DimaX: A Fault-Tolerant Multi-Agent Platform

Nora Faci [1]
faci@leri.univ-reims.fr

Zahia Guessoum [1,2]
Zahia.Guessoum@lip6.fr

Olivier Marin [2]
Olivier.Marin@lip6.fr

[1] University Reims Champagne-Ardenne
CReSTIC-MODECO Team
Rue des Crayeres BP 1035
51687 Reims, FRANCE

[2] University Pierre and Marie Curie
LIP6- OASIS and SRC Teams
8 Rue du Capitaine Scott
75015 Paris, FRANCE

ABSTRACT

Fault tolerance is an important property of large-scale multi-agent systems as the failure rate grows with both the number of the hosts and deployed agents, and the duration of computation. Several approaches have been introduced to deal with some aspects of the fault-tolerance problem. However, most existing solutions are ad hoc. Thus, no existing multi-agent architecture or platform provides a fault-tolerance service that can be used to facilitate the design and implementation of reliable multi-agent systems. So, we have developed a fault-tolerant multi-agent platform (named DimaX). DimaX deals with fail-stop failures like bugs and/or breakdown machines. It brings fault-tolerance for multi-agent applications by using replication techniques. It is based on a replication framework (named DARX).

Categories and Subject Descriptors

I.2.11 [**Distributed Artificial Intelligence**]: Multi-Agent Systems; D.2.10 [**Software Engineering**]: Design and Implementation

General Terms

Design, Reliability

Keywords

Large Scale Multi-Agent Systems, Fault-Tolerance, Replication, Multi-Agent Platform

1. INTRODUCTION

Fault tolerance is a relevant problem in multi-agent systems (MAS). Nowadays, the multi-agent systems are naturally employed to build distributed applications. Both the size and the complexity of the problems are high, an important number of agents is required. As the failure rate grows with both the number of hosts and deployed agents, and the duration of computation, these applications are subject to more failures.

To deal with some aspects of the fault-tolerance problem, several approaches [12, 7, 11] were introduced. For detecting and recovering faults in MAS, Hagg [7] introduced the sentinel concept. In his project, agents interact for achieving functionalities. The designer associates to each functionality a sentinel. These sentinels observe the different agents and detect functionality deviations in order to diagnose faults and to repair them. Kumar et al. [12] introduced a different approach. They proposed a brokers team to recover faults unregarding the fault reasons. A broker offers several services like searching appropriate agents for given tasks. As a task can be performed by several agents, an agent failure remains transparent as long as there are safe agents. These techniques provide interesting solutions. However, they are ad hoc and are suitable for small-scale multi-agent applications; they could not be reused to build other multi-agent applications and are not suitable for large-scale MAS. For instance, the Hagg sentinels are specific to each MAS like Kumar brokers that use domain knowledge for delivering their services. Thus, no existing multi-agent architecture or platform provides a fault-tolerance service that can be used to facilitate the design and implementation of reliable multi-agent systems.

So, we have developed a fault-tolerant multi-agent platform (named DimaX). The design of fault-tolerant MAS requires to deal with problems related to distribution and fault tolerance. DimaX offers several services like naming, fault detection and recovery. To make MAS robust, DimaX uses replication techniques. Moreover, DimaX provides developers with libraries of reusable components for building MAS. DimaX presents some interesting features like robustness and reusability.

The remainder of this paper is organized as follows. Sections 2 and 3 present our DimaX platform and an development example. Section 4 gives the DimaX features provided for fault-tolerant MAS development. Section 5 discusses the related work. Finally, Section 6 summarizes our approach and ongoing work.

SELMAS'06 May 22-23, 2006, Shanghai, China

2. DIMAX

The present section aims at defining the type of failures DimaX deals with. Then, it presents the DimaX services for developing fault-tolerant MAS.

2.1 Fault Model

The most generally accepted failure classification can be found in [15]:

1. A crash failure means a component stops producing output; it is the simplest failure to contend with.
2. An omission failure is a transient crash failure: the faulty component will eventually resume its output production.
3. A timing failure occurs when output is produced outside its specified time frame.
4. An arbitrary (or byzantine) failure equates to the production of arbitrary output values at arbitrary times.

