ENGINEERING CONTROL STRATEGIES FOR REPLICATION-BASED FAULT-TOLERANT MULTI-AGENT SYSTEMS

J.-P. BRIOT[1,2] AND Z. GUESSOUM[1] AND S. AKNINE[1]
AND A.L. ALMEIDA[1] AND N. FACI[3] AND J. MALENFANT[1]
AND O. MARIN[1] AND P. SENS[1]
[1] *LIP6, Paris, France ;* [2] *LES, PUC-Rio, Brazil ;* [3] *CReSTIC, Reims, France*
Contact: Jean-Pierre.Briot@lip6.fr

This work focuses on the engineering of software replication techniques for distributed cooperative applications designed as multi-agent systems. Such applications are often very dynamic: e.g., new agents can join or leave, they can change roles or strategies. Also, the relative importances of agents may evolve during the course of computation and cooperation, as opposed to traditional static approaches of replication, e.g., for data bases, where critical servers may be identified at design time. Thus, 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. An important issue is then: what kind of information could be used to estimate which agents are most critical agents? In this paper, we first introduce our approach and prototype architecture for adaptive replication. Then, we discuss various kinds of information and strategies to estimate criticality of agents: dynamic dependences, roles, and plans. Some preliminary measurements and future directions are also presented.

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), some more *curative* e.g., based on exception handling and cooperative recovery [13], and some more *preventive*, notably based on the concept of replication, applied e.g. to data bases.

As discussed by [5], 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. 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. 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 - and not addressed in this paper -, is how to combine various approaches for fault-tolerance in a single articulated methodology.

Software replication is generally applied explicitly and statically, at design time. Thus, it is the responsibility of the designer of the application to identify explicitly what 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 communications, roles, and plans.

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 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 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 [5]. 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*

Many toolkits include replication facilities to build reliable applications. However, many of them are not flexible enough to implement an adaptive replication. For instance, MetaXa [3] implements in Java active and passive replication in a flexible way. Authors extended Java with a meta-level architecture. However, MetaXa relies on a modified Java interpreter. GARF [4] realizes fault-tolerant Smalltalk machines using active replication. Similar to MetaXa, GARF uses a reflexive architecture and provides different replication strategies. But, it does not provide adaptive mechanism to apply these strategies.

3. Principles of our Approach 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). As discussed above, 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. For portability and compatibility issues, DarX is implemented in Java.

A replication group is an opaque entity underlying every application agent. The number of replicas and the internal 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 provides global naming. Each agent has a global name which is independent of the current location of its replicas. The underlying system allows to handle the agent's execution and communication. Each agent is itself wrapped into a `TaskShell` (see Figure 1), which acts as a replication group manager and is responsible for delivering received messages to all the members of the replication group, thus preserving the transparency for the supported application. Input messages are intercepted by the `TaskShell`, enabling message caching. Hence all messages get to be processed in the same order within a replication group.

An agent can communicate with a remote agent, regardless whether it is a single agent or a replication group, by using a local proxy implemented by the `RemoteTask` interface. Each `RemoteTask` references a distinct remote entity considered as its replication group leader. The reliability features are

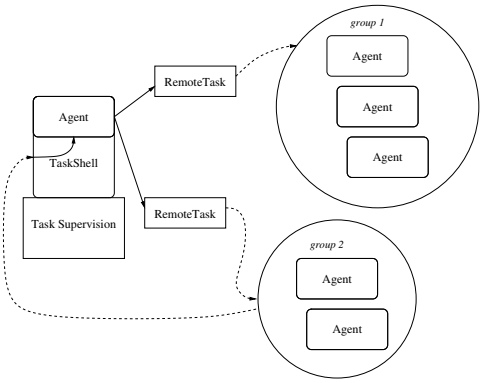

Figure 1. DarX application architecture

brought to agents by an instance of a DarX server (`DarxServer`) running on every location. Each DarxServer implements the required replication services, backed up by a common global naming/location service. See e.g., [12] 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 agents should be replicated and with what strategy (active or passive, how many replicas, where to create the replicas, etc.).[a] For dynamic applications,[b] 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 assis-

[a]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 [6].

