Adaptive Replication of Large-Scale Multi-agent Systems – Towards a Fault-Tolerant Multi-agent Platform

Zahia Guessoum[1,2], Nora Faci[2], and Jean-Pierre Briot[1]

[1] LIP6, Université Pierre et Marie Curie (Paris 6),
8 rue du Capitaine Scott, 75015 Paris, France
Zahia.Guessoum@lip6.fr, Jean-Pierre.Briot@lip6.fr
[2] MODECO-CReSTIC - IUT de Reims,
51687 Reims Cedex 2, France
faci@leri.univ-reims.fr

Abstract. In order to construct and deploy large-scale multi-agent systems, we must address one of the fundamental issues of distributed systems, the possibility of partial failures. This means that fault-tolerance is an inevitable issue for large-scale multi-agent systems. In this paper, we discuss the issues and propose an approach for supporting fault-tolerance of multi-agent systems. The starting idea is the application of replication strategies to agents, the most critical agents being replicated to prevent failures. As criticality of agents may evolve during the course of computation and problem solving, and as resources are bounded, we need to dynamically and automatically adapt the number of replicas of agents, in order to maximize their reliability and availability. We will describe our approach and related mechanisms for evaluating the criticality of a given agent (based on application-level semantic information, e.g. interdependences, and also system-level statistical information, e.g., communication load) and for deciding what strategy to apply (e.g., active or passive replication) and how to parameterize it (e.g., number of replicas). We also will report on experiments conducted with our prototype architecture (named DimaX).

1 Introduction

The possibility of partial failures is a fundamental characteristic of distributed applications. The fault-tolerance research community has developed solutions (algorithms and architectures), mostly based on the concept of replication, and notably applied to data bases. But, these techniques are almost always applied explicitly and statically, at design time. In such approaches, this is the responsibility of the designer of the application to identify explicitly which critical servers should be made robust and also to decide which strategies (active or passive replication...) and their configurations (how many replicas, their placement...).

A. Garcia et al. (Eds.): SELMAS 2005, LNCS 3914, pp. 238–253, 2006.
© Springer-Verlag Berlin Heidelberg 2006

New cooperative applications, e.g., air traffic control, cooperative work, and e-commerce, are much more dynamic and large scale. Such cooperative applications are now increasingly designed as a set of autonomous and interactive entities, named agents, that interact and coordinate (multi-agent system). In such applications, the roles and relative importance of the agents can greatly vary during the course of computation, of interaction and of cooperation, the agents being able to change roles, strategies. Also, new agents may also join or leave the application (open system). It is thus very difficult, or even impossible, to identify in advance the most critical software components of the application. Furthermore, criticality can vary over run time.

In addition, such applications may be large scale. And the fact that the underlying distributed system is large scale makes it unstable by nature, at least in currently deployed technologies. This increases the needs for mechanisms for adaptive fiabilisation (improving robustness) of the application.

Our approach is consequently to give the capacity to the multi-agent system itself to dynamically identify the most critical agents and to decide which fiabilisation strategies to apply to them. This is analog to load balancing but for fiabilisation. In other words, we would like to **automatically** and **dynamically** apply fiabilisation (mostly through replication mechanisms) **where** (to which agents) and **when** they are most needed. To guide the adaptive fiabilisation, we intend to use various levels of information, system level, like communication load, and application/agent level, like roles or plans.

This paper is organized as follows: Section 2 presents the related work and Section 3 introduces our multi-agent monitoring architecture. Sections 4 and 5 introduce a dynamic and adaptive control mechanism of replication. Section 6 presents the DimaX platform that we developed to implement this solution and the realized experiments.

2 Related Work

Several approaches address the multi-faced problem of fault tolerance in multi-agent systems. These approaches can be classified in two main categories: corrective (e.g., [10],[5]) and preventive (e.g., [12],[11],[14]). The preventive approach deals with the ability to continue to deliver services when faults occur. In the corrective approach, the process consists of fault diagnostic and repair. Moreover, several works address the difficulties of making reliable mobile agents, that are more exposed to security problems [1] such as intrusion detection. This category is beyond the scope of this paper.

Kaminka et al. [12] introduce a monitoring approach in order to detect and recover faults. They use models of relations between mental states of agents. They adopt a procedural plan-recognition based approach to identify the inconsistencies. However, the adaptation is only structural, the relation models may change but the contents of plans are static. Their main hypothesis is that any failure comes from incompleteness of beliefs. This monitoring approach relies on agent knowledge. The design of such multi-agent systems is very complex. Moreover, the agent behavior cannot be adaptive and the system cannot be open.

