

Towards Fault-Tolerant Massively Multiagent Systems

Zahia Guessoum^{1,2}, Jean-Pierre Briot¹, and Nora Faci²

¹ LIP6, Université Pierre et Marie Curie (Paris 6)
8 rue du Capitaine Scott, 75015 Paris, France

`Zahia.Guessoum@lip6.fr`, `Jean-Pierre.Briot@lip6.fr`

² MODECO-CReSTIC - IUT de Reims, 51687 Reims Cedex 2, France
`faci@leri.univ-reims.fr`

Abstract. In order to construct and deploy massively multiagent systems, we must address one of the fundamental issues of distributed systems, the possibility of partial failures. In this paper, we discuss the issues and propose an approach for fault-tolerance of massively multiagent systems. The starting idea is the application of replication strategies to agents. As criticality of agents may evolve during the course of computation and problem solving, and as resources are bounded, we need to dynamically and automatically adapt the number of replicas of agents, in order to maximize their reliability and availability. We will describe our approach and related mechanisms for evaluating the criticality of a given agent and how to parameterize it (e.g., number of replicas). We also will report on experiments conducted with our prototype architecture (named DarX).

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), mostly based on the concept of replication, applied for instance to data bases. But, these techniques are almost always applied explicitly and statically, at design time. In such approaches, this is the responsibility of the designer of the application to identify explicitly which critical servers should be made robust and also to decide which strategies (active or passive replication...) and their configurations (how many replicas, their placement...).

New cooperative applications, e.g., air traffic control, cooperative work, and e-commerce, are much more dynamic and massive. It is thus very difficult, or even impossible, to identify in advance the most critical software components of the application. Furthermore, criticality can vary over run time, information that should be used to best allocate the scarce replication resources. Such cooperative applications are now increasingly designed as a set of autonomous and interactive entities, named agents, which interact and coordinate (multiagent system). 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, strategies. Also, new agents may also join or leave the application (open system).

In addition, such applications may be massive. And the fact that the underlying distributed system is massive makes it unstable by nature, at least in currently deployed technologies. That increases the needs for mechanism for adaptive fiabilisation of the application.

Our approach is in consequence to give the capacity to the multiagent system itself to dynamically identify the most critical agents and to decide which fiabilisation strategies to apply to them. This is analog to “load balancing” but for fiabilisation. We want to **automatically** and **dynamically** apply fiabilisation (mostly through replication mechanisms) **where** (to which agents) and **when** they are most needed. To guide the adaptive fiabilisation, we intend to use various levels of information, system level, like communication load, and application/agent level, like roles or plans.

This paper is organized as follows: Section 2 presents the related work and Section 3 presents fault tolerance concepts and replication principles. Section 4 presents the DarX framework that we developed to replicate agents. This framework introduces novel features for dynamic control of replication. Section 5 describes our approach to compute agent criticality in order to guide replication. Section 6 describes the implementation of this solution and our preliminary experiments.

2 Related Work

Several approaches address the multi-faced problem of fault tolerance in multiagent systems. These approaches can be classified in two main categories. A first category focuses especially on the reliability of an agent within a multiagent 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 [12]. This second category is beyond the scope of this paper.

Within the family of reactive multiagent systems, some systems offer high redundancy. A good example is a system based on the metaphor of ant nests. Unfortunately:

- we cannot design any application in terms of such reactive multiagent systems. Basically we do not have yet a good methodology. Moreover, these systems are more suitable for simulations.
- we cannot apply such simple redundancy scheme onto more cognitive multiagent systems as this would cause inconsistencies between copies of a single agent. We need to control its redundancy.

S. Hagg introduces sentinels to protect the agents from some undesirable states [7]. Sentinels represent the control structure of their multiagent 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 multiagent system. This sentinel handles the different agents which interact to achieve the functionality. The analysis of its beliefs on the other agents enables the sentinel to detect a fault when it occurs. Adding sentinels to multiagent systems seems to be a good approach, however the sentinels themselves represent failure points for the multiagent system. Moreover, the problem solving agents themselves participate in the fault-tolerance process.

