Towards Reliable Multi-Agent Systems An adaptive Replication Mechanism

Zahia GUESSOUM, [a,b] Jean-Pierre BRIOT, [a] Nora FACI, [b] and Olivier MARIN [a]

[a] *LIP6, University of Paris 6*

[b] *CReSTIC, University of Reims*

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. In this paper, we will focus on crash failures, with usual preventive approaches by replication. But, as criticality of agents may evolve during the course of computation and problem solving, static design (as, e.g., for critical database servers, well identified at design time) is not appropriate. 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. In this paper, we describe a prototype architecture, supporting adaptive replication. We also discuss and compare various control strategies for replication, one using agent roles, and another using inter-agent dependences as types of information to infer and estimate criticality of agents. Experiments and measurements are also reported.

Keywords. Agent, multi-agent system, reliability, fault-tolerance, monitoring, control, adaptation, replication, role, dependence, organization.

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 successfully applied e.g. to databases. As implied by [16], 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, we note that the replication techniques are in general applied explicitly and statically, at design time. This means that it is the responsibility of the designer of the application to identify explicitly which critical components should be made robust and also to decide upon strategies (active or passive replication) and their parameters (how many replicas and their placement).

Meanwhile, new cooperative applications, e.g., e-commerce, air traffic control, crisis management systems, ambient intelligence, are much more dynamic. They are in-

creasingly designed as a set of autonomous and interactive entities, named agents, which interact and coordinate, thus named multi-agent systems (MAS) [11] [27].

In such applications, the roles and relative importance of the agents can greatly vary during the course of computation, of interaction and of cooperation, the agents being able to 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. In addition, such applications may be large scale [20]. And the fact that the underlying distributed system is large scale makes it unstable by nature, at least in currently deployed technologies. Last, future collaborative applications may use *ad hoc* network infrastructures, where possibilities of disconnection are even greater.

These new challenges reach the limit of a traditionally static approach for introducing fault tolerance in a system, and motivate the study of adaptive fault tolerance mechanisms. One of the key issue is then the identification of the most critical components (agents) of the application at a certain time. For that purpose, we study the use of distinct levels of information: the system level (cpu, memory, network loads, and such like), and the application/agent level (roles or dependences). Note that our objective is somehow analog to automatic *load balancing*, yet we focus on fault tolerance.

Outline

This paper is organized as follows. Section 1 introduces the context of this work, notably the type of faults that we consider here. Section 2 introduces a new approach for dynamic and adaptive replication control. Section 3 describes the DARX framework that we developed to replicate agents. This framework introduces novel features for dynamic control of replication. Section 4 describes our approach to compute agent criticality in order to guide replication. Section 5 describes a first strategy, using agent roles, to infer and estimate agent criticality. Section 6 describes another strategy based on the inter-agent dependences. Section 7 describes the implementation of this solution and our preliminary experiments. Section 8 discusses related work before section 9 concludes this paper.

1. Context of this Work

Any software/hardware component may be subjected to faults resulting in output errors, which can lead to a deviation of its specified behaviour, ie. a failure. In distributed systems, and even more so in scalable environments, failures are unavoidable. Reliability is a vast domain: a considerable amount of techniques allow to account for the existence of faults, and to guarantee – to a certain extent – the continuation of computations. A subdomain of reliability, fault-tolerance aims at allowing a system to survive in spite of faults, ie. *after* a fault has occurred, by means of redundancy in either hardware or software architectures.

1.1. Failure Model

The most generally accepted failure classification can be found in [32]:

- A *crash* failure means a component stops producing output; it is the simplest failure to contend with.
- An *omission* failure is a transient crash failure: the faulty component will eventually resume its output production.
- A *timing* failure occurs when output is produced outside its specified time frame.
- An *arbitrary* (or *byzantine*) failure equates to the production of arbitrary output values at arbitrary times.

Given this classification, two types of failure models are usually considered in distributed environments:

- *fail-silent*, where the considered system allows only crash failures,
- *fail-uncontrolled*, where any type of failure may occur.

In this work we focus on the fail-silent model. However, in various cases our solution allows to deal with every other type of failure. Ways to extend the failure model are currently being investigated, but will not be considered in this paper.

1.2. Replication Techniques

Replication is an effective way of achieving fault-tolerance in a fail-silent environment. A replicated software component is defined as a software component that possesses a representation on two or more hosts [15]. There are two main types of replication protocols:

- 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. If the degree of replication is n, the n replicas are activated simultaneously to produce one result.

Passive replication minimizes processor utilization by activating redundant replicas only in case of failures. That is: if the active replica is found to be faulty, 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 approach but it requires a checkpoint management which remains expensive in processing time and space.

1.3. Limitations of Current Replication Techniques

Many toolkits (e.g., [15] and [41], and see Section 8) include replication facilities to build reliable applications. However, most of them are not quite suitable for implementing large-scale MAS and adaptive replication mechanisms. For example, although the 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. Besides, as each group structure is left to be designed by the user, the task of conceiving a large-scale software appears tremendously complex.

2. 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.

2.1. Dynamic Replication

First, we need a replication architecture which allows dynamic replication and dynamic adaptation of the replication policy (e.g., changing the number of replicas...). As discussed in Section 1.3, current replication architectures rarely support such dynamicity. Therefore we designed a novel replication architecture, named DARX, with such dynamic features. It will be introduced in Section 3.

2.2. Adaptive Control

Then, 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...). For dynamic applications,[1] 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.

2.3. Criticality of an Agent

The control mechanism will try to estimate which agents are the most critical within the application and this information will be regularly updated. Here we may define the *criticality* of an agent, informally as follows: the criticality of an agent, regarding an organization of agents it belongs to, is the measure of the potential impact of the failure of that individual agent on the behavior of the whole organization.

We are experimenting with several strategies 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 ? (statically or dynamically, 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.).

This will be further discussed in Section 4.3, and two strategies will be described in Sections 5 and 6.

[1] 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.

3. 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 and with different replication strategies (passive and active). Note that we are working on the integration of other replication strategies in DARX, including quorum-based strategies [2]. However, this paper does not address the design of particular strategies, but describes the infrastructure that will enable to switch to the most suitable dependability protocol. The number of replicas may be adapted dynamically. Also, and this is a novel feature, the replication strategy is reified so that the replication strategy may be dynamically changed.

