Role of Replication Planning for Fault Tolerant Multiagent Systems

Samir Aknine, Olivier Marin

LIP6, Université Paris 6
8, rue du Capitaine Scott
75015 PARIS Cedex 15, France
Samir.Aknine@lip6.fr, Olivier.Marin@lip6.fr

Abstract. Fault tolerance is a research topic seldom addressed in multiagent systems up until recently. The increasing interest is justified by the growing importance of multiagent applications and the need for a higher quality of service in these systems. In this article, we tackle this problem by proposing an original method for fault tolerant multiagent systems through dynamic replication. Our method is different from other work addressing the same issue in the sense that our research focuses on building a viable, dynamic and evolutionary replication policy for agents. This policy is determined by taking into account different data, the most critical pieces of information resulting from the collective and individual behaviours of the agents in the application. The set of replication strategies applied at a given moment to an agent is fine tuned gradually by the replication system. For this purpose, it gathers information from every agent. The replication system also guarantees the dynamic evolution of the replication policy in order to fit to the properties of adaptation and dynamic evolution of the multiagent application.

1. Introduction

Today multiagent systems tend to support complex applications that require higher levels of dependability. In particular, fault tolerance is important because it provides a higher quality of service for these systems. Fault tolerance problems have been thoroughly studied for distributed systems in general. Recently the multiagent community has shown interest for this kind of problematic because of the fundamental issue of reliability within multiagent systems. On the other hand, the potential of the agent paradigm as a facilitator for the resolution of distributed problems has also aroused the interest of the fault tolerance community. Indeed, in general fault tolerance solutions focus on standard strategies defined for every software element before runtime. These strategies may not reflect adequately the requirements of the application as it is processed. This is all the truer in large-scale environments where system behaviour is very dynamic and varies greatly from one part of the network to another. Another issue linked to scalability and which most fault tolerance solutions fail to address is the impossibility of achieving a global view of the system. Replication strategies are obviously predefined in the solution we

propose for multiagent systems. However we move a step further with the ability to adapt strategy parameters, and even to swap strategies as a function both of evolving system characteristics and of agent activities in the application itself. These activities include the collective and individual behaviours of the agents that we will detail in this article. The suggested approach differs from preceding work in that we propose to build an adaptive replication scheme for the agents which satisfies three essential properties: viability, dynamicity and evolution.

Based on the multiagent application, the replication system extracts information that it uses to determine the right level of fault tolerance support it will grant to each agent of the application. This information makes it possible to compute a *criticity* for each agent. A *criticity* is a value associated to an agent in order to reflect the effects of its failure on the overall system. Needless to say criticity evolves over time. Since it is impossible to determine such a value quickly and simply, we propose in this paper to approach it by analysing the behaviour of the agent with respect to the rest of the application : the collective and individual actions of the agent, actions to come, its commitments, etc. Knowing that a multiagent system is evolutionary and that it is impossible to predict in advance all the actions of the agents — since these are decided dynamically and could be modified at any time —, it seems that computing this criticity should take into account the dynamics imposed by the agents themselves. The solution we developed makes it possible to define a temporal criticity map for each agent. This map is computed dynamically and gradually by the replication system on short intervals of the execution cycle of each agent since some of the actions of the agents may be known while others remain uncertain or even unknown.

This paper is organised as follows. Section 2 introduces the problem for fault-tolerant multiagent systems we are interested in. Section 3 presents our solution for assessing agent criticity. We first introduce an example describing the behaviours of agents during the collective resolution of a problem; this example enables us thereafter to illustrate our approach. Then we introduce the concepts of our approach and detail the replication method we propose and explain how the replication strategy is defined for the agents. Section 4 describes the architecture of the middleware developed by our team for fault tolerant agents. Section 5 compares our approach with related work. Section 6 concludes with a quick summary of our approach and its advantages, and with our future research.

2. Problem description

The fault tolerance problem described in this paper considers a set of agents $A = \{ a_1, a_2, a_3 \ldots a_n \}$ that have to complete a set of tasks. Tasks consist of a set T such that $T = \{ T_1, T_2, T_3 \ldots T_m \}$. The envisaged replication strategies range from highly efficient active strategies that are very costly both in time and resources to cheaper but less efficient passive strategies. The multiagent system has limited resources: only the most critical agents should be replicated agressively, whereas less crucial agents may use more optimistic strategies and some others none at all. The scheme applied to an agent will depend on its criticity. The problem consists in finding a replication scheme which allows a better reliability for the overall system along its life cycle.

