recover. In *Proceedings of the 17th Symposium on Reliable Distributed Systems (SRDS'98), West Lafayette, IN, USA*, pages 280–286, October 1998.

[HR83] T. Härder and A. Reuter. Principles of transaction oriented database recovery — a taxonomy. *ACM Computing Surveys*, 15(4):289–317, December 1983.

[HT94] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, CS, University of Toronto; CS, Cornell University, May 1994.

[ISO96] ISO/IEC. *Information Technology - Distributed Transaction Processing - The XA Specification*, 1st edition, 1996. ISO/IEC 14834.

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

[Kar00] G. Karjoth. 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)*, LNCS 1882, pages 44–56, Zurich, Switzerland, October 2000. Springer-Verlag.

[KLS90] H.F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In D. McLeod, R. Sacks-Davis, and H.-J. Schek, editors, *Proc. of 16th Int. Conference on Very Large Data Bases*, pages 95–106, Brisbane, Queensland, Australia, August 1990. Morgan Kaufmann.

[Kna87] E. Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys*, 19(4):303–328, 1987.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LD98] C. Loosley and F. Douglas. *High-Performance Client/Server*. Wiley Computer Publishing, New York, USA, 1998.

[LM93] P.-Y. Li and B. McMillin. Fault-tolerant distributed deadlock detection/resolution. In *Proc. of 17th Int. Computer Software and Applications Conference (COMPSAC)*, pages 224–230, November 1993.

[LO98] D.B. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley, Reading, Massachussetts, 1998.

[LO99] D.B. Lange and M. Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 45(3):88–89, March 1999.

[Mat98] F. Mattern. Mobile agenten. *it+ti - Informationstechnik und Technische Informatik*, (4):12–17, 1998. In German.

[Maz96] K. R. Mazouni. *Étude de l'invocation entre objets dupliqués dans un système réparti tolérant aux fautes*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, January 1996. In French.

[MBB+98] D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran, and J. White. MASIF, The OMG Mobile Agent System Interoperability Facility. In Kurt Rothermel and F. Hohl, editors, *Mobile Agents, Proceedings of the Second International Workshop, MA'98*, LNCS 1477, pages 50–67. Springer Verlag, September 1998.

[MGG95a] K. R. Mazouni, B. Garbinato, and R. Guerraoui. Invocation support for replicated objects. Technical Report 95/120, École Polytechnique Fédérale de Lausanne, Switzerland, May 1995.

[MGG95b] K.R. Mazouni, B. Garbinato, and R. Guerraoui. Filtering duplicated invocations using symmetric proxies. In *Proc. of the 4th IEEE International Workshop on Object Orientation in Operating Systems (IWOOOS'95)*, Lund, Sweden, August 1995.

[MGM99] P. Maes, R.H. Guttman, and A.G. Moukas. Agents that buy and sell. *Communications of the ACM*, 42(3):81–91, March 1999.

[MLC98] D.S. Milojicicc, W. LaForge, and D. Chauhan. Mobile objects and agents (MOA). In *In Proc. of 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Santa Fe, New Mexico, April 1998.

[Mos85] J.E.B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, MA, USA, 1985.

[MP99] A.L. Murphy and G.P. Picco. Reliable communication for highly mobile agents. In *Proc. of 1st Int. Conference on Agent Systems and Applications/Mobile Agents (ASAMA'99)*, pages 141–150, Palm Springs, CA, USA, October 1999. IEEE.

[MPT00] A. Mohindra, A. Purakayastha, and P. Thati. Exploiting non-determinism for reliability of mobile agent systems. In *Proc. of Int. Conference on Dependable Systems and Networks (DSN'00)*, pages 144–153, New York, June 2000.

[MS95] C.P. Malloth and A. Schiper. View synchronous communication in large scale distributed systems. In *Proceedings of the 2nd Open Workshop of the ESPRIT project BROADCAST (nb 6360)*, Grenoble, France, July 1995.

[Mul93] S. Mullender, editor. *Distributed Systems*. Addison-Wesley, Reading, Massachusetts, USA, 2nd edition, 1993.

[MvRSS96] Y. Minsky, R. van Renesse, F.B. Schneider, and S.D. Stoller. Cryptographic support for fault-tolerant distributed computing. In *In Proc. of 7th European Workshop on ACM SIGOPS*, pages 109–114, Connemara, Ireland, 1996.