Horling et al. [11] present a distributed system of diagnosis. The faults can directly or indirectly be observed in the form of symptoms by using a fault model. The diagnosis process modifies the relations between tasks, in order to avoid inefficiencies. The adaptation is only structural because they do not consider the internal structure of tasks. The different diagnosis subsystems perform local updates on the task model. However, performance is optimized locally but not globally.

The work of Malone et al. [14] on coordination relies on a characterization of the dependencies between activities in terms of goals and resources. These dependencies represent situations of conflict, and the different coordination mechanisms represent the solutions to manage them. The main contribution of this approach is the proposed taxonomy of these dependencies. The authors offer a framework of coordination study, that provides the basic stone to build a monitoring approach. However, this monitoring approach has not yet been developed. This work has been reused by Klein et al. [16] to detect exceptions in multi-agent systems.

These corrective approaches present useful solutions to the problem of monitoring in multi-agent systems. However, the monitoring component is often centralized and its design relies on the agents' knowledge [19].

The fault-tolerance research community has developed preventive solutions (algorithms and architectures), mostly based on the concept of replication, and notably applied to data bases. Replication of data and/or computation is thus an effective way to achieve fault tolerance in distributed systems. A replicated software component is defined as a software component that possesses a representation on two or more hosts [6].

Many toolkits (e.g., [6] and [18]) include replication facilities to build reliable applications. However, most of them are not quite suitable for implementing large-scale, adaptive replication mechanisms. For example, although the strategy can be modified in the course of the computation, no indication is given as to which new strategy ought to be applied; moreover, such a change must have been devised by the application developer before runtime. Besides, as each group structure is left to be designed by the user, the task of designing a large-scale software appears tremendously complex.

S. Hagg introduces sentinels to protect the agents from some undesirable states [10]. Sentinels represent the control structure of a multi-agent system. They need to build models of each agent and monitor communications in order to react to faults. Each sentinel is associated by the designer to one functionality of the multi-agent system. This sentinel handles the different agents that interact to achieve the functionality. The analysis of its beliefs on the other agents enables the sentinel to detect a fault when it occurs. Adding sentinels to multi-agent systems seems to be a good approach. However the sentinels themselves represent failure points for the multi-agent system. Moreover, the problem solving agents themselves participate in the fault-tolerance process.

A. Fedoruk and R. Deters [5] 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. They point out the specific problems of read/write consistency, resource locking also discussed in [23]. This approach lacks flexibility and reusability in particular concerning the replication control. The experiments have been done with FIPA-OS, which does not provide any replication mechanism. The replication is therefore realized by the designer before run time.

The work by Kraus et al. [13] proposes a solution for deciding allocation of extra resources (replicas) for agents. They proceed by reformulating the problem in two successive operational research problems (knapsack and then bin packing). Their approach and results are very interesting but they are based on too many restrictive hypothesis to be made adaptive.

In the next section, we will introduce our monitoring multi-agent architecture, which allows to control automatically and dynamically the agent replication.

3 Monitoring Multi-agent Architecture

The deployment of large-scale multi-agent systems that must operate continuously faces several problems:

- the existing multi-agent architectures are often not well scalable [2],
- failures affect often a subset of the agents,
- the environment is often dynamic and the number of resources is limited.

One of the prime motivation behind the proposed monitoring multi-agent architecture is to improve the robustness of large-scale distributed multi-agent systems in dynamic environments and with limited number of resources. Monitoring consists thus in acquiring necessary information to dynamically and automatically apply replication to agents when it is most needed. This information may be based on standard measurements (communication load, processing time...) or multi-agent characteristics such as the roles of agents [8] or their interdependences.

3.1 Interdependence Graph

In a multi-agent system, each agent is defined as an autonomous entity. However, the agents do not always have all the required competencies or resources and thus depend on other agents to provide them. Interdependence graphs [3] [21] [22] were introduced to describe the interdependences of these agents. These graphs are defined by the designer before the execution of the multi-agent system. However, complex multi-agent systems are characterized by emergent structures [20], that thus cannot be statically defined by the designer.

In our architecture, a multi-agent system is therefore represented by a graph that reflects an emergent organizational structure. This structure can be interpreted to define each agent criticality.

