Towards Autonomic Fault-Tolerant Multi-Agent Systems

Alessandro Almeida
LIP6 (Laboratoire d'Informatique de Paris 6)
Université Paris 6 - CNRS
Case 169, 4 place Jussieu
75252 Paris Cedex 05, France
Email: Alessandro.Luna-Almeida@lip6.fr

Jean-Pierre Briot
LIP6
& Laboratório de Engenharia de Software (LES), DI, PUC-Rio
Rua Marquês de São Vicente 225, Gávea
Rio de Janeiro, RJ 22451-900, Brazil
Email: Jean-Pierre.Briot@lip6.fr

Samir Aknine
LIP6
Email: Samir.Aknine@lip6.fr

Zahia Guessoum
LIP6
Email: Zahia.Guessoum@lip6.fr

Olivier Marin
LIP6
Email: Olivier.Marin@lip6.fr

***Abstract*—This paper describes the progress of our work on autonomic fault-tolerant multi-agent systems. Our starting point is that new generation distributed applications are increasingly dynamic and open, e.g., new agents can join or leave, they can change roles or strategies and interdependences with other agents. Thus the traditional approach of configuring fault tolerance techniques (e.g., through replication) at design time may turn out inadequate. We propose to make autonomous the management of fault tolerance, more precisely by replication, along the course of the computation. Therefore, in a first step, we propose a framework for dynamic replication with some specific features (e.g., dynamic adaptation of replication strategies). We then propose several metrics for identifying dynamically what agents are more critical. In this paper, after surveying 2 metrics based on dependences and on roles, we detail our most recent metric, based on plans. We also discuss the issue of creation of replicas as a resource allocation problem.**

I. Introduction

The possibility of partial failures is a fundamental characteristic of distributed applications. Standard solutions for fault tolerance (e.g., replication, exception handling) are usually designed and configured at design time. In other words, this is the task of the designer to identify in advance the most critical software components of the application and to decide what strategies (e.g., active or passive replication) and their configurations (how many replicas, their placement, etc.) will be applied. But this traditional approach reaches its limits when considering new generation of cooperative distributed applications (e.g., multi-agent systems) which are more open and dynamic. Examples of such applications are: e-commerce, air traffic control, crisis management systems, and ambient intelligence. In these applications, the roles and relative importance of the agents can greatly vary during the course of computation and interaction, because the agents may change roles, plans and strategies as well as the nature of their interdependences. 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. As a consequence, early design decisions on where and how to apply fault tolerance techniques (e.g., decision to replicate an agent) may turn out inadequate. Therefore, in this paper we propose an approach of self-adaptation of fault tolerance techniques to make this fault tolerance management dynamic and automatic.

Our starting point is the technique of software replication. As discussed by [10], software replication in distributed environments has some 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. Another important advantage, on the design perspective, is that the use of software replication is relatively generic and transparent to the application domain. The designer does not have to explicitly specify the nature of the possible abnormal behaviors and the way to handle them.[1]

As we show in the paper, our solution is furthermore transparent, as the task of deciding what entities to replicate and how to parameterize replication is handled automatically, in an autonomic way. A 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, or/and application/agent level, e.g., roles or plans, as metrics to estimate criticality. This paper will report on some of our current strategies and experiments, in order to estimate criticality and to control replication.

II. Context of this Work

A. Model of Failure Considered

Any software/hardware component may be subject to faults resulting in output errors, which can lead to a deviation of

[1]That said, software replication focuses mostly on processor or network faults, and does not address the whole spectrum of possible faults (design, timing…). Thus, a general issue, and still subject of open research - not addressed in this paper -, is how to combine various approaches for fault tolerance in a single articulated methodology.

its specified behavior, 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 the 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.

B. 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 [10]. 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. Note that when applied to agents with a deterministic behavior, active replication strategies ensure consistency between replicas, thanks to the total ordering of messages. For non deterministic agents, a (light weight) additional consistency management mechanism is required.