[Nar99] P. Narasimhan. *Transparent Fault Tolerance for CORBA*. PhD thesis, University of California, Santa Barbara, USA, September 1999.

[Obj99] ObjectSpace. *Voyager: ORB 3.1 Developer Guide*, 1999. http://www.objectspace.com/products.

[Obj00] Object Management Group (OMG). *Mobile Agent System Interoperability Facilities Specification, OMG Document formal/00-02-01*, February 2000. http://www.omg.org.

[OGS97] R. Oliveira, R. Guerraoui, and A. Schiper. Consensus in the crash-recover model. Technical Report 97/239, École Polytechnique Fédérale de Lausanne, Switzerland, August 1997.

[OH98] R. Orfali and D. Harkey. *Client/Server Programming with JAVA and CORBA*. John Wiley, New York, USA, 1998.

[OMGa] OMG. Common Object Request Broker Architecture (CORBA). http://wwww.corba.org.

[OMGb] Object Management Group (OMG). http://www.omg.org.

[OMG98a] OMG. *The Common Object Request Broker: Architecture and Specification, Version 2.2*, February 1998. http://www.omg.org/corba/corbaiiop.html.

[OMG98b] OMG. *CORBAservices: Common Object Services Specification*, December 1998. http://www.omg.org/library/csindx.html.

[Ple99] S. Pleisch. State of the art in mobile computing – security, fault tolerance, and transaction support. Technical Report rz3152, IBM Research, 1999.

[Pol94] S. Poledna. Replica determinism in distributed real-time systems: A brief survey. *Real-Time Systems*, 6(3):289–316, May 1994.

[Pow91] D. Powell. Delta4: A generic architecture for dependable distributed computing. In *ESPRIT Research Reports*, volume 1. Springer Verlag, 1991.

[PPG00] H. Pals, S. Petri, and C. Grewe. FANTOMAS - fault tolerance for mobile agents in clusters. In J.D.P. Rolim, editor, *Proc. of IPDPS 2000 Workshop*, LNCS 1800, pages 1236–1247. Springer Verlag, 2000.

[PS00] S. Pleisch and A. Schiper. Modeling fault-tolerant mobile agent execution as a sequence of agreement problems. In *Proc. of 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, pages 11–20, Nuremberg, Germany, October 2000.

[PS01a] S. Pleisch and A. Schiper. Approaches for fault-tolerant mobile agents. Technical Report rz 3333, IBM Research, January 2001.

[PS01b] S. Pleisch and A. Schiper. 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)*, pages 215–224, Goteborg, Sweden, July 2001.

[PS01c] S. Pleisch and A. Schiper. TranSuMA: Non-blocking transaction support for mobile agent execution. Technical Report rz 3386, IBM Research, 2001.

[PS02] S. Pleisch and A. Schiper. Non-blocking transactional mobile agent execution. In *Proc. of 22nd IEEE Int. Conference on Distributed Computing Systems (ICDCS'02)*, pages 443–444, Vienna, Austria, July 2002.

[Rak94] A. Rakotonirainy. Exploiting transaction and object semantics to increase concurrency. In C. Girault, editor, *Proceedings of IFIP*, pages 155–164. Elsevier Science B.V., 1994.

[RS98] K. Rothermel and M. Strasser. A fault-tolerant protocol for providing the exactly-once property of mobile agents. In *Proc. of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98)*, pages 100–108, West Lafayette, Indiana, October 1998.

[RSS97] A. Reuter, K. Schneider, and F. Schwenkreis. Contracts revisited. In S. Jajodia and L. Kerschberg, editors, *Advanced Transaction Models and Architectures (ATMA)*, pages 127–151. Kluwer Academic Publ., Boston, MA, 1997.

[Ruk91] M. Rukoz. Hierarchical deadlock detection for nested transactions. *Distributed Computing*, (4):123–128, 1991.

[SAE01] R. Sher, Y. Aridor, and O. Etzion. Mobile transactional agents. In *Proc. of 21st IEEE Int. Conference on Distributed Computing Systems (ICDCS'01)*, pages 73–80, Phoenix, Arizona, April 2001.

[SBH97] M. Strasser, J. Baumann, and F. Hohl. Mole - a java based mobile agent system. In M. Mühlhäuser, editor, *Special Issues in Object Oriented Programming*, pages 301–308. dpunkt Verlag, 1997.

