Plan-Based Resource Allocation for Providing Fault Tolerance in Multi-agent Systems

Alessandro de Luna Almeida[1], Samir Aknine[1], Jean-Pierre Briot[1,2]

[1] Université Pierre et Marie Curie-Paris6,UMR 7606
4 Place Jussieu, Paris, F-75005 France

[2]Laboratório de Engenharia de Software (LES), DI, PUC-Rio
Rua Marquês de São Vicente 225, 22451-900, Gávea, Rio de Janeiro, RJ, Brazil

{Alessandro.Luna-Almeida, Samir.Aknine, Jean-Pierre.Briot}@lip6.fr

Abstract. *In this article, we propose an original method for providing fault tolerance in multi-agent systems through replication. Our method focuses on building an automatic, adaptive and predictive replication policy to solve the resource allocation problem of determining where agents must be replicated to minimize the impact of failures. This policy is determined by taking into account the criticality of the plans of the agents, which contain the collective and individual behaviors of the agents in the application. Some measurements assessing the efficiency of our approach and future directions are also presented.*

1. Introduction

The possibility of partial failures is a fundamental characteristic of distributed applications. In order to prevent that a system stops working properly due to the occurrence of faults, many fault tolerance approaches have been proposed, some more curative e.g., based on exception handling and cooperative recovery [Romanovsky et al. 2001], and some more preventive, notably based on the concept of replication, i.e. creation of copies of a component in distant machines.

As discussed by [Guerraoui and Schiper 1997], software replication in distributed environments has some advantages over other fault-tolerance solutions: it provides the groundwork for the shortest recovery delays, it is less intrusive with respect to execution time, and it scales much better.

Replicating every single component of the application in every machine is not a feasible approach due to the limit on the resources available per machine. In general, it is the responsibility of the designer of the application to explicitly identify what critical components should be made robust and how to parameterize replication. This can be decided either statically before the application starts [Fedoruk and Deters 2002, Kraus et al. 2003] or in a non-automatic way during the execution of the system [Cukier et al 1998, Favarim et al. 2003, Kalbarczyk et al. 1999].

However, those works are not suitable for multi-agent systems (MAS) applications for two main reasons: firstly, MASs can be very dynamic and thus, it is very difficult, or even impossible, to identify in advance the most critical agents; secondly, for large-scale applications, a manual control is not realistic, as the

application designer cannot monitor the evolution of a distributed application of a significant scale.

In this paper, we will introduce our approach to building reliable multi-agent systems. It is based on the concept of criticality, a value (evolving in time) associated to each agent in order to reflect the effects of its failure on the overall system.

Previous work [Briot et al. 2006] of our project uses different kinds of information to calculate the criticality of an agent (e.g., processing time, messages exchanged between agents). However, those types of information do not necessarily capture with precision the behavior of the agents and thus are not enough to determine their criticality. Additionally, neither the problem of optimal allocation of resources nor the analysis of a long term replication strategy is taken into account in this work.

As presented in [Almeida et al. 2006], our current work goes further by taking into consideration the plans of the agents (the actions that each agent has planned to execute in the near future) to determine the criticality in a more precise way. A plan-based fault-tolerant mechanism acts as a promising preventive method since it takes into account the prediction of the future behavior of the agents and their influence over the other agents of the society.

In this paper, we adopt a new representation of plans using Recursive Petri Nets, we propose the definition of a problem of resource allocation which uses a refined model of probability of failures to take into account timing aspects and we introduce our current collaboration with Eurocontrol, which brings in applications in air traffic management as primary test fields for our approach.

The remainder of this paper is organized as follows. Section 2 introduces the fault tolerance problem we deal with in this paper. Section 3 explains how the plans of the agents can be used as an approach to this problem. Section 4 defines the fault tolerance problem as a resource allocation problem and proposes a heuristic solution to it. Section 5 describes the general architecture of the experimental platform. Section 6 shows some preliminary results. Finally, in section 7 we present our conclusions and perspectives for future work.

