IV Workshop de Testes e Tolerância a Falhas (WTF2003) - XXI Simpósio Brasileiro de Redes de Computadores (SBRC 2003)

The MARES Platform: Support for Transactional and Fault-tolerant Execution of Mobile Agent-based Applications

Flávio M. Assis Silva, Raimundo J.A. Macêdo, Ana Vitoria Piaggio Freitas

LaSiD – Distributed Systems Laboratory
Federal University of Bahia (UFBA)
Av. Adhemar de Barros, S/N
Bairro de Ondina, Salvador/BA – CEP 40170-110

{fassis, macedo, piaggio}@ufba.br

Abstract

Support for transactional behaviour and fault tolerant executions of mobile agent-based applications is a fundamental issue in the development of mobile agent systems. At LaSiD/UFBA we are developing the MARES platform, which supports the modelling of mobile agent-based applications as distributed transactions. The executions of mobile agent-based transactions on top of the MARES platform are fault-tolerant, in the sense that if the location in the distributed environment where a part of a global transaction is being executed becomes faulty for a long time, the system performs a recovery procedure to resume the execution of that part of the transaction at another location. In this paper we present the trasnaction model used in the MARES platform, we outline the interface for modelling mobile agent-based transactions, and we discuss alternatives for implementing mobile agent fault tolerance that increase the level of flexibility for modelling reliable mobile agent-based applications when compared with previously proposed approaches.

1. Introduction

A *mobile agent* (or simply *agent*) is a self-contained software element responsible for executing a programmatic process, which is capable of *autonomously migrating* through a network. An agent migrates in a distributed environment between logical ”places“ referred here to as *agencies*. When an agent migrates, its execution is suspended at the original agency, the agent is transported (i.e. program code, data, execution state and control information) to the destination agency, and, after being re-instantiated, it resumes execution. The mobile agent concept is being proposed to support different types of applications, including electronic commerce, workflow management systems and network management due to the potential benefits that might be achieved by exploring its asynchronous way of execution [1, 3].

For at least some types of mobile agent-based applications (such as electronic commerce or workflow applications) it is fundamental that the executions of these applications exhibit strong reliability properties. Among the reliability requirements are the need for the executions of the applications to be fault tolerant and to exhibit some transactional semantics, and that groups of mobile agents can coordinate their activites by using a mechanism of reliable group communication [5].

At LaSiD/UFBA (Distributed Systems Laboratory / Federal University of Bahia) we are developing and implementing a platform called MARES, which provides solutions for these

reliability requirements. This platform provides for its applications an interface through which *mobile agent-based transactions* can be modelled. A mobile agent-based transaction is a distributed transaction that is carried out by a set of mobile agents. The control of the execution of such a transaction becomes thus decentralized and the components that execute parts of the execution control (the agents) might change their location in the distributed environment. In the MARES platform the executions of mobile agent-based transactions are fault tolerant. Not only the execution of a part of a transaction being executed at an agency is recovered locally when that agency restarts normal execution, but also the execution of that part of the transaction is resumed at *another* agency, if the failure lasts long.

In this paper we present the transaction model provided by the MARES platform and we discuss alternatives for implementing mobile agent fault tolerance. Approaches for mobile agent fault tolerance and the modelling of transactions on top of such approaches have been proposed in the literature [e.g. 1, 4, 7]. The transaction model implemented in the MARES platform is based on the model presented in [1]. Beyond a brief description of the interface for modelling mobile agent-based transactions in MARES, the alternatives for implementing mobile agent fault tolerance presented in this paper provide more flexibility for modelling relible mobile agent-based applications than the existing approaches.

This paper is structured as follows. In section 2 we present the architecture of the MARES platform. In section 3 we present the transaction model used in the platform and the interface for specifying these transactions. In section 4 we discuss alternatives for implementing mobile agent fault tolerance. Finally, section 5 concludes the paper.

2. The MARES Platform

The MARES Platform is structured in three layers, as shown on Figure 1.

