Dynamic and Adaptive Replication for Large-Scale Reliable Multi-Agent Systems

Zahia Guessoum, Jean-Pierre Briot, Olivier Marin, Athmane Hamel, and Pierre Sens

OASIS and SRC teams, LIP6, Université Pierre et Marie Curie (Paris 6)
8 rue du Capitaine Scott, 75015 Paris, France
Zahia.Guessoum@lip6.fr, Jean-Pierre.Briot@lip6.f, Olivier.Marin@lip6.fr, hamel@poleia/lip6.fr, Pierre.Sens@lip6.fr
http://www-src.lip6.fr/darx/

Abstract. In order to make large-scale multi-agent systems reliable, we propose an adaptive application of replication strategies. Critical agents are replicated to avoid failures. As criticality of agents may evolve during the course of computation and problem solving, we need to dynamically and automatically adapt the number of replicas of agents, in order to maximize their reliability and availability based on available resources. We are studying an approach and mechanisms for evaluating the criticality of a given agent (based on application-level semantic information, e.g. messages intention, and also system-level statistical information, e.g., communication load) and for deciding what strategy to apply (e.g., active or passive replication) and how to parameterize it (e.g., number of replicas).

1 Introduction

A multi-agent system is a set of autonomous and interactive entities called agents [1]. Recent real-life applications (e.g., intensive care monitoring, air traffic control and process control) are often distributed at large scale and must run continuously without any interruption. As distributed systems, multi-agent systems are exposed to possibility of failure of their hardware and/or software components. The failure of one component can often evolve into the failure of the whole system. To make these large-scale systems reliable, an obvious solution is the introduction of redundancy: duplication (replication) of the critical components.

Replication mechanisms have been successfully applied to various distributed applications [10], e.g. data-bases. But in most cases, replication is decided by the programmer and applied statically, before the application starts. This works fine because the criticality of components (e.g., main servers) may be well identified and remains stable during the application session.

Opposite to that, in the case of dynamic and adaptive multi-agent applications, the criticality of agents may evolve dynamically during the course of computation. Moreover, the available resources are often limited. Thus, simultaneous replication of all the components of a large-scale system is not feasible.

Our idea is thus to **automatically** and **dynamically** apply replication mechanisms **where** (to which agents) and **when** it is most needed. In this paper, we will introduce our approach to this objective and the software architecture to help at building reliable multi-agent systems.

This paper is organized as follows: Section 2 presents fault tolerance concepts and replication principles. Section 3 introduces a new approach of dynamic and adaptive control of replication. 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 Requirements and Techniques for Fault-Tolerance

2.1 A First and Simple Example

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

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

If the assistant agent of one important participant (initiator or prime participant) in a meeting fails (e.g., his 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.

2.2 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 [9]. 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 needs a checkpoint management which remains expensive in processing time and space.

2.3 Limits of Current Replication Techniques

Many toolkits (e.g., [9] and [19]) include replication facilities to build reliable applications. However, most of them are not quite suitable for implementing large-scale, adaptive replication mechanisms. For example, although 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.

Therefore we designed a specific and novel framework for replication, named DarX (see details in Section 4), 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.

3 Towards Dynamic Replication and Adaptive Control

Several solutions have been proposed to replicate distributed systems. These solutions are often used by the designer to replicate the system components before run time. The number of replicas and the replication strategy are explicitly and statically defined by the designer before run time. However, these solutions are not suitable to multi-agent systems. The solution we propose is mainly characterized by dynamic replication and adaptive control.

3.1 Dynamic Replication

The two replication strategies (active and passive) can be used to replicate agents. Active replication provides a fast recovery delay. So, it is dedicated to applications with real-time constraints. Moreover, passive replication provides a low overhead under failure but it does not provide short recovery delays. So, the choice of the most suitable strategy relies on the environment context. Active replication must be chosen when the failure rate becomes too high or when the application has real-time constraints. Otherwise, passive replication is most suitable.

In most multi-agent applications, the environment context is very dynamic.

So, the choice of the replication strategy of each component, which relies on a part of this environment, must be determined dynamically and adapted to the environment changes.

Moreover, a multi-agent system component which can be very critical at a moment can loose its criticality later. If we consider the replication cost which is very high, the number of replicas of these components must be therefore dynamically updated.