Given this classification, two types of failure models are usually considered in distributed environments:

- fail-silent, where the considered system allows only crash failures, and
- fail-uncontrolled, where any type of failure may occur.

In this work we focus on the fail-silent model. An agent failure is defined as its abnormal termination due to failure in an underlying resource. This could be either a bug in the underlying operating system, or a local host crash or a network disconnection.

2.2 DimaX Services

DimaX is the result of an integration of a multi-agent platform (named DIMA[5]) and a fault tolerance framework (named DARX[13, 1]). Figure 1 gives an overview of DimaX and its main components and services. DimaX is founded on three levels: system (i.e., DARX middleware), application (i.e., agents) and monitoring. At the application level, DIMA provides a set of libraries to build multi-agent applications. Moreover, DARX provides the mechanisms necessary for distributing, and replicating agents as services. These mechanisms operate at the middleware level. DimaX also endowed with an observation service operating at both middleware and application levels.

At the monitoring level, the control of replication is automatically performed with cooperation of the observation service [6]. Thus, a DimaX server provides the following services: naming, fault detection, observation and replication.

2.2.1 Naming Service

One of the problem related to multi-agent systems distribution is the agent localization at the time of message sending. A naming server maintains the list (i.e., white pages) of all the agents within its administration domain. When an agent is created, it is registered at both the DimaX server and the naming server.

To send messages to another, an agent needs to know the application-level identifier of the receiver. However, the transmission of these messages through DimaX servers, requires some knowledge about the physical localization (i.e., the IP address and a port number). The local DimaX server requests this information from the naming server and locally stored it in a cache. So, the cache contains the list of agents which have been contacted. This avoids that a DimaX server initiates new searches.

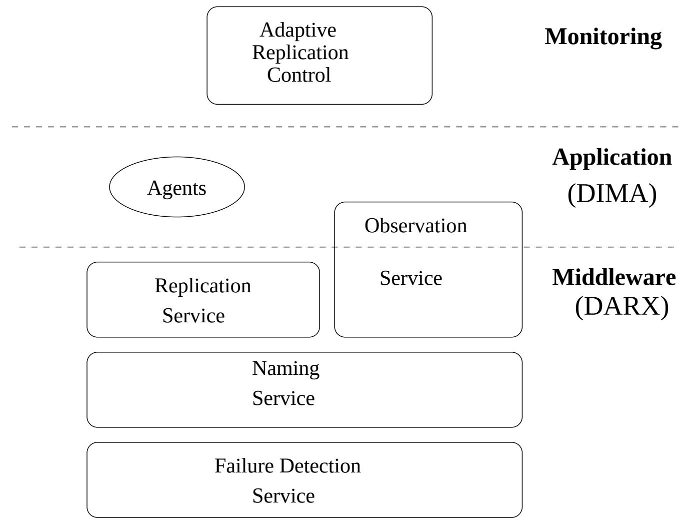

Figure 1: Overview of DimaX

2.2.2 Fault Detection Service

Failure detection is an essential aspect of any fault-tolerant system; indeed it is necessary to recognize a faulty agent. DARX fault detection service is based on the heartbeat technique; a process sends an *I am alive* message to other processes for informing that it is safe (see Figure 2). This technique has two parameters:

- the heartbeat period: the time between two emissions of the i^{th} *I am alive* message,
- the timeout delay: the time between the last reception of an *I am alive* message from p and the time where q suspects p, until an *I am alive* message from p is received.

The detection results may be incorrect; q detects that p is crashed while p is actually safe but its transmissions are delayed for some reason (e.g., communication load). To overcome this problem, one solution is to estimate the arrival date of the following *I am alive* message, with a dynamic margin. These values are functions of the quality of service of the network and the application [1].

When a server detects a failure of another DimaX server, its naming module removes all the replicated agents the faulty server hosted from the list and replaces these agents by their replicas located on other hosts. The replacement is initiated by the failure notification.