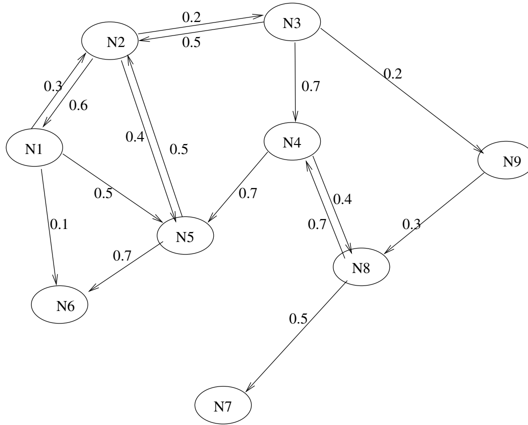

Fig. 1. Example of interdependence graph

For each domain agent[1], we associate a node. The set of nodes (see Figure 1), named interdependence graph, is represented by a labelled oriented graph (N, L, W). N is the set of nodes of the graph, L is the set of arcs and W the set of labels.

$$N = \{N_i\}_{i=1,n} \tag{1}$$

$$L = \{L_{i,j}\}_{i=1,n,j=1,n} \tag{2}$$

$$W = \{W_{i,j}\}_{i=1,n,j=1,n} \tag{3}$$

$L_{i,j}$ is the link between the nodes N_i and N_j and $W_{i,j}$ is a real number that labels $L_{i,j}$. $W_{i,j}$ reflects the importance of the interdependence between the associated agents ($Agent_i$ and $Agent_j$).

A node is thus related to a set of other nodes that may include all the nodes of a system. This set is not static: it can be modified when a new domain agent is added, or when an agent disappears, or when an agent starts interacting with another agent.

Our hypothesis is that the criticality of an agent relies on the interdependences of other agents on this agent. So, the agent $Agent_i$ is critical if the weights $wji_{j=1,n}$ are important. In this case, the failure of $Agent_i$ may be propagated to the agent $Agent_j$. It thus affects a subset of agents that form a connex component in the interdependence graph.

The interdependence graph is initialized by the designer. It is then dynamically adapted by the system itself. The proposed adaptation algorithms of the interdependence graph are described in Section 4. These adaptation algorithms are used by the monitoring agents that are described in the following section.

[1] In the following, we will name *domain agents*, agents from the application domain. In the following section, we will introduce other types of agents, named *monitoring agents*, to monitor them.

3.2 Multi-agent Architecture

In most existing multi-agent architectures, a monitoring mechanism is centralized. The acquired information is typically used off-line to explain and to improve the system's behavior. Moreover, the considered application domains typically only involve a small number of agents and *a priori* well-known organizational structures.

These centralized monitoring architectures are not suited for large-scale and complex systems where the observed information needs to be analyzed in real-time to adapt the multi-agent system to the evolution of its environment.

We thus propose to distribute the observation mechanism to improve its efficiency and robustness. This distributed mechanism relies on a reactive-agent organization. These agents have several roles:

- Observe the domain agents and their interactions,
- Build global information,
- Update the interdependence graph,
- Use the interdependence to define the agent criticality,
- Use the agent criticality to manage the resources.

These roles are assigned to two kinds of agents: domain agent monitors (named agent-monitors) and host-monitors (named host-monitors). An agent-monitor is associated to each domain agent and a host-monitor is associated to each host (see Figure 2).

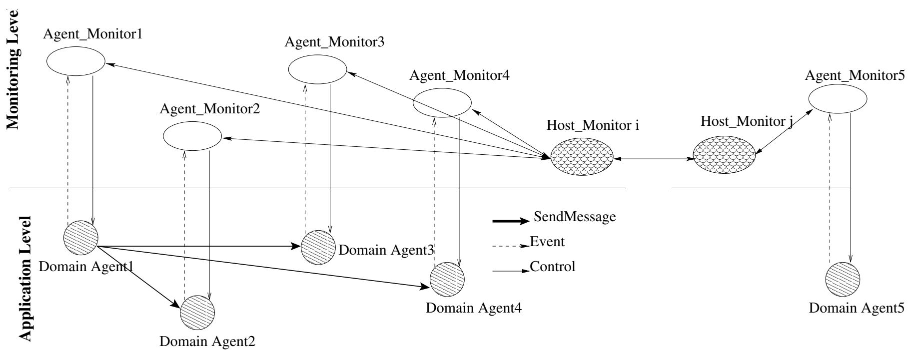

Fig. 2. Multi-agent architecture, domain agents are the agents of the application

An agent-monitor is thus associated to each agent of the application (named domain agent) and a host-monitor is associated to each host. These monitoring agents (agent-monitors and host-monitors) are hierarchically organized. Each agent-monitor communicates only with one host-monitor. Host-monitors exchange their local information to build global information (global number of messages, global exchanged quantity of information, ...).

