Experience and Prospects for Various Control Strategies for Self-Replicating Multi-Agent Systems

J.-P. Briot[1,2]
Z. Guessoum[1]

Jean-Pierre.Briot@lip6.fr et al.
[1] LIP6, Paris, France

S. Aknine[1]
A.L. Almeida[1]
N. Faci[3]
M. Gatti[2]
[2] LES, PUC-Rio, Brazil

C. Lucena[2]
J. Malenfant[1]
O. Marin[1]
P. Sens[1]
[3] CReSTIC, Reims, France

ABSTRACT

Distributed cooperative applications (e.g., e-commerce) are now increasingly being designed as a set of autonomous entities, named agents, which interact and coordinate (thus named a multi-agent system). Such applications are often very dynamic: new agents can join or leave, they can change roles, strategies, etc. This high dynamicity creates new challenges to the traditional approaches of fault-tolerance. As relative importance of agents may evolve during the course of computation and problem solving, we need to dynamically and automatically identify the most critical agents and to adapt their replication strategies (e.g., active or passive, number of replicas), in order to maximize their reliability and their availability. One important issue is then: what kind of information could be used to estimate which agents are most critical agents? In this paper, we will first introduce our prototype architecture for adaptive replication. Then, we will discuss various kinds of information and strategies to estimate criticality of agents: static dependences, dynamic dependences, roles, norms, and plans. Some preliminary measurements and future directions will also be presented.

Categories and Subject Descriptors

B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance

General Terms

Reliability

Keywords

Agent, multi-agent system, dependability, fault-tolerance, control, adaptive, replication, criticality, estimation, strategy, dependence, role, norm, plan.

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), notably based on the concept of replication, which have been applied e.g. to data bases. As implied by [11], software replication in distributed environments has significant advantages over other fault-tolerance solutions. First and foremost, it provides the groundwork for the shortest recovery delays. Also, generally it is less intrusive with respect to execution time. Finally, it scales much better.

But these techniques are in general applied explicitly and statically, at design time. Thus, it is the responsibility of the designer of the application to identify explicitly which critical components should be made robust and also to decide what strategies (e.g., active or passive replication) and their configurations (how many replicas, their placement, etc.).

Meanwhile, new cooperative applications, e.g., e-commerce, air traffic control, crisis management systems, ambient intelligence, increasingly designed as multi-agent systems (MAS), are much more dynamic. In such applications, the roles and relative importance of the agents can greatly vary during the course of computation, of interaction and of cooperation, because the agents may change roles, plans and strategies. Also, new agents may join or leave the application (as an open system). It is thus very difficult, or even impossible, to identify in advance the most critical software components of the application.

Such new challenges reach the limits of traditional static approaches of replication, and motivate the study of adaptive replication mechanisms. One key issue is then the identification of the most critical components (agents) of the application at a certain time. Therefore, we consider using various levels of information: system level, e.g., communication load, and application/agent level, e.g., roles or plans, to estimate criticality. This paper will report on our past and current experiments using various types of informations, notably references, communications, roles, plans, and norms.

2. CONTEXT OF THIS WORK

2.1 Model of Failure Considered

Any software/hardware component may be subject to faults resulting in output errors, which can lead to a deviation of its specified behaviour, i.e. a *failure*. In distributed systems, and even more so in scalable environments, failures are unavoidable. A subdomain of reliability, fault-tolerance aims at allowing a system to survive in spite of faults, i.e. *after* a fault has occurred, by means of redundancy in either hardware or software architectures.

In this work, we consider crash type of failures, that is when a component stops producing output. It is the simplest type of failure to contend with. However, in various cases our solution allows to deal with other types of failures (omission, timing, byzantine). They are currently being investigated, but will not be considered in this paper.

2.2 Types of Techniques Considered

Replication is an effective way to achieve fault-tolerance for crash types of failures. A replicated software component has representations (replicas) on two or more hosts [11]. 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.