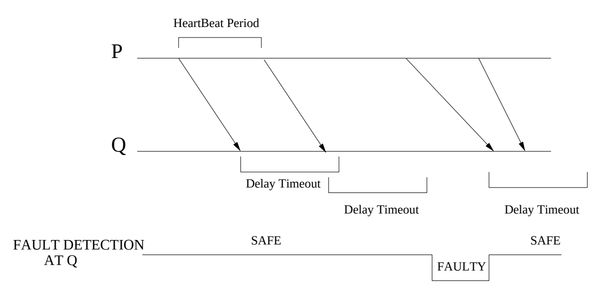

Figure 2: The heartbeat technique

2.2.3 Observation Service

The functionalities of the observation service are fundamental for controlling replication. An observation module collects data at two levels:

- system level: data about the execution environment of the MAS like CPU time and mean time between failures,
- application level: information about its dynamic characteristics like the interaction events among agents (e.g., the sent and received messages).

The observation service relies on an reactive-agents organization (named host monitors) (see Figure 3). These agents collect and process the observation data to compute local information like the number of exchanged messages between two agents during a given period. They also exchange this local information to build global information like the number of messages exchanged in the system. After each interval of time Δt, the host monitors deliver the collected data and events to particular agents (named monitor agents) (see [6] for more details).

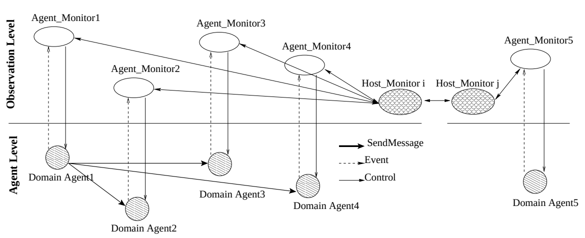

Figure 3: Architecture of the Observation Service

2.2.4 Replication Service

To deal with agent failures, several diagnostic approaches are proposed (see for instance [10, 11]). These techniques correct or adapt the agents behaviors to deal with failures; for instance, they update belief models of faulty agents. They are attractive solutions. However, these approaches are complex; they need a deep knowledge about the behavior of the system. It is not always possible to have a precise description of the whole multi-agent system. Moreover, they are not suitable for critical applications where the diagnosis and correction must be done in real-time. Contrary to these approaches, replication is a simple solution; there is no need for a complex analysis of the system. Also, replication has been used in distributed systems community and has proved its efficiency. We propose therefore to use replication mechanisms to avoid failures of multi-agent systems. Replication enables to run multi-agent systems without interruption, in spite of failures. A replicated agent (see Section 2.3) is an entity that possesses two or more copies of its behavior (or replicas) on different hosts. There are two main types of replication protocols:

- active replication, in which all replicas process concurrently all input messages, and
- passive replication, in which only one of the replicas processes all input messages and periodically transmits its current state to the other replicas in order to maintain consistency.

Active replication strategies provide fast recovery but lead to a high overhead. If the degree of replication is n, the n replicas are activated simultaneously. Passive replication minimizes processor utilization by activating redundant replicas only in case of failures. That is: if the active replica is found to be faulty, a new replica is elected among the set of passive ones and the execution is restarted from the last saved state. This technique requires less CPU resources than the active one but it needs a checkpoint management which remains expensive in processing time and space.

Many toolkits (e.g., see [4, 17]) use only one of these techniques. So, they may suffer from the disadvantages of the used technique. Contrary to these approaches, DimaX relies on the DARX replication framework [13] which uses these both techniques, in an adaptive manner, depending on the evolution of the MAS context. The designer can dynamically change replication strategies during the MAS execution.

2.3 DimaX Agents

DimaX offers several libraries and mechanisms to facilitate the design and implementation of fault-tolerant multi-agent systems. These libraries and mechanisms are provided by DIMA and DARX.