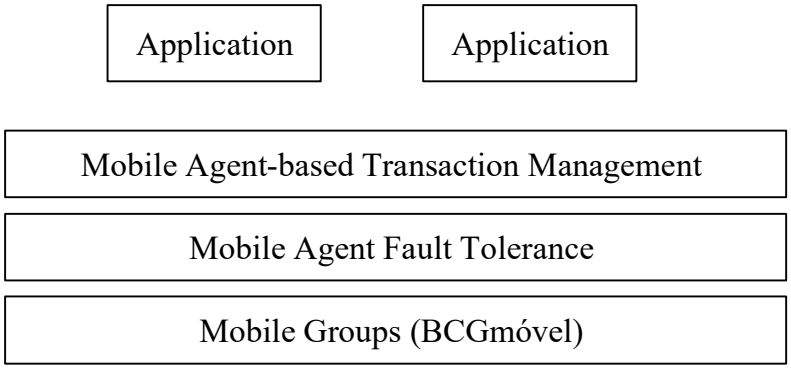

Figure 1: The Layers of the MARES Platform

The platform is based on a group communication system that implements the concept of *mobile groups* (lowest layer). The concept of mobile group was introduced by Macêdo and Assis Silva [6] and is an extension of the traditional concept of a group of cooperating processes in which a member process can move from a location in a distributed system to another while continuing to be a member of the group. Analogously to traditional group systems, mobile groups also provide message delivery guarantees and virtual synchrony. Traditionally, mobile group systems (such as Horus, Transis, NewTop, among others) do not inherently support process migration. Making process migration an integrated functionality of a process group system makes the implementation of migration of a group member more

efficient and allows the synchronization of movement with other group events (such as joins and crashes of processes). The group communication system being developed at LaSiD that implements the mobile groups concept is called BCGmóvel (*Reliable Group Communication System with Mobility Support*). As discussed in [5], a mobile group can be considered to be an underlying functionality in a mobile agent system to fulfill some of the reliability requirements of mobile agent-based applications.

On top of the layer providing mobile group support is a layer implementing a protocol for mobile agent fault tolerance. If the agency where an agent is running fails, the execution of that agent remains blocked while the agency is faulty. Long unavailability periods of that agency have the obvious undesirable effect of delaying the execution of an agent-based application. It is desirable to have a mechanism that makes it possible to recover the execution of a computation being performed by an agent at an agency that has remained faulty for a long time from some other agency in the distributed environment (for possibly following an alternative execution path, restarting from a previous consistent execution state).

The protocolos proposed so far for mobile agent fault tolerance are based on mobile agent replication [e.g., 1, 4, 7]. When a migration is performed, the code and state of a mobile agent-based application is sent to a set of locations, instead of just to one. This set of agents executes as a group. If one of them fails, the other agents might recover the execution of the group and continue its execution from a previous consistent state. The coordination of this set of agents is the goal of the *Mobile Agent Fault Tolerance* layer of the MARES Platform.

On top of the Mobile Agent Fault Tolerance layer is the *Mobile Agent-based Transaction Management* layer. This layer provides the needed functionality for enabling mobile agent-based applications to run as global distributed transactions. The functionality implemented at this layer guarantees that the whole system will remain in a consistent state independently of concurrent executions of global transactions. One or more mobile agents can be used to execute each global transaction.

As examples of applications to run on top of the MARES platform are electronic commerce and workflow applications. At LaSiD we are developing a workflow management system based on mobile agents, called MAB*flow* (*Mobile Agent-based Workflow Management System*). In this system, workflows are specified in UML (*Unified Modelling Language*). Tools are provided that generate an XML (*Extended Markup Language*) document that describes the structure of the workflow specified in UML. From this XML document, Java code is automatically generated that models the workflow as a global transaction using the interface provided by the MARES platform.

The platform will include also tools for supporting the analysis of correctness of mobile agent-based applications. The goal of these tools will not be discussed here.

3. The Mobile Agent-based Transaction Management Layer

The application program interface (API) of the MARES platform is the interface provided by the *Mobile Agent-based Transactional Management Layer*. This API consists of a set of Java classes and interfaces. By using this API, applications can be modelled as mobile agent-based transactions.

3.1. The Transaction Model

A transaction is a concept used to guarantee correct and reliable executions of programs accessing resources concurrently in an environment subject to faults. The most simple transaction model is the *atomic (flat) transactions* model. Atomic transactions satisfy a set of properties commonly referred to as ACID (*Atomicity, Consistency, Isolation and Durability*) properties [2]. In this model, intermediate results of a transaction are not visible to other

transactions (the *isolation property*) and all of the effects produced by the transacion are made durable in the system or none of them will be (*atomicity* and *durable* properties). Although atomic transactions represent a suitable abstraction for modelling concurrent accesses to shared data, it is well known that they do not fulfill requirements on transaction processing of complex applications, for example, due to the need for modelling long duration activities [8].