Active replication strategies provide fast recovery but lead to a high overhead. Passive replication minimizes processor use by activating redundant replicas only in case of failures. Then a new replica is elected among the set of passive ones and the execution is restarted from the last saved state. This technique requires less CPU resources than the active strategy but it needs an expensive checkpoint management.

2.3 Limitations of Current Replication Techniques and Toolkits

Many toolkits (e.g., [10]) 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 in some toolkits the replication 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.

3. PRINCIPLES OF OUR APPROACH FOR DYNAMIC REPLICATION

To overcome the limitations of standard way of replication where decision is fixed by the configuration at design time, we propose an approach with automatic and dynamic control of replication.

At first, we need a replication architecture which allows dynamic replication and dynamic adaptation of the replication policy (e.g., passive to active, changing the number of replicas). As discussed in Section 2.3, current replication toolkits rarely support such dynamicity. Therefore, we designed a novel replication framework, named DarX, with such dynamic features.

3.1 DarX: A Framework for Dynamic Replication

DarX is a framework for designing reliable distributed applications based on adaptive replication. Each agent can be replicated an unlimited number of times, with different replication strategies (main ones are: passive and active). A novel feature is the reification of the replication strategy, so that it may be dynamically changed. DarX includes group membership management to dynamically add or remove replicas. It also provides atomic and ordered multi-cast for the replication groups' internal communication. Messages between agents, that is communication external to the group, are also logged by each replica, and sequences of messages can be re-emitted for recovery purposes. DarX also includes an original failure detection service, based on a hierarchy of adaptive failure detectors [2]. DarX was designed to easily integrate various agent architectures, and the mechanisms that ensure dependability are kept as transparent as possible to the application. See e.g., [22, 21] for further details on DarX.

3.2 Need for Automatic and Adaptive Control

Provided the architecture for dynamic replication, we need a control mechanism for deciding which agent should be replicated and with what strategy (active or passive, how many replicas, where to create the replicas, etc.).[1] For dynamic applications,[2] a manual control is not realistic, as the application designer cannot monitor the evolution of a distributed cooperative application of a significant scale. Therefore, the control mechanism should be automatic, although it may use some information as provided by the designer of the application.

3.3 A Simple Scenario

As a simple example of scenario, let us consider a distributed multi-agent system that helps at scheduling meetings. Each user owns a personal assistant agent which manages his calendar. This agent interacts with: the user to receive his meeting requests and the associated information (a title, a description, possible dates, participants, priority, etc.) ; the other agents of the system to schedule meetings, based on preferences of its human owner.

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

3.4 Notion of Criticality

The control mechanism will estimate the most critical agents of the application and this information will be regularly updated. Here we may informally define the *criticality* of an agent as follows: the criticality of an agent, relative to an organization of agents it belongs to, is the measure of the potential impact of the failure of that individual agent on the failure of the organization. In the following, we consider criticality of an agent as a numerical value within the

[1]Here, we only discuss the decision about which agents to replicate and with how many replicas. Other issues are addressed elsewhere, e.g., where to create the replicas in [14].

[2]For multi-agent applications which are very static (fixed organization, fixed behaviors, etc., and with a small number of agents), the most critical agents may be identified by the application designer at design time. Thus, replication may be decided at configuration time, as for traditional replication techniques.

interval [0 1]. Various strategies to estimate the criticality of an agent are discussed in Section 4.

3.5 Replication Control

Once we have a strategy for computing (estimating) the criticality of each agent, we may compute the number of replicas nb_i of an agent as follows:

$$nb_i = rounded(rm + w_i * Rm/W)$$

- w_i: the criticality of the agent,
- W: the sum of the domain agents' criticalities,
- rm: the minimum number of replicas,
- Rm: the available resources, i.e., the maximum number of replicas.

The numbers of replicas is then used by DarX to control and update replication for each agent.