After each interval of time Δt, the host-monitor sends the collected events and data to the corresponding agent-monitors. An agent-monitor has then the following behavior (see algorithm 1):

Algorithm 1. Agent-Monitor Behavior

1: Read the messages received from the host-monitor,
2: Update the local data and interdependences,
3: Compute the domain-agent criticality,
4: Determine the number of its replicas,
5: Inform the associated host-monitor of local changes that are important.

When the criticality of the domain agent is significantly modified, the agent-monitor notifies its host-monitor. The latter informs the other host-monitors to update global information. In turn, agent-monitors are informed by their host-monitors when global information changes significantly. Algorithm 2 describes the behavior of host-monitors.

Algorithm 2. Host-Monitor Behavior

1: Read messages received from the agent-monitors,
2: Update local statistics that define aggregation of the host-monitors parameters,
3: Send the new parameters to the agent monitors of the local host,
4: Send to the other host monitors the observed parameters that have significantly changed.

4 Adaptation Algorithms

Several parameters may be used to define the interdependences between agents such as communication load, executed tasks, roles of agents or their goals. An adaptation algorithm gives an outline of the adaptation mechanism of the interdependence graph. This adaptation relies on local information (communication load, cpu time ...) and on global information, which is defined as an aggregation of the local information of the various agents and hosts. The adaptation algorithm is thus used by each agent-monitor to manage the associated node of the interdependence graph (see Section 3.1).

Let us consider an interval of time Δt. The agent-monitors are activated each Δt. At each step, an agent-monitor executes an adaptation algorithm (see the two following subsections). However, the domain agents act continuously according to their goals and the evolution of their environment.

These algorithms are automatically used by each agent-monitor to update its interdependences with the set of agents. This set includes agents that communicate with it. Our hypothesis is that if an $Agent_i$ does not communicate with $Agent_j$ then then $Agent_i$ does not depend on $Agent_j$.

In this section, we propose two algorithms to compute the interdependence between two agents. The first one considers only the number of messages exchanged by agents and the second one deals with speech acts (performatives).

The first algorithm (see Algorithm 3) relies on the global number of sent messages $NbM(\Delta t)$, which is calculated by the host-monitor as follows:

$$NbM(\Delta t) = \sum_{i=1,n} \sum_{j=1,n \; \imath \neq j} NbM_{i,j}(\Delta t) \tag{4}$$

where $NbM_{i,j}(\Delta t)$ is the number of messages received by $agent_i$ from $agent_j$ during the interval of time Δt.

Algorithm 3. Basic adaptation of the interdependences

1: **for** each j different of i **do**
2: Update the weights by using the following rule:

$$W_{i,j}(t + \Delta t) = W_{i,j}(t) + NbM_{i,j}(\Delta t)/NbM(\Delta t) \tag{5}$$

3: **end for**

Algorithm 3 is very simple, thus the cost of monitoring is very low. Consequently, it is useful for applications where the semantics of messages is not required. However, several applications rely on semantics of messages. So, we propose a new algorithm that deals with performatives. This algorithm is described in the following section.

The second algorithm (see Algorithm 4) relies on the semantics proposed by FIPA and the influence of the reception of a message on the receiver. Based on the work of Colombetti and Verdicchio [4], we propose the following six classes of performatives:

- class 1 =request, request-whenever, query-if, query-ref, subscribe
- class 2 = inform, inform-done, inform-ref
- class 3 = cfp, propose
- class 4 = reject-proposal, refuse, cancel
- class 5 = accept-proposal, agree
- class 6 = not-understood, failure.

To represent the influence of a message on its receiver, we use a graduation of the interval of possible variations [0, 1], where:

- 0 corresponds to no influence,
- 1 corresponds to the maximum influence.

Table 1. Symbolic values of the six classes

Classes	Symbolic Value
classes 4, 6	Low
classes 2, 3, 5	medium
class 1	high

We propose then to represent influences by symbolic values such as *low, medium, high*, that correspond respectively to the intervals: $[0, 0.35],]0.30, 0.65]$ and $]0.60, 1]$. The average value of each symbolic value is the median of its interval. It is used to define the weight of a message.