Thus, the solution we propose allows to dynamically adapt the number of replicas and the replication strategy. This solution is provided by the framework DarX (see Section 4).

3.2 Adaptive Control

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 2.1), we can consider more than 100 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.

We distinguish two cases: 1) the agent's criticality is static and 2) the agent's criticality is dynamic. In the first case, multi-agent 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, multi-agent 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. This approach is based on two kinds of information: semantic-level information and system-level information (see Section 5).

4 DarX: A Framework for Dynamic Replication

DarX 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 [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 such as one may dynamically change the replication strategy.

4.1 System model

We consider a distributed system consisting of a finite set of n processes (or agents) $\Pi = \{p_1, p_2, \ldots, p_n\}$ that are spread throughout a network. These processes communicate only by sending and receiving messages.

Processes can fail by crashing only, and this crash is permanent. Our algorithm does not need synchronized clocks, but local clocks must not drift with respect to real time. They must measure time intervals accurately.

We consider the model of partial synchrony proposed by Chandra and Toueg in [3]. This model stipulates that, for every execution, there are bounds on process speeds and on message transmission times. However, these bounds are not known and they hold only after some unknown time.

4.2 DarX Architecture

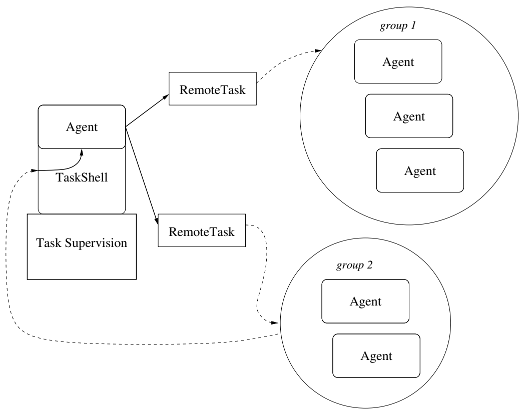


Fig. 1. DarX application architecture

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 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.

4.3 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.

4.4 Failure of the DarX Server

Assuming that a DARX server will never fail contradicts the very existence of DARX. Indeed, software fault tolerance becomes useless in a zero-failure context. Besides, in our solution the fault tolerance protocols are agent-dependent, and not place-dependent, i.e. the mechanisms built for providing the continuity of the computation are integrated in the replication groups, and not in the servers. For every agent there corresponds a replication group: a set of software entities - replicas - which contains 0 to N followers/backups kept consistent with respect to a single leader. The consistency protocol as well as the replication group information are handled by the TaskShell which wraps every replica. A machine crash - server failure - is handled in three steps within every replication group:

- detection of an eventual failure within the group,
- evaluation of the context: new criticality, leader failure, ...
- recovery: If the missing replica was the group leader, a new one is elected. In the other case, it depends on the evaluation; a new follower/backup may or may not be instantiated.

Obviously, if a leader without any follower/backup fails, then it will not be recovered. This derives from the original assumptions we made: the criticality of an agent evolves during the computation, and there are phases when an agent need not be fault-tolerant.

4.5 Measurements

Our first measurements of DarX are very promising. We evaluated several costs and made comparisons with other systems (see [14]).

In this paper, we just show the cost of sending a message to a replication group using the active replication strategy. Figure 2 presents three configurations with different replication degrees. In the RD-1 configuration, the task is local and not replicated. In the RD-2 (resp. RD-3) configuration, there is one (resp. two) replica(s); the leader being on the sending host and the other replica(s) residing on one (or two) distinct remote host(s).

Measures were obtained using JDK1.1.6 on a set of UltraSparc II 333 MHz linked by a fast Ethernet (100Mb/s).

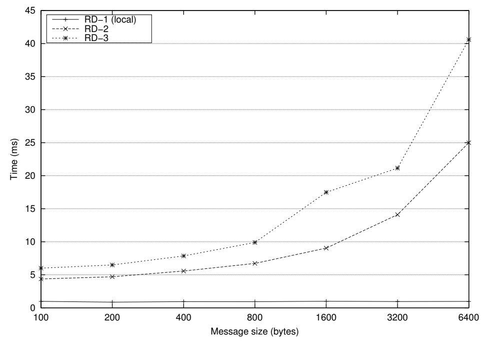


Fig. 2. Communication cost as a function of the replication degree

5 Adaptive Control of Replication

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 [16][13], because it captures the importance of an agent in an organization, and his dependencies to other agents.