III. A Framework for Dynamic Replication

To overcome the limitations of static or explicit replication, 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). Current replication toolkits rarely support such dynamicity. Therefore, we designed a novel replication framework, named DarX, with such dynamic features.

DarX is a framework for designing reliable distributed applications based on adaptive replication. One of DarX specific features is the reification of the replication strategy, so that it may be dynamically changed. In DarX, a replication group is an opaque entity underlying every application agent. The number of replicas and the replication strategy of a specific agent are totally hidden to the other application agents. Each replication group has exactly one leader which communicates with the other agents. The leader also checks the liveness of each replica and is responsible for reliable broadcasting. In case of failure of a leader, a new one is automatically elected among the set of remaining replicas. 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. See, e.g., [15] for further details about DarX.

IV. Autonomous Control of Replication

Provided the architecture for dynamic replication, we need a control mechanism for deciding which agents should be replicated and with what strategy (active or passive, how many replicas, where to create the replicas, etc.). As discussed in Section I, in dynamic and open applications, control should be dynamic. Moreover, a manual control is not realistic, as the application designer cannot directly monitor the evolution of a distributed cooperative application of a significant scale. Therefore, the control mechanism should be autonomous although it may use some information as provided by the designer of the application.

A. An Example of Scenario

In this paper, 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. Each controller is assisted by an assistant agent who cooperatively monitors the air traffic to suggest decisions about traffic control (see Figure 1). Agents communicate in order to assist with collaborative procedures, e.g., hand off procedures, i.e., when a controller passes the responsibility of an airplane exiting from its supervision sector to the controller of the sector the plane is entering.

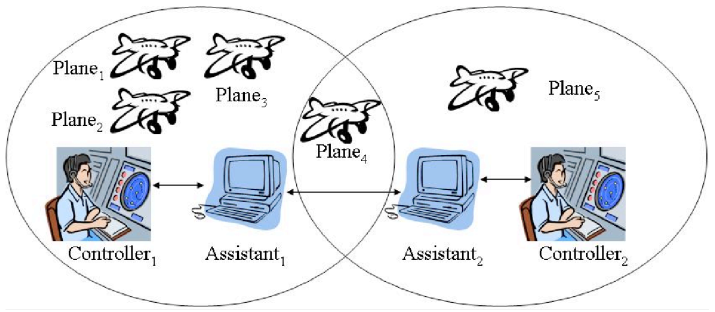

Fig. 1. Air traffic control

While trying to accomplish their tasks, agents can be faced to different kinds of failures. In our work, we initially consider the crash type of failures, which can be caused by internal (operating system crashes, hardware problems) or external factors (malicious attacks, power failures, environmental disasters). Additionally, the failure of one agent can also impact on the agents who depend on it (hand off procedures). To minimize the impact of failures, agents can be replicated. It is clear that replicating every agent in every machine 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. Thus, 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 contexts of tasks are dynamic and, thus, the criticalities of the agents vary at runtime.

B. Replication Control = Metric + Allocation

To control replication in an autonomous way, we will address two successive issues: 1) a *metric* problem - estimating the criticality of agents; 2) an *allocation* problem - which agents should be replicated and where to deploy the replicas. 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.

V. Metrics for Estimating the Criticality

Some metrics have been proposed for identifying most critical software components (see, e.g., [5]). However, these metrics are often based on the static structure and local nature of software components, e.g., size, complexity, and dependences. Some recent directions are considering more dynamic properties, e.g., frequency, that is the number of times that a function is executed during a period of time, and mechanisms for updating them automatically (see e.g., [7]).

We believe that, because of the very dynamic nature of multi-agent systems (agents changing roles, non deterministic behavior), as well as thanks to the higher conceptual level (organizational and cognitive levels of knowledge), more appropriate metrics should be explored. Therefore, we have designed alternative metrics for estimating agent criticality (static dependences, dynamic dependences, roles, norms, and plans) [4]. Note that each metric has its pros and cons: static or dynamic, cost (overhead), and nature of assumptions of abstractions available (e.g., messages, roles, norms, plans). In this paper, we will survey three of them, and detail our most recent one, based on plans, which has the advantage of estimating future criticality and not just instantaneous one.