A. Fedoruk and R. 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. 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 [13]. 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. [8] 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 multiagent systems is very complex. Moreover, the behavior of agent cannot be adaptive and the system cannot be open.

In distributed computing, many toolkits include replication facilities to build reliable application. However, many of products are not enough flexible to implement an adaptive replication. MetaXa [10] implements in Java active and passive replication in a flexible way. Authors extended Java with a reactive metalevel architecture. Like in DarX, the replication is transparent. 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.

The work by Kraus et al [9] 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.

3 Requirements and Techniques for Fault-Tolerance

3.1 Principles of Replication

Replication of data and/or computation is an effective way to achieve fault tolerance in distributed systems. A replicated software component is defined as a software component that possesses a representation on two or more hosts [4]. 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 short recovery but lead to a high overhead. Passive replication minimizes processor utilization by activating redundant replicas only in case of failures. It requires less CPU resources than the active approach but it needs a checkpoint management which remains expensive in processing time and space.

3.2 Limits of Current Replication Techniques

Many toolkits (e.g., [4] and [14]) include replication facilities to build reliable applications. However, most of them are not quite suitable for implementing 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 software appears tremendously complex.

Therefore we designed a specific and novel framework for replication, named DarX (see details in Section 5), which allows dynamic replication and dynamic adaptation of the replication policy (e.g., passive to active, changing the number of replicas). 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.

4 DarX: A Framework for Dynamic Replication

DarX [1] is a framework to design reliable distributed applications which include a set of distributed communicating entities (agents). 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. 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 such as one may dynamically change the replication strategy.

4.1 DarX Architecture

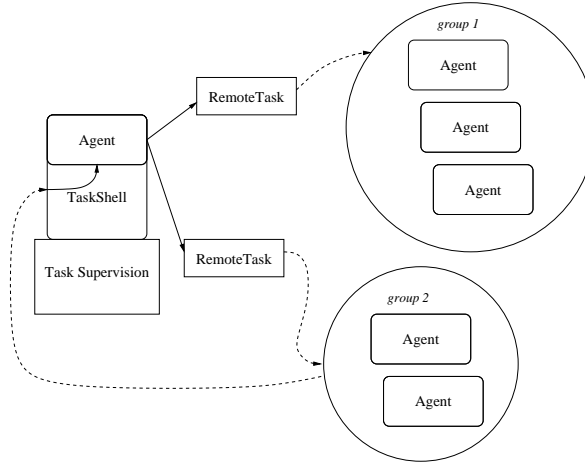


Fig. 1. DarX application architecture

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 are 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.

4.2 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. 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 (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.

5 Adaptive Control of Replication

DarX provides the needed adaptive mechanisms to replicate agents and to modify the replication strategy. Meanwhile, we cannot always replicate all the agents of the system because the available resources are usually limited. In the given example (Section 6.1), we can consider more than 100 distributed assistant agents and resources that do not allow to duplicate more than 60 agents. The problem therefore is to determine the most critical agents and then the needed number of replicas of these agents. The resources are thus allocated to critical agents.

We distinguish two cases: 1) the agent’s criticality is static and 2) the agent’s criticality is dynamic. In the first case, multiagent systems have often static organization structures, static behaviors of agents, and a small number of agents. Critical agents can be therefore identified by the designer and can be replicated by the programmer before run time.

In the second case, multiagent systems may have dynamic organization structures, dynamic behaviors of agents, and a large number of agents. So, the agents criticality cannot be determined before run time. The agent criticality can be therefore based on these dynamic organizational structures. The problem is how to determine dynamically these structures to evaluate the agent criticality?

Thus, we propose a new approach for observing the domain agents and evaluating dynamically their criticality. We will now detail our approach for dynamically evaluating criticality of each agent in order to perform dynamic replication where and when best needed.