A set of so-called *extended transaction models* were then proposed to fulfill the requirements of such applications [8]. In the models that do not provide the isolation property, the recovery of transactions is usually performed by using *compensation*. A transaction the effects of which must be cancelled (the *compensated-for* transaction) has an associated *compensating transaction*. The compensating transaction is able to cancel the effects produced by the compensated-for transaction *from a semantic point-of-view*. I.e., the resulting state produced in the system by the compensating transaction is not necessarily equal to the state read by the compensated-for transaction before its execution. The resulting state is only *related* to the original one according to the semantics of the applications.

The transaction model implemented in the MARES platform is an *open nested transaction model*. An open nested transaction can be composed of other transactions, called its *subtransactions*. Each subtransaction might have its own subtransactions, resulting this way in a transaction tree. The root of the transaction tree is an open nested transaction. The subtransactions of an open nested transaction can be (*flat) atomic* or also *open nested*. Differently from atomic transactions, the intermediate states of an open transaction are made visible to other transactions and its recovery is based on compensation.

Atomic and open transactions are used to model different types of activities in the MARES platform. A flat transaction executes on a single agency and its code consists typically of a set of invocations of methods of objects providing services in the distributed environment (for example, an application database). These methods are to be invoked *locally*, i.e., with the agent performing the transaction and the object providing the service in an environment where the invocations can be made with a good quality of service (for example, with both the agent and the service provider object at the same host or at different hosts but at the same local area network). The control flow of an open transaction, on the other hand, defines how its subtransactions are to be executed. They do not access directly services at the interface of objects. Subtransactions of an open transaction can be executed sequentially or in parallel. Sets of alternative transactions can also be defined. These sets are composed by transactions which provide equivalent results to the application, but at most one of the transactions in the set can commit.

Compensating transactions are defined as follows. Each transaction, with exception of the root transaction, has associated with it a compensating transaction. The compensating transaction of an atomic transaction is another atomic transaction, written by the application programmer. The compensating transaction of an open transaction is another open transaction, but defined automatically during runtime. The compensating transaction of an open transaction cancels the effects of the subtransactions that were committed, by executing the compensating transactions of its subtransactions, in the reverse order of their executions.

A transaction can also be *vital* or *non-vital*, depending on the impact that a failure of this transaction has on its parent transaction. A transaction is vital if a failure during its execution implies that the effects of its parent transaction must also be cancelled. If the execution of a non-vital transaction fails, the execution of its parent transaction can continue.

Transactions of an alternative set have priorities. The transaction with the highest priority is the one which is executed first. If this transaction fails, the transaction with the next priority is executed. The process is repeated until either one of the transactions in the set is executed successfully or there is no more transactions to execute in the set.

3.2. The Transaction Support Interface

Two general classes were defined, that represent the basic types of transactions:

- *FlatTransaction*: used to model leaf (atomic) transactions in a transaction tree;
- *OpenTransaction*: corresponds to non-leaf nodes in a transaction tree (open transactions).

The class *OpenTransaction* has three specializations: *SequentialOpenTransaction, ParallelOpenTransaction* and *AlternativeOpenTransaction*. Each of these specializations has associated with it a set of transactions, each of which might be, by its turn, either a *FlatTransaction* or one of the specializations of *OpenTransaction*.

The different specializations of *OpenTransaction* define specific ways of executing the transactions in the associated set as follows.

In the case of *SequentialOpenTransaction*, each transaction in the set is associated with a set of conditions that must be satisfied in order for that transaction to be executed. The conditions are boolean expressions that can involve values of data items processed in the transaction or the outcome of previously executed transactions. When a *SequentialOpenTransaction* is to be executed, the sets of conditions associated with the transactions in the set are repeatedly evaluated. If all the conditions in the set associated with a transaction are evaluated to true, that transaction is executed. If that happens to more than one transaction in the set, one of them is randomly chosen.

In the case of *ParallelOpenTransaction*, all transactions in the associated set are executed in parallel.

In the case of *AlternativeOpenTransaction*, each transaction in the associated set has a different priority. The execution of an *AlternativeOpenTransaction* begins with the execution of the transaction with the highest priority. If that transaction fails, the transaction with the next highest priority is executed. This process is repeated until some of the transactions is executed successfully or until there is no more transaction in the set to be executed.