A. Dependences

This first metric is based on the concept of *dependence* (between agents). We explicitly represent dependences between agents as an oriented graph, where each node represents a domain agent and each arc between two nodes is labelled with a weight which reflects the importance of the dependence between the associated agents.

Intuitively, the more an agent has other agents depending on it, the more it is critical in the organization. Thus, the criticality of an agent is computed as the aggregation of the weights representing dependences of other agents on it.

At design time, the interdependence graph is initialized by the designer. It is then dynamically and automatically adapted at run time. Several parameters may be used to update the interdependences between agents. A first strategy is using communication load (number of messages) as the parameter. We also proposed using performatives as additional input information (e.g., *request* has a weight greater than *cancel*).

B. Roles

An alternative metric that we studied is based on the concept of *role*. A role, within an organization, represents a pattern of services, activities and relations. For example, in an e-commerce organization, agents can play the roles of service provider, client, broker, etc. A role will be fulfilled (played) by one or more agents, and the same agent may simultaneously play several roles in different organizations. The notion of role captures some information about the relative importance of roles and their interdependences. We ask the designer to grade the various roles along their criticality (relative importance). We compute the estimated criticality of an agent as the aggregation of the weights of the roles it is currently playing.

Note that, in order to monitor roles, we may assume that agents signal explicitly when they take or leave a role (this information may be made explicit for several models of organizations and also for some models of interaction protocols). In order to be more general, we also designed a role recognition mechanism, based on the recognition of patterns of interaction.

C. Plans

This most recent metric is based on the plans of an agent, i.e., the actions that the agent has planned to execute in the near future. We consider that each agent of the system knows which sequence of actions (plan) must be executed in order to accomplish its current goals. We represent the plan of an agent as an acyclic Recursive Petri Net (RPN) [6], 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.

In the example of Figure 2, we show two plans elaborated by two assistant agents. After solving the conflict between *plane1* and *plane2*, *Assistant1* asks *plane3* to land and hands off *plane4* to *sector2* (controlled by *Assistant2*). The action *SolveConflict* is an abstract action (transition in black) and, thus, *Assistant1* needs to refine it during execution. The action *HandOff* is a joint action which needs synchronization between the two agents. As a consequence, when *Assistant1* proposes to hand off *plane4* to *sector2*, it must ensure that *Assistant2* is ready to receive the details of the transfer. At the implementation level, this is done by a synchronizing transition.

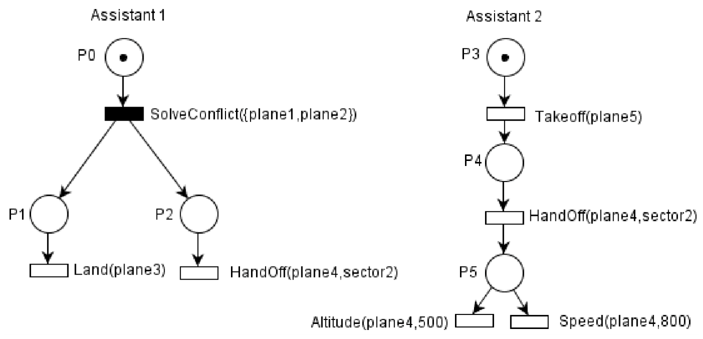

Fig. 2. Examples of two interacting plans

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. 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 dependent 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. 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. 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.

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 [2].

VI. Resource Allocation

Our mechanism for deciding which agents should be replicated and where to deploy the replicas is considering the failure probability of the replicas. Indeed, it is better to have only one replica which will have in the future an almost zero probability of failure than having many replicas which are not reliable. We formally define replica allocation as an optimization problem and use event-driven policy for updating the probability of machines failures and then of replicas failures, based on the history of machine failures. Because of space limitation, more details are presented in [1].

VII. Experiments

A preliminary serie of experiments were run using our plan-based strategy on a simplified air traffic control scenario. 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. We set the number of resources available at the machine as half of the number of agents. 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 using the corresponding mechanism. 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 the probability of failure of its resources. 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.