4. ESTIMATING THE CRITICALITY

In order to estimate the criticality of an agent, the issues are: What kind of information will be pertinent ? And how can we obtain it ? (explicitly stated by the application designer, inferred by external observation, e.g., amount of messages exchanged, or by internal observation, e.g., plans of an agent, etc.). We describe below various strategies: some are completely general and use basic information (references, messages), some make some assumption of higher-level abstractions (performatives, roles, plans, norms) which may or not be supported by a given multi-agent architecture.

4.1 Static Dependences

The first strategy that we studied is based on the concept of dependence (between agents). Intuitively, the more an agent has other agents depending on it, the more it is critical in the organization. Interdependence graphs [6] were introduced as a way to specify interdependences between agents. But as we want control to be as much automatic as possible, we would like to estimate and infer such dependences. A first estimation of dependences may be done statically by using message sending instructions from the code of agents.

Starting from the code of the whole multi-agent system, we automatically extract message sending instructions. In current implementation, we made the assumption that the behavior of an agent is structured through <condition,action> transition rules.[3] Thus, the code extractor can use the causality information about condition (message reception) and action (message sending).

Then, an algorithm automatically computes the graph of communication dependences. An example of resulting communication dependences graph is shown at the left side of Figure 1, where each node is a communication expression ($x \rightarrow y$ means: agent x sends a message to agent y).

The complexity of that first algorithm is $O(N^p)$, where N is the aggregated number of transitions, p the maximum number of transitions matching a specific message sending.

[3]Such as used in the DIMA multi-agent architecture [12]. Meanwhile, this technique is actually more general and the extraction algorithm (code parser) could be adapted to other kinds of structurations (agent architectures), as long as at least message sending instructions are made accessible.

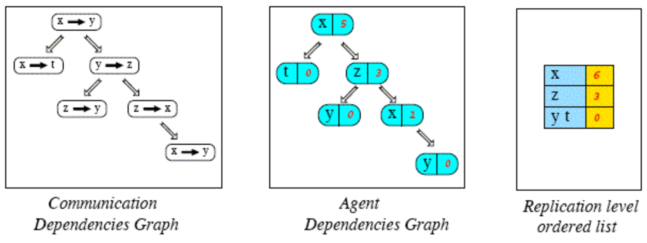

Figure 1: Static extraction of dependences

p is at most $N - 1$, but in practice with a lower bound, so the algorithm is polynomial.

A second algorithm transforms the communication dependences graph into an agent dependences graph, where each node corresponds to an agent and a label representing how many agents depend on it (can receive messages from it). It is shown at the middle of Figure 1.

Last, from that second graph, we can extract, in a standard way, its connex parts, minimal covering trees, and finally an ordered list, as a guide for ordering criticality of agents (shown at the right side of Figure 1). Further details were described in [20].

4.2 Dynamic Dependences

A limitation of the first strategy is that it is static and based on communication expressions, thus only capturing *potential* communications. Also, complex multi-agent systems are characterized by emergent structures, which thus cannot be always statically defined by the designer. Our second strategy is then to explicitly represent dependences between agents as a weighted graph, and to provide a mechanism to automatically update its respective weights according to communications between respective agents.

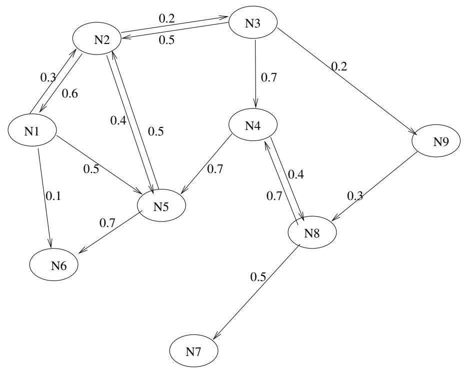

Figure 2: Example of interdependence graph

We consider the interdependence graph as a labeled oriented graph (see Figure 2), where each node represents a domain agent and each labeled arc between two nodes represents a dependence between the associated agents. The label of an arc (oriented) is a real number which reflects the importance of the dependence (oriented) between the associated agents. The interdependence graph is dynamic as it can be modified when a new domain agent is added, or disappears, or when interaction patterns evolve.