Moreover, DARX has been designed to easily integrate various agent architectures, and the mechanisms that ensure dependability are kept as transparent as possible to the application.

DARX also includes an original failure detection service, based on a hierarchy of adaptive failure detectors [3]. Every failure detector is composed of two layers: a basic layer which adapts the detection to the behaviour of the underlying network, and a service layer which allows client applications to specify their required quality of the detection. For instance in our case, the latter allows to distinguish between the detection regarding a single host and that regarding a whole cluster. That work will not be further detailed here.

We consider a distributed system consisting of a finite set of n processes (or agents) that are spread throughout a network. These processes communicate only by sending and receiving messages.

3.1. DARX Architecture

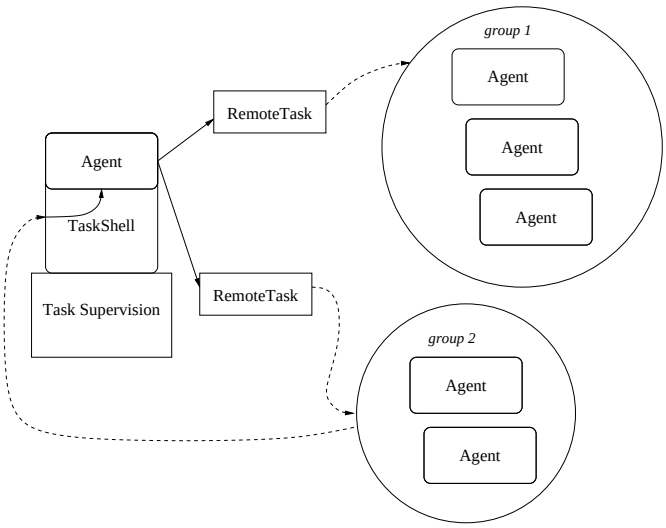

Figure 1. DARX application architecture

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.

3.2. Agent Replication

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. Every replication group has exactly one representative, named a *leader* which communicates with the other agents. All other replicas within the replication group are called *subjects*, and the point is to keep these consistent with respect to their *leader*. The *leader* acts both on the composition and the consistency maintenance of its replication group, and is responsible for reliable broadcasting within the group. 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, unregarding 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 thus 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.

4. Adaptive Control of Replication

We will now detail our monitoring and control architecture [2] for dynamically estimating the criticality of each agent and deciding on dynamic replication where and when best needed. The estimation of criticality of each agent will be used to decide, depending on available resources and the requirements of the designer of the application, which agent should be replicated and with what strategy (active or passive, how many replicas, where to create the replicas?).[3]

[2] In the following, named monitoring architecture

[3] In this paper, we will only discuss the decision about which agents to replicate, and about how many replicas. Other issues are addressed elsewhere, e.g., where to create the replicas in [20].

4.1. A Simple Scenario

As a first simple scenario, let us consider a distributed multi-agent system that helps at scheduling meetings. Each user owns a personal assistant agent which manages his/her calendar. This agent interacts with:

- the user to receive his/her 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., its 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 – any decision for replication should be done automatically and dynamically.

4.2. Monitoring Architecture

In order to compute criticality and replication decision for each agent, we propose a distributed monitoring architecture.

In most existing multi-agent architectures, the monitoring mechanism is centralized. The acquired information is typically used off-line to explain and to improve the system's behavior. Moreover, the considered application domains typically only involve a small number of agents and *a priori* well-known organizational structures. These centralized monitoring architectures are not suited for large-scale and complex systems where the observed information needs to be analyzed in real-time to adapt the multi-agent system to the evolution of its environment.

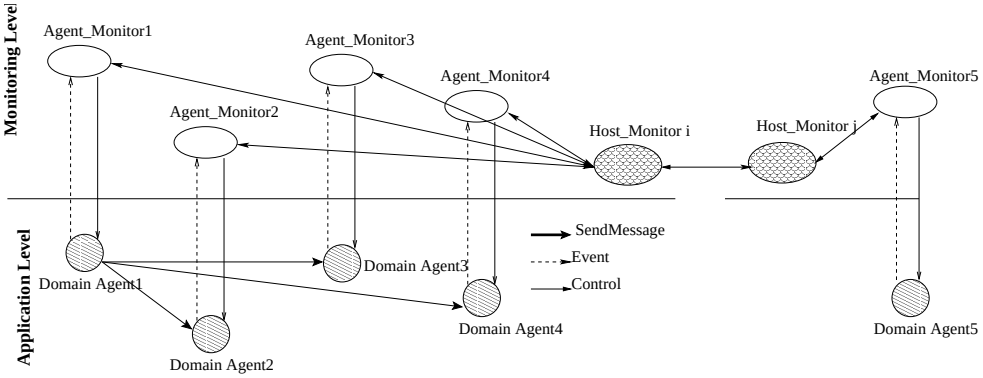

Figure 2. Distributed monitoring architecture

We thus propose a distributed monitoring architecture to improve the efficiency and robustness of MASs [29] [25]. This architecture (see Figure 2) includes:

- *agent-monitors*, one being associated to each application agent (also named domain agent),
- *host-monitors*, one being associated to each host (machine).

The monitoring agents (agent-monitors and host-monitors) are hierarchically organized. Each agent-monitor communicates only with one host-monitor. Host-monitors exchange their local information to build global information (global number of messages, global exchanged quantity of information...).

Each agent-monitor has several purposes:

- Observe its associated domain agent and its interactions,
- Update local information (this behavior depends on the strategy for criticality estimation, e.g. updating the dependences, see Section 6),
- Compute the criticality of its domain agent,
- Use the agent criticality to decide the replication strategy for its domain agent.

In order to collect the data, we use the observation module. In DARX, every host runs a `DarxServer` which comprises an observation module. This module will collect events and data, both at system-level (failure detections, increase in network latencies ...) and at application-level (message receptions, replica creations ...). All this information is used for determining criticalities.

When the criticality of a domain agent is significantly modified, the agent-monitor notifies its host-monitor. The latter informs the other host-monitors to update global information. In turn, agent-monitors are informed by their host-monitors when global information changes significantly.