This scheme must also be revised over time, considering that the multiagent execution context of tasks is dynamic and does not allow to make predictions on how tasks will be carried out, by which agents and when. These predictions are difficult to make since decisions concerning tasks will be made by the agents and become known to the replication system only at runtime.

3. Our criticity assessment method

Before detailing the replication scheme for multiagent systems we propose in this article, we will first present an application example that illustrates a multiagent problem solving method. Then we will explain the different steps of our method for assessing agent *criticities* in this application. We will also give the basic concepts which underlie this mechanism.

3.1. Problem illustration

While preserving the original constraints, the basic assumptions for a collective resolution of a problem in the multiagent approach, and the genericity of the proposed method, we have voluntarily chosen a simple application example to illustrate, in our method, a collective resolution of a problem with agents. The application example presented below describes well the fundamental aspects of our approach. It concerns agents behaving in a world of blocks.
In this example, three agents a_1, a_2 and a_3 behave in a common world which contains six blocks (A_k, B_k) with $k=1..3$. Blocks with the same index share the same colour and texture. Initially, all blocks are in a given position. At the end of their execution, agents must have stacked the blocks in a predefined order and at another position such as that described in Figure 1. Each agent a_i is responsible for its two blocks (A_k, B_k) such as $i=k$. To solve their problem, the agents have a set of operators for domain actions and others for control actions such as those for negotiation, group formation, etc. These operators are given below with their semantics:

- *Push* ($<a_i> ; x$) is the action agent a_i executes in order to push block x from one position to another in the world.
- *Unstack* ($<a_i, a_j> ; x ; y$) is the action agents a_i and a_j execute in order to unstack block x from block y.
- *Stack* ($<a_i, a_j> ; x ; y$) is the action agents a_i and a_j execute so as to stack block x on block y.

Amongst control actions, we give below only those which help understanding the approach we propose:

- *Negotiate* ($<a_i> ; S ; X ; C$) is the control action agent a_i executes so as to negotiate a domain action X, for instance *Push,* with the agents in set S such that : $S \subset A$. C is the set of constraints.
- *Group* ($<a_i> ; S ; C$) is the control action agent a_i executes so as to form a group with the agents in set S such that : $S \subset A$. C is the set of constraints.

- *Inform* (<a_i> ; *S* ; *M*) is the control action agent a_i executes in order to notify message *M* to the agents in *S*.

These operators can be divided into three sets. The operators which allow to carry out individual tasks, for example *Push*, the operators for collective tasks, for example *Stack*, and the operators which are useful for interactions, for example *Inform*.

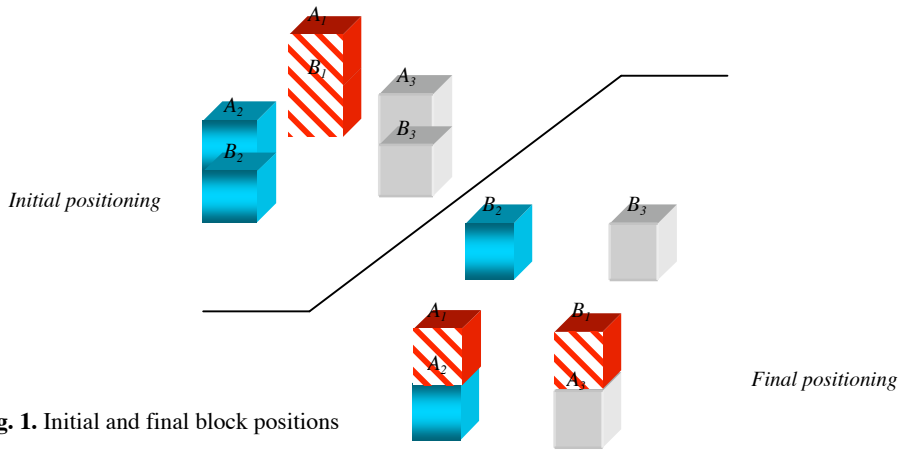

Fig. 1. Initial and final block positions

3.2. Multiagent resolution

Our replication mechanism, described in Section 4, is completely transparent to the multiagent problem solving method. The resolution of the problem must remain completely independent from the way agents will be replicated. As we said before, the replication system needs information on the actions of the agents in order to determine the corresponding schemes. On the previous example, the multiagent resolution can be carried out using different methods. In the following, we describe one resolution method which only aims at illustrating our approach.