2. Problem Definition

The fault tolerance problem described in this paper considers a set of n_a agents $S = \{Agent_1, Agent_2, ..., Agent_{na}\}$ that have to complete a set of tasks. Let us consider an application of assistance for air traffic control through assistant agents. (This is a simplified scenario of an ongoing collaborative project with EuroControl, the European Organisation for the Safety of Air Navigation). The airspace is divided into sectors, each sector being controlled by a human controller (see Figure 1). Each controller is assisted by an assistant agent who cooperatively monitors the air traffic to suggest decisions about traffic control. Agents communicate in order to assist with collaborative procedures, e.g. hand off procedures, that is when a controller passes the responsibility of an airplane exiting from its supervision sector to the controller of the sector the plane is entering.

While trying to accomplish their tasks, agents can be faced to different kinds of failures. In our work, we initially consider just the crash type of failures, i.e. when a component stops producing output. In our model of failure, machines can crash due to

internal (operating system crashes, hardware problems) or external factors (malicious attacks, power failures, environmental disasters), and as a consequence all the agents executing in it will crash as well. Additionally, the failure of one agent can also impact on the agents who depend on it.

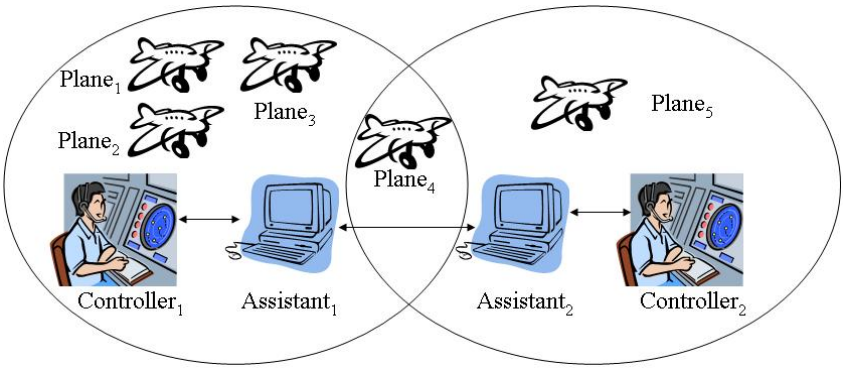

Figure 1. Example of air traffic control

To minimize the impact of failures, agents can be replicated. The two main types of replication protocols are: active replication, in which all replicas process concurrently all input messages; 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. It is important to notice that using replication also minimizes the impact of network failures.

It is clear that replicating every agent an unlimited number of times is not a feasible approach since not only the available resources are often limited, but also the overhead imposed by the replication could degrade performance. The problem consists in finding a replication scheme which minimizes the probability of failure of the most critical agents. This scheme must also be revised over time, considering that the multi-agent execution context of tasks is dynamic and, thus, the criticalities of the agents vary at runtime.

3. Our Plan-Based Criticality Assessment Method

In our approach to solve the problem defined in the last section, we consider that each agent of the system knows which sequence of actions (plan) must be executed in order to accomplish its current goals. The generation and update of plans are out of the scope of our work. We assume that they are generated either by a central planner or by the agents themselves in a distributed way.

We represent the plan of an agent as an acyclic Recursive Petri Net (RPN) [Seghrouchni and Haddad 1996] where each place represents a state of the plan and a transition models an action. As opposed to classical Petri Nets, RPNs allow the representation not only of elementary actions (an irreducible task which can be performed without any decomposition) but also of abstract actions (the execution of which requires its substitution by a new sub-plan). Actually, using RPNs, one can represent partial plans which briefly describe the actions of the agents at a certain iteration of the resolution. These plans can then be refined according to the evolution of the execution of the agents.