Table 1 gives the symbolic values of the six classes.

Let us consider:

- ΔW: an aggregation of the variations of $W_{i,j}$, as defined below:

$$\Delta W(t) = \sum_{i=1,n} \sum_{j=1,n \; \imath \neq j} \Delta W_{i,j}(\Delta t) \tag{6}$$

- $S_{i,j}$: the set of messages received by $Agent_i$ from $Agent_j$.

The weight of a message is defined by the median of the interval corresponding to the fuzzy value of its performative.

Algorithm 4. Performative-based adaptation of the interdependences

1: **for** each j different from i **do**
2: Update $W_{i,j}$ by using the following rule:

$$W_{i,j}(t + \Delta t) = W_{i,j}(t) + \sum_{m \in S_{i,j}} weight(m)/\Delta W(t) \tag{7}$$

3: **end for**

Algorithm 4 cost seems higher than that of the first Algorithm 3. However, the semantics of messages is very useful when dealing with interdependences in some application domains such as e-commerce.

5 Agent Criticality

The analysis of events and measures (system data and interaction events) provides two kinds of information: the interdependence and the degree of activity of each agent. To evaluate the degree of the agent activity, we use system data that are collected at the system level. We are considering two kinds of measures: CPU time and communication load. We are currently evaluating the significance of these measures as indicators of agent activity, to be useful to estimate agent criticality.

For an agent $Agent_i$ and a given time interval Δt, these measures provide:

- The used time of CPU (cp_i),
- The communication load (cl_i).

cp_i and cl_i may be then used to measure the agent degree of activity aw_i as follows:

$$aw_i(t) = (d_1 * cp_i/\Delta t + d_2 * cl_i/CL)/(d_1 + d_2) \tag{8}$$

where:

– CL is the global communication load,
– d_1 and d_2 are weights introduced by the user.

The estimation of the criticality of the agent $Agent_i$ is computed as follows:

$$w_i(t) = (a_1 * aggregation(W_{ij,j=1,m}) + a_2 * aw_i(t))/(a1 + a2) \quad (9)$$

Where a_1 and a_2 are the weights given to the two kinds of parameters (interdependences and degree of activity). They are introduced by the designer.

Note that in our experiment (see Section 6), we do not consider the activity. So, $a_1 = 1$ and $a_2 = 0$.

For each Agent A_i, its estimated criticality w_i is used to compute the number of its replicas and decide where to replicate the agents (see our SELMAS'2005 paper [9] for the resources management problem).

6 Implementation and Experiments

This section gives an overview of the realized platform (named DimaX) that implements our adaptive replication mechanism. It then describes the example that we use for the experiments and give some results.

6.1 Overview of DimaX

DIMA [7] and DarX [15] [2] have been integrated to build a fault-tolerant multi-agent platform (named DimaX). DimaX provides multi-agent systems with several services such as distribution, replication, and naming service [15]. In order to benefit from fault tolerance mechanisms, the agent behavior is wrapped in a task of the DarX framework (see Figure 3). Moreover, for a dynamic control of replication, the monitoring architecture has been introduced. Figure 3 gives an overview of DimaX.

We consider a distributed system consisting of a finite set of agents $Ai = \{A_1, A_2, \ldots, A_n\}$ that are spread through a network. These agents communicate only by sending and receiving messages.

DarX provides global naming. Each agent has a global name that is independent of the current location of its replicas. The underlying system allows to handle the agent's execution and communication.

The failure of a machine or a connection often involves the failure of the associated DarX 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 servers. For instance, the monitoring agents are built as active components associated to the domain agents.

Moreover, DarX provides a fault-detection mechanism. A machine crash - server failure[2] - is handled in three steps within every replication group:

[2] In this work, we consider fail-silent (crash) model of faults.