2.3.1 DIMA Agent Behaviors

DIMA is a Java multi-agent platform built as an extension of Object-Oriented Programming (OOP) [5]. Its kernel is a framework of proactive components which represent autonomous and proactive entities. A simple DIMA agent architecture consists of: a proactive component, an agent engine, and a communication component. The proactive component (the AgentBehavior class (see Table 1)) represents the agent behavior. This proactive component includes a decision component to select appropriate actions. For instance, a finite machine state or a rule-based system could be used to describe the decision process. The selected actions may be message sending. So, a communication component is used to send and deliver messages. An Agent Engine is provided to launche and support the agent activity:

```
public class AgentEngine extends ProactiveComponentEngine
implements Runnable{
protected ProactiveComponent proactivity;
public Thread thread;
}
public void run(){
proactivity.startUp();
}
```

DIMA can be used easily to build MASs. To make them reliable, we realized an integration of DIMA agents and DarX tasks. Before desccribing the result of this integration, we define the DarX tasks.

2.3.2 DarX Tasks

DARX [13] is a framework to design reliable distributed applications which include a set of distributed communicating entities (named DarX tasks). It includes transparent replication management. While the application deals with tasks, DARX handles replication groups. Each of these groups consists in software entities (the replicas) which are the representation of the same DarX task (see Figure 4). In DARX, a DarX task can be replicated several times and with different replication strategies. It is wrapped into a

Table 1: Main methods of AgentBehavior Class

Methods	Description
public abstract boolean isAlive()	Tests if the agent has not reached its goal.
public abstract void step()	Represents an execution cycle of the agent.
public void readMailBox()	Removes a message from the mail box, if the latter is not empty, and processes it.
public void sendMessage()	Sends a message to one or several agents. The structure of a message corresponds to the FIPA ACL specification [3].

void proactivityLoop()	Represents the control of agent behavior.
	
	public void proactivityLoop()
	{ while (this.isAlive()) {
	this.preActivity();
	this.step();
	this.postActivity();}}

public void startUp()	Initializes and activates the control of agent behavior.
	
	public void startUp() {
	this.proactivityInitialize();
	this.proactivityLoop();
	this.proactivityTerminate();}

TaskShell which is responsible for replication group management. To maintain coherence between the different replicas, the TaskShell delivers received messages to all active replicas. Also, it periodically updates the state of the passive replicas; this requires to suspend the DarX task then to resume it. When it receives several identical replies from different replicas of the same task, it uses a filter mechanism to forward the first reply and discard the other redondant ones.

The TasKShell sends outgoing messages through its encapsulated DarXCommInterface. The communication between distinct TaskShells is performed via a proxy: the RemoteTask (see Figure 6).

Thus, the sender of a message does not need to know the replicas number of the receiver; the RemoteTask of the receiver delegates the messages to the corresponding TaskShell which transmits them to all replicas of the same agent. The replication has a cost in communication but DARX optimizes it by piggybacking application-level messages on the *I am alive* messages (see Section 2.2.2).

2.3.3 *Fault-Tolerant Agents*

Figure 5 gives the main classes of our agents and the required components to achieve fault tolerance. A fault-tolerant agent (called DimaX agent) is an agent built on our DimaX fault-tolerant multi-agent platform. Each DimaX agent has the structure of a DarXTask. However, the DarXTask is not autonomous. To make it autonomous, we encapsulate the DIMA agent behavior into the DarXTask