Using each initial and final state described below, each agent initially defines a partial plan of actions which enables it to reach its final state. In this plan, each agent specifies, in a partial order, the individual and collective actions to carry out on the blocks it is responsible for. Using the operators given in section 3.1, agent a_1 defines the first plan presented in Figure 2. This plan, obviously a simplified example, can be obtained either by a regression method starting from the goals of the agent or by an exploration method. Several plans could be obtained after this step. The example given in Figure 2 indicates that, since it is responsible for blocks A_1 and B_1, agent a_1 must initially negotiate the *unstack* action on block A_1 with a set of agents denoted here < ? >. This means that the set of agents remains unspecified and will be decided at the time of the negotiation. It must then apply the *unstack* action on the block, followed by a negotiation for *unstack* actions on blocks A_2 and A_3. Once block A_1 is unstacked, a_1 can then push both block A_1 and block B_1 towards their right positions. Hereafter, the agent is free negotiate the *unstack* actions on both blocks A_2 and A_3 in whichever sequence fits best in the application progress. It is then possible to negotiate the stacking of blocks A_1 et B_1 in their final positions, and the current plan

finishes.
In the same way, agent a_2 and a_3 build a first plan of actions to perform in order to achieve their own goals. While agents execute their plans, they can refine these plans, update some of their actions or even re-plan other actions in case the conditions under which these plans were built have evolved. Through the example, we notice that all agent behaviours belong to either of the three categories mentioned earlier: individual, collective interactive. We base our approach on these different types of agent behaviours.

3.3. Replication planning design

Using the previous example, we will explain the behaviour of the replication planning system (RPS). This fault tolerance planning system receives the partial plan of every agent at the end of the first phase of planning. Coinciding plans will be used to draw a first temporal criticity map for every agent. This temporal map defines the criticity evolution of an agent in the multiagent system. The criticity of an agent strives to measure the consequences of the failure of this agent on the rest of the multiagent system. It is a function that takes two parameters: one for the actions the agent achieves and another for its dependencies with other agents of the system. All parameters are computed using the agent plans. It is on the basis of agent *criticities* that the RPS later defines the scheme it will associate to a precise state of any agent.
Nevertheless, it is essential that an agent criticity does not vary too frequently for fear of impeding significantly on the performances of the whole system. Indeed altering the replication scheme involves overhead that may become strainful for the system. It is thus necessary that criticity changes be as infrequent as possible. For this reason, in our approach, the computation of the criticity of an agent is not performed at random points in the execution of an agent but on consequent execution intervals in the life cycle of each agent. For a better understanding as to the construction of this criticity map, we will first introduce some necessary parameters for the computation of an agent criticity.

3.3.1. Parameters for agent criticity computation

We should first remind that, in the context of multiagent systems, guaranteeing dependability for a single agent regardless of the other ones is pointless. Agents often work collectively and towards the same objectives. This implies that improving the fault tolerance for one agent depends not only on the replication scheme which is applied to it, but also on the scheme decided for the rest of the agents with which it has relationships. Some parameters related to this purpose follow.

- Criticity of an agent-planned action. Basically, the criticity of an agent at a given state is computed based on the set of actions which the agent carries out, or which it plans to carry out. The absolute criticity of an action is the importance of the action for the multiagent system. It is a value fixed *a priori* and invariable in time. It can be defined according to the length of the action, the number of agents intervening in its

execution, the sensitivity of the action, etc. In order to compute the relative criticity of an action, we distinguish two types of actions.

- Criticity of an individual action. An individual action is an action that an agent planned alone and will also carry out alone. The relative criticity of this action is a function of its absolute criticity but also of the consequences of a failure in the execution of this action on the rest of the actions of the other agents depending on it.

- Criticity of a collective action. A collective action is an action that an agent planned alone but will carry out with other agents of the system once the cooperation agreement is obtained from them. This agreement is established through the formation of groups or coalitions; thus there is a cost for such formations, and a cost for the collective execution of actions. The relative criticity of a collective action planned by an agent must thus take into account the resources consumed for the group/coalition formation, as well as the consequences a commitment breach would have. The absolute criticity of the action should also be considered.