At design time, the interdependence graph is initialized by the designer,[4] and at run time, it is automatically dynamically adapted. Several parameters may be used to update the interdependences between agents. Our primary updating strategy is using communication load (number of messages) as the parameter. The adaptation algorithm updates the interdependence graph, based on local information (communication load) and on global information, which is defined as an aggregation of the local information of the various agents and hosts.

The algorithm is very simple: only the number of messages is considered, independent of their contents, thus the cost of monitoring is very low. We also proposed and experimented with an extension of this algorithm, using performatives as additional input information (e.g., `request` has a weight greater than `cancel`) [14]. Note that monitoring of communication, is implemented by a general monitoring distributed architecture, which can also be used by other strategies (e.g., for monitoring roles [14]).

4.3 Roles

Another strategy that we studied is based on the concept of role. A role, within an organization, represents a pattern of services, activities and relations. As an example, in some e-commerce organization, roles are: service provider, client, broker, etc. A role will be fullfilled (played) by one or more agents, and the same agent may simultaneously play several roles in different organizations.

Roles are usually defined relatively to some organization, but they may also be defined relatively to some protocol. An example is the standard Contract Net Protocol [23] which considers two roles: manager (which broadcasts the call for proposal), and bidder (who proposes a bid). In fact, protocol roles can be considered as some specific case of organizational role, where an organization is created dynamically during the scope of the protocol activation.

For example, for the simple scenario introduced in Section 3.3, we may use the contract net protocol (CNP) as following (see Figure 3):

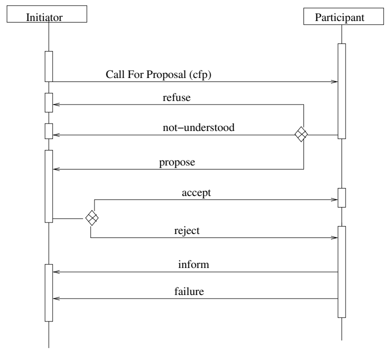

Figure 3: Contract net protocol

- A call for proposals message is sent to the participants from the initiator following the FIPA CNP.
- The participants reply (propose or refuse) to the initiator with the proposed meeting times.
- The initiator sends accept or reject messages to participants.
- The participants which agree to the proposed meeting inform (confirm) the initiator.

The notion of role captures some information about relative importance of roles and their interdependences. Thus we thought that a role is a pertinent concept for estimating criticality. We ask the designer to grade the various roles along their *criticality* (relative importance). In the scenario, two roles are considered: `Initiator` and `Participant`. Their respective weights could be set by the application designer to e.g., respectively `0.7` and `0.4`.

In order to monitor roles, we must consider if we can make the assumption that agents signal explicitly when they play a role (role-taking and then role-leaving) or if we cannot.

Signaling explicitly when an agent starts (and stops) playing a role is usually the case for organizational roles, where organizational actions are usually made public to the organization. For protocol roles, if agents use FIPA ACL and specify explicitly the protocol used (within the messages), that information can then also be used.

Meanwhile, as we want our role strategy to be general, we also considered the case where agents do not necessarily signal their roles. We only suppose that they communicate with some agent communication language such as FIPA ACL. We thus designed a role monitoring mechanism, described in [13]. It uses a description language for specifying protocols, stored in a library, and a recognition algorithm.

Last, the criticality of an agent is computed as the aggregation of the weights set to the roles it is currently playing [13].

4.4 Norms

A strategy under current investigation extends the previous strategy using roles, with two kinds of additional informations: time outs and norms. The underlying assumption is that time outs and norms (permissions, obligations, prohibitions) also capture some indication of criticality. We start from a description language (with its associated control architecture), named XMLaw, for law-based governance of multi-agent systems [5]. In XMLaw, we specify an interaction protocol (with its transitions), and a set of norms and clocks (see [5] for details on XMLaw).