5.1 Hypothesis and principles

We want some automatic mechanism for generality reasons. But in order to be efficient, we also need some prior input from the designer of the application. This designer can choose among several approaches of replication: static and dynamic.

In the proposed dynamic approach, the agent criticality relies on two kinds of information:

- System-level information. It will be based on standard measurements (communication load, processing time...). We are currently evaluating their significance to measure the activity of an agent.
- Semantic-level information.

Several aspects may be considered (importance of agents, independence of agents, importance of messages...). We decided to use the concept of role [11], because it captures the importance of an agent in an organization, and its dependencies to other agents.

Note that our approach is generic and that it is not related to a specific interaction language or application domain. We just suppose that agents communicate with some agent communication language such as ACL [3].

5.2 Architecture

In order to track the dynamical adoption of roles by agents, we propose a role recognition method. Our approach is based on the observation of the agent execution and their interactions to recognize the roles of each agent and to evaluate its processing activity. This is used to dynamically compute the criticality of an agent.

In most existing multiagent architectures, the observation 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 observation architectures are not suited for large-scale and complex systems where the observed information needs to be analyzed in real-time to adapt the multiagent structure to the evolution of its environment.

We thus propose to distribute the observation and monitoring mechanism to improve its efficiency and robustness. This distributed mechanism relies on a reactive-agent organization. These reactive agents have two roles:

- they observe and control the domain agents,
- they build global information and minimize communication.

These two roles are assigned to two kinds of agents: domain agent monitors (named agent-monitors) and host monitors (named host-monitors). An agent-monitor is associated to each domain agent and a host-monitor is associated to each host (see Figure 2).

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...).

Distribution and observation mechanisms are provided by the middleware. Our implementation relies on DarX (see Section 4). In this middleware, each machine runs a DarX server and each server provides an observation module.

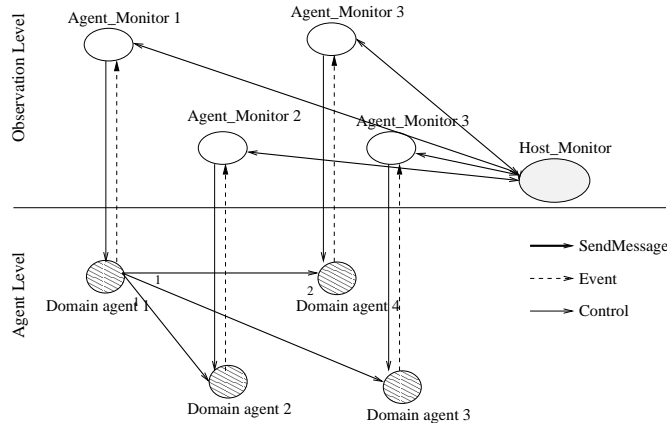


Fig. 2. Multiagent architecture

This module collects events (sent and received messages) and data. Each agent-monitor registers to its host-monitor to receive information and events related to the associated domain agent.

After each interval of time Δt , the host-monitor sends the collected events and data to the corresponding agent-monitors. An agent-monitor executes then the following behavior:

Begin

- read messages received from the host-monitor,
- activate the role analysis (see section 5.3),
- compute the domain-agent criticality (see sections 5.4 and 5.5),
- determine the number of its replicas (see section 5.6).

End

When the criticality of the 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.

The next sections describe the role analysis and activity analysis methods that we propose.

5.3 Role Analysis

We consider two cases. In the first case, each agent displays explicitly its roles or interaction protocols. The roles of each agent are thus easily deduced from its interaction events. In the second case, agents do not display their roles nor their

interaction protocols. The agent roles are deduced from the interaction events by the role analysis module.

In this analysis, attention is focused on the precise ordering of interaction events. The `role analysis` module captures and represents the set of interaction 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.