- Criticity of an individual agent commitment. An individual commitment is an action that an agent will carry out alone for the others. The criticity of an individual commitment of an agent is a function of the absolute criticity of the associated action, the consequence of the failure in the execution of this action with respect both to the other actions of the agent and to the actions of other agents concerned with this commitment.

- Criticity of a collective agent commitment. A collective commitment is an action that an agent committed to carrying out with others. This action has been proposed by another agent or more. The computation of its criticity is done on the same bases as for the computation of a collective action.

- Time to recovery. The time to recovery from failure at a given state of the execution is a function of the time to failure detection, and also of the time needed for solving the failure. This time is variable since it depends on the system state when the failure occurs, and on the replication scheme that is applied at the time of the failure.

3.3.2. Building the temporal criticity map for agents

In order to build the temporal map of the criticity of an agent, the replication system uses the information obtained from the agent itself. However the system needs to focus on defining a coherent replication map for the overall multiagent system where each agent will be considered. Once information on the agent actions is obtained, the replication system starts constructing replication schemes for every agent using the partial plans they give. All the plans corresponding to our example are detailed in Figure 2. Based on the above definitions for parameters intervening in the criticity computation, the RPS starts with analysing the partial plans. From these plans, the RPS recognises the individual actions planned by each agent, as well as the collective actions and interactions alongside their temporal order, partial or total. As mentioned

above, it is necessary to detect collective actions since on the time interval where those are carried out, all agents involved must be replicated harmoniously. Indeed, if one of the participating agents fails without any possibility of recovery, the collective action may fail too. As an example, the *unstack* action creates a relationship between two agents which will carry it out. Both agents are equally critical for this action since both of them intervene collectively in it. The RPS explores the partial plans with an algorithm for graph exploration and unfolds the actions in these plans and raise any imprecision. Typically, an imprecision may have to be cleared as to whom an agent will cooperate with for a collective action.

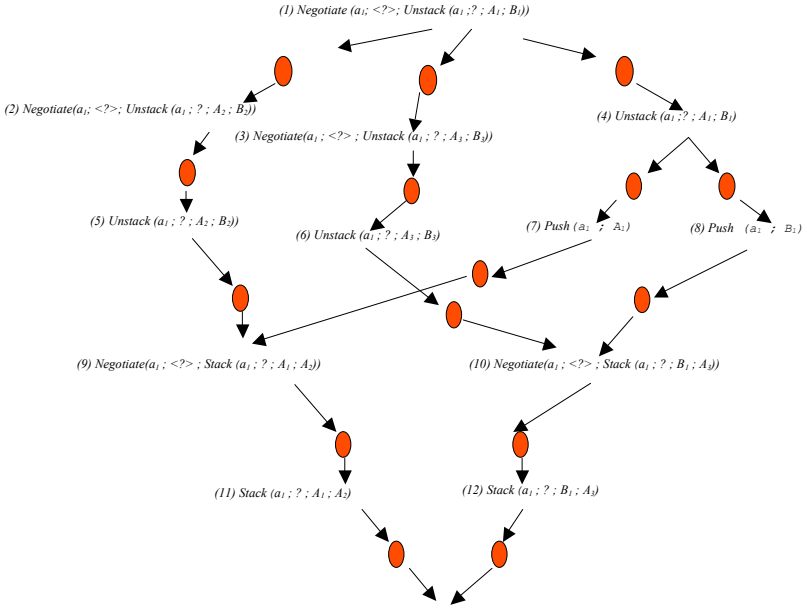

Fig. 2. Partial plan of agent a_1

The RPS then interweaves the partial plans. This step enables it to recognise the dynamic relationships created between agents through their individual and collective commitments, their collective actions, etc. The RPS tries to use all the information provided in the initial partial plans. If some partial plans have not been submitted yet the RPS will rely on the available ones and define an overly pessimistic temporal criticity map, designed to guarantee a safe level of fault tolerance for the multiagent application despite the lack of information. In this case, not all of the previously presented criticity computation parameters (cf. section 3.3.1) will be taken into account. The resulting map will be improved gradually as the partial plans unfold. From the plan of agent a_1, the RPS traces the partial criticity map defined in Figure 3.
This map reproduces information from the plan of a_1; each action is represented on this map with its corresponding number in the plan. The RPS associates a criticity to every action of the plan. For instance, we can observe that the criticity defined for the *unstack* action is more significant than that of the *negotiate* action. This map enables to define, thereafter, the amount of allocated resources as well as an adequate replication scheme for every agent on the considered time interval. Nevertheless it is

necessary that the changes in the replication scheme be as infrequent as possible: such changes imply additional processing, which augment the computational complexity of the overall system. Eventually the complexity may become so important as to make the processing times involved prohibitive, and hence bring the whole system to collapse.