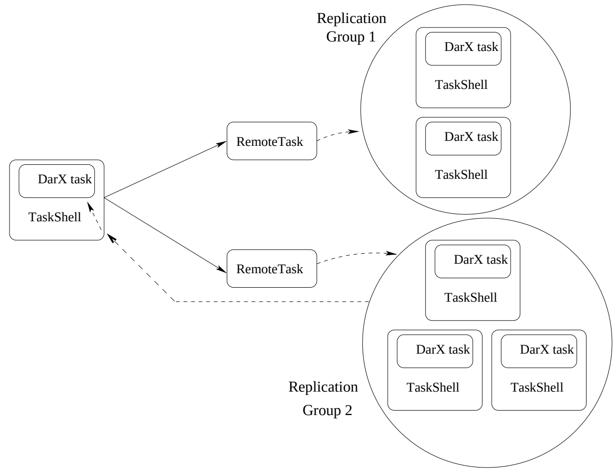

Figure 4: Communication between DarX tasks

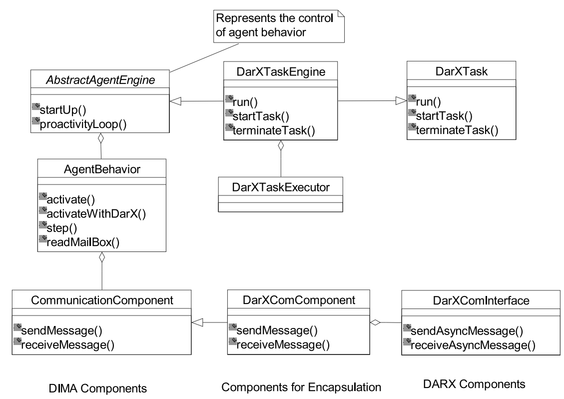

Figure 5: Fault-tolerant Agent Model

(see Figure 6). This agent architecture enables to replicate the agent several times. As the DARX middleware and the DIMA platform both provide mechanisms for execution control, communication and naming but at different levels, their integration requires a set of some additional components; This set calls, transparently, for DARX services (e.g., replication, naming) when executing multi-agent applications developed with DIMA; at the application level, any code modification is required. It controls the execution of agents built under DimaX and offers a communication interface between remote agents, through DimaX servers.

A DimaX agent is a DIMA agent encapsulated in a particular entity, the DarXTaskEngine. The DarXTaskEngine is a DarXTask, with autonomous behaviors of the original DIMA agent. It includes the agent engine (called the DarXTaskExecutor) which executes the lifecycle of the agent (see below the proactivityLoop method). For coherence reasons, the execution of the agent lifecycle may be suspended during the creation and/or updates of the replicas. When a DimaX agent sends messages to other agents, DimaX provides communication mechanisms to localize agents and deliver them messages. This delivery is realized through the communication component of DimaX agents (DarXComComponent) which delegates the DIMA message transmissions to the as-

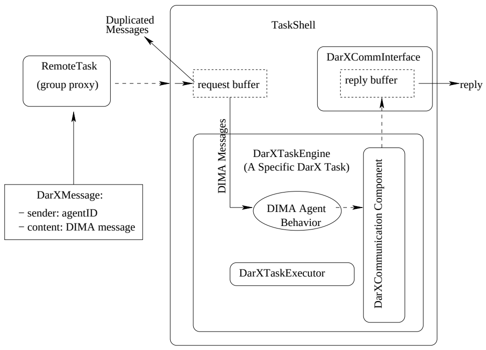

Figure 6: DimaX Agent Architecture

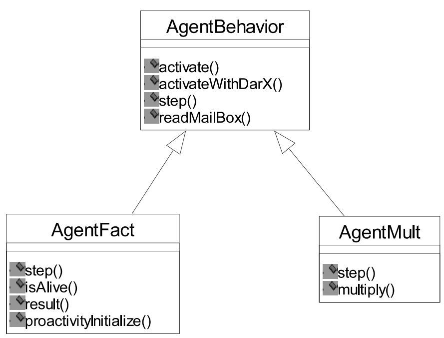

Figure 7: UML Diagram of Factorial Application

sociated DarXCommInterface. This communication interface enables DARX entities to communicate between them. So, at the application level, the agents communicate DIMA messages which are transmitted via the DARX middleware.

In our work, we assume that the agents directly interact using the FIPA-ACL language; they do not act on a common environment to avoid concurrency problems. Considering such issues, the filter mechanism, used at the TaskShell for the messages, could be applied to the environment.