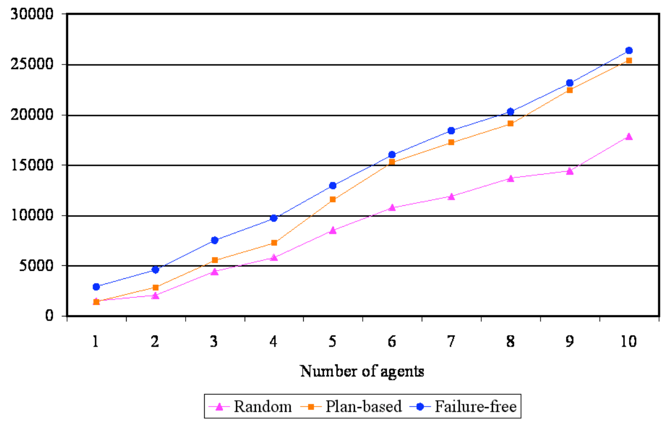

Fig. 3. Quality of the replication mechanism used vs. a failure-free execution

Figure 3 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. 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 than a random strategy. In fact, the probability that a critical agent fails with our strategy is lower than with a random strategy. We also ran experiments to compare the CPU time required by our mechanism and by the execution of the multi-agent system with no replication at all. Using no replication always outperforms our replication mechanism, but the overhead of our mechanism is negligible (less than 4%).

VIII. Related Work

A seminal project about autonomic computing is the Autonomic Computing Program of IBM. They propose a general blueprint architecture (monitor, analyze, plan, execute). A prototype architecture, named ABLE (Agent Building and Learning Environment) [3], partially implements it and provides a toolbox of components (implemented as JavaBeans) for manipulating and using monitored information (rules, neural networks, statistics…). An example application is on autonomic load balancing of application servers. Accord [14] is another framework, focusing on the development of autonomic applications using dynamic composition of autonomic elements. However, ABLE or Accord does not provide built-in mechanisms for fault tolerance such as replication strategies.

[17] is a general analysis on how to incorporate fault tolerance concerns in an autonomic setting. It identifies fault tolerance at the level of service registry, and discusses mechanisms for identifying faulty servers. It considers using replication but does not address adaptation mechanisms.

[16] addresses the issue of fault-tolerance in the context of the use of mobile agents, both for workflow management and grid computing contexts. They propose a model and a mechanism for automatic selection of checkpointing mechanisms (local or remote) in order to minimize the workflow execution duration. Their choice is based on some variables specific to the workflow execution domain (e.g., agent size, number of stages in the sequence path). The use of model characteristics (namely the variables in their model) as input for selection and mapping is analog to our use of information and metrics to estimate and control the application of fault-tolerance techniques (in our case, replication). However, the type of information we use is more generic to the multi agent-systems domain.

Chameleon [12] is an adaptive fault tolerance system using reliable mobile agents. The methods and techniques are embodied in a set of specialized agents supported by a fault tolerance manager (FTM) and by host daemons for handshaking with the FTM via the agents. Adaptive fault tolerance is achieved by making the Chameleon infrastructure reconfigurable. Static reconfiguration guarantees that the components can be reused for assembling different fault tolerance strategies. Dynamic reconfiguration allows component functionalities to be extended or modified at runtime by changing component composition, and components to be added to or removed from the system without taking down other active components. However, this reconfiguration must be specified and applied in a non-automatic way during the execution of the system.

Hagg introduces sentinels to protect the agents from some undesirable states [11]. Sentinels represent the control structure of their multi-agent system. They need to build models of each agent and monitor communications in order to react to faults. Each sentinel is associated by the designer to one functionality of the multi-agent system. This sentinel handles the different agents which interact to achieve the functionality. The analysis of his believes on the other agents enables the sentinel to detect a fault when it occurs. Adding sentinels to multi-agent systems seems to be a good approach, however the sentinels themselves represent failure points for the multi-agent system.

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.