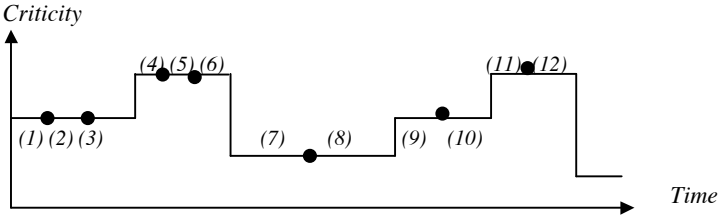

Fig. 3. Partial criticity map for agent a_1

Consequently it is necessary for the RPS to smooth down the criticity map of each agent in order to avoid repetitive replication scheme switching on short intervals of time. This smoothing down of the map is computed according to the type of actions the agent will perform. For each agent state, the RPS increases or decreases the criticity of agent actions in order to make it correspond to that of their succeeding or preceding actions in the map. In this manner, the imperfections due to the timely imprecision on the actions of the partial plans will be increasingly reduced.
To perform this smoothing down, the RPS can use several metrics, for instance the relationship between the criticity of the action of the agent and the length of its actions. Thus on the first segment of the map of criticity of the agent a_1 (cf. Figure 4), we have two successive intervals Δt_1 and Δt_2 with two different criticity degrees, respectively C_1 and C_2, for agent a_1. For this criticity value alteration to remain, both sub-intervals Δt_1 and Δt_2 need to be sufficiently significant, i.e., higher than a reference interval Δt. When this property is not satisfied, the RPS simplifies the map by smoothing down one of the two intervals. For instance, in our system the only state remaining will be the one with the highest value for $\Delta t_i * C_i$.

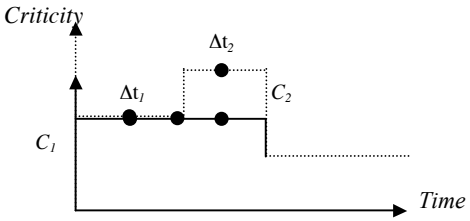

Fig. 4. Fitting of the criticity map

It is thus necessary to compare the fault tolerance provided for each agent, in each state, to that of all other agents. In this new approach, the RPS seeks the relationships between the different agents, deduced from the partial plans. In order to determine these relationships, the system uses a parallel exploration algorithm on the plans. On each plan, it seeks invocations of other agents through collective tasks and

commitments. It then builds links between the plans sharing sequences of actions or simply dependence relationships between the agents. Interacting agents must be analysed as a whole since they must be made fault tolerant in a consistent way. Indeed, as mentioned before the failure of an agent may ultimately jeopardise the collective task or the link it was involved in.