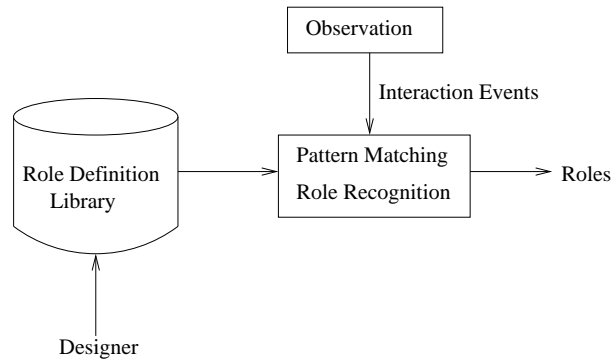


Fig. 3. Roles recognition

To represent the agent interactions, several methods have been proposed such as state machines and Petri nets. For our application, state machines provide a well suitable representation. Each role interaction model is represented by a state machine. A transition represents an interaction event (sending or receiving a message). Figure 4 shows an example of machine state that represents the interaction model of the role Initiator.

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:

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

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 (see Section 6.1), the assistant agents use the contract net protocol to schedule a meeting. The interaction models of

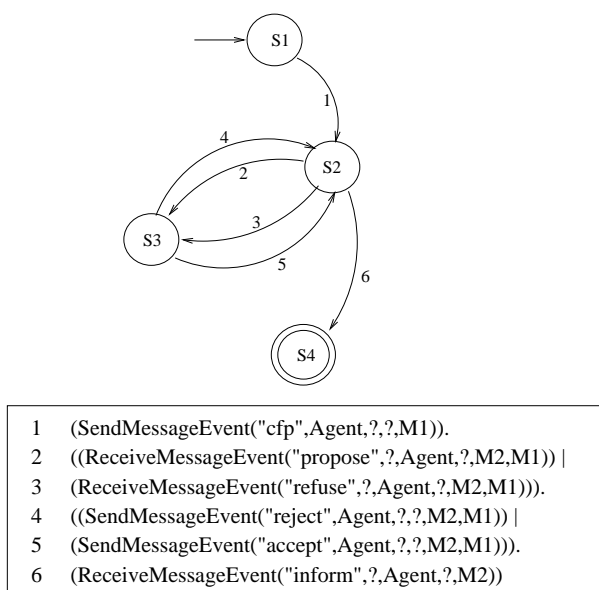


Fig. 4. Machine State for the Initiator

the initiator and the participant are deduced from the contract net protocol. The initiator is described in Figure 4. The description represents the different steps (sent and received messages) of the role. The description of the Initiator can be interpreted as follows:

- A call for proposals message is sent to the participants from the initiator following the FIPA Contract Net protocol.
- 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 agent-monitor may therefore have one or more active role recognition process.

5.4 Activity Analysis

In multiagent systems, the internal activity of agents cannot be observed, because it is private. The observation is restricted to events. To evaluate the degree of the agent activity, we use system data that are collected at the system level. We are considering two kinds of measures: CPU time and communication load. We are currently evaluating the significance of these measures as indicators of agent activity, to be useful to calculate agent criticality.

For an agent $Agent_i$ and a given time interval Δt , these measures provide:

- The used time of CPU (cp_i),
- The communication load (cl_i).

cp_i and cl_i may be then used to measure the agent degree of activity aw_i as follows:

$$aw_i(t) = (d_1 * cp_i / \Delta t + d_2 * cl_i / CL) / (d_1 + d_2) \quad (1)$$

where:

- CL is the global communication load,
- d_1 and d_2 are weights introduced by the user.

5.5 Agent Criticality

The analysis of events and measures (system data and interaction events) provides two kinds of information: the roles and the degree of activity of each agent. This information is then processed by the agent's criticality module. The latter relies on a table T that defines the weights of roles. This table is initialized by the application designer.

The criticality of the agent $Agent_i$ which fulfills the roles r_{i1} to r_{im} is computed as follows:

$$w_i(t) = (a_1 * aggregation(T[r_{ij}]_{j=1,m}) + a_2 * aw_i(t)) / (a_1 + a_2) \quad (2)$$

Where a_1 and a_2 are the weights given to the two kinds of parameters (roles and degree of activity). They are introduced by the designer.