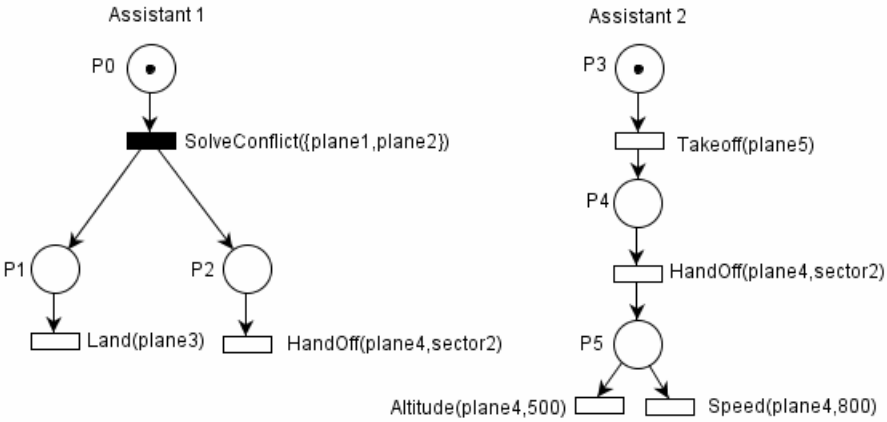

Figure 2. Example of two interacting plans (transitions in black represent abstract actions)

In the example of Figure 2, we show two plans elaborated by two assistant agents. After solving the conflict between *plane1* and *plane2*, $Assistant_1$ asks *plane3* to land and hands off *plane4* to *sector2* (controlled by $Assistant_2$). The action *SolveConflict* is an abstract action (transition in black) and, thus, $Assistant_1$ needs to refine it (by another RPN) during execution. The action *HandOff* is a joint action which needs synchronization between the two agents. As a consequence, when $Assistant_1$ proposes to hand off *plane4* to *sector2*, it must ensure that $Assistant_2$ is ready to receive the details of the transfer. At the implementation level, this is done by a synchronizing transition.

Definition 1: An *external* transition is a transition representing an action belonging to the plan of an agent which will be executed by others.

Definition 2: The *set of children* of a transition *t* in a plan *p* (denoted by *Children (t, p)*) is the set of places which are directly connected to the transition *t* in the plan *p*. Similarly, the set of children of a place is the set of transitions which are directly connected it.

For example, in Figure 2, $Children\ (SolveConflict,\ Plan_1) = \{P1,\ P2\}$ and $Children\ (P5,\ Plan_2) = \{Altitude,\ Speed\}$.

3.1. Agent Criticality

The criticality of an agent at any time can be calculated based on the criticalities of the forthcoming actions which belong to its plan. An agent who executes critical actions must be considered critical. In a given time *t*, the criticality of the agent will be given by the relative criticality of the initial place of its plan (where the token is initially located).

Before defining the relative criticality, let's first introduce the concept of absolute criticality. The *absolute criticality (AC)* of an action (transition) is defined without taking into account the current plans of the agents. It is given a priori by the system designer and can be determined in function of a number of factors: number of agents capable of performing the action, resources required for the execution of the action, application dependant information.

The *relative criticality (RC) of a place* in a plan estimates the aggregation of the criticalities of the children (transitions) of the place in the plan. We assume that the

children transitions of a place have the same probability of being executed. Thus, we use the mean as the aggregation function for estimating what is, on average, the risk of not executing the portion of the plan which has the corresponding place as a root. Let *s* be one place belonging to a plan *p*, and *f* be the aggregation function (*MEAN*) of the children of *s* in *p*. Then, the relative criticality of *s* in *p* (*RC (s, p)*) is formally defined by:

$$RC(s,p) = f(RC(t_1,p), RC(t_2,p), \ldots), \forall t_i \in Children(s,p) \quad \textbf{(1)}$$

Finally, the *relative criticality of a transition* executed by an agent (possibly jointly with other agents) estimates the impact of its failure to the multi-agent system as a whole. The *RC* depends on the absolute criticality of the action and on the usefulness of its results to all the agents which depend on it to perform their tasks. It is given by:

$$RC(t,p) = \begin{cases} \sum_{s_i \in Children(t,p)} RC(s_i,p) & \text{if } t \text{ is external} \\ AC(t) + \sum_{p_j | t \in p_j} \sum_{s_i \in Children(t,p_j)} RC(s_i,p_j) & \text{otherwise} \end{cases} \quad \textbf{(2)}$$

In other words, for an external transition, the *RC* is equal to the sum of the children relative criticalities. For a non-external transition, the *RC* is equal to its absolute criticality plus the sum of the relative criticalities of all of its children in all the plans to which it belongs.