[SBS00] L.M. Silva, V. Batista, and J.G. Silva. Fault-tolerant execution of mobile agents. In *Proc. of Int. Conference on Dependable Systems and Networks (DSN'00)*, pages 135–143, New York, June 2000.

[Sch93] F.B. Schneider. Replication management using the state-machine approach. In Mullender [Mul93], pages 169–198.

[Sch97] F.B. Schneider. Towards fault-tolerant and secure agentry. In *Proc. of the 11th Int. Workshop on Distributed Algorithms, Saarbrücken, Germany*, September 1997. Invited paper.

[SG84] A. Spector and D. Gifford. The space shuttle primary computer system. *Communications of the ACM*, pages 874–900, September 1984.

[SGS84] F.B. Schneider, D. Gries, and R.D. Schlichting. Fault-tolerant broadcasts. *Science of Computer Programming*, 4(1):1–15, April 1984.

[SM94] L.S. Sabel and K. Marzullo. Simulating fail-stop in asynchronous distributed systems. In *Proc. of the 13th IEEE Symposium on Reliable Distributed Systems (SRDS'94)*, pages 138–147, Dana Point, CA, October 1994.

[SM96] J.B. Sussman and K. Marzullo. Comparing primary-backup and state machines for crash failures. In *Proc. of the 15th ACM Symposium on Principles of Distributed Computing (PODC'96)*, page 90, Philadelphia, Pennsylvania, May 1996. Brief announcement.

[SR98] M. Strasser and K. Rothermel. Reliability concepts for mobile agents. *International Journal of Cooperative Information Systems*, 7(4):355–382, 1998.

[SR00] M. Strasser and K. Rothermel. System mechanisms for partial rollback of mobile agent execution. In *Proc. of 20th IEEE Int. Conference on Distributed Computing Systems (ICDCS'00)*, pages 20–28, Taipei, Taiwan, April 2000.

[Sun] Sun Microsystems, Palo Alto, California, USA. *Java Remote Method Invocation – Specification*. http://java.sun.com/j2se/1.4/docs/guide/rmi/index.html.

[Tac] University of Tromsø, Norway; Cornell University, Ithaca, New York, USA; University of California, San Diego, USA. *Tromsø and Cornell Mobile Agents (Tacoma)*. http://www.tacoma.cs.uit.no/.

[TFWR97] C. Tham, B. Friedman, J. White, and T. Rutkowski. An assessment of the mobile agent facility proposal. http://www.agent.org/pub/satp/papers/maf-assessment.html, January 1997.

[TR00] W. Theilmann and K. Rothermel. Optimizing the dissemination of mobile agents for distributed information filtering. *IEEE Concurrency*, pages 53–61, April 2000.

[TST01] K. Takashio, G. Soeda, and H. Tokuda. 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)*, pages 202–207, April 2001.

[TW96] D. Tennenhouse and D. Wetheral. Towards an active network architecture. In *ACM Computer Communications Review*, volume 26, pages 5–18, April 1996.

[VKM97a] H. Vogler, T. Kunkelmann, and M.-L. Moschgath. 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, December 1997. IEEE Computer Society.

[VKM97b] H. Vogler, T. Kunkelmann, and M.-L. Moschgath. 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, October 1997. IEEE Computer Society.

[WPM99] D. Wong, N. Paciorek, and D. Moore. Java-based mobile agents. *Communications of the ACM*, 42(3):93–102, March 1999.

[WS92] G. Weikum and H.-J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 515–553. Morgan Kaufmann, 1992.

Appendix A

DIV Consensus

A.1 Deadlock-Free DIV Consensus

Algorithm 1 presents DIV consensus using the approach discussed in Section 6.4.2 in order to prevent deadlocks. The modifications compared to the original DIV Consensus [DS00, Déf00] are highlighted using "$\rightarrow$". Note that we have always assumed that a_i^0 is the first coordinator in DIV consensus. This has simplified the discussion. In Algorithm 1, a_i^1 is acting as the first coordinator, as in this case the replica ID (e.g., 1 for a_i^1) of the coordinator corresponds to the round number in DIV consensus.