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

IX. 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. In this paper, we discussed some metrics for estimating criticality of agents, inferred automatically from various kinds of information (messages, roles, 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 metrics and strategies, and classify their respective classes of applications.

Acknowledgments: The authors thank the other members of our project, and notably Nora Faci, Jacques Malenfant, and Pierre Sens, for their contributions to the project. We acknowledge support from the following research programs: ANR Sécurité & Informatique, EuroControl CARE INO III, CAPES-COFECUB and CNPq.

References

[1] Almeida, A.L., Aknine, S., Briot, J.-P., Guessoum, Z. and Marin, O., Plan-based resource allocation for providing fault tolerance in multi-agent systems. *3rd Workshop on Software Engineering for Agent-oriented Systems (SEAS'2007)*, co-located with the 21th Brazilian Conference on Software Engineering (SBES'2007), João Pessoa, PB, Brazil, October 2007.

[2] Almeida, A.L., Aknine, S., Briot, J.-P. and Malenfant, J., A predictive method for providing fault tolerance in multi-agent systems, *IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'2006)*, Hong Kong, H.K., December 2006, pp. 226–232.

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

[4] Briot, J.-P., Guessoum, Z., Aknine, S., Almeida L., Faci, N., Gatti, M., Lucena, C., Malenfant, J., Marin, O. and Sens, P., Experience and Prospects for Various Control Strategies for Self-Replicating Multi-Agent Systems, *ICSE'06 International Workshop on Software Adaptive and Self-Managing Systems (SEAMS'06)*, Shanghai, China, ACM Press, May 2006, pp. 37–43.

[5] Ebert, C., Metrics for Identifying Critical Components in Software Projects, In *Handbook of Software Engineering and Knowledge Engineering*, Volume 1, edited by S.-K. Chang, World Scientific Publishers, 2001.

[6] El Fallah Seghrouchni, A. and Haddad, S., A Recursive Model for Distributed Planning, *2nd International Conference on Multi-Agent Systems (ICMAS'96)*, AAAI Press, 1996, pp. 307–314.

[7] El Yamany, H.F., Capretz, M.A.M. and Capretz, L.F., A Multi-Agent Framework for Testing Distributed Systems, *IEEE COMPSAC 3rd International Workshop on Quality Assurance and Testing Web-Based Applications (QATWBA'2006)*, Chicago, IL, USA, September 2006, pp. 151–156.

[8] Fedoruk A. and Deters, R., Improving Fault-Tolerance by Replicating Agents, *International Conference on Autonomous Agents And Multi-Agent Systems (AAMAS'2002)*, Bologna, Italy, July 2002, pp. 737–744.

[9] Guerraoui, R., Garbinato, B. and Mazouni, K., Lessons from Designing and Implementing GARF, In *Object-Based Parallel and Distributed Computation*, edited by J.-P. Briot, J.-M. Geib and A. Yonezawa, LNCS, No 791, Springer-Verlag, 1995, pp. 238–256.

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

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

[12] Kalbarczyk, Z., Iyer, R.K., Bagchi, S. and Whisnant, K., Chameleon: A Software Infrastructure for Adaptive Fault tolerance, *IEEE Transactions on Parallel Distributed Systems*, 10(6):560–579, 1999.

[13] Kraus, S., Subrahmanian, V.S. and Cihan Tacs, N., Probabilistically Survivable MASs, *International Joint Conference on Artificial Intelligence (IJCAI'03)*, Acapulco, Mexico, August 2003, pp. 789–795.

[14] Liu, H. and Parashar, M., Accord: A Programming Framework for Autonomic Applications, *IEEE transaction on Systems, Man, and Cybernetics*, Special issue on Engineering Autonomic Systems, edited by R. Sterritt and T. Bapty, IEEE Press, 2005.

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

[16] Nichols, J., Demirkan, H. and Goul, M., Autonomic Workflow Execution in the Grid, *IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews*, 36(3):353–364, May 2006.

[17] Toma, Y., Incorporating Fault-Tolerance into an Autonomic-Computing Environment, *IEEE Distributed Systems Online*, 5(2), February 2004.