In dynamic and unreliable environments, actions with a late start time will be executed less possibly than actions with an early starting time, since the plans can change or failures can happen. Consequently, we have also refined this strategy by considering the expected starting time of actions. We compute the estimated starting time of the actions using a topological sorting in the graph (top-down) considering the elapsed times of the antecedents and siblings' actions.

To deal with the dynamicity of multi-agent systems, criticalities need to be updated along time. We proposed two main types of strategies to revise the criticality: time-driven strategies and event-driven strategies (action completion, failure). More details are presented in [Almeida et al. 2006].

4. Resource Allocation Problem

Once estimated the criticality of the agents (using the strategies described in the last section, for example), one may ask which agents should be replicated and where to deploy the replicas. It is important to notice that it is not enough to determine the number of replicas that should be given to each agent. It is also essential to address the problem of where deploying efficiently the replicas so as to take into consideration the failure probability of the replicas. In fact, it is better to have only one replica which will have in the future a probability of failure equals to 0.1 than having five replicas with a probability of failure equals to 0.9 each.

Hence, we propose the definition of a resource allocation problem which considers the probability of failures and a mechanism which solves this problem in a satisfactory way.

The problem of resource allocation considers a set of agents $S = \{Agent_1, Agent_2, ..., Agent_{na}\}$, a set of machines $M = \{m_1, m_2, \ldots, m_{nm}\}$ and a set of resources $R = \{r_1, r_2, \ldots, r_{nr}\}$, where n_a, n_m and n_r are respectively, the total number of agents, of machines and of replication resources (a resource for us is a place in a machine where the agent can be deployed for execution).

Definition 3: We define *membership function* as a total surjection $b: R \rightarrow M$, where the image of a resource under *b* is the machine to which it belongs.

Definition 4: We define *resource allocation function* as an allocation of replication resources to the agents, i.e. a total function $g: S \rightarrow 2^R$, where 2^R is the power set of *R*. A same replication resource cannot be allocated to two different agents. In other words, let $R_i = \{r_{i1}, r_{i2}, \ldots, r_{ini}\}$ be the value of *g* at $Agent_i$. Then:

$$\forall i, j \cdot R_i \cap R_j = \emptyset \quad \textbf{(3)}$$

Since our model of failure considers machine failures, it is useless to deploy replicas of the same agent using different resources in the same machine. Thus, we add the following restriction to the resource allocation functions:

$$\forall a \in S \cdot g(a) = R_a \Rightarrow (\forall r_j, r_k \in R_a \cdot b(r_j) \neq b(r_k)) \quad \textbf{(4)}$$

Finally, since every agent must be deployed somewhere in the system, we add this last condition:

$$\forall a \in S \cdot g(a) \neq \emptyset \quad \textbf{(5)}$$

Definition 5: For each machine m_i of the system, we define the *mean time between failures (MTBF)* as the mean (average) time between two failures produced at m_i. We will denote the MTBF by θ_i.

The value of each θ_i can be computed by collecting statistical data on past failures of each node. Let T_i be the total time when m_i has been up and N_i be the number of failures during T_i. Then the MTBF can be calculated as follows:

$$\theta_i = \frac{T_i}{N_i} \quad \textbf{(6)}$$

Definition 6: For each machine m_i of the system, we define the *failure rate* (denoted as λ_i) as the *frequency* with which m_i fails. One can calculate the failure rate as the inverse of the MTBF:

$$\lambda_i = \frac{1}{\theta_i} = \frac{N_i}{T_i} \quad (7)$$

Failure rates are generally modeled by the reliability *bathtub curve*. In this model, the life of a product is divided in three phases. The first phase (early life) is characterized by a decreasing failure rate. As the time passes, the failure rate becomes nearly constant, and we enter what is considered the useful life period. Finally, as the product exceeds its design lifetime, failures occur at increasing rates.