Host-monitors have several purposes:

- Compute local statistics with aggregation of the agent-monitors parameters,
- Send the new statistics to the agent monitors of the local host when changes are above some given threshold.

As part of the means to supply adequate support for large-scale applications, the monitoring service follows a hierarchical organization. It comprises two levels: a local (host) and a global one (LAN), mapped upon the network topology. Our monitoring architecture includes this a third category of agents: $LAN - monitors$. These agents are not described in this paper.

4.3. Estimating the Criticality

As already discussed in Section 2.3, we are experimenting with several strategies to estimate the criticality of an agent. The issues are: What kind of information will be pertinent? And how can we obtain it? (statically or dynamically, 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.).

The first strategy we studied is based on the concept of role [12]. A role, within an organization, represents a pattern of services, activities and relations. As such, it captures some information about relative importance of roles and their dependences. Thus a role is a promising concept for estimating criticality. This strategy will be described in Section 5.

A second, alternative, strategy that we studied is based on the concept of dependence. Intuitively, the more an agent has other agents depending on it, the more it is critical in the organization. The dependences may be inferred, e.g., through the analysis of communication between agents. That second strategy will be described in Section 6.

Other strategies, e.g., using agent plans, are also currently being investigated [7], but will not be presented in this paper (for space limitation reasons).

4.4. Replication

An agent may be replicated according to:

- w_i: its criticality,
- W: the sum of the domain agents' criticality,
- rm: the minimum number of replicas which is introduced by the designer,
- Rm: the available resources which define the maximum number of possible simultaneous replicas.

The number of replicas nb_i of $Agent_i$ can be determined as follows:

$$nb_i = rounded(rm + w_i * Rm/W). \quad (1)$$

The numbers of replicas are then used by DARX to update the number of replicas of each agent.

Note that this initial formula does not address the issue of where to create the replicas nor the cost of replication. We designed a first mechanism, based on resource valuation and on a bidding mechanism using the contract net protocol (see details in [19] [20]) and 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. Another refined mechanism takes into consideration the failure probability of the replicas, considers replica allocation as an optimization problem and uses event-driven policy for updating the probability of machines failures and then of replicas failures, based on the history of machine failures. It is detailed in [8].

5. Using Roles

In this first strategy, we use the concept of role [30], because it captures the importance of an agent in an organization, and its dependences to other agents. A role, within an organization, represents a pattern of services, activities and relations. As such, it captures some information about relative importance of roles and their interdependences.[4]

It will be fulfilled by a particular agent. In a e-commerce organization, examples of roles are: service provider, client, broker, etc. An agent may play several roles at the same time, e.g., for composite services, a single agent may be at the same time both client and provider within two hierarchical organizations. And a same role may be played simultaneously by several agents.

[4] Intuitively, the CEO of an organization is usually supposed to be more important/critical role than the role of a simple worker.

5.1. Relative Importance of Roles

The designer is asked to grade the various roles along their *criticality*. In the simple scenario introduced in Section 4.1, we are considering only two roles: `Initiator` and `Participant`. Their respective weights will be set by the application designer to respectively `0.7` and `0.4` (see Table 1).

Table 1. Examples of roles and their weights

Role	*Weight*
`Meeting Initiator`	`0.7`
`Meeting Participant`	`0.4`

5.2. Types of Roles Considered

Roles are usually defined relatively to some organization, but they actually may also be defined relatively to some protocol. An example is the standard Contract Net Protocol (CNP) [40] which considers two roles: manager (which broadcasts the call for proposal), and bidder (who propose bids). In fact, protocol roles can be considered as some specific case of organizational role, where some fine grained organization is created dynamically during the scope of the protocol activation.

5.3. Monitoring Roles

We may consider two cases for monitoring roles:

- role-adoption and role-abandon is signaled by agents,
- role-adoption and role-abandon is not signaled by agents.

In the first case, such role-adoption or role-abandon events are monitored by the agent-monitor of the agent.

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

Meanwhile, as we want our role monitoring strategy to be general, we also considered the case where agents do not always signal their role-adoption and role-abandon. We just suppose that they communicate with some agent communication language such as FIPA-ACL [13], but without any assumption of signaling explicitly the protocols. Moreover, the agents may have different architectures (cognitive, reactive...).

5.4. Recognizing Roles

In this analysis, attention is focused on the precise ordering of interaction events (exchanged messages). The *role analysis* module captures and represents the set of interac-

tion events resulting from the domain agent interactions (sent and received messages). These events are then used to determine the roles of the agent. Figure 3 illustrates the various steps of this analysis which is done by an agent-monitor.

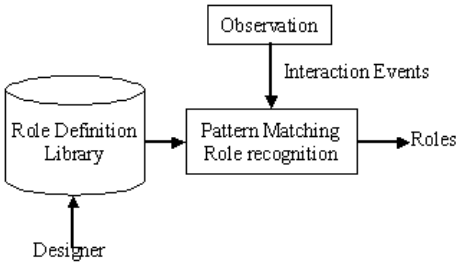

Figure 3. Roles recognition by an agent monitor

To represent the agent interactions, several methods have been proposed such as state machines and Petri nets [28]. For our application, state machines provide a well suitable representation. Each role interaction model is thus represented by a state machine. A transition represents an interaction event (sending or receiving a message). Figure 4 and Figure 5 show two examples of state machines that represent the interaction models of the roles `Initiator` and `Participant` described below.

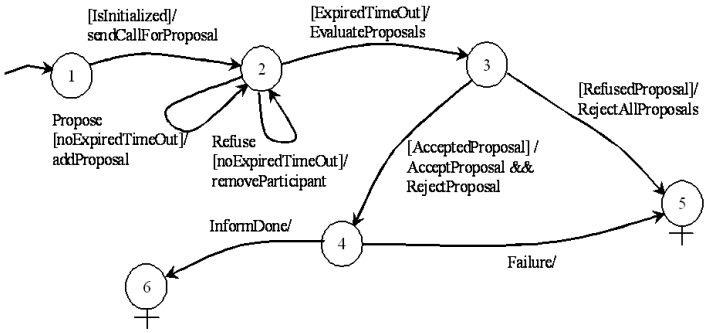

Figure 4. State machine for `Initiator` role

A library of role definitions is used to recognize the active roles. To facilitate the initialization of this library, we have introduced a role description language. Each role is represented by a set of interaction events. This language is based on a set of operators (similar to those proposed in [42], see Table 2), interaction events and variables.