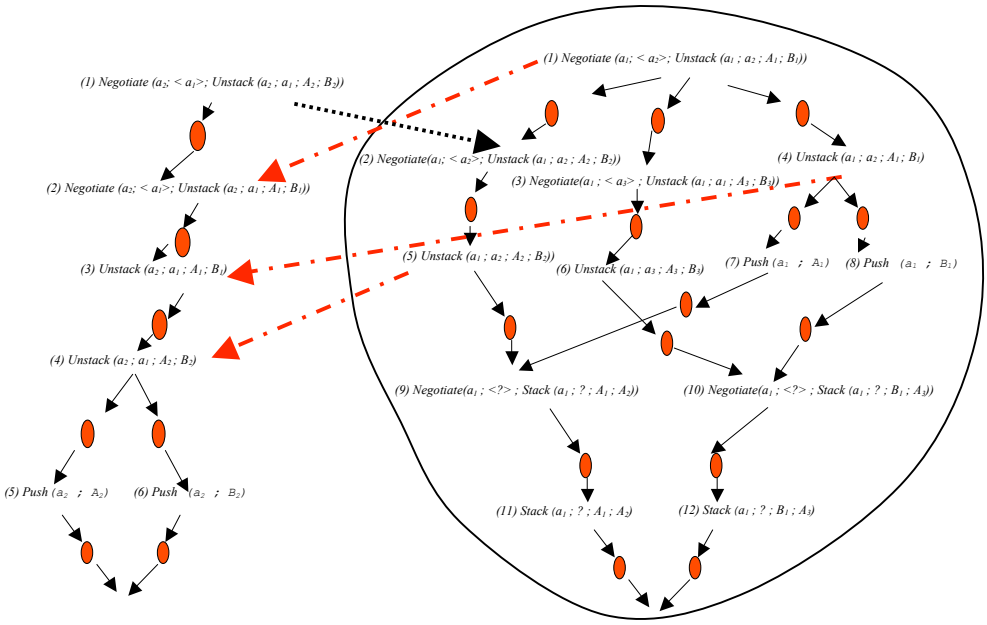

Fig. 5. Fusion of the partial plans for agents a_1, and a_2

In our example, a parallel plans exploration gives us the results described by Figure 5. This figure shows that after the first step of negotiation between the three agents, a_1 and a_2 agreed to unstack blocks A_1 and A_2. As for the block A_3, it will be unstacked by agents a_1 and a_3. Once these blocks are rightly positioned, the negotiations over successive stacks of the blocks in their final positions will be performed.

In order to build temporal criticity map of the whole multiagent system, the RPS initially computes the criticity map of each agent. Once these maps are smoothed down, the RPS compares them.

Figure 6 proposes a simplified representation on a time interval of the criticity maps after their smoothing down. The maps of agents a_1, a_2 and a_3 are obtained from example 1 on the basis of Figure 6. The comparison of the criticity maps leads to the resolution of a linear program in order to define the replication scheme to be applied to each agent. In a temporal segment of the criticity map, the replication scheme applied to an agent is determined by a function whose parameters are: the criticity associated to it, the status of available resources in the local network vicinity, and the quality of service required by the agent. We will not develop this part of work in this paper since it concerns another, quite different aspect of the problem we are addressing here.

The map comparison gives a solution on a given time interval. The global temporal map is regularly updated so as to take in the changes of plans by agents according to their new actions; as fits the role of the RPS.

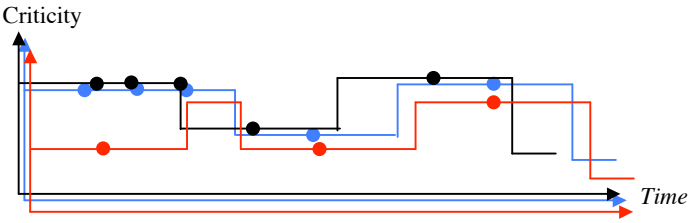

Fig. 6. Crossing the temporal criticity maps of agents a_1, a_2 and a_3

4. DARX: middleware & framework

As shown in Figure 7, DARX is a framework combined with a middleware comprising several services such as failure detection and monitoring, all of which are intended for the support of an adaptive replication mechanism designed for agents over a large-scale network [1]. **Failure detection** enables to suspect host and process failures; without it there is no way to decide whether an agent is still alive and running in an environment that is not fully synchronous [12]. **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. **System-level observation** extracts data from every host, gathers it and uses it to process and then diffuse information concerning system behaviour and its evolution; an example of valuable information is the average MTBF (mean time between failures) for a random set of η hosts over a period τ. The **application analysis** service monitors the supported agent applications and assesses the requirements of every agent in terms of fault tolerance; the present paper focuses on this issue. The **replication** service uses the information, provided by both the application analysis and the system-level observation, to assess and enforce a suitable replication scheme for every agent. DARX is coded in Java 1.4 and uses its RMI feature as a means to simplify the coding of network issues.

Another, even more schematic representation for DARX is composed of three layers:

1. At the top, an **application** layer assesses the overall criticity of every agent dynamically.
2. A **system** layer at the bottom evaluates which resources (network bandwidth, host CPU, and so on...) are available, and for how long. It also enforces replication policies once they are determined.
3. In between them, a **strategy** layer takes input both from the application and from the system layer in order to determine the best replication scheme available (ie., the scheme that is closest to what the application requires, given the available resources and replication strategies.)