Let's take a simple example of e-commerce, with a seller and a customer, as shown in Figure 4. The customer requests for products through a call for proposal (`cfp` message, based on CNP, as in Section 4.3) sent to various sellers. When a seller proposes a product (`propose` message), a clock (time out) is activated in order to check that the customer answers (accept or refuse the offer) within a specific time frame (which then deactivates the clock). If the customer accepts the offer in time, then the seller sends the bill to the customer (`inform (bill)` message). A norm - to be more precise, an obligation - is then activated to ensure that the customer sends a proof of payment to the seller (which deactivates the norm). The specification of the norm in XMLaw, with its associated activation and deactivation events, is shown below. It is a fragment of the whole XMLaw specification of the example.

```
<Norms>
   <Owner>Customer</Owner>
   <Obligation id="obligation-customer-to-pay">
      <Activations>
```

[4]Note that it can actually be automatically initialized by the previous strategy, based on static analysis of dependences.

```
         <Element ref="customer-receive-bill"
                event-type="transition_activation"/>
      </Activations>
      <Deactivations>
         <Element ref="customer-send-proof-payment"
                event-type="transition_activation"/>
      </Deactivations>
   </Obligation>
</Norms>
```

These specifications about role taking/leaving,[5] clock activation/deactivation, and norm activation/deactivation, are used as inputs to automatically adjust the criticality along the various steps of the protocol, as shown in Figure 4. These three contributions (role, clock, norm) to the change of criticality are then aggregated to produce the estimation.

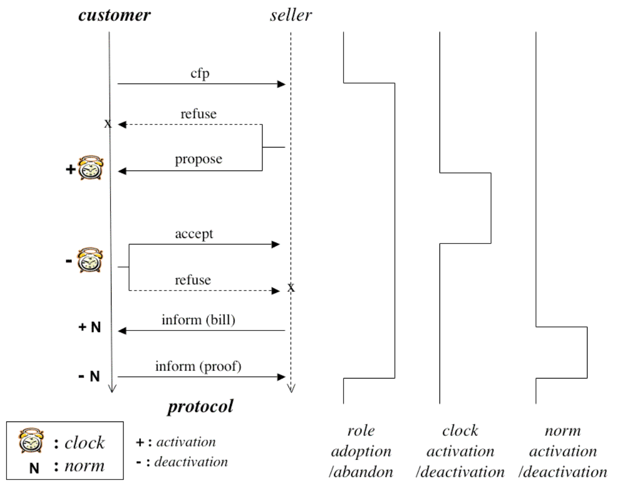

Figure 4: Evolution of criticality for customer role

The designer may also have a finer grain control explicit of the weights associated to each activation or deactivation event. Further details of that strategy may be found in [9].

4.5 Plans

That last strategy uses the plans of an agent, i.e., the actions that the agent has planned to execute in the near future. In our model, we consider that each agent of the system knows which sequence of actions (plan) must be executed in order to accomplish its current goal.[6] We assume that at each given instant of time, the agent is executing at most one action.

Using the same approach established by [16], we represent the plan of an agent as a directed acyclic AND/OR graph where each node represents an action. The nodes are connected by AND or OR edges.

In the example of Figure 5, after performing the action A, $Agent1$ needs to have both actions B and C executed in order to accomplish its plan. However, after C, only one of D or E needs to be performed so that $Agent1$ accomplishes its plan. We call an *external action* an action belonging to the plan of an agent which will be executed by other agents. For example, consider the action C belonging to the plan of $Agent1$ in Figure 5. Since this action is performed by $Agent2$, it is an external action in the current plan of $Agent1$. A *terminal action* is an action after which no other known action will be performed.

[5]The use of roles is analog to Section 4.3.

[6]Since unexpected events may occur in dynamic environments, agents usually interleave planning and execution. Consequently, their plans are established just for the short term.

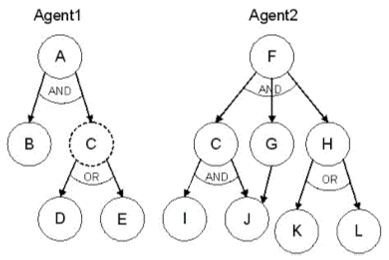

Figure 5: Examples of plans

In order to calculate the criticality of an action, we distinguish its *absolute criticality* from its *relative criticality*. The *absolute criticality (AC)* of an action 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, duration of the action, resources required for the execution of the action, application dependent information.