In our model, we suppose that the machines used in the system are neither in their starting phase nor in the wear out period. Hence, we will assume a constant failure rate for the duration of the execution. It is important to notice that our model remains

sound even if the machines are not in the useful life period, as long as the duration of the execution is not too long (in those cases failure rates are also almost constant).

Definition 7: Let λ_i be the constant failure rate of the machine m_i, then we can define its *failure density function* by the following exponential density function:

$$f_i(t) = \lambda_i e^{-\lambda_i t} \quad \textbf{(8)}$$

The failure density function represents the time to failure of the corresponding machine. Intuitively, the expected value of the failure density function should be the mean time between failures. In fact, one can prove easily that it is given by:

$$E_i(t) = \int_0^\infty t f_i(t)\, dt = \frac{1}{\lambda_i} = \theta_i \quad \textbf{(9)}$$

Given the failure density function, one can obtain the corresponding probability distribution of the time to failure:

$$F_i(t) = P(TimeToFailure \leq t) = \int_0^\infty f_i(t)\, dt = 1 - e^{-\lambda_i t} \quad \textbf{(10)}$$

Definition 8: We define the *reliability* of the replication resource r_k at the interval of time *[0, t]* (denoted by $v_k(t)$), as the probability that it will not crash before the time *t*. Let $m_i = b(r_k)$, then:

$$v_k(t) = P(TimeToFailure \geq t) = 1 - F_i(t) = e^{-\lambda_i t} \quad \textbf{(11)}$$

If we assume that the failures of the resources R_i allocated to an agent $Agent_i$ are independent (which is often the case when they belong to different machines), it is easy to show that the probability that an agent $Agent_i$ will fail is given by the equation:

$$P(Failure(Agent_i) = 1) = (1 - v_{i1}) \times (1 - v_{i2}) \times \ldots \times (1 - v_{in_i}) \quad \textbf{(12)}$$

This implies that the probability p_i that $Agent_i$ will not fail is:

$$p_i = 1 - (1 - v_{i1}) \times (1 - v_{i2}) \times \ldots \times (1 - v_{in_i}) \quad \textbf{(13)}$$

Definition 9: Let g be a resource allocation function and c_i the criticality of $Agent_i$, a value aggregating the importance of the actions executed by $Agent_i$ (this value can be calculated using the method described in Section 3). Then, we define the *utility* of the multi-agent system (denoted as $u(g)$) as the total importance of the actions executed by it:

$$u(g) = c_1 + c_2 + \ldots + c_{n_a} \quad \textbf{(14)}$$

Definition 10: Let g be a resource allocation function, c_i the criticality of $Agent_i$ and p_i the probability that $Agent_i$ will not fail. The *expected utility* of the MAS deployed using g is defined as the expected value of the utility function. It can be calculated as follows:

$$E(u(g)) = c_1 \times p_1 + c_2 \times p_2 + \ldots + c_{n_a} \times p_{n_a} \quad \textbf{(15)}$$

The higher the value of $E(u(g))$, the more efficient and fault-tolerant the allocation function g.

Definition 11: We define the *resource allocation problem* as the optimization problem of finding the resource allocation function g_{max} which gives a maximum value for the expected utility of the MAS ($E(u(g_{max})$ is maximum).

4.1. Agent Replication Mechanism

Our agent replication mechanism tries to solve the resource allocation problem previously defined in a heuristic way. In our mechanism, let V be the sum of the reliabilities of all the resources of the system. Then, an agent $Agent_i$ is allowed to be deployed using a set of resources R_i whose sum of reliabilities (w_i) does not exceed a certain limit. This limit is proportional to the percentage of its criticality (c_i) with respect to the sum of agents' criticalities (C), as given by the equation:

$$w_i = \frac{c_i \times V}{C} \qquad (16)$$

Among the sets of resources R_i which satisfy the equation 16, we look for the one with the minimal probability of failure. For that, the system of replication will first sort the machines by decreasing reliability. For each resource available in each machine, we allocate it to the most critical agent $Agent_i$ who can have it (if the agent $Agent_i$ has not yet one resource in this machine and if the sum of reliability of the resources that have already been allocated to $Agent_i$ plus this one does not exceed its limit w_i).