Interaction events represent the exchanged messages. We distinguish two kinds of interaction events: `ReceiveMessage` and `SendMessage`. The attributes of the `SendMessage` and `ReceiveMessage` interaction events are similar to the attributes of ACL messages:

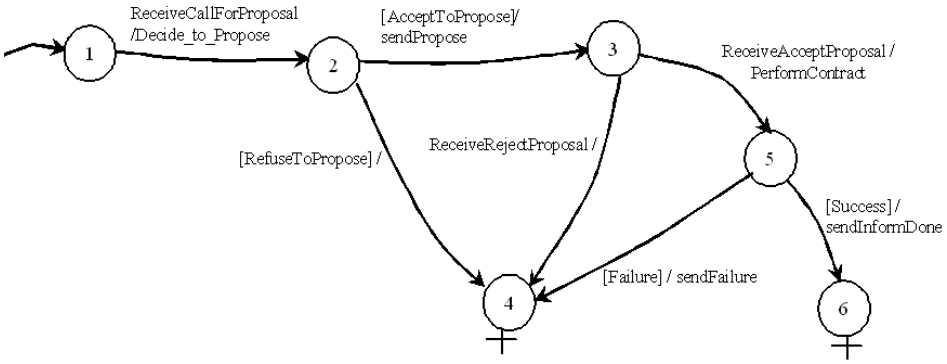

Figure 5. State machine for `Participant` role

- `SendMessage(`*Communicative act*`,` *sender*`,` *receiver*`,` *content*`,` *reply-with*`, ...)`.
- `ReceiveMessage(`*Communicative act*`,` *sender*`,` *receiver*`,` *content*`,` *reply-with*`, ...)`.

Table 2. Operators

Operator	*Interpretation*
`A.B`	Separate two consecutive events
`A|B`	Or
`A||B`	Parallel events
`(A)*`	0 time or more
`(A)+`	1 time or more
`(A)n`	n time or more
`[A]`	Facultative

In order to be able to filter various messages, we introduce the *wild card* character `?`. For example, in the interaction event `ReceiveMessage("CFP", "X", "Y", ?)`, the content is unconstrained. So, this interaction event can match any other interaction event with the communication act `CFP`, the sender `"X"`, the receiver `"Y"` and any content.

In the example of scheduling meetings, the assistant agents use the CNP (Contract Net Protocol) [13] (see Figure 6) to schedule a meeting. The interaction models of the initiator and the participant are deduced from the CNP. They are described in Figures 4 and 6.

The description represents the different steps (sent and received messages) of the role. The description of the Initiator can be interpreted as follows [13]:

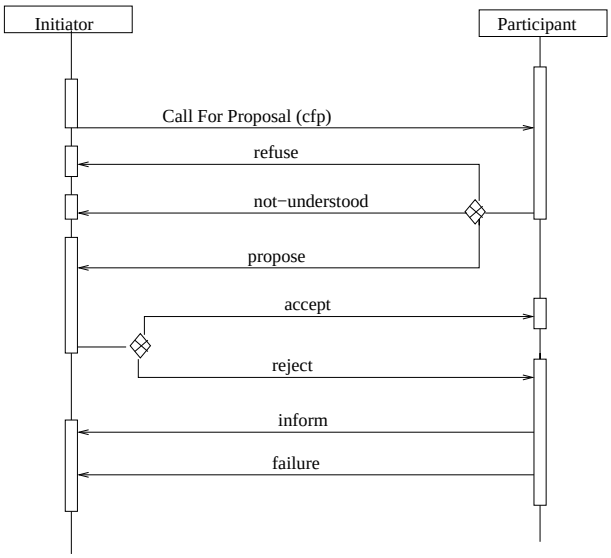

Figure 6. Contract Net Protocol

- A call for proposal (CFP) message is sent to the participants from the initiator following the FIPA CNP.
- The participants reply to the initiator with the proposed meeting times. The form of this message is either a proposal or a refusal.
- The initiator sends accept or reject messages to participants.
- The participants which agree to the proposed meeting inform the initiator that they have completed the request to schedule a meeting (confirm).

An agent may simultaneously fulfill more than one role. Each monitoring agent may therefore have one or more active role recognition process which are identified by the attributes `reply-to` and `reply-with`.

5.5. Recognition Algorithm

The process of recognition of a role is based on Algorithm 1, where R is the set of roles which match the current sequence of events. This set includes initially all the roles of the library.

Note that in many cases, roles can be deduced before the end of the associated sequence of interaction events (final state of the associated state machine). In the scheduling meetings example, the role `Initiator` may be recognized as soon as the `CFP` message is received, as it is unique to this role.

5.6. Estimating the Criticality

The recognized roles are then processed by the agent's criticality module. It relies on a table T (an example is given in Table 1) that defines the weights of roles. This table is initialized by the application designer.

Algorithm 1 Role recognition algorithm

Require: $R \leftarrow$ Roles of the library

```
for each event e do
  for each role r ∈ R do
    if current state of r accepts e then
      update current state of r
    else
      remove r from R
    end if
  end for
  if R has only one role then
    return r
  end if
end for
```

The estimation of criticality w_i of the agent $Agent_i$ which has adopted m roles (r_{i1} to r_{im}) is computed as follows:

$$w_i = aggregation(T[r_{ij}]_{j=1,m}) \tag{2}$$

6. Interdependences

In this second strategy, we use the notion of dependences between agents as a clue for estimating criticality.

Interdependence graphs [6] [36] [37] were introduced to describe the interdependences of agents. These graphs are defined by the designer before the execution of the multi-agent system. However, complex multi-agent systems are characterized by emergent structures [35] which thus cannot be statically defined by the designer.

Our approach is 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. This graph can be interpreted to define each agent criticality.