3. EXAMPLE

Our aim is to facilitate the development of fault-tolerant MAS for developers not trained in fault-tolerance techniques. They need only to focus on problem solving issues like the agents behavior and their interactions. DimaX has been used to distribute several already developed multi-agent applications with the replication of their agents. The distribution and replication have not required any code modification. The distribution cost is therefore minimal. Actually, after the designer builds the agents behavior by using the DIMA multi-agent platform, he/she uses the *activateWithDarX* method to deploy his/her MAS and to endow it with fault-tolerance capabilities (i.e., replication). This activation method enables to encapsulate the agent behavior in a DarXTask (see Section 2.3) and register the agent in the system (i.e., the naming service). Its parameters are:

- the url and port: where the agent will be replicated,
- the degree of replication: the number of its replicas, and
- the replication strategy: how it is replicated (e.g., active or passive).

To exemplify DimaX, we propose the Factorial toy problem (n!). This toy problem gives some insights to distributed problem solving. We consider two kinds of agents:

1. AgentFact: these agents have the needed behavior to compute a factorial but they do not have the behavior to multiply numbers.

2. AgentMult: these agents have a behavior to compute a multiplication.

These agents are implemented as subclasses of BehaviorAgent class (see Figure 7). To compute n!, AgentFact creates a list (named Couple) with the numbers from 1 to n:

```
pubilc void proactivityInitialize(){
    for (int i=1, i<=n, i++){
        Couple.addElement(i);
    }
}
```

Then, it sends requests to AgentMult with all possible couples of numbers. When it receives a result, it puts it into the list:

```
pubilc void result(int i){
    Couple.addElement(i);
    // nbRequests is the number of resquests
    nbRequests −−;
}
```

If the list has more than one element, new requests are then sent to AgentMult. It repeats this action while the list contains more than one number or AgentFact has not the responses to all the sent requests. This test is performed by its isAlive() method as follows:

```
pubilc boolean isAlive(){
    return ((Couple.size() > 1) not(hasAllResponses()));
}
```

The AgentFact behavior is defined by:

```
pubilc void step(){
    while (hasMail())
        readMailBox();
    while (Couple.size()>1) {
            sendMessage("multiply",Couple.elementAt(0),
Couple.elementAt(1),AgentMultIdentifier);
        Couple.remove(0);
        Couple.remove(1);
        nbRequests++;
    }
}
```

The AgentMult behavior is as follows:

```
public void step(){
    while (hasMail())
        readMailBox();
}
```

The multiply action of MultAgent is:

```
public void multiply(int a, int b){
    int c=a*b;
    sendMessage("result",c, AgentFactIdentifier);
    }
}
```

The initialization of the MAS is performed as:

```
public void main (String [] args) {
    // agents creation
     AgentFact a= new AgentFact("requesterId");
     AgentMult b=new AgentMult("performerId");
     // agents activation on the same machine
     a.activate();
     b.activate();
}
```

With DimaX, the deployment can be performed as follows:

```
public void main (String [] args) {
     AgentFact a= new AgentFact("requesterId");
     AgentMult b=new AgentMult("performerId");
// agents activation on two different machines
     a.activateWithDarX(url1, port1, repDegree1, replica-
tionStrategy1);
     b.activateWithDarX(url2, port2, repDegree2, replica-
tionStrategy2);
}
```

As we can see, no code modification is performed by the designer in order to build a fault tolerant MAS. Moreover, the designer does not have to care of the observation feature and the replication management. So, the development effort is actually reduced.

4. DIMAX FEATURES

This section presents DimaX main features which are provided to the development and deployment of fault-tolerant large-scale MAS: scalability, reusability, robustness, and adaptability.

1. Scalability. A platform is said to be scalable if it can handle the increasing of the problem size (number of agents) and complexity without suffering a noticeable loss of performance. In DimaX, the proposed solution is to organize hierarchically the components of the different services in order to minimize the communication overload caused by them. DimaX also provides global state of MAS (e.g., the average number of exchanged messages), in a distributed manner. Indeed, this reduces remote access and avoids bottleneck, contrary to the case of a central component. Moreover, the messages used by the failure detection service are piggybacked by the other services messages and those of the application.