One can apply the same possible strategies used as the agent criticality update policy (time-driven or event-driven) to decide when to re-calculate the values of w_i. For instance, one can use a variable window of time Δt for each agent $Agent_i$. If the quantity of resources (whose total value does not exceed w_i) that the agent $Agent_i$ can acquire does not change significantly, the window of time Δt can be increased, otherwise it is decremented. Another possibility is to recalculate the value of w_i whenever the value of c_i is updated.

5. Architecture and Implementation

To implement the agent replication mechanism described in the preceding section, we have extended the framework DARX [Marin et al. 2003]. We describe in this section this extension.

5.1. DARX

DARX (Dynamic Agent Replication eXtension) relies on the notion of replication group (RG). Every agent of the application is associated to an RG, which DARX handles in a way that renders replication transparent to the application at runtime. Each RG has exactly one ruler, which communicates with the other agents. Other RG members, referred to as subjects, are kept consistent with their ruler according to the replication strategies. Several different strategies, ranging from passive to active, may be applied within a replication group. The number of subjects and the replication strategy may be adapted dynamically.

DARX provides atomic and ordered multi-cast for the replication groups' internal communication. Messages between agents are also logged by each replica, and sequences of messages can be re-emitted for recovery purposes.

As shown in Figure 3, DARX offers several services. Failure detection enables to suspect host and process failures based on a hierarchy of adaptive failure detectors. Naming and localisation provides a means to supply agents and their replicas with unique identifiers throughout the system, and to retrieve their location whenever the application requires it.

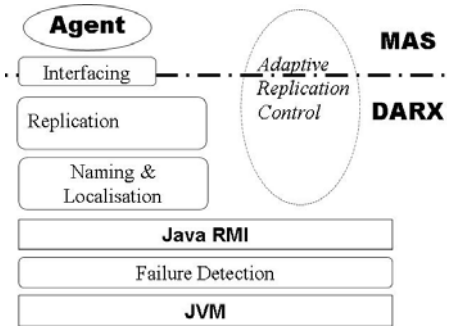

Figure 3. DARX framework architecture

DARX is coded in Java 1.4 and uses RMI as a means to simplify the coding of network issues. It can be easily integrated to any agent platform by means of an interfacing component. We implemented the proposed replication mechanism in an adaptive replication control module, which we have coupled to the DARX platform. This module is described in the next section.

5.2. Adaptive Replication Control Architecture

The adaptive replication control module, shown in Figure 4, is completely distributed and uses the replication service of DARX to provide a suitable replication scheme for every agent. We associate an agent monitor to each agent of the system and a machine manager to each machine.

The agent monitor receives the local plans of the monitored agent and is responsible for the calculation and update of its criticality. As we have seen in the section 3.1, the computation of the criticality of an agent may rely on the criticality of other agents (because of possible dependence between their tasks). Thus their respective monitoring agents need to communicate information.

Each machine manager contains a piece of the global information of the application, such as:

- The criticalities and reliabilities of the agents deployed in its machine (obtained from the monitoring agents, as shown by the arrows in Figure 4);

- The number of replication resources available in its machine;

- The failure rate of the machine.

Machine managers exchange messages with their local information in order to keep their vision about other machines up to date (total number of resources in the system, sum of the criticalities of all the agents, …) and, consequently, to make it viable the mechanism of replication described in section 4.1.