The *relative criticality (RC)* of an action belonging to the plan of an agent is proportional to the criticality of the agent when it is executing the action or waiting for some other agent to execute it. As a consequence, the relative criticality of an action may vary depending on the agent plan it belongs to. The relative criticality is calculated as follows:

- For an external action, it is equal to the *local relative criticality (LRC)*. The LRC is obtained using the AND-aggregation function if the action is connected to its children by means of AND edges or the OR-aggregation function if it is connected by OR edges. If the action has only one child, its LRC is equal to the relative criticality of its child. If the action is terminal (i.e. it has no child), its local relative criticality is equal to zero.

- For a non-external action a, its relative criticality is equal to its absolute criticality plus the sum of the local relative criticalities of a in each plan to which it belongs.

We have also refined this strategy (summarized in this paper) by considering the expected duration 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. Another issue is the possible dynamicity of plans of agents, because of, e.g.: lack of resources, failed commitments, etc. We proposed two main types of strategies to update criticality: time-driven strategies and event-driven strategies (e.g., action completion, failure). More details of the strategies are presented in [1]. Note that one expected advantage of this strategy of using plans is that we can estimate not only the immediate criticality, but also estimate future criticality.

5. DISCUSSION

Each strategy has its pros and cons: static, dynamic, cost, and nature of assumptions of abstractions available (messages, roles, norms, plans). The last strategy (plans) has the advantage of estimating future criticality and not just instantaneous one. Note that various strategies that we proposed are mostly bottom-up, as they use or infer information from the program elements or/and from execution, to estimate criticality of agents. We are also planning to study a dual direction, top down, based on first analysis and specifications of general dependability requirements, and then in using that information to guide replication control. Some directions are in using a dependability risk-driven approach [4] or/and dependability cases [24].

We are currently conducting experiments to compare strategies. In this paper, we summarize some of our experiments, based on the scenario of meetings scheduling (see Section 3.3). They were carried out on twenty machines with Intel(R) Pentium(R) 4 CPU at 2 GHz and 526 Mb of RAM. To compare accuracy of strategies, we used a fault simulator which randomly chooses an agent and stops its thread. If the killed agent was playing the role of an initiator, then its associated meeting scheduling negociation (protocol) fails, unless the agent has been replicated. Thus, that experiment provides some measure of the accuracy of the strategy to identify most critical agents and protect them.

We considered a multi-agent system with 200 agents distributed on 10 machines. We run each experiment 10 minutes and we introduced 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: $SR = \frac{NSS}{TNS}$, where NSS is the number of succeeded simulations and TNS is the total number of simulations.

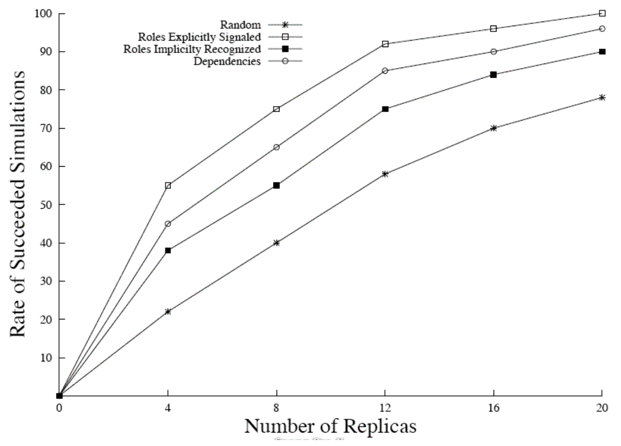

Figure 6: Rate of succeeded simulations for each number of replicas

In Figure 6, we compare four strategies: 1) random, 2) roles strategy with roles explicitly signaled, 3) roles strategy with role monitoring, 4) dependences strategy. For each strategy, we display the success rate SR as a function of the number of extra replicas.