[b]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.

tant agent which manages his calendar. This assistant 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 assistant agents of the system to schedule meetings, based on preferences of their human owners.

In practice, the assistant agents use the Contract Net Protocol (CNP) [14], as following (see Figure 2): a call for proposals message is sent (broadcasted) to the participants by the initiator ; 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.

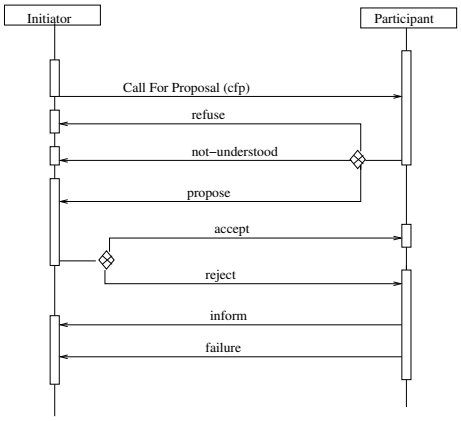

Figure 2. Contract net protocol

If the assistant agent of one important participant (initiator or prime participant) fails (e.g., his machine or PDA crashes), this may disorganize the whole meeting planning. 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 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: $nb_i = rounded(rm + w_i * Rm/W)$, where: w_i is the criticality of the agent, W is the sum of the domain agents' criticalities, rm is the minimum number of replicas, Rm are the available resources, i.e., the maximum number of replicas. The numbers of replicas computed 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 some strategies: some are completely general and use basic information (references, messages), some make some assumption of higher-level abstractions (performatives, roles, plans) which may or not be supported by a given multi-agent architecture.

4.1. *Dynamic Dependences*

This strategy 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. We explicitly represent dependences between agents as a weighted graph, and we provide a mechanism to automatically update its respective weights. The criticality of an agent is computed as the aggregation of the weights representing dependences of other agents on it.

We consider the interdependence graph as a labeled oriented graph (see Figure 3), 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.

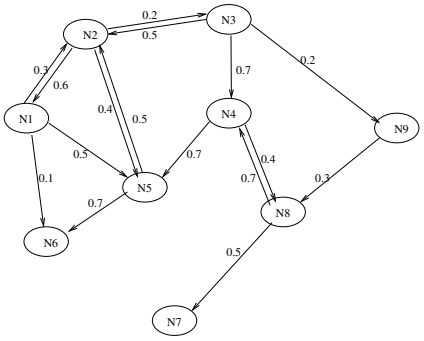

Figure 3. Example of interdependence graph

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,[c] and at run time, it is dynamically and automatically 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`) [6]. 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, see next section).

4.2. *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 fulfilled (played) by one or

[c]Note that it can actually be automatically initialized by another strategy, e.g., based on static analysis of dependences, which has been implemented.

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 contract net protocol which is used in the scenario (see Section 3.3). It considers two roles: manager/initiator and bidder/participant. 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.

The notion of role captures some information about the 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., `0.7` and `0.4`.

In order to monitor roles, we must assume or not if agents signal explicitly when they play a role (role-taking and then role-leaving). 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 (agent communication language) 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 minimal agent communication language. We thus designed a role monitoring mechanism. It uses a description language for specifying protocols, stored in a library, and a recognition algorithm. Lastly, the criticality of an agent is computed as the aggregation of the weights set to the roles it is currently playing.

4.3. *Plans*

The 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.[d] We assume that at each

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

given instant of time, the agent is executing at most one action.

Using the same approach established by [8], 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.

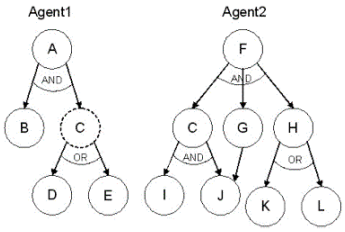

Figure 4. Examples of plans

In the example of Figure 4, 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 4. 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.

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. For an external action, it is computed as the aggregation (OR and AND, see details in [1]) of its children relative criticality. 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.

See [1] for further refinements, for considering the expected duration of actions, and for considering the possible dynamicity of plans of agents, because of, e.g., lack of resources, or failed commitments.