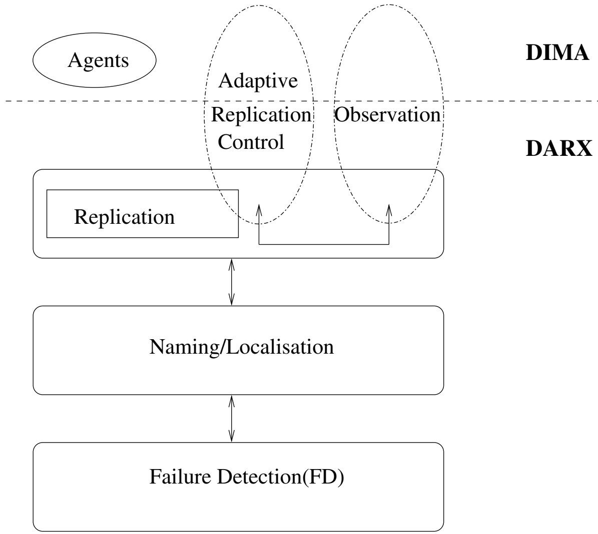

Fig. 3. Overview of DimaX

- detection of an eventual failure within the group,
- evaluation of the context: new criticality, leader failure, ...
- recovery: If the missing replica was the group leader, a new one is elected and an agent monitor is automatically activated. In the other case, it depends on the evaluation; a new follower/backup may or may not be instantiated.

Obviously, if a leader without any follower/backup fails, then it is not recovered. This derives from the original assumptions we made: the criticality of an agent evolves during the computation, and there are phases when an agent do not need to be fault-tolerant [15].

To validate DimaX, we realized several series of experiments. The first series evaluates the performances of the proposed multi-agent architecture and the proposed adaptation algorithms. The second one evaluates the robustness of the multi-agent systems that are based on the proposed monitoring architecture.

The following sections describe our example and the experiments.

Note: The experiments presented in this section were carried out on twenty machines with Intel(R) Pentium(R) 4 CPU at 2 GHz and 526 Mb of RAM.

6.2 Example

In our experiments, we consider the example of a distributed multi-agent system that helps at scheduling meetings. Each user has a personal assistant agent that manages his/her calendar. This agent interacts with:

- the user to receive his meeting requests and associated information (a title, a description, possible dates, participants, priority, etc.),
- the other agents of the system to schedule a meeting.

If the assistant agent of one important participant (initiator or prime participant) in a meeting fails (e.g., its machine crashes), this may disorganize the whole process. As the application is very dynamic - new meeting negotiations start and finish dynamically and simultaneously - decision for replication should be done automatically and dynamically.

6.3 Performances

The proposed monitoring multi-agent architecture is very useful to implement the proposed adaptive replication mechanism. However, the monitoring cost does not seem insignificant. So, our first series of experiments measures the monitoring cost in the proposed architecture. We consider, a multi-agent system with n distributed agents that execute the same scenario (a fixed set of meetings to schedule). We realized several experiments with various number of agents. For each n (100, 150, ..., 300), we considered m meetings (20, 40, ..., 80) and we realized two kinds of measures (with and without monitoring). We used 20 machines for each experiment and we repeated each experiment 10 times. We considered three cases: 1) a multi-agent system without monitoring, 2) a multi-agent system with monitoring based on *Algorithm 1*, and 3) a multi-agent system with monitoring based on *Algorithm 2*.

Figure 4 shows the average global execution time for these three different monitoring solutions. We found that monitoring cost is almost a constant function. The monitoring activity does not increase when the number of agents (domain agents and associated monitoring agents) increase. That can be explained by the proposed optimization within the multi-agent architecture, such as the hierarchical organization of monitoring agents and the communication between the agent-monitors

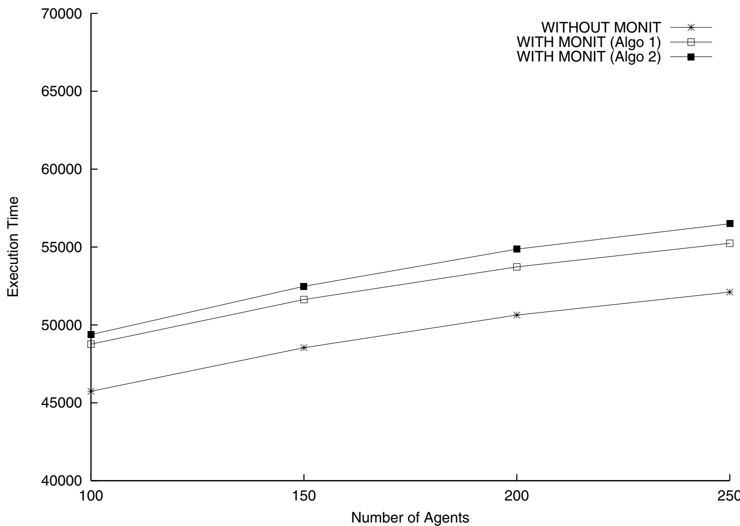

Fig. 4. Fault-Tolerant Multi-Agent Systems Costs

Table 2. Monitoring Cost and Comparison of the two Algorithms

Number of Agens	Monitoring Cost of Algorithm 1	Monitoring Cost of Algorithm 2	Difference
100	3025	3635	610
150	3094	3945	851
200	3089	4227	1138
250	3130	4387	1257

and host-monitors. These agents are organized hierarchically. For instance, to build the global information (global communication load ...), the host-monitors communicate only if the local information changes. Moreover, the host-monitors exchange local information only when there is an important change. Therefore, the number of communications between these agents is optimized.