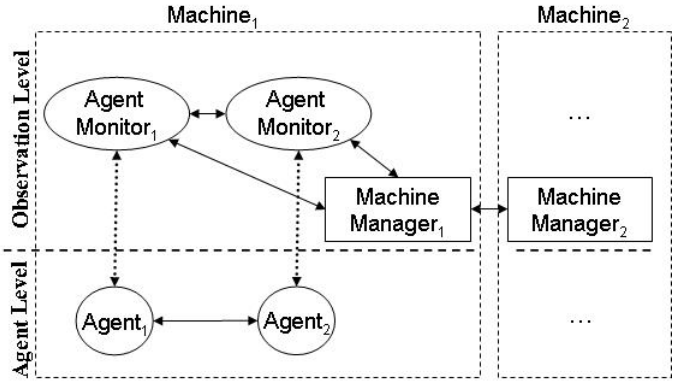

Figure 4. Architecture for replication control

6. Experimental Results

We are currently conducting experiments, whose preliminary results we summarize in this paper. In our experiments, each agent has to accomplish its own sequence of 5 plans, one at a time, each with 10 actions. The average duration of actions is of 2 seconds. We repeated ten times each experiment (the results shown are the mean of those several runs). We maintained the same sequence of plans and actions that each agent must execute in those runs. Whenever replication is present, the number of resources available at the machine is half of the number of agents.

In the first place, we ran each experiment considering a completely reliable environment (no failures) and calculated the CPU time (in milliseconds) required for the completion of all the plans by all the agents. Figure 5 shows the effect of changing the number of agents on the CPU time required (y-axis) by our replication mechanism and by the execution of the multi-agent system with no replication at all. Whenever replication is present, the number of replicas available at the machine is half of the number of agents and the strategy used is the passive one. One can notice that using no replication always outperform our replication mechanism, but the overhead of our mechanism is negligible (less than 4%).

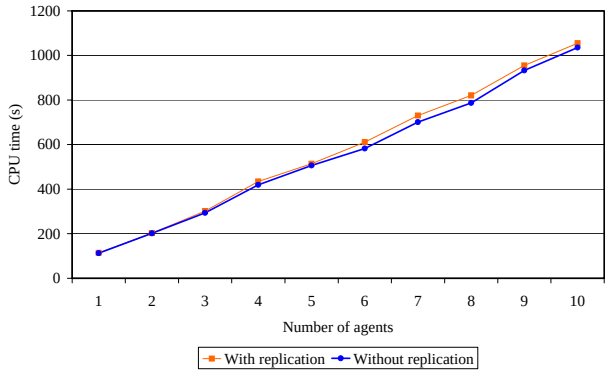

Figure 5. Quality of the replication mechanism used vs. a failure-free execution

In order to assess the quality of a replication mechanism, we considered the sum of the absolute criticalities of the actions which were executed with success. During the execution of each experiment, at each interval of 2s and for each agent, a failure generator will cause the agent to fail with a probability equal to its probability of failure

given by Equation 12. Whenever an agent fails (because all its resources failed), its current plan fails, the agent is restarted with its next plan and all the resources which were allocated to it are made available for use.

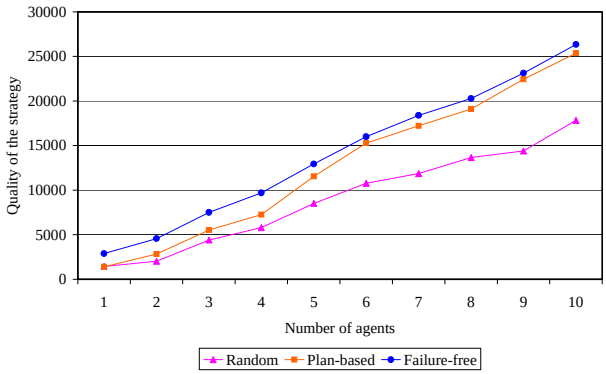

Figure 6. Quality of the replication mechanism used vs. a failure-free execution

Figure 6 shows the maximum quality that could be obtained (in a completely reliable environment) compared to the quality of our strategy of replication and to a random one, which allocates randomly each resource available. We varied the failure rate of the resources, but due to space constraints, only a fixed value of 10 failures per minute is reported in Figure 6. The results are encouraging in the sense that the quality of our mechanism is quite close (80% at average) to the maximum value that could be obtained in a failure-free execution. Additionally, our strategy is more accurate to determine and replicate the most critical agents as the random one. In fact, the probability that a critical agent fails with our strategy is lower than with a random strategy.