For each Agent A_i , its criticality w_i is used to compute the number of its replicas.

5.6 Replication

An agent is 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:

$$nb_i(t) = rounded(rm + w_i(t) * Rm / W) \quad (3)$$

The number of replicas is then used to update the number of replicas of the domain agent. When this number of replicas increases, the associated monitor-agent sends a call for proposal to all the host-monitors. Each host monitor sends

a proposal if it has a non-allocated resource. The agent-monitor then selects the needed resources. The selection process relies mainly on the communication cost between the two hosts. Moreover, when the number of replicas decreases, the agent-monitor selects $nb_i(t + \Delta t) - nb_i(t)$ of its replicas and sends a message to the corresponding host-monitors to cancel the location of the resources.

6 Experiments

To validate the proposed approach, we realized an integration of DarX with the multiagent platform DIMA [5]. This integration provides a generic fault-tolerant multiagent platform. In order to validate this fault-tolerant multiagent platform, we carried out several experiments.

Measures were obtained using a set of 20 Pentium PCs running linux with JDK1.2 and linked by a fast Ethernet (10Mb/s).

6.1 A First and Simple Example

We consider the example of a distributed multiagent system that helps at scheduling meetings. Each user has a personal assistant agent which manages its calendar. This agent interacts with:

- the user to receive its meeting requests and the associated information (a title, a description, possible dates, participants, priority, etc.),
- the other agents of the system to schedule meetings.

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

6.2 Performances

Monitoring is a useful mechanism. However, its cost seems important. Thus, our first experiment measures the monitoring cost in the proposed architecture. We consider, a multiagent system with n distributed agents that execute the same scenario, each agent has a fixed scenario. The number of agents (n) is an important factor because our framework was specially designed for massively multiagent systems. For each n (100, 150, ..., 400), we realized two kinds of measures (with and without monitoring). Each machine runs 20 agents. We use thus the 20 machines for the last experiment ($n=400$) and we repeat each experiment 10 times.

Figure 5 gives the average execution time for each n . It shows that the monitoring cost is almost a constant function. It does not increase with the number of agents. That can be explained by the proposed optimization in the multiagent architecture such as the communication between the agent-monitors and host-monitors. For instance, to build global information (global communication load ...), the host-monitors communicate only if the local information changes.

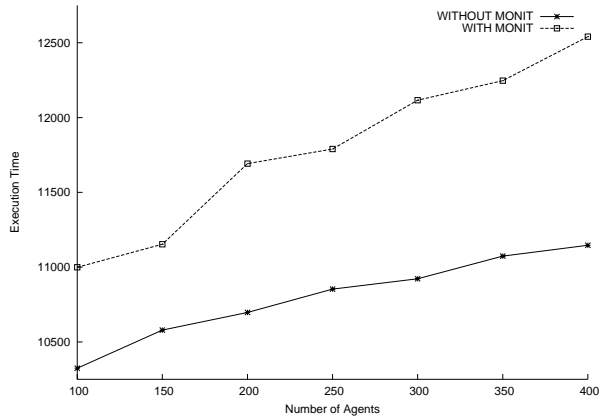


Fig. 5. Monitoring cost

6.3 Robustness

We considered 100 assistant agents which are distributed on 10 machines. These agents may have two roles: initiator and participant. We define a scenario with 20 initiators. These agents have thus two roles: initiator and participant. Moreover, we consider 80 agents which have one role: participant. These agents are thus less critical.

We run each experiment 10mn and we introduced 100 faults. To simulate the presence of faults, we implemented a failure simulator randomly stopping the thread of an agent (chosen randomly). We repeated several times the experiments with a variable number of extra resources (number of replicas that can be used).

We consider here the following variables:

$$ReplicationRate = \frac{NumberOfExtraReplicas}{NumberOfAgents} \quad (4)$$

and the rate of simulations which succeeded (i.e., which did not fail):

$$SuccessRate = \frac{NumberOfSuccessfulSimulations}{NumberOfSimulations} \quad (5)$$

Figure 6 shows the success rate as a function of the replication rate.