First, it shows that all strategies show better results than the random strategy. The strategy with roles explicitly signaled is the most accurate (actually it is also the less costly). This can be explained for the example scenario by the importance of the initiator in the negotiation. For application domains where the roles have similar importance, the strategy based on dependences may lead to better results. We are currently conducting further measures on different types of applications. The objective is to try to empirically identify possible features of applications, correlated to the relative accuracy of different strategies.

6. RELATED WORK

Some work [7] offers dynamic cloning of specific agents in multi-agent systems. But their motivation is different to ours, as their objective is to improve the availability of an agent if it is too congested. The agents considered seem to have only functional tasks (with no changing state) and fault-tolerance aspects are not considered.

Hagg introduces sentinels to protect the agents from some undesirable states [15]. Sentinels 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 and handles agents which achieve that functionality. Adding sentinels to multi-agent systems is an interesting approach, however sentinels represent themselves failure points for the multi-agent system.

Fedoruk and Deters [8] 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. A proxy manages the state of the replicas. All external and internal communications of the group are redirected to the proxy. But 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. Their approach lacks some flexibility and reusability, in particular concerning replication control. Replication is indeed set up by the designer before run time.

Kaminka et al. [18] adapt 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. Thus, the behavior of agent cannot be adaptive and the system cannot be open.

Horling et al. [17] present a distributed system of diagnosis. The faults can directly or indirectly be observed in the form of symptoms by using a failure 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. However, a problem of performances can occur in this approach because the global performance improvement is based on a local performance improvement.

The work by Kraus et al. [19] 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 it is based on too many restrictive hypothesis to be made adaptive.

Last, a related and much more general project is the Autonomic Computing Program of IBM. They propose a general blueprint architecture (monitor, analyze, plan, execute). The ABLE prototype architecture/framework [3], partially implements it and provides a toolbox of information analysis and management components. Although fault-tolerance is one of its crucial part, the ABLE architecture by itself does not solve the problem, but the blueprint guidelines are an interesting direction.

7. CONCLUSION

Large-scale multi-agent systems are often distributed and must run without any interruption. To make these systems reliable, we proposed an architecture (DarX) for dynamic replication and its control [13]. In this paper we discussed various strategies for estimating criticality of agents, infered automatically from various kinds of information (references, messages, roles, norms, plans). The agent criticality is then used to replicate agents in order to maximize their reliability and availability based on available resources.

We believe that our current results are promising. Meanwhile, more experiments are needed to better evaluate our approach, various strategies, and classify their respective classes of applications.

Acknowledgments. The authors would like to thank Marin Bertier, Alain Cardon, Sébastien Charpentier, Athmane Hamel, Guillaume Lacôte and Jakob Zimmerman, for their past contributions to the project.

8. REFERENCES

[1] A. L. Almeida, S. Aknine, J.-P. Briot, and J. Malenfant. Plan-based Replication for Fault-tolerant Multi-Agent Systems. To appear in *11th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS'2006)*, IPDPS'2006, Rhodes Island, Greece, April 2006.

[2] M. Bertier, O. Marin, and P. Sens. Performance Analysis of a Hierarchical Failure Detector. *Int. Conf. on Dependable Systems and Networks*, San Francisco, CA, USA, June 2003.

[3] J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Mills, and Y. Diao. ABLE: A Toolkit for Building Multiagent Autonomic Systems. *IBM Systems Journal*, 41(3):350–371, 2002.

[4] G. Carvalho, R. Paes, R. Choren, and C. Lucena. Towards a Risk Driven Method for Developing Law Enforcement Middleware. *3rd Int. Workshop on Agent-Oriented Methodologies*, OOPSLA'2004, Vancouver, BC, Canada, October 2004.

[5] G. Carvalho, R. Paes, R. Choren, P. Alencar, and C. Lucena. Increasing Software Infrastructure Dependability through a Law Enforcement Approach. *1st Int. Symposium on Normative Multiagent Systems (NorMAS'2005)*, AISB'2005, U.K., April 2005.