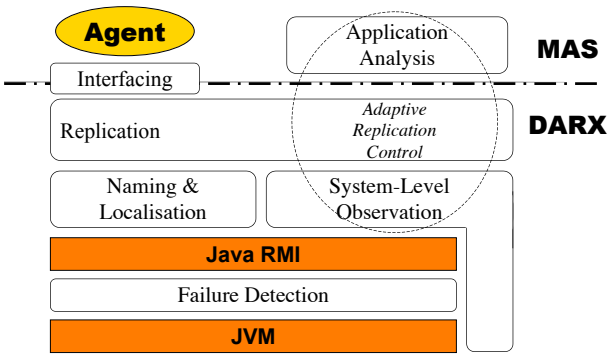

Fig. 7. DARX framework design

Technically speaking, DARX relies on the notion of replication group (RG). Every agent of the application has an RG associated to it, which DARX handles in a way that renders replication transparent to the application at runtime. An RG contains at least one replica, and possesses a unique *ruler*. Other RG members, referred to as *subjects*, are kept consistent with their *ruler* by applying various replication strategies. Several different strategies, ranging from passive to active, may be applied within a same RG; the set of RG members combined with the set of strategies applied inside the RG constitute its replication policy. An RG *ruler* gathers information both from the multi-agent application and from the underlying operating system in order to evaluate the correctness of the current replication policy within its RG. If it appears inadequate, then the *ruler* determines a new policy and applies it to its RG.

5 Related Work and discussion

Several approaches have addressed the multi-faceted problem of fault tolerance in multiagent systems. This research domain can be divided in two main categories. A first research subdomain focuses on the reliability of an agent within a multiagent system. This approach handles the serious problems of communication, interaction, and coordination of the agent and possibly of its replicas with the other agents of the multiagent system. The second research subdomain addresses the difficulties of making agents dependable, mobile agents in particular as they are more likely confronted to security problems as they move through insecure places [5][10]. This work is beyond the scope of this paper.

Hagg introduced the use of sentinels in multiagent systems so as to keep agents from reaching undesirable states [4]. Sentinels represent the control structure of the multiagent system. They need to build models for every agent which perform functionalities and monitor communications in order to react to agent failures. Each sentinel is associated to one functionality of the multiagent system by the designer of the multiagent system. This sentinel handles the different interacting agents which cooperate in order to achieve the functionality. The analysis of the beliefs of every agent it monitors enables a sentinel to detect a failure when it occurs. Adding

sentinels to multiagent systems seems to be a good approach, however the sentinels themselves represent failure points for the multiagent system. A similar approach is that of [7]. They presented a fault tolerant multiagent architecture that regroups agents and brokers. They have specifically addressed the problem of recovering from its broker failures.
To overcome such problems, [2] proposes to use proxies. This approach tries to make the use of agent replication transparent, i.e. enable agent replicas to represent their group with respect to the other agents of the system; other agents will not be made aware that they are interacting with a group of replicas. The proxy manages the state of the replicas. To do so, 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, for instance, a hierarchy of proxies for each group of replicas. They point-out the specific problems of read/write consistency, resource locking also discussed in [11]. [6] proposes a probabilistic approach for solving the problem of agents deployment in a multiagent application. Our approach is different from this one in the sense that the replication scheme we propose takes into account not only several parameters related to the activity of the agents but also the kind of replication to adopt (active, passive, mixed, etc.).
The previous replication approach proposed within the DARX framework is an event-driven approach [8]. It is essentially based on the concept of role, which is associated to given states of an agent. The occurrence of a change of role, possibly announced by the agent itself to the RPS or detected by means of its observation service, implies the computation of a new criticity and then the assessment of the replication scheme to be applied. Although it allows for a better optimisation, this method makes it hard to take into account situations that have not been forethought. In our new approach, we propose a method where replication schemes are determined quite objectively. Determination of a replication scheme is based on the partial plans of agents: their actions and their tasks are defined on temporal intervals. These intervals vary according to the activity of the agents and the frequency of changes made on the initial previsions of these agents. To solve the replication problem in a multiagent context, our method belongs to preventive approaches. It recommends analysing the actions planned by the agents in more or less short intervals and then providing this information to a RPS, which can determine a replication scheme following these plans.
As agents are autonomous systems behaving in large systems, and as it is insufficient to make one agent fault-tolerant independently from the rest of the agents with which it shares some actions, our method draws a global fault tolerance scheme which adapts itself for every entity of the overall multiagent system.