A.2 Blocking-Free DIV Consensus

In this section, we present a version of DIV consensus (see Algorithm 2) that sends undo information to the other replicas in order to prevent the blocking situation discussed in Section 7.3.4. In Algorithm 2 we have only highlighted using "$\rightarrow$" the important modifications compared to DIV consensus given in [DS00, Déf00].

Algorithm 1 DIV Consensus (code of process p) [Déf00]: the danger of deadlocks when executing the function $\mathcal{GIV}()$ is avoided by the use of an additional thread (see Lines 16 and 27).

```
 1: procedure Lazy Consensus ( function GIV : ∅ ↦ v)
 2:   estV_p ← ⊥                                              {p's estimate of the decision value}
 3:   state_p ← undecided
 4:   r_p ← 0                                                 {r_p is p's current round number}
 5:   ts_p ← 0                                                {ts_p is the last round in which p updated estV_p, initially 0}
→ 6:   evalStarted_p ← false                                  {true if evaluation of the function GIV() is running}

 7:   while state_p = undecided do                            {rotate through coordinators until decision reached}
 8:     c_p ← (r_p mod n) + 1                                 {c_p is the current coordinator}
 9:     r_p ← r_p + 1
10:     Phase 1:                                             {all processes p send estV_p to the current coordinator}
11:       if r_p > 1 then
12:         send (p, r_p, estV_p, ts_p) to c_p
13:     Phase 2:                                             {coordinator gathers ⌈(n+1)/2⌉ estimates and proposes new estimate}
14:       if p = c_p then
15:         if r_p = 1 then
→ 16:           fork: estV_p ← eval giv()                    {p proposes a value}
→ 17:           evalStarted_p ← true
→ 18:           wait until [estV_p ≠ ⊥ or timeout]
→ 19:           if estV_p ≠ ⊥ then                           {execution of GIV() has succeeded}
→ 20:             evalStarted_p ← false
21:         else
22:           wait until [for ⌈(n+1)/2⌉ processes q : received (q, r_p, estV_q, ts_q) from q]
23:           msgs_p[r_p] ← {(q, r_p, estV_q, ts_q) | p received (q, r_p, estV_q, ts_q) from q}
24:           t ← largest ts_q such that (q, r_p, estV_q, ts_q) ∈ msgs_p[r_p]
25:           if estV_p = ⊥ and ∀(q, r_p, estV_q, ts_q) ∈ msgs_p[r_p] : estV_q = ⊥ then
→ 26:             if ¬evalStarted_p then
→ 27:               fork: estV_p ← eval giv()                {p proposes a value}
→ 28:               evalStarted_p ← true
→ 29:               wait until [estV_p ≠ ⊥ or timeout]
→ 30:               if estV_p ≠ ⊥ then                       {execution of GIV() has succeeded}
→ 31:                 evalStarted_p ← false
32:           else
33:             estV_p ← select one estV_q ≠ ⊥ s.t. (q, r_p, estV_q, t) ∈ msgs_p[r_p]
→ 34:         if estV_p ≠ ⊥ then
→ 35:           send (p, r_p, estV_p) to all                 {Only send suggestion if not ⊥}
36:     Phase 3:                                             {all processes wait for new estimate proposed by current coordinator}
37:       wait until [received (c_p, r_p, estV_cp) from c_p or c_p ∈ D_p]   {query failure detector D_p}
38:       if [received (c_p, r_p, estV_cp) from c_p] then    {p received estV_cp from c_p}
39:         estV_p ← estV_cp
40:         ts_p ← r_p
41:         send (p, r_p, ack) to c_p
42:       else                                               {p suspects that c_p crashed}
43:         send (p, r_p, nack) to c_p
44:     Phase 4:  {the current coordinator waits for ⌈(n+1)/2⌉ replies. If they indicate that ⌈(n+1)/2⌉ processes adopted its
          estimate, the coordinator R-broadcasts a decide message}
45:       if p = c_p then
46:         wait until [for ⌈(n+1)/2⌉ processes q : received (q, r_p, ack) or (q, r_p, nack)]
47:         if [for ⌈(n+1)/2⌉ processes q : received (q, r_p, ack)] then
48:           R-broadcast (p, r_p, estV_p, decide)

49: when R-deliver (q, r_q, estV_q, decide)                  {if p R-delivers a decide message, p decides accordingly}
50:   if state_p = undecided then
51:     decide(estV_q)
52:     state_p ← decided
```