[6] C. Castelfranchi. Dependence Relations in Multi-Agent Systems. *Decentralized AI*, Elsevier, 1992.

[7] K. Decker, K. Sycara, and M. Williamson. Cloning for Intelligent Adaptive Information Agents. *ATAL'97*, LNAI, p. 63–75, Springer-Verlag, 1997.

[8] A. Fedoruk and R. Deters. Improving Fault-Tolerance by Replicating Agents. *AAMAS'2002*, p. 737–744, 2002.

[9] M. Gatti, C. Lucena, J.-P. Briot, and Z. Guessoum. Towards a Fault-Tolerant Open Multi-Agent Platform based on a Law-Governed Approach, *Monografias em Ciència da Computação*, No 06/01, Dept. de Informàtica, PUC-Rio, Brazil, January 2006.

[10] R. Guerraoui, B. Garbinato, and K. Mazouni. Lessons from Designing and Implementing GARF. *Object-Based Parallel and Distributed Computation*, No 791, p. 238–256, LNCS, Springer, 1995.

[11] R. Guerraoui and A. Schiper. Software-based Replication for Fault Tolerance. *IEEE Computer*, 30(4):68–74, 1997.

[12] Z. Guessoum and J.-P. Briot. From Active Objects to Autonomous Agents. IEEE Concurrency, 7(3):68–76, July-September 1999.

[13] Z. Guessoum, J.-P. Briot, O. Marin, A. Hamel, and P. Sens. Dynamic and Adaptive Replication for Large-Scale Reliable Multi-Agent Systems. *Software Engineering for Large-Scale Multi-Agent Systems*, No 2603, p. 182–198, LNCS, Springer, 2003.

[14] Z. Guessoum, N. Faci, and J.-P. Briot. Adaptive Replication of Large-Scale Multi-Agent Systems - Towards a Fault-Tolerant Multi-Agent Platform. *ACM Software Engineering Notes*, 30(4):1–6, July 2005.

[15] S. Hagg. A Sentinel Approach to Fault Handling in Multi-Agent Systems. *Multi-Agent Systems, Methodologies and Applications*, No 1286, p. 190–195, LNCS, Springer, 1997.

[16] B. Horling et al. The TAEMS White Paper, ISI, USC, L.A., CA, USA, January 1999.

[17] B. Horling, B. Benyo, and V. Lesser. Using Self-Diagnosis to Adapt Organizational Structures. *5th Int. Conf. on Autonomous Agents*, p. 529–536, 2001.

[18] 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.

[19] S. Kraus, V.S. Subrahmanian, and N. Cihan Tacs. Probabilistically Survivable MASs. *IJCAI'03*, pages 789–795, 2003.

[20] G. Lacôte, J.-P. Briot, Z. Guessoum, and P. Sens. Towards Fault-Tolerant Agents. *Workshop on Distributed Objects Programming Paradigms*, ECOOP'2000, Cannes, France, June 2000.

[21] O. Marin, M. Bertier, and P. Sens. DARX - a Framework for the Fault-Tolerant Support of Agent Software. *14th IEEE Int. Symposium on Software Reliability Engineering (ISSRE'2003)*, p. 406–417, Denver, CO, USA, 2003.

[22] O. Marin, P. Sens, J.-P. Briot, and Z. Guessoum. Towards Adaptive Fault-Tolerance for Distributed Multi-Agent Systems. *4th European Research Seminar on Advances in Distributed Systems (ERSADS'2001)*, p. 195–201, Bertinoro, Italy, 2001.

[23] R.G. Smith. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *IEEE Trans. Computers*, 29(12):1104–1113, 1980.

[24] C.B. Weinstock, J.B. Goodenough, J.J. Hudak. Dependability Cases. *Technical Notes*, CMU/SEI-2004-TN-016, Software Engineering Institute, CMU, Pittsburgh, PA, USA, 2004.