5. Discussion

Each strategy has its pros and cons: static or dynamic, cost, and nature of assumptions of abstractions available (messages, roles, 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 or dependability cases [15].

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 negotiation (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.

In Figure 5, 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.

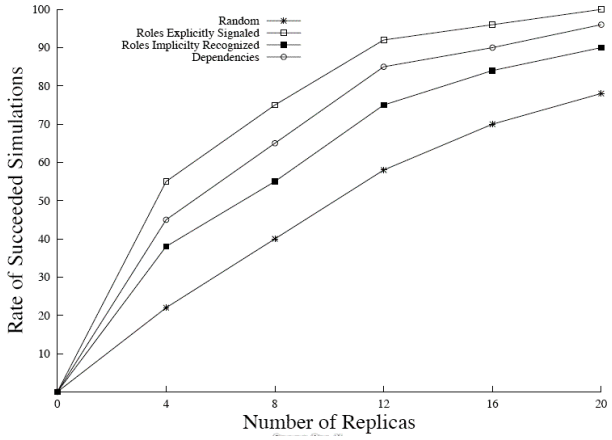

Figure 5. Rate of succeeded simulations for each number of replicas

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

Chameleon [10] 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. Unfortunately, through its centralized FTM, this architecture is not scalable and the FTM represents a bottle-neck as well as a failure point for the system.

Hagg introduced sentinels to protect the agents from some undesirable states [7]. 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. This approach is interesting, however sentinels represent themselves failure points for the multi-agent system.

Fedoruk and Deters [2] 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. [11] adopted 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. 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. [9] presented 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.

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

References

1. A. L. Almeida, S. Aknine, J.-P. Briot, and J. Malenfant. Plan-based Replication for Fault-tolerant Multi-Agent Systems. *11th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS'2006)*, IPDPS'2006, Rhodes Island, Greece, April 2006.
2. A. Fedoruk and R. Deters. Improving Fault-Tolerance by Replicating Agents. *International Conference on Autonomous Agents And Multi-Agent Systems (AAMAS'2002)*, p. 737–744, 2002.
3. M. Golm. MetaXa and the Future of Reflection. *OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, p. 238–256, 1998.
4. R. Guerraoui, B. Garbinato, and K. Mazouni. Lessons from Designing and Implementing GARF. *Object-Based Parallel and Distributed Computation*, LNCS, No 791, p. 238–256, Springer, 1995.
5. R. Guerraoui and A. Schiper. Software-based Replication for Fault Tolerance. *IEEE Computer*, 30(4):68–74, 1997.
6. 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.
7. S. Hagg. A Sentinel Approach to Fault Handling in Multi-Agent Systems. *Multi-Agent Systems, Methodologies and Applications*, LNCS, No 1286, p. 190–195, Springer, 1997.
8. B. Horling et al. The TAEMS White Paper, ISI/USC, L.A., CA, USA, 1999.
9. B. Horling, B. Benyo, and V. Lesser. Using Self-Diagnosis to Adapt Organizational Structures. *5th International Conference on Autonomous Agents*, p. 529–536, 2001.
10. Z. Kalbarczyk, R.K. Iyer, S. Bagchi, and K. Whisnant. Chameleon: A Software Infrastructure for Adaptive Fault tolerance. *IEEE Transactions on Parallel Distributed Systems*, 10(6):560–579, 1999.
11. 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.
12. O. Marin, M. Bertier, and P. Sens. DARX - a Framework for the Fault-Tolerant Support of Agent Software. *14th IEEE International Symposium on Software Reliability Engineering (ISSRE'2003)*, p. 406–417, Denver, CO, USA, 2003.
13. A. Romanovsky, C. Dony, J.L. Knudsen, and A. Tripathi (eds). Advances in Exception Handling Techniques, LNCS, No 2022, Springer, 2001.
14. R.G. Smith. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *IEEE Transactions on Computers*, 29(12):1104–1113, 1980.
15. C.B. Weinstock, J.B. Goodenough, J.J. Hudak. Dependability Cases. *Technical Notes*, CMU/SEI-2004-TN-016, SEI, CMU, Pittsburgh, PA, USA, 2004.