From these experiments, we notice that the rate of succeeded simulation becomes interesting (greater than 90%) when the number of extra resource reaches the number of critical agents. Therefore, the number of extra resources should be at least equal to the number of critical agents. Each critical agent should thus have at least one replica.

Although preliminary, we believe these results are encouraging. Note that the results are similar for the two replication strategies.

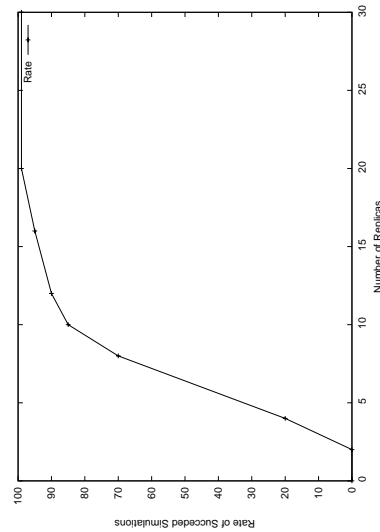


Fig. 6. Rate of succeeded simulations for each number of replicas

7 Conclusion

Massively multiagent systems are often distributed and must run without any interruption. To make these systems reliable, we proposed a new approach to evaluate dynamically the criticality of agents [6]. This approach is based on the concepts of roles and degree of activity. 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 multiagent platform (DIMA [5]) to implement multiagent systems. The integration of DarX with the multiagent platform DIMA provides a generic fault-tolerant multiagent platform. In order to validate this fault-tolerant multiagent platform, two small applications have been developed (meetings scheduling and crisis management system). They are intended at evaluating our model and architecture viability.

In the realized experiments the role definition library (see Section 5.3) and the role criticality are defined by the designer before runtime. It seems interesting to improve the monitor-agent behavior to learn new roles and learn their criticality by observing agent interactions.

Moreover, the obtained results are interesting and promising. However, more experiments with real-life applications are needed to validate the proposed approach.

Acknowledgement

The authors would like to thank the members of *Fault-Tolerant Multi-agent Systems* project for many useful discussions and suggestions.

References

1. M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *the International Conference on Dependable Systems and Networks*, Washington, USA, 2002.
2. A. Fedoruk and R. Deters. Improving fault-tolerance by replicating agents. In *AAMAS2002*, Boulogna, Italy, 2002.
3. FIPA. Specification. part 2, agent communication language, foundation for intelligent physical agents, geneva, switzerland. <http://www.cselst.stet.it/ufv/leonardo/fipa/index.htm>, 1997.
4. R. Guerraoui, B. Garbinato, and K. Mazouni. Lessons from designing and implementing garf. In *Proceedings Objects Oriented Parallel and Distributed Computatio*, volume LNCS 791, pages 238–256, Nottingham, 1989.
5. Z. Guessoum and J.-P. Briot. From active objects to autonomous agents. *IEEE Concurrency*, 7(3):68–76, 1999.
6. Z. Guessoum, J.-P. Briot, and S. Charpentier. Dynamic and adaptative replication for large-scale reliable multi-agent systems. In *Proceedings of the ICSE'02 First International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'02)*, Orlando FL, U.S.A., may 2002. ACM.
7. 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. Springer Verlag, 1997.
8. 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.
9. S. Kraus, V.S. Subrahmanian, and N. Cihan Tacs. Probabilistically survivable MASs. In AAAI, editor, *IJCAI'03*, 2003.
10. M. Golm. Metaxa and the future of reflection. In *OOPSLA -Workshop on Reflective Programming in C++ and Java*, pages 238–256. Springer Verlag, 1998.
11. J. J. Odell, H. V. Dyke Parunak, and B. Bauer. Representing agent interaction protocols in UML. In *Fourth International Conference on Autonomous Agents*, 2000.
12. 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.
13. 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.
14. R. van Renesse, K. Birman, and S. Maffei. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.