7. Conclusion

Large-scale multi-agent systems are often distributed and must run without any interruption. To make these systems reliable, we proposed an original predictive method to evaluate dynamically the criticality of agents. Our approach takes profit of the specificities of multi-agent applications and analyses the agents' plans to determine their importance to the system. This approach allows us to obtain a more precise value of the criticality and it takes into account the future behaviors of the agents. The agent criticality is then used to replicate agents in order to maximize their reliability and availability based on available resources.

The proposed mechanism was implemented over the DARX replication platform. We have tested it and we believe that our current results are promising. In fact, the algorithms have a negligible overhead and provide a satisfactory reliability.

One of the perspectives of this work is to refine the problem of fault tolerance in multi-agent systems and its evaluation measures, in order to compare the different proposed strategies with an optimal one. In that process, we are studying how to define and classify different variations for the expected global utility function. Additionally, we are currently adding other constraints to the resource allocation function such as allowing the definition of the minimum level of reliability required by the agents.

Additionally, we have implemented and we are currently testing other heuristic solutions to the resource allocation problem, such as branch and bound, greedy, hill climbing and tabu search algorithms.

Last but not least, we intend to run more large-scale experiments using the air traffic control application to validate our approach. This work is being co-funded by the following research programs: CAPES-COFECUB, CNPq, ANR Sécurité & Informatique and EuroControl CARE INO III.

References

Almeida, A. L., Aknine, S., Briot, J.P. and Malenfant, J. (2006) “A predictive method for providing fault tolerance in multi-agent systems”, In: Proc. of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2006), Hong Kong, p. 226-232.

Briot, J.-P., Guessoum, Z., Aknine, S., Almeida, A. L., Faci, N., Gatti, M., Lucena, C., Malenfant, J., Marin, O., Sens, P. (2006) “Experience and Prospects for Various Control Strategies for Self-Replicating Multi-Agent Systems”, In: Workshop on Software Engineering for Adaptive and Self-Managing Systems, Shangai, p. 37-43.

Cukier, M. et al (1998) “AQuA: an adaptive architecture that provides dependable distributed objects”, In: Proc. of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98), West Lafayette, Indiana, p. 245-253.

Favarim, F., Siqueira, F. and Fraga, J. S. (2003) “Adaptive fault-tolerant CORBA components”, In: Middleware Workshops 2003, p. 144-148.

Fedoruk, A. and Deters, R. (2002) “Improving fault-tolerance by replicating agents”, In: Proc. AAMAS02, Bologna, Italy, p. 737-744.

Guerraoui, R. and Schiper, A. (1997) “Software-based Replication for Fault Tolerance”, In: IEEE Computer, vol. 30, no. 4., p. 68-74.

Kalbarczyk, Z., Bagchi, S., Whisnant, K. and Iyer, R.K. (1999) “Chameleon: a software infrastructure for adaptive fault tolerance”, In: IEEE Transactions on Parallel and Distributed Systems, p. 560-579.

Kraus, S., Subrahmanian, V.S. and Cihan, N. (2003) “Probabilistically survivable MASs”, In: IJCAI-03, p. 789-795.

Marin, O., Bertier, M. and Sens, P. (2003) “DARX - a Framework for the Fault-Tolerant Support of Agent Software”, In: 14th IEEE International Symposium on Software Reliability Engineering (ISSRE’2003), Denver, USA, p. 406-417.

Romanovsky, A., Dony, C., Knudsen J.L. and Tripathi, A. (eds.) (2001), Advances in Exception Handling Techniques, LNCS 2022, Springer.

Seghrouchni, A. E. F. and Haddad, S. (1996) “A recursive model for distributed planning”, In: Proc. Of Second International Conference on Multi-Agent Systems (ICMAS-96), AAAI Press, p. 307-314.