Algorithm 2 DIV Consensus (code of process p) [Déf00]: invocations to servers by failed primaries are undone when needed.

```
 1: procedure Lazy Consensus ( function GIV : ∅ ↦ v)
 2:   estV_p ← ⊥                                              {p's estimate of the decision value}
 3:   state_p ← undecided
 4:   r_p ← 0                                                 {r_p is p's current round number}
 5:   ts_p ← 0                                                {ts_p is the last round in which p updated estV_p, initially 0}

 6:   while state_p = undecided do                            {rotate through coordinators until decision reached}
 7:     c_p ← (r_p mod n) + 1                                 {c_p is the current coordinator}
 8:     r_p ← r_p + 1
 9:     Phase 1:                                              {all processes p send estV_p to the current coordinator}
10:       if r_p > 1 then
11:         send (p, r_p, estV_p, ts_p) to c_p
12:     Phase 2:                                              {coordinator gathers ⌈(n+1)/2⌉ estimates and proposes new estimate}
13:       if p = c_p then
14:         if r_p = 1 then
15:           estV_p ← eval giv()   {p proposes a value. No undo information needed as first time to evaluate GIV()}
16:         else
17:           wait until [for ⌈(n+1)/2⌉ processes q : received (q, r_p, estV_q, ts_q) from q]
18:           msgs_p[r_p] ← {(q, r_p, estV_q, ts_q) | p received (q, r_p, estV_q, ts_q) from q}
19:           t ← largest ts_q such that (q, r_p, estV_q, ts_q) ∈ msgs_p[r_p]
20:           if estV_p = ⊥ and ∀(q, r_p, estV_q, ts_q) ∈ msgs_p[r_p] : estV_q = ⊥ then
→ 21:             if [received undo information message from processes q] then
→ 22:               send termination request to all T invoked by processes q
→ 23:             send undo information to all processes      {p sends its undo information to all processes}
24:             estV_p ← eval giv()                           {p proposes a value}
25:           else
26:             estV_p ← select one estV_q ≠ ⊥ s.t. (q, r_p, estV_q, t) ∈ msgs_p[r_p]
27:         send (p, r_p, estV_p) to all
28:     Phase 3:                                              {all processes wait for new estimate proposed by current coordinator}
29:       wait until [received (c_p, r_p, estV_{c_p}) from c_p or c_p ∈ D_p]   {query failure detector D_p}
30:       if [received (c_p, r_p, estV_{c_p}) from c_p] then                   {p received estV_{c_p} from c_p}
31:         estV_p ← estV_{c_p}
32:         ts_p ← r_p
33:         send (p, r_p, ack) to c_p
34:       else                                                                 {p suspects that c_p crashed}
35:         send (p, r_p, nack) to c_p
36:     Phase 4:  {the current coordinator waits for ⌈(n+1)/2⌉ replies. If they indicate that ⌈(n+1)/2⌉ processes adopted its
        estimate, the coordinator R-broadcasts a decide message}
37:       if p = c_p then
38:         wait until [for ⌈(n+1)/2⌉ processes q : received (q, r_p, ack) or (q, r_p, nack)]
39:         if [for ⌈(n+1)/2⌉ processes q : received (q, r_p, ack)] then
40:           R-broadcast (p, r_p, estV_p, decide)

41: when R-deliver (q, r_q, estV_q, decide)                   {if p R-delivers a decide message, p decides accordingly}
42:   if state_p = undecided then
43:     decide(estV_q)
44:     state_p ← decided
```

Curriculum Vitae

Stefan Pleisch was born in Langenthal (BE, Switzerland) in 1972. After attending primary and secondary school, he graduated from Gymnasium Langenthal with the "Maturität" in 1991. From 1992 to 1997 he studied computer science at the Swiss Federal Institute of Technology (EPFL) in Lausanne, Switzerland. During this time, he spent two terms at the University of Waterloo, Ontario, Canada, and worked on his diploma thesis at the Swiss Federal Institute of Technology (ETHZ) in Zurich, Switzerland. After graduating from university in April 1997 he was employed by ELCA Informatik AG, a Swiss IT services supplier, as a programmer/systems architect. Since October 1998, he has been working at the IBM Zurich Research Laboratory in Rüschlikon (Switzerland) and has been pursuing a PhD. His advisor is Prof. André Schiper from the Distributed Systems Laboratory at EPFL.