Note that our approach is generic and that it is not related to a specific interaction language or application domain. Also agents can be either reactive or cognitive. We just suppose that they communicate with some agent communication language such as ACL [8] and KQML [7].

5.2 Example

The application designer will manually evaluate criticality of the roles, corresponding to their "importance" in the organization and in the computation.

In the example introduced in section 2.1, we are considering two roles: Initiator and Participant [7]. 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

Roles	Weights
Initiator	0.7
Participant	0.4

5.3 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 his processing activity. This is used to dynamically compute the criticality of an agent.

The basic architecture controlling the replication of agents is shown in Figure 4.

In order to collect the data, we associate an observation module to each DarxServer on each machine (see Section 4.2). This module will collect events and dara (provided by DarxServer). A monitoring agent (called Mi and Mj in Figure 3) is then associated to each agent (the leader of replica group). This monitoring agent realizes the role analysis and activity analysis of the associated agent by considering his sent and received interaction events, and his system data. He then uses the obtained roles and degree of activity to compute the agent criticality.

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

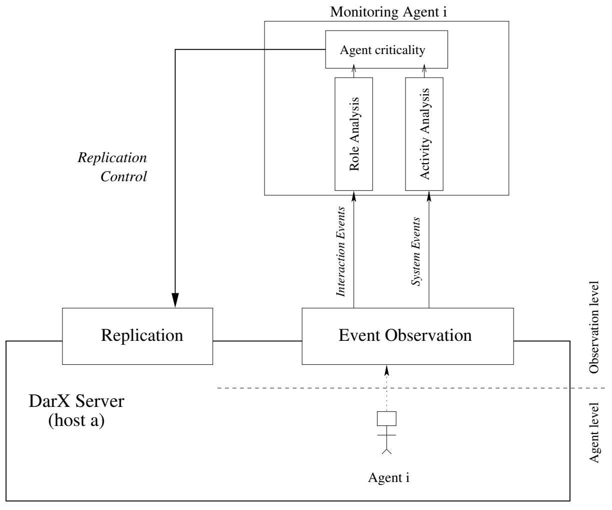


Fig. 3. General architecture for replication control

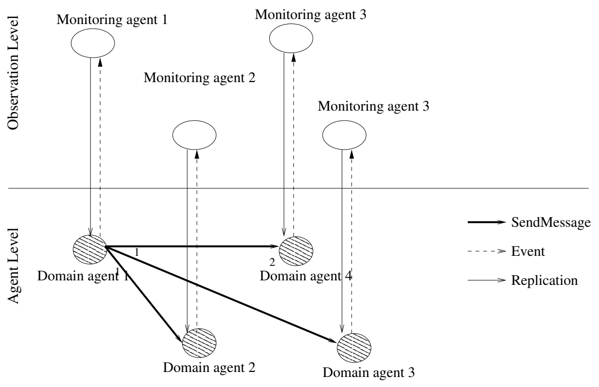


Fig. 4. General architecture for replication control

5.4 Role Analysis

We consider two cases. In the first case, each agent displays explicitly his roles or interaction protocols. The roles of each agent are thus easily deduced from his 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 (exchanged messages). The role module captures and represents the set of interaction events resulting from the domain agent interactions (sent and received messages).

We associate to each agent an entity that analyses the associated interaction events. This analysis determines the roles of the agent. Figure 5 illustrates the various steps of this analysis.

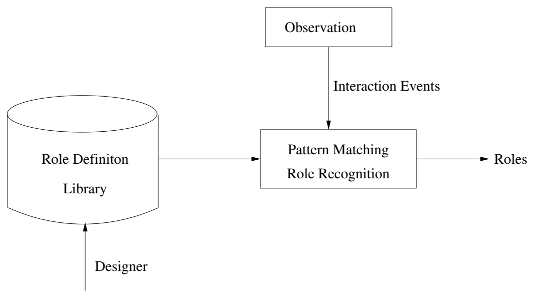


Fig. 5. Roles recognition

To represent the agent interactions, several methods have been proposed such as state machines and Petri nets [5]. For our application, state machines provide a well suitable representation. Each role interaction model is represented by an augmented transition network (ATN) [20]. A transition represents an interaction event (sending or receiving a message).