2. Reusability. To faciliate the design and implementation of fault-tolerant large-scale MAS, for developers not trained in fault-tolerance techniques, DimaX provides several component libraries to build multi-agent systems: decision components, communication components, interaction protocols. For example, the library of interaction protocols provides a generic implementation of interaction protocols. Interaction protocols are resuable components.

3. Robustness. The robustness of MAS is almost always a major concern when they are applied to critical domains like spacecraft, or medicine. It is important that this kind of application runs without interruption, in spite of failures, like crashes. DimaX achieves robustness of MASs by using adaptive replication mechanisms. To evaluate the reliability of the platform, we have run the robustness test based on fault injection techniques [9]. The results show that our platform achieves a robustness degree interesting of the application. Also, the platform must continue to deliver its services in despite of one of its services components failure. The failure of a machine or a connection often involves the failure of the associated DimaX server. However, in our solution, the fault tolerance protocols are agent-dependent and not-place dependent, i.e., the mechanisms built for providing the continuity of the computation are integrated in the replication groups, and not in the server.

4. Adaptability. To deal with limited resource problem for replicating agents, a good replication mechanism should adapt the replication strategy to the evolution of the environment. Thus, we have introduced, in DimaX, a multiagent monitoring architecture to control replication. This architecure implements our adaptation mechanisms to define the agent criticality. These mechanisms rely on organizational concepts like role and interdependence graph [6]. Moreover, due to the heterogeneous resource problem (i.e., different and dynamic characteristics of the hosts), DimaX uses an adaptive approach to resource management for determining the number of replicas and their placement.

5. RELATED WORK

In the multi-agent literature [16], we can find a large number of multi-agent platforms but only few ones offer fault-tolerance mechanisms.

Kumar et al. [12] advocate fault-tolerance approach by using broker teams. A broker accepts requests, locates capable agents, routing requests and responses, etc. They use multiple brokers which form a team with appropriate commitments. The team members should recover from broker failures insofar they have team and/or individual commitments like to connect to a registered agent which gets disconnected. In other words, this brokering knowledge is shared among the members. This work presents some interesting results, but stays at the theoretical stage. Moreover, they don't address scalability and reusability issues.

Cougaar [8] is a Java-based architecture for the construction of large-scale distributed agent-based applications. An agent is a set of problem solving behaviors interacting via blackboards. If an agent is unable to contact a member of its community it could send a health alert message to a health monitor. This agent is responsible for the recovery of agents.

For instance, the recovery of a domain agent consists either to retrieve an appropriate community state needed to pursue the problem solving or to re-join its community which has began a new problem solving stage. However, the approach lacks adaptability; no guarantee is given that the MAS will correctly pursue its goals, in spite of agent failures. Failures could cause interblocked situations; the progress of the problem solving depends on each other.

The FATMAS methodology [14] provides mainly four models used to design and implement the target system and a fault-tolerance technique where only a certain number of agents will be replicated. Here, an agent is critical as it performs at least one task that cannot be performed by any other agent in the system. If the agent is non-critical, then it is not replicated and its tasks are replicated in other agents. If it is a critical agent, then it must be replicated. FATMAS proposes guidelines for the analysis and the design of fault-tolerant MAS. Moreover, it provides agent and task replication. This enables to reduce the replication cost. However, the approach addresses to closed MAS; the agent criticality is defined at design time. The replication is static.

A. Fedoruk and R. Deters [2] propose to use proxies to make transparent the use of agent replication, i.e. enabling the replicas of an agent to act as a same entity regarding the other agents. The proxy manages the state of the replicas. All the external and internal communications of the group are redirected to the proxy. However this increases the workload of the proxy, which is a quasi central entity. To make it reliable, they propose to build a hierarchy of proxies for each group of replicas. This approach lacks reusability; in particular concerning the replication control.