A mobile agent-based transaction is thus formed by choosing at each level the type of transaction that apropriately models the type of control flow desired at that level. If the chosen transaction is an open transaction, then transactions of appropriate classes are inserted in the set. The whole transaction tree is thus built by recursively repeating this procedure.

```
1 Class ExampleTransaction extends SequentialOpenTransaction {
2       ...
3  protected boolean controlFlow() {
4       ExampleFlatTransaction t1 = new ExampleFlatTransaction( “t1”, true, “ag1”);
5       ExampleFlatTransaction t2 = new ExampleFlatTransaction(“t2”, true,”ag2”);
6       this.addSubTransaction(t1);
7       this.addSubTransaction(t2);
8       this.addCondition(“t2”, newCommitCondition(t1));
9       ...
10 }
```

Figure 2: Example of part of a mobile agent-based transaction

An example of part of the specification of a mobile agent-based transaction is shown on Figure 2. In this example, the code a sequential open transaction is defined. In lines 4 and 5 two atomic transactions were defined, *t1* and *t2*. In lines 6 and 7 these transactions are defined

as subtransactions of the *SequentialOpenTransaction* being written. In line 8 the condition for the execution of transaction *t2* is defined. Transaction *t2* will only execute if transaction *t1* commits. Since there is no condition associated with *t1* it is eligible to be executed when its parent transaction begins executing.

At any time, if the execution of a vital transaction fails, the compensation process starts. This process executes the compensating transaction for each of the committed transactions, in the reverse order of their execution.

4. The Mobile Agent Fault Tolerance Layer

4.1. Executing Mobile Agent based Transactions

Agent-based transactions are executed by a set of one or more mobile agents. As previously highlighted, mobile agent fault tolerance is achieved by replicating agents. With replication, an agent execution is seen as being performed in a sequence of *stages*. The first stage begins when the application execution starts. A new stage then begins when the mobile agent execution reaches a movement operation, to continue at a new agency. Whenever a new stage of execution is achieved, new replicas of the agent with the current execution state are created at a new set of agencies. The agent copies at the terminating stage are destroyed. Stages are created and terminated in this way until the end of the execution of the application.

In order to illustrate how the management of the agent replicas is done, consider the case of the execution of an *AlternativeOpenTransaction* which, for example, has three atomic transactions in its corresponding transaction set, each of them to be executed at a different agency. This set of transactions will be executed by a set of mobile agents, all with the same code and initially with the same execution state information (for instance, the current state of the transaction which includes the *AlternativeOpenTransaction* as one of its subtransactions). One agent is sent to each of the agencies where transactions in the set are to be executed. The agent at the agency where the transaction with the highest priority is to be executed starts executing. If that agent fails, the other agents in the group *elect* the agent in the group with the transaction with the next highest priority to resume the execution of the *AlternativeOpenTransaction*. If the agency where that agent is fails, the process is repeated. If one of the transactions in the set executes successfully or all of the transactions in the set has been unsuccessfully executed, the execution of the *AlternativeOpenTransaction* will have been terminated. A new configuration of agents at agencies will thus be created to continue the execution of that *AlternativeOpenTransactions*'s parent transaction.

4.2. Alternatives for Managing the Execution of Agent-based Applications by Sets of Agents

There are two issues involved in the management of replicated agents: preserving the computation and data integrity while executing a stage; and managing the creation and destruction of stages.

Preserving computation and data integrity relates to managing how the members of a stage will execute their actions in such a way that the consistency of the whole system is guaranteed. For example, if an agent executes some action at an agency, this agency fails and some other member of the group resumes the execution of the transaction, specific actions must be carried out when that agency recovers from the failure in order to cancel the effects performed there before the failure.

Managing the creation and destruction of stages relates to coordinating the creation of the configuration of agent copies that will participate in a stage and destroying the configuration that was used to execute the previous stage. An important issue involved in that

is the guarantee of *exactly-once semantics*. Informally, exactly-once semantics means in this context that each step of an application executed by mobile agents must be executed exactly once. Exactly-once semantics is achieved by guaranteeing that during a stage the results produced by only one of the replicas can be committed (and used to form the replicas forming a new stage). For example, if more than an agent tries to create a new stage, only one of them can succeed.

The approaches proposed so far for solving these two aspects of the execution of agent-based applications by sets of agents [e.g., 1, 4, 7] consider alternatives for executing the former aspect, but all of them consider exactly-once semantics for the management of stages. For example, as described above, copies of an agent could be the members of a stage to execute the set of transactions of an *AlternativeOpenTransaction*. We could alternatively have a similar set of transactions all with the same priority. In this case, a set of agents could be used to execute these transactions, but now all the transactions *can be executed in parallel*. However, only one of them must commit. In this case the exactly-once must still be enforced.