For each domain agent, we associate a node. The set of nodes (see Figure 7), named interdependence graph, is represented by a labeled oriented graph (N, L, W). N is the set of nodes of the graph, L is the set of arcs and W the set of labels.

$$N = (N_i)_i = 1, n \tag{3}$$

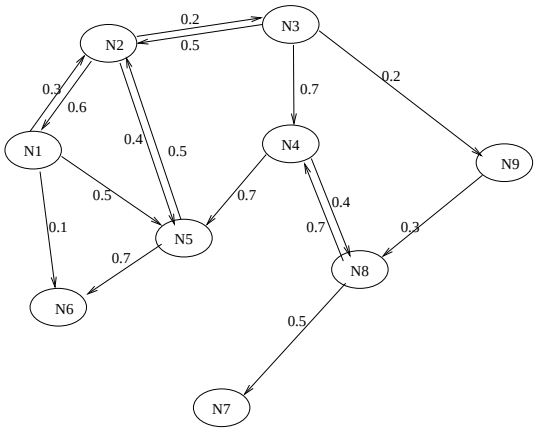

Figure 7. Example of interdependence graph

$$L = \{L_{i,j}\}_{i=1,n,j=1,n} \tag{4}$$

$$W = \{W_{i,j}\}_{i=1,n,j=1,n} \tag{5}$$

$L_{i,j}$ is the link between the nodes N_i and N_j and $W_{i,j}$ is a real number which labels $L_{i,j}$. $W_{i,j}$ reflects the importance of the interdependence between the associated agents ($Agent_i$ and $Agent_j$).

A node is thus related to a set of other nodes that may include all the nodes of a system. This set is not static: it can be modified when a new domain agent is added, or when an agent disappears, or when an agent starts interacting with another agent.

Our hypothesis is that the criticality of an agent $Agent_i$ relies on the interdependences of other agents on this agent. So, the agent is critical if the weights $wij_{j=1,n}$ are important. In this case, the failure of $Agent_i$ may be propagated to the agent $Agent_j$. It thus affects a subset of agents which form a connex component in the interdependence graph.

6.1. Adaptation of Interdependences

The interdependence graph is initialized by the designer [5]. It is then dynamically adapted by the system itself. The proposed adaptation algorithm of the interdependence graph is described below.

Several parameters may be used to define the interdependences between agents, such as communication load, executed tasks, roles of agents, or their goals. An adaptation algorithm updates the interdependence graph. This adaptation relies on local information (e.g., communication load, CPU time) and on global information, which is defined as an

[5] Note that it can actually be initialized by another strategy, based on static analysis of agent references, which will not be detailed here.

aggregation of the local information of the various agents and hosts. Currently we use communication load as the local information. Our hypothesis is that if an $Agent_i$ does not communicate with $Agent_j$ then $Agent_i$ does not depend on $Agent_j$.

Let us consider an interval of time Δt. The agent-monitors are activated every Δt. At every step, an agent-monitor executes the adaptation algorithm.

6.2. Algorithm

This algorithm relies on the global number of sent messages $NbM(\Delta t)$ which is calculated by the host-monitor:

$$NbM(\Delta t) = \sum_{i=1,n} \sum_{j=1,n \text{ ß} \neq j} NbM_{i,j}(\Delta t) \quad (6)$$

where $NbM_{i,j}(\Delta t)$ is the number of messages received by $agent_i$ from $agent_j$ during the interval of time Δt.

$W_{i,j}$ is defined as follows (Algorithm 2):

This algorithm is very simple, thus the cost of monitoring is very low. Consequently, it is useful for applications where the semantics of messages is not required. However, several applications rely on semantics of messages. So, we have proposed a new algorithm that deals with performatives as additional input information (e.g., `request` has a weight greater than `cancel`) [20].

7. Experiments

The following sections describe the experiments. Experiments were carried out on the $GRID5000$ platform (see $www.grid5000.fr/$). The scenario used for the experiments is the scheduling of meetings introduced in Section 4.1. This scenario was implemented with the platform DimaX [19] which implements our adaptive replication mechanism. It is based on DarX (see Section 3).

7.1. Performances

The proposed monitoring multi-agent architecture is very useful to implement the proposed adaptive replication mechanism. However, the monitoring cost overhead does not seem insignificant. So, our first series of experiments measures the monitoring cost in the proposed architecture. We consider, a multi-agent system with n distributed agents that execute the same scenario (a fixed set of meetings to schedule). A multi-agent system goal is to schedule all the meetings. We realized several experiments with various numbers of agents. For each n (100, 200, ..., 45000), we considered m meetings (2, 4, ..., 900). We used $\frac{n}{50}$ machines for each experiment and we repeated each experiment 20 times. We considered four cases: 1) a multi-agent system without monitoring, 2) a multi-agent system with monitoring where roles are explicitly signaled, 3) a multi-agent system where roles are implicilty recognized (see Section 5.4) and 4) a multi-agent system with monitoring based on dependences (see Section 5.5).

Algorithm 2 Basic adaptation of the interdependences

Let threshold1 a threshold defined by the designer. Its default value is 0.001.

Let $NbM_{i,j}(\Delta t)$ be the number of messages sent by $agent_i$ to $agent_j$ during some interval of time Δt.

Let n be the number of agents.

Let $NbMax(\Delta t)$ (resp. $NbMin(\Delta t)$ and $NbMin(\Delta t)$) be the maximum (resp. minimum and average) number of messages between couples of agents (i, j). More precisely:

$$NbMax(\Delta t) = \max_{i=1,n,j=1,n,i\neq j}(NbM_{i,j}(\Delta t)) \tag{7}$$

$$NbMin(\Delta t) = \min_{i=1,n,j=1,n,i\neq j}(NbM_{i,j}(\Delta t)) \tag{8}$$

$$NbAver(\Delta t) = \sum_{i=1,n,j=1,n,i\neq j}(NbM_{i,j}(\Delta t))/n*(n-1) \tag{9}$$