6. CONCLUSION

In this paper, we presented a new fault-tolerant multiagent platform named DimaX. The design and the implementation of fault tolerant large scale multiagent systems require to deal with problems related to distribution and fault-tolerance. For that, DimaX provides several services namely naming service for agent localization, fault detection service for reconginizing faulty agents, observation service for collecting relevant information, and replication service for supporting replication techniques. Thanks to these services and their implementation, DimaX has interesting features like scalability, reusability, robustness, and adaptability for fault-tolerant MAS development. Thus, we achieve robustness by using replication techniques. Contrary to other approaches (i.e., diagnosis), replication enables us to run the critical multiagent applications without interruption. Moreover, our control of replication enables to change dynamically replication strategies, for better adapting to the evolution of the MAS context.

To generalize our approach, the futur work will propose a design methodology for fault-tolerant large-scale MAS. The principles developed in our approach to the failure problem in MAS will be the basis of the methodology.

7. ACKNOWLEDGMENTS

The authors would like to thank the members of Fault-Tolerant Multi-agent System project of the LIP6 for their many useful suggestions regarding the faulttolerant multi-agent systems. We are also grateful to Fabien Michel for providing valuable feedback on drafts of this paper.

8. REFERENCES

[1] M. Bertier, O. Marin, and P. Sens. Performance analysis of a hierarchical failure detector. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'2003)*, pages 635–644, San Francisco, USA, June 2003.

[2] A. Fedoruk and R. Deters. Improving fault-tolerance in mas with dynamic proxy replicate groups. In *IAT*, pages 364–370, 2003.

[3] FIPA Foundation for Intelligent Physical Agents. Fipa acl message structure specification.

[4] R. Guerraoui and A. Schiper. Software-based replication for fault-tolerance. *IEEE Computer*, 30(3):68–74, 1997.

[5] Z. Guessoum and J.P. Briot. From active object to autonomous agents. *IEEE Concurrency*, 7(3):68–78, 1999.

[6] Z. Guessoum, N. Faci, and J-P. Briot. Adaptive replication of large scale mass: Towards a fault-tolerant multiagent platform. In *Springer Verlag*, 2006.

[7] S. Hagg. A sentinel approach to fault handling in multi-agent systems. volume 1286 of *LNCS*, pages 190–195. Springer-Verlag, 1997.

[8] A. Helsinger, M. Thome, and T. Wright. Cougaar: a scalable, distributed multi-agent architecture. In *SMC (2)*, pages 1910–1917, 2004.

[9] M. Hsueh, T.K. Tsai, and R.K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.

[10] G.A. Kaminka, D.V. Pynadah, and M. Tambe. Monitoring teams by overhearing: A multi-agent plan-recognition approach. *Journal of Intelligence Artificial Research*, 17(1):83–135, 2002.

[11] M. Klein, J. Rodriguez-Aguilar, and C. Dellarocas. Using domain-independent exception handling services to enable robust open multi-agent systems: the case of agent death. *Journal of Autonomous Agents and Multi-Agent Systems*, 7(1-2):179–189, 2003.

[12] S. Kumar and P.R. Cohen. Towards a fault-tolerant multiagent system architecture. In *Proc.of 4th International Conference on Autonomous Agents*, pages 459–466, New York, USA, June 2000.

[13] O. Marin, P. Sens, J.P. Briot, and Z. Guessoum. Towards adaptive fault-tolerance for distributed multi-agents systems. In *Proc. Fourth European Research Seminar on Advances in Distributed Systems (ERSADS'01)*, pages 195–201, Bertinoro, Italy, May 2001.

[14] S. Mellouli, B. Moulin, and G.W. Mineau. Towards a modelling methodology for fault-tolerant multi-agent systems. In *Informatica Journal 28*, pages 31–40, 2004.

[15] D. Powell. Delta-4: A generic architecture for dependable distributed computing. In *Springer Verlag*, 1991.

[16] P-M. Ricordel and Y. Demazeau. From analysis to deployment: A multi-agent platform survey. volume 1972 of *LNAI*, pages 93–106. Springer-Verlag, 2004.

[17] R. van Renesse, K. Birman, and S. Maffeis. A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.