6.4 Robustness

For this second series of experiments, we use a failure simulator. This simulator chooses randomly an agent and stops its thread. If the killed agent is critical then the multi-agent application fails. We considered a multi-agent system with 200 agents distributed on 10 machines. We run each experiment 10 minutes and we introduce 100 faults. We repeated several times the experiment with a variable number of extra resources Rm. Here, Rm defines the number of extra replicas that can be used by the whole multi-agent system. This experiment measures the rate of succeeded simulations SR, which is defined as follows:

$$SR = \frac{NSS}{TNS} \tag{10}$$

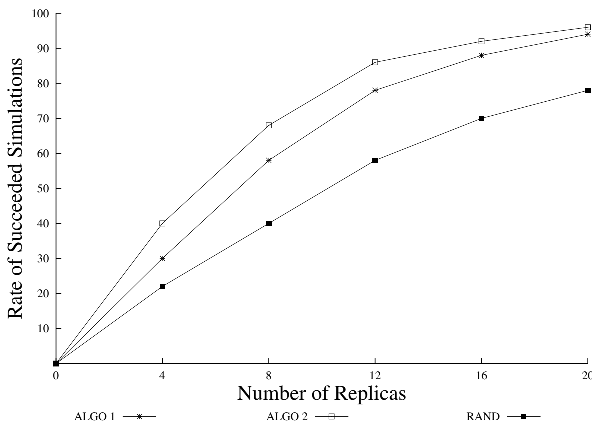

Fig. 5. Rate of succeeded simulations for each number of replicas with the two Algorithms

where NSS is the number of simulations that did not fail and TNS is the total number of simulations. Let us remind that a simulation fails when the fault simulator stops a critical agent that has not been replicated.

We considered three cases: 1) the replication is random, 2) the replication is based on algorithm 1 and 3) the replication is based on algorithm 2.

Figure 5 shows the success rate SR as a function of the number of extra replicas. It compares the two algorithms. It shows that algorithm 2 gives the best results for the considered application. Meanwhile the two algorithms require a number of extra resources that is at least equal to the number of critical agents.

Moreover Figure 5 compares the two algorithms with a random replication. In this case the agents criticality is defined randomly. We thus show that our algorithms for adaptive replication are more accurate than random replication.

6.5 Discussion

In the example, the monitoring cost of the second algorithm is more important than the cost of the first one. The difference corresponds to the cost of the message analysis. However, multi-agent systems using the second algorithm are more robust. Indeed, in our application the content of messages is important. This is not the case for some application domains such as network management where all the messages have the same weight. In this kind of application it is recommended to use the first algorithm to reduce the monitoring cost.

It is thus useful to study classes of application domains for each algorithm. The results of this study can be then used to help the designer to choose the most suited algorithm for his/her application.

7 Conclusion

This paper presented a new approach to make large-scale multi-agent systems reliable. This approach is based on the concepts of interdependence, where an agent criticality is estimated through its interdependences with other agents. The agent criticality is then used to replicate agents in order to maximize their reliability and availability based on available resources and their costs.

We thus proposed a generic architecture to extend an already built where an agent criticality is estimated through its interdependences multi-agent system with a basic adaptation mechanism to dynamically and automatically update the replication strategy. To make concrete this architecture, we have implemented a fault-tolerant multi-agent platform (named DimaX). DimaX is the result of an integration of the DIMA multi-agent platform [7] and the DarX replication framework [2].

The obtained results are interesting and promising. However, more experiments with a large-scale real-life applications and several local area networks (e.g. the one of our two teams : LIP6 and CReSTIC) are needed to validate the proposed approach and to analyze the proposed algorithms. Moreover, the proposed classification of performatives needs to be evaluated and compared with other classifications and different weights.

Finally, we are working on a methodology based on the Model Driven Architecture [17] to facilitate the design of fault-tolerant multi-agent systems and their implementation with DimaX.

Acknowledgment

The authors would like to thank the members of *Fault-Tolerant Multi-agent System* project of the LIP6 for their many useful suggestions regarding fault-tolerant multi-agent systems.