A library of roles definition 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 [13], 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:

– 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, the assistant agents use the contract net protocol [8] (see Figure 6) to schedule a meeting. The interaction model of

Table 2. Operators

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

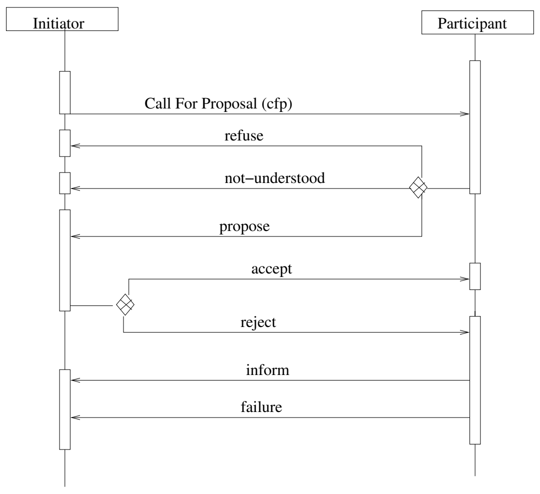


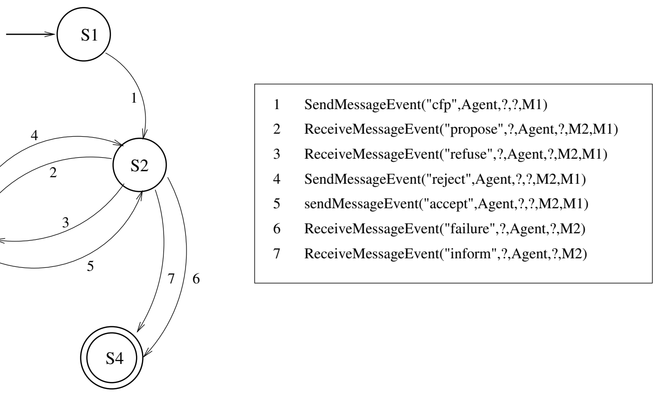
Fig. 6. Contract net protocol

the initiator role is deduced from the contract net protocol. It is described in Figure 3.

This description represents the different steps (sent and received messages) of the Initiator. It can be interpreted as follows [8].

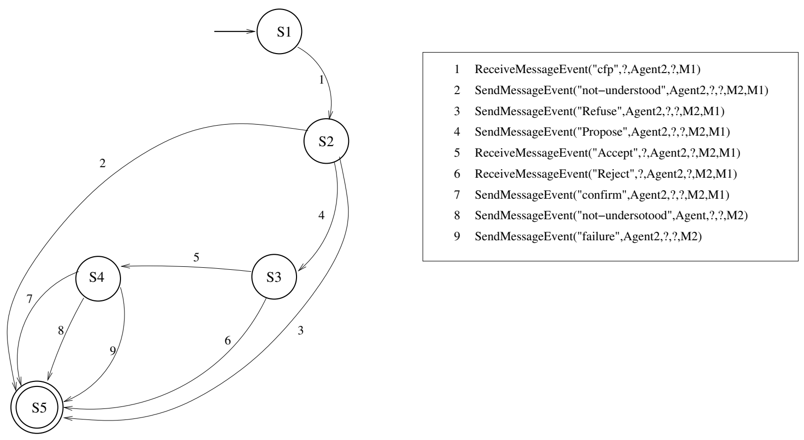
- A call for proposals message is sent to the participants from the initiator following the FIPA Contract Net protocol.
- The participants reply to the initiator with the proposed meeting times. The form of this message is either a proposal or a refusal.
- The initiator sends accept or reject messages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

Figure 7 and Figure 8 show examples of ATN that represent the interaction models of the roles Initiator and Participant described below.