Exactly-once property is not, however, necessary in all cases. With exactly-once semantics, at any point in the execution, the results of the execution of only one of the agents in the group can commit. We can decouple the management of stages from the need for having fault tolerance of the mobile agent-based application. For example, consider that at a certain point in the execution of an application, three independent execution threads exist (for example, three sequences of atomic transactions that can be executed in parallel). In this case, we could have three agents, each executing one of the threads. The group of three agents is still used to provide fault tolerance for the application, but now they can migrate independently one from the others. The agents might exchange messages during execution in order to store information about the execution of their threads (in case of failure of one of the agents, a recovery procedure can use that information to resume execution from a more recent previous consistent state).

4.3. The Mobile Groups Layer

Mobile groups [6] provide the necessary mechanism to implement the alternatives discussed in the previous section. The virtual synchrony of Mobile Groups allows all replicas of an agent to identify agent failure (e.g., the leader failure) in a mutually consistent manner. Atomicity and FIFO message delivery properties can be used to assert that all replicas perceive the same set of actions executed by other agents (e.g., the leader). The exactly-once semantics is a direct benefit of using Mobile Groups and its total order of views. In general terms, it can be achieved by having just one leader at any instant and allowing only the leader to create a new stage. A leader can be uniquely chosen by applying a function on the identifiers of the view members. When a leader decides to move and the movement succeeds, a new view is generated and a new stage begins. The new view will describe the new stage configuration. When the current leader is suspected to have failed, it will leave the group and its actions will not have effects on the group any longer (if it was trying to create a new stage, for example, this new stage will not be created). A new leader is then elected to continue the stage execution. The current execution state information can be transmitted by the leader to the other group members by multicasting it before moving. Other actions, such as the creation of a new group of agents, can be achieved similarly.

5. Conclusion

With the development of the MARES platform we are analysing a set of alternatives for the development of reliable mobile agent-based applications. Although a set of approaches for addressing reliability requirements of such applications has been proposed in the literature, there is still many aspects to be analysed in more depth yet. In particular, we are more deeply interested in decreasing the cost of solutions for mobile agent-based applications reliability, in identifying fundamental components of a platform that implements such solutions, and in relaxing the restrictions that are imposed by the existing proposals. The platform described in this paper corresponds to some of the results we achieved in these directions so far.

Currenlty the transaction support as described here is already implemented. It runs, however, on top of a simulation of the *Mobile Agent Fault Tolerance* layer. Although a prototype of the mobile groups concept is already implemented, there are still some componentes to be implemented yet to turn it into a completely functional tool.

We are currently porting a workflow management system, the MAB*flow*, in development at LaSiD to run on top of the MARES platform. The current version of *MABflow* is operational at LaSiD, but without fault tolerance and transactional support. A mapping from the XML description of a workflow to a corresponding modelling of the workflow in terms of a mobile agent-based transaction is already implemented and is under test. We are already able to specify a workflow in UML and to simulate its execution with transactional and fault tolerance properties with the MARES platform. We are currently investigating how the support for disconnected operation that is implemented in the current version of MABflow can be best integrated when MARES is used to execute the workflows.

Referências

[1] Flávio M. Assis Silva. *A Transaction Model based on Mobile Agents.* PhD thesis, Universidade Técnica de Berlim, Berlim, Alemanha, 1999.

[2] J.Gray, A.Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers. California, USA. 1993

[3] A. Fuggeta, G. Picco and G. Vigna. Understanding Code Mobility. IEEE Transactions of Software Engineering, 24(5), Maio 1998.

[4] K.Rothermel, M.Strasser. A Fault-Tolerant Protocol for Providing the Exactly-Once Property of Mobile Agents. *Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS´98).* West Lafayette, USA. Outubro, 1998

[5] F.M.Assis Silva, R.J.A.Macêdo. Reliability Requirements in Mobile Agent Systems. *Proceedings of the Second Workshop on Tests and Fault Tolerance (II WTF 2000)*. Curitiba, Brazil. July, 2000.

[6] R.J.Araújo Macêdo, F.M.Assis Silva. Coordination of Mobile Processes with Mobile Groups. *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN2002)*. Washington, DC, USA. June 2002. Pp. 177-186

[7] S.Pleisch, A.Schiper. Modeling Fault Tolerant Mobile Agent Execution as a Sequence of Agreement Problems. *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS)*. Nuremberg, Germany. Oct. 2000

[8] A.K.Elmagarmid (ed.). *Database Transaction Models for Advanced Applications*. Morgan-Kaufmann Publishers. USA. 1992