$\alpha(t) = 0.5$

for each j different of i **do**

  if If link $l_{i,j}$ exists **then**

    Update the weights by using the following rule:

$$W_{i,j}(t+\Delta t) = W_{i,j}(\Delta t)+(Nb_{i,j}(\Delta t)-NbAver(\Delta t))/(NbMax(\Delta t)-NbMin(\Delta t))+\alpha(t) \tag{10}$$

  end if

end for

$\alpha(t + \Delta t) = \alpha(t) - 0.01$

if $W_{i,j}(t + \Delta t) < threshold1$ **then**

  $W_{i,j}(t + \Delta t) = 0$

end if

We then measure the global execution time for scheduling the m meetings. Figure 8 shows the average global execution time for these four different monitoring approaches. We found that, for each strategy, the monitoring cost is almost a constant function. The monitoring activity does not increase very much when the number of agents (domain agents and associated monitoring agents) increases. That can be explained by the pro-

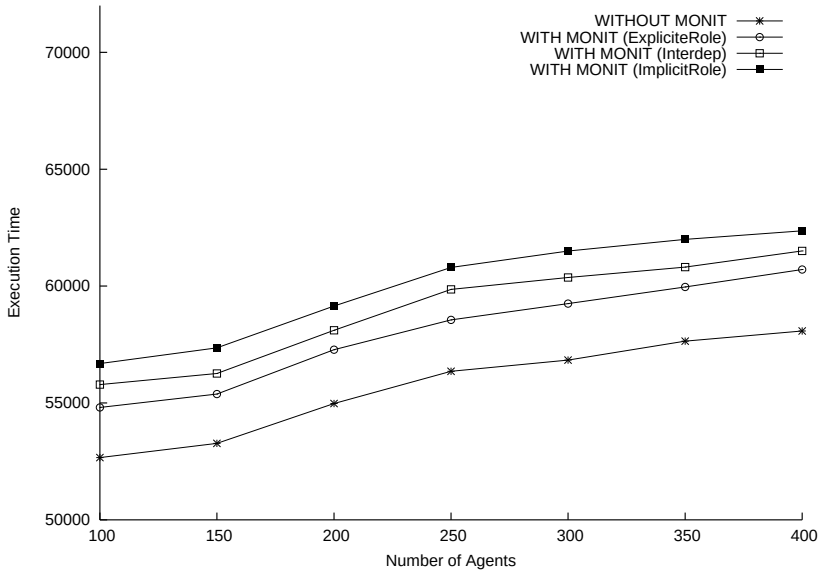

Figure 8. Fault-Tolerant Multi-Agent Systems Costs

posed optimization within the multi-agent architecture, such as the hierarchical organization of monitoring agents and the communication between the agent-monitors and host-monitors. For instance, to build the global information (global number of sent messages ...), the host-monitors communicate only if the local information changes significantly. Moreover, the host-monitors exchange local information only when there is an important change. Therefore, the number of communications between these agents is optimized.

7.2. Robustness

For this second series of experiments, we used a fault simulator. The fault simulator stops an agent currently playing the role of initiator and which has not been replicated. This simulator chooses randomly 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 measurement provides some measure of the accuracy of the strategy to identify most critical agents and protect them.

We considered a multi-agent system with 1500 agents. We run each experiment 10 minutes and we introduce 1000 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:

$$SR = \frac{NSS}{TNS} \tag{11}$$

where NSS is the number of simulations which did not fail and TNS is the total number of simulations. Let us remind that a simulation fails when the fault simulator stops a critical agent which has not been replicated.

We considered three strategies: 1) the replication is random, 2) the replication is based on role-based strategy and 3) the replication is based on the dependences algorithm.

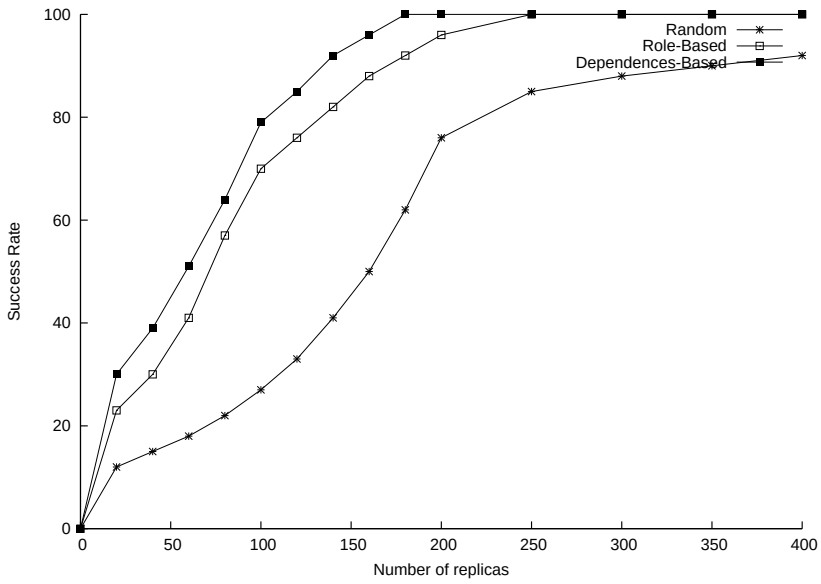

Figure 9. Rate of succeeded simulations for each number of replicas with the four strategies

Note that in our experiments, we compute the number of terminated negotiations instead of the number of successful negotiations. Figure 9 compares the success rate *SR* as a function of the number of extra replicas, for the four strategies. First, it shows that two strategies give better results than the random strategy. The strategy with dependences is the most accurate. This can be explained by fact for application domains where the roles have the same importance, the strategy based on dependence may lead to better results.

We are currently conducting 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.

8. Related Work

8.1. General Replication Architectures

In distributed computing, many toolkits include replication facilities to build reliable applications. However, many products are not flexible enough to implement an adaptive

replication [5] [14] [15]. MetaXa [14] implements in Java active and passive replication in a flexible way. Java was extended with a metalevel architecture. However, MetaXa relies on a modified Java interpreter. GARF [15] 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 mechanisms to apply these strategies.

Chameleon [23] 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 refers to the ability to dynamically adapt to the evolving fault tolerance requirements of an application. This 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. 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.