1	SendMessageEvent("cfp",Agent,?,?,M1)
2	ReceiveMessageEvent("propose",?,Agent,?,M2,M1)
3	ReceiveMessageEvent("refuse",?,Agent,?,M2,M1)
4	SendMessageEvent("reject",Agent,?,?,M2,M1)
5	sendMessageEvent("accept",Agent,?,?,M2,M1)
6	ReceiveMessageEvent("failure",?,Agent,?,M2)
7	ReceiveMessageEvent("inform",?,Agent,?,M2)

Fig. 7. Machine State for the Initiator



1	ReceiveMessageEvent("cfp",?,Agent2,?,M1)
2	SendMessageEvent("not−understood",Agent2,?,?,M2,M1)
3	SendMessageEvent("Refuse",Agent2,?,?,M2,M1)
4	SendMessageEvent("Propose",Agent2,?,?,M2,M1)
5	ReceiveMessageEvent("Accept",?,Agent2,?,M2,M1)
6	ReceiveMessageEvent("Reject",?,Agent2,?,M2,M1)
7	SendMessageEvent("confirm",Agent2,?,?,M2,M1)
8	SendMessageEvent("not−undersotood",Agent,?,?,M2)
9	SendMessageEvent("failure",Agent2,?,?,M2)

Fig. 8. Machine State for the Participant

Note that in many cases, roles can be deduced before the end of the associated sequence of interaction events (final state of the associated ATN). 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.

5.5 Activity Analysis

In multi-agent 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 mesures: 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 mesures 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:

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

where:

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

5.6 Agent Criticality

The analysis of events and mesures (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 (an example is given in table 1) 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:

$$w_i = (a_1 * aggregation(T[r_{ij]}]_{j=1,m}) + a_2 * aw_i)/(a1 + a2) \tag{2}$$

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, his criticality w_i is used to compute the number of his replicas.

5.7 Replication

An agent is replicated according to:

- w_i: his 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:

$$nb_i = rounded(rm + w_i * Rm/W) \tag{3}$$

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

6 Experiments

We made some preliminary experiments using the scenario of agents scheduling their meetings, as introduced in Section 2.1.

Agents take randomly roles of Initiator, choose Participants for scheduling meetings or remain inactive (without any role). Several meetings are scheduled simultaneously. The number of critical agents (which can be either Initiator or Participant) is 60% of the number of agents.

In order to simulate the presence of faults, we implemented a failure simulator randomly stopping the thread of an agent (chosen randomly).

Measures were obtained using a set of Pentium IV/2GHz PCs running linux with JDK1.2 and linked by a fast Ethernet (10Mb/s). We considered 100 agents which are distributed on 10 machines. We run each experiment 10 mn and we introduce 100 faults. 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:

$$ReplicationRate = \frac{NumberOfExtraReplicats}{NumberOfAgents} \tag{4}$$

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

$$SuccessRate = \frac{NumberOfSuccessfulSimulations}{NumberOfSimulations} \tag{5}$$

Figure 9 shows the success rate as a function of the replication rate.

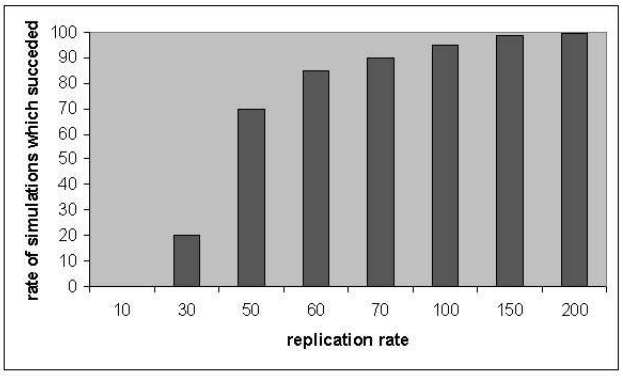


Fig. 9. Rate of succeeded simulations for each rate replication

From these experiments, we found that the number of extra resources should be at least equal to the number of critical agents.

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

7 Related Work

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 [17]. 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. Unfortunately:

– we cannot design any application in term of such reactive multi-agent 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 multi-agent systems as this would cause inconsistencies between copies of a single agent. We need to control its redundancy.

Some work [4] 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 agents considered seem to have only functional tasks (with no changing state) and fault-tolerance aspects are not considered.