6 Conclusion

In recent years, research on multiagent systems has addressed the problem of agent reliability within these systems. Most proposed works were based on previous methods proposed in distributed systems: using proxies, sentinels, or algorithms for

the deployment of agents on different sites. In this work we propose an original method for this problem. Our approach pays much attention to the specificities of multiagent applications. Hence within our method we distinguish the different agents behaviours, varying between collective, individual, interaction, and so on. Every behaviour is integrated in our method as a parameter to be used for evaluating the criticity of agents.

The notion of criticity is particularly important. Even in our targeted environment, large-scale systems, agents can only rely on limited amounts of locally available resources. Moreover system characteristics are not only very different from one part of the system to another, but they are also subject to important variations. It is therefore impossible to determine a fixed scheme for every possible situation, as the number of situations is virtually infinite. Hence it is crucial to scale the problem down to a single variable which allows to compare agents globally.

This research has been undergone as a joint venture between two research groups working respectively on distributed systems and multiagent systems. We believe the coupling of both perspectives has allowed for a deeper understanding of the problem at hand, and for a sound resolution of all the adjoining issues. Our work combines proven methods from both domains: namely dynamic collective resolution on one hand, and replication on the other. The result is an original method which welds effective solutions from both domains.

One aspect that remains to be thoroughly explored is the decentralisation of the criticity assessment. As mentioned before, there is no way to build an overview of the whole system in a large-scale environment. Hence, our scalability issue prohibits the definition of a coherent replication map for the overall multiagent. We are currently working on the construction of views covering subsets of the agent application so as to enable a consistent use of resources available in vicinity-based locations.

References

[1] M. Bertier, O. Marin, P. Sens, J-P.Implementation and Performance Evaluation of an Adaptable Failure Detector. DSN, 354-363, 2002.

[2] A. Fedoruk and R. Deters. Improving Fault-Tolerance by Replicating Agents. First International Joint Conference on Autonomous Agents and Multiagent Systems. July 15-19, Bologna, Italy, 2002.

[3] S. Hagg. A Sentinel Approach to Fault Handling in Multiagent Systems, Second Australian Workshop on Distributed AI, Cairns, Australia, 1996.

[4] D. Johansen, K. Marzullo, F. B. Schneider, K. Jacobsen, and D. Zagorodnov. NAP: Practical fault-tolerance for itinerant computations. 19th IEEE International Conference on Distributed Computing Systems (ICDCS), Austin, Texas, June 1999.

[5] Z. Kalbarczyk, R. K. Iyer, S. Bagchi, K. Whisnant, Chameleon: A Software Infrastructure for Adaptive Fault Tolerance, IEEE Transactions on Parallel and Distributed Systems, Vol. 10, no.6, pp.560-579, 1999

[6] S. Kraus, V.S. Subrahmanian and N. Cihan Tacs. Probabilistically Survivable MASs, Proc. IJCAI-03.

[7] S. Kumar, R. Philip, H. Levesque. The Adaptive Agent Architecture: Achieving Fault-Tolerance Using Persistent Broker Teams. ICMAS, July 7-12, 2000.

[8] Z. Guessoum, J.-P. Briot, O. Marin, A. Hamel and P. Sens. Dynamic and Adaptive Replication for Large-Scale Reliable Multi-Agent Systems. In Software Engineering for Large-Scale Multi-Agent Systems (SELMAS), LNCS 2603, pp. 182-198, April 2003.

[9] A. Mohindra, A. Purakayastha, and P. Thati. Exploiting nondeterminism for reliability of

mobile agent systems. International Conference on Dependable Systems and Networks, 144–153, New York, June 2000.
[10] S. Pleisch and A.Schiper. Fatomas - A fault-tolerant mobile agent system based on the agent-dependent approach. DSN, 2001.
[11] L. Silva, V. Batista, and J. Silva. Fault-tolerant execution of mobile agents. International Conference on Dependable Systems and Networks, 135–143, 2000.
[12] M. Fischer, N. Lynch, and M. Patterson. Impossibility of distributed consensus with one faulty process. JACM, 32(2):374--382, 1985.