AQuA [33] is a middleware built above Ensemble [34]; it offers an adaptive fault tolerance scheme for CORBA-compliant software. AQuA provides its own built-in replication strategies and uses the monitoring features of Ensemble in order to determine and configure the right strategy for each component at runtime. Such decision making is left to AQuA's most prominent element: its Proteus dependability manager. Proteus operates in a way that is more user-independent than the pre-defined plans used in Chameleon: it samples the QoS requirements of every component it manages, and determines the right configuration with respect to information provided by monitoring entities, called observers. AQuA suffers from the same flaw as Chameleon: although the Proteus dependability manager is set as a replicated component within AQuA, thus decreasing the risk of its becoming a failure point, it still represents a bottle-neck for entire chunks of the software it supports. Also, AQuA limits its own potential by preventing more than one strategy to be applied in the same replication group.

8.2. Fault-Tolerant Multi-Agent Architectures

Several approaches address the multi-faced problem of fault tolerance in multi-agent systems. These approaches can be classified in two main categories. A first category focuses especially on the reliability of an agent within a multi-agent system. This approach handles the serious problems of communication, interaction and coordination of agents with the other agents of the system. The second category addresses the difficulties of making reliable mobile agents which are more exposed to security problems [38]. This second category is beyond the scope of this paper.

Within the family of reactive multi-agent systems, some systems offer high redundancy. A good example is a system based on the metaphor of ant nests [1]. Unfortunately:

- we cannot design any application in term of such reactive multi-agent systems. Basically we do not yet have a good methodology. Moreover, these systems are more suitable for simulations.

- we cannot apply such a simple redundancy scheme onto more cognitive multi-agent systems as this would cause inconsistencies between copies of a single agent. We need to control its redundancy.

Some work [9] offers dynamic cloning of specific agents in multi-agent systems. But their motivation is different, the objective is to improve the availability of an agent if it is too congested. The considered agents 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 [21]. 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 its beliefs on 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. Moreover, the problem solving agents themselves participate in the fault-tolerance process.

Fedoruk and Deters [10] propose to use proxies to hide the agent replication, i.e. enabling the replicas of an agent to act as a single entity towards other agents. The proxy manages the state of the replicas. All the external and internal communications of the group are redirected to the proxy. However this increases the workload of the proxy which is a quasi central entity. To make it reliable, they propose to build a hierarchy of proxies for each group of replicas. They point out the specific problems of read/write consistency, resource locking also discussed in [39]. This approach lacks flexibility and reusability in particular concerning the replication control. The experiments have been done with FIPA-OS which does not provide any replication mechanism. The replication is therefore realized by the designer before run time.

Kaminka et al. [24] 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. This monitoring approach relies on agent knowledge. The design of such multi-agent systems is very complex. Moreover, agent behaviors cannot be adaptive and the system cannot be open.

Horling et al. [22] present a distributed diagnosis system. The faults can directly or indirectly be observed in the form of symptoms by using a fault 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. The different diagnosis subsystems perform local updates on the task model. 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. [26] proposes a solution for deciding the 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 they are based on too many restrictive hypotheses to be made adaptive.

At the Free University of Amsterdam, the Intelligent Interactive Distributed Systems Group (headed by Pr. Frances Brazier), runs a research project related to our general objectives. They designed an agent architecture intended for replication, named AgentScape. They separate the public part of an agent, which is immutable and may be freely replicated, from its private part which should be kept unique. They are studying how they may combine their agent architecture and our current adaptive replication platform, named DARX [31]. Note that they do not address the issue of adaptive replication control yet.

Last, a related and much more general project is the Autonomic Computing Program of IBM. This program aims to design adaptive computing systems, being able to self adapt (self configure, self optimize, self repair...). They propose a general blueprint architecture (monitor, analyze, plan, execute). A prototype architecture/framework, named ABLE (Agent Building and Learning Environment) [4], partially implements it and provides a toolbox of components (implemented as JavaBeans) for manipulating and using monitored information (rules, neural networks, statistics...). Autonomic computing has very wide spectrum and is a long term goal. Although fault-tolerance is one of its crucial parts, the ABLE architecture by itself does not solve the problem, but the blueprint guidelines are an interesting direction.

9. Conclusion

Large-scale multi-agent systems are often distributed and must run without any interruption. To make these systems reliable, we proposed a new approach to evaluate the criticality of agents dynamically[18]. In this paper, we presented two strategies for estimating the criticality of each agent, the first one based on the notion of role, and the second one based on the notion of dependence between agents. The agent criticality is then used to replicate agents in order to maximize their reliability and availability based on available resources.

To validate the proposed approach, we realized a fault-tolerant framework (DARX) and we used a multi-agent platform (DIMA [17]) to implement multi-agent systems. The integration of DARX with the multi-agent platform DIMA provides a generic fault-tolerant multi-agent platform. In this paper, we evaluated and compared both our proposed strategies for estimating criticality (based on roles and on dependences) on a common small scenario. We believe that the obtained results are promising. Meanwhile, more experiments with various scenarios (crisis management, cooperative patrolling, cooperative control), and with alternative strategies (notably, using plans) are being conducted to further validate our approach.

Acknowledgements

The authors would like to thank all members, present and past, of the *Fault-Tolerant Multi-agent Systems* project at LIP6 for their contributions.

Moreover, the authors would like to thank he Grid 5000 project (www.grid5000.fr), which allows them to realize their experiment with a large number of agents.

References

[1] N. A. Avouris and L. Gasser, editors. *Distributed Artificial Intelligence: Theory and Praxis*, chapter Using Reactive Multi-Agent Systems in Simulation and Problem Solving. Kluwer Academic, London, 1992.

[2] F. Belkouch, M. Bui, and L. Chen. Self-stabilizing quorum systems for reliable document access in fully distributed information systems. *Studies in Informatics and Control*, 7(4):311–326, 1998.

[3] M. Bertier, O. Marin, and P. Sens. Performance analysis of a hierarchical failure detector. In *Proc. of the International Conference on Dependable Systems and Networks*, San Francisco, CA, USA, june 2003.

[4] 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.

[5] K. P. Birman. Replication and fault-tolerance in the isis system. *SIGOPS Oper. Syst. Rev.*, 19(5):79–86, 1985.

[6] C. Castelfranchi. *Decentralized AI*, chapter Dependence relations in multi-agent systems. Elsevier, 1992.