S. Hagg introduces sentinels to protect the agents from some undesirable states [12]. Sentinels represent the control structure of their multi-agent system. They need to build models of each agent and monitor communications in order to react to faults. Each sentinel is associated by the designer to one functionality of the multi-agent system. This sentinel handles the different agents which interact to achieve the functionality. The analysis of his believes on the other agents enables the sentinel to detect a fault when it occurs. Adding sentinels to multi-agent systems seems to be a good approach, however the sentinels themselves represent failure points for the multi-agent system. Moreover, the problem solving agents themselves participate in the fault-tolerance process.

A. Fedoruk and R. Deters [6] 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 [18]. 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.

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 [15] 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 [9] 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.

Pre-defined library of replication mechanisms enable more flexibility to be obtained. The users or the system themselves can tailor their own fault tolerance mechanisms to suit their needs by using these replication mechanisms and updating the strategy to the evolution of their environment. Indeed, the system must have the ability to observe the agents to determine their criticality and to replicate them according to this criticality (see Section 5).

8 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 dynamically the criticality of agents. 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 multi-agent framework (DIMA [11]) to implement multi-agent systems. The integration of DarX with the multi-agent platform DIMA provides a generic fault-tolerant multi-agent platform. In order to validate this fault-tolerant multi-agent platform, two small applications have been developed (meetings scheduling and crisis management system). They are intended at evaluating our model and architecture viability. The obtained results are interesting and promising. However, more experiments with real-life applications are needed to validate the proposed approach.

References

1. N. A. Avouris and L. Gasser. *Distributed Artificial Intelligence: Theory and Praxis*, chapter Object-Oriented Concurrent Programming and Distributed Artificial Intelligence, pages 81–108. Kluwer Academic Publisher, 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. T.D. Chandra and S. Toueg. Unreliable failure detector for asynchronous distributed systems. In *10 th Annual ACM Symposium on Principles of Distributed Computing*, pages 325–340, 1992.

4. K. Decker, K. Sycara, and M. Williamson. Cloning for intelligent adaptive information agents. In *ATAL'97*, LNAI, pages 63–75. Springer Verlag, 1997.
5. A. El Fallah-Seghrouchni, S. Haddad, and H. Mazouzi. Protocol engineering for multiagent interactions. In *MAAMAW'99*, number 1647 in LNAI, pages 128–135. Springer Verlag, 1999.
6. A. Fedoruk and R. Deters. Improving fault-tolerance by replicating agents. In *AAMAS2002*, Boulogna, Italy, 2002.
7. T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an agent communication language. In *Third international conference on information and knowledge management.* ACM Press, November 1994.
8. FIPA. Specification. part 2, agent communication language, foundation for intelligent physical agents, geneva, switzerland. http://www.cselt.stet.it/ufv/leonardo/fipa/index.htm, 1997.
9. R. Guerraoui, B. Garbinato, and K. Mazouni. Lessons from designing and implementing *garf*. In *Proceedings Objects Oriented Parallel and Distributed Computations*, volume LNCS 791, pages 238–256, Nottingham, 1989.
10. R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, April 1997.
11. Z. Guessoum and J.-P. Briot. From active objects to autonomous agents. *IEEE Concurrency*, 7(3):68–76, 1999.
12. 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.
13. N. Jennings M. Wooldridge and D. Kinny. The methodology gaia for agent-oriented analysis and design. *AI*, 10(2):1–27, 1999.
14. O. Marin, P. Sens, J.-P. Briot, and Z. Guessoum. Towards adaptive fault-tolerance for distributed multi-agent systems. In *ERSADS'2001*, pages 195–201, 2001.
15. M.Golm. Metaxa and the future of reflection. In *OOPSLA - Workshop on Reflective Programming in C++ and Java*, pages 238–256. Springer Verlag, 1998.
16. James J. Odell, H. Van Dyke Parunak, and Bernhard Bauer. Representing agent interaction protocols in uml. In Paolo Ciancarini and Michael J. Wooldridge, editors, *Agent-Oriented Software Engineering*, number 1957 in LNCS, pages 121–140. Springer Verlag, 2000.
17. 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.
18. 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.
19. R. van Renesse, K. Birman, and S. Maffeis. Horus: A flexible group communication system. *CACM*, 39(4):76–83, 1996.
20. W. Woods. Transition network grammar for natural language analysis. *Communication of Association of Computing Machinery*, 10(13):591–606, 1970.