References

1. F.M. Assis-Silva and R. Popescu-Zeletin. An approach for providing mobile agent fault tolerance. In *Second International Workshop on Mobile Agents*, number 1477 in LNCS. Springer-Verlag, 1998.
2. M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *the International Conference on Dependable Systems and Networks*, Washington, USA, 2002.
3. C. Castelfranchi. *Decentralized AI*, chapter Dependence relations in multi-agent systems. Elsevier, 1992.
4. Marco Colombetti and Mario Verdicchio. An analysis of agent speech acts as institutional actions. In *AAMAS'2002*, pages 1157–1164, 2002.
5. A. Fedoruk and R. Deters. Improving fault-tolerance by replicating agents. In *AAMAS'2002*, pages 373–744, Bologna, Italy, 2002.
6. R. Guerraoui, B. Garbinato, and K. Mazouni. Lessons from designing and implementing GARF. In *Object-Based Parallel and Distributed Computation*, number 791 in LNCS, pages 238–256, 1995.
7. Z. Guessoum and J.-P. Briot. From active objects to autonomous agents. *IEEE Concurrency*, 7(3):68–76, 1999.
8. Z. Guessoum, J.-P. Briot, O. Marin, A. Hamel, and P. Sens. *Software Engineering for Large-Scale Multi-Agent Systems*, chapter Dynamic and Adaptative Replication for Large-Scale Reliable Multi-Agent Systems, pages 182–198. Number 2603 in LNCS. April 2003.
9. Z. Guessoum, N. Faci, and J.-P. Briot. Adaptive replication of large-scale multi-agent systems - towards a fault-tolerant multi-agent platform. In *Proceedings of the ICSE'05 Fourth International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'05)*, Saint Louis, U.S.A., may 2005. ACM.
10. S. Hagg. A sentinel approach to fault handling in multi-agent systems. In C. Zhang and D. Lukose, editors, *Multi-Agent Systems, Methodologies and Applications*, number 1286 in LNCS, pages 190–195, 1997.
11. B. Horling, B. Benyo, and V. Lesser. Using self-diagnosis to adapt organizational structures. In *5th International Conference on Autonomous Agents*, pages 529–536, Montreal, 2001. ACM Press.
12. G. A. Kaminka, D. V. Pynadath, and M. Tambe. Monitoring teams by overhearing: A multi-agent plan-recognition approach. *Journal of Intelligence Artificial Research*, 17:83–135, 2002.
13. S. Kraus, V.S. Subrahmanian, and N. Cihan Tacs. Probabilistically survivable MASs. In *IJCAI'03*, pages 789–795, 2003.

14. T. W. Malone and K. Crowston. The interdisciplanary study of coordination. *ACM Computing Surveys*, 26(1):87–119, March 1994.
15. O. Marin, M. Bertier, and P. Sens. DARX - a framework for the fault-tolerant support of agent software. In *14th International Symposium on Software Reliability Engineering (ISSRE'2003)*, pages 406–417, Denver, Colorado, USA, 2003. IEEE.
16. M.Klein, J.A. 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.
17. OMG TC Document ormsc/2001 07-01. Model driven architecture (mda). Technical report, OMG, 2001.
18. R. Van Renesse, K. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.
19. N. Roos, A.t. Teije, and C. Witteveen. A protocol for multi-agent diagnosis with spatially distributed knowledge. In ACM, editor, *First Workshop on Programming Multiagent Systems: Languages, frameworks, techniques, and tools (ProMAS03), AAMAS'03*, pages 655–661, July 2003.
20. J. S. Sichman and R. Conte. Multi-agent dependence by dependence graphs. In *AAMAS'2002*, pages 483–490, Bologna, Italy, 2002. ACM.
21. J. S. Sichman, R. Conte, and Y. Demazeau. Reasoning about others using dependence networks. In *Actes de Incontro del gruppo AI*IA di interesse speciale sul inteligenza artificiale distribuita*, Roma, Italia, 1993.
22. J. S. Sichman, R. Conte, and Y. Demazeau. A social reasoning mechanism based on dependence networks. In *Proceedings of ECAI'94 - European Conference on Artificial Intelligence*, Amsterdam, The Netherlands, August 1994.
23. L. Silva, V. Batista, and J. Silva. Fault-tolerant execution of mobile agents. In *International Conference on Dependable Systems and Networks*, pages 135–143, 2000.