[7] A. de Luna Almeida, S. Aknine, J.-P. Briot, and J. Malenfant. Méthode de réplication basée sur les plans pour la tolérance aux pannes des systèmes multi-agents. In *Journées Francophones sur les Systèmes Multi-Agents (JFSMA'05)*, Special Issue, Revue des Sciences et Technologies de l'Information (RSTI), pages 183–186, november 2005.

[8] A. de Luna Almeida, S. Aknine, J.-P. Briot, and J. Malenfant. A predictive method for providing fault tolerance in multi-agent systems. In *IAT*, pages 226–232, 2006.

[9] K. Decker, K. Sycara, and M. Williamson. Cloning for intelligent adaptive information agents. In *ATAL'97*, LNAI, pages 63–75. Springer Verlag, 1997.

[10] A. Fedoruk and R. Deters. Improving fault-tolerance by replicating agents. In *AAMAS*, pages 373–744, Bologna, Italy, 2002.

[11] J. Ferber. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[12] J. Ferber and O. Gutknecht. Aalaadin: a meta-model for the analysis and design of organizations in multi-agent systems. In *ICMAS'98*, July 1998.

[13] FIPA. Agent communication language, foundation for intelligent physical agents, Geneva, Switzerland. http://www.cselt.stet.it/ufv/leonardo/fipa/index.htm, 1997.

[14] M. Golm. Metaxa and the future of reflection. In *OOPSLA -Workshop on Reflective Programming in C++ and Java*, pages 238–256, 1998.

[15] R. Guerraoui, B. Garbinato, and K. Mazouni. Lessons from designing and implementing GARF. In *Object-Based Parallel and Distributed Computation*, number 791 in LNCS, pages 238–256, 1995.

[16] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, 1997.

[17] Z. Guessoum and J.-P. Briot. From active objects to autonomous agents. *IEEE Concurrency*, 7(3):68–76,

1999.

[18] Z. Guessoum, J.-P. Briot, O. Marin, A. Hamel, and P. Sens. *Software Engineering for Large-Scale Multi-Agent Systems*, chapter Dynamic and Adaptative Replication for Large-Scale Reliable Multi-Agent Systems, pages 182–198. Number 2603 in LNCS. April 2003.

[19] Z. Guessoum and N. Faci. Dimax: A fault- tolerant multi-agent platform. In *In the ACM Electronic Proceedings of the ICSE'05 4th International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'05)*, Shanghai, China, May 2006. ACM Software Engineering Notes.

[20] Z. Guessoum, N. Faci, and J.-P. Briot. Adaptive replication of large-scale multi-agent systems - towards a fault-tolerant multi-agent platform. In *In the ACM Electronic Proceedings of the ICSE'05 4th International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'05)*, St Louis, MI, USA, May 2005. ACM Software Engineering Notes.

[21] S. Hagg. A sentinel approach to fault handling in multi-agent systems. In C. Zhang and D. Lukose, editors, *Multi-Agent Systems, Methodologies and Applications*, number 1286 in LNCS, pages 190–195, 1997.

[22] B. Horling, B. Benyo, and V. Lesser. Using self-diagnosis to adapt organizational structures. In *5th International Conference on Autonomous Agents*, pages 529–536, Montreal, 2001. ACM Press.

[23] Z. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. Whisnant. Chameleon: A software infrastructure for adaptive fault tolerance. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):560–579, 1999.

[24] 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.

[25] G. Karjoth. Access control with ibm tivoli access manager. *ACM Trans. Inf. Syst. Secur.*, 6(2):232–257, 2003.

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

[27] V. R. Lesser and L. Gasser, editors. *Proceedings of the First International Conference on Multiagent Systems, June 12-14, 1995, San Francisco, California, USA*. The MIT Press, 1995.

[28] H. Mazouzi, A. ElFallah-Seghrouchni, and S. Haddad. Open protocol design for complex interactions in multi-agent systems. In *AAMAS*, Bologna, Italy, July 2002. ACM.

[29] N. J. Muller. The openview enterprise management framework. *Int. Journal of Network Management*, 6(5):271–283, 1996.

[30] J. J. Odell, H. V. Dyke Parunak, and B. Bauer. Representing agent interaction protocols in UML. In *Fourth International Conference on Autonomous Agents*, pages 121–140, 2000.

[31] B. J. Overeinder, F. M. T. Brazier, and O.Marin. Fault Tolerance in Scalable Agent Support Systems: Integrating DARX in the AgentScape Framework. In *CCGRID*, pages 688–, 2003.

[32] D. Powell, editor. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag, 1991.

[33] Y. Ren. *AQuA: A Framework for Providing Adaptive Fault Tolerance to Distributed Applications*. PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[34] O. Rodeh, K. Birman, and D. Dolev. The architecture protocols in the Ensemble group communication system. In *Information Systems and Security (TISSEC)*, 2001.

[35] J. S. Sichman and R. Conte. Multi-agent dependence by dependence graphs. In *AAMAS2002*, pages 483–490, Bologna, Italy, 2002. ACM.

[36] J. S. Sichman, R. Conte, and Y. Demazeau. Reasoning about others using dependence networks. In *Actes de Incontro del gruppo AI*IA di interesse speciale sul inteligenza artificiale distribuita*, Roma, Italia, 1993.

[37] J. S. Sichman, R. Conte, and Y. Demazeau. A social reasoning mechanism based on dependence networks. In *Proceedings of ECAI'94 - European Conference on Artificial Intelligence*, Amsterdam, The Netherlands, August 1994.

[38] F. De Assis Silva and R. Popescu-Zeletin. An approach for providing mobile agent fault tolerance. In S. N. Maheshwari, editor, *Second International Workshop on Mobile Agents*, number 1477 in LNCS, pages 14–25. Springer Verlag, 1998.

[39] L. Silva, V. Batista, and J. Silva. Fault-tolerant execution of mobile agents. In *International Conference on Dependable Systems and Networks*, pages 135–143, 2000.

[40] 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.

[41] R. van Renesse, K. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.

[42] M. Wooldridge, N. Jennings, and D. Kinny. The methodology Gaia for agent-oriented analysis and design. *AI*, 10(2):1–27, 1999.