Comparative Study of the Various Decision/Information Strategies and Models for Controlling Replication-Based Fault Tolerant Multi-Agent Systems

INO CARE III

14 September 2007

Jean-Pierre Briot\textsuperscript{1,3} Zahia Guessoum\textsuperscript{1,2} Samir Aknine\textsuperscript{1} Alessandro Almeida\textsuperscript{1}
Nora Faci\textsuperscript{2} Jacques Malenfant\textsuperscript{1} Olivier Marin\textsuperscript{1} Pierre Sens\textsuperscript{1}

\textsuperscript{1} Laboratoire d’Informatique de Paris 6 (LIP6), Universit\'e Paris 6 - CNRS
Case 169, 4 place Jussieu, 75252 Paris Cedex 05, France
\textsuperscript{2} MODECO-CRoSTIC, IUT de Reims, 51687 Reims Cedex 2, France
\textsuperscript{3} Laborat\’orio de Engenharia de Software (LES), DI, PUC-Rio
Rua Marqu\’es de S\’ao Vicente 225, G\’avea, Rio de Janeiro, RJ 22451-900, Brazil

\{Jean-Pierre.Briot, Zahia.Guessoum, Samir.Aknine, Alessandro.Luna-Almeida,
faci@leri.univ-reims.fr

Abstract

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

The possibility of partial failures is a fundamental characteristic of distributed applications. The fault tolerance research community has developed solutions (algorithms and architectures), some more curative e.g., based on exception handling and cooperative recovery [32], and some more preventive, notably based on the concept of replication, applied e.g. to databases.

As discussed by Guerraoui and Schiper [17], software replication in distributed environments has some advantages over other fault tolerance solutions. First and foremost, it provides the groundwork for the shortest recovery delays. Also, generally it is less intrusive with respect to execution time. Finally, it scales much better. Another important advantage, on the design perspective, is that the use of software replication is relatively generic and transparent to the application domain. The designer does not have to explicitly specify the nature of the possible abnormal behaviors and the way to handle them. As we show in the report, our solution is furthermore transparent, as the task of deciding what entities to replicate and how to parameterize replication is handled automatically. That said, software replication focuses mostly on processor or network faults, and does not address the whole spectrum of possible faults (design, timing...). Thus, a general issue, and still subject of open research - and not addressed in this report -, is how to combine various approaches for fault tolerance in a single articulated methodology.

Software replication is generally applied explicitly and statically, at design time. Thus, it is the responsibility of the designer of the application to identify explicitly what critical components should be made robust and also to decide what strategies (e.g., active or passive replication) and their configurations (how many replicas, their placement, etc.). Meanwhile, new cooperative applications, e.g., e-commerce, air traffic control, crisis management systems, ambient intelligence, increasingly designed as multi-agent systems (MAS), are much more dynamic. In such applications, the roles and relative importance of the agents can greatly vary during the course of computation, of interaction and of cooperation, because the agents may change roles, plans and strategies. Also, new agents may join or leave the application (as an open system). It is thus very difficult, or even impossible, to identify in advance the most critical software components of the application.

Such new challenges reach the limits of traditional static approaches of replication, and motivate the study of adaptive replication mechanisms. One key issue is then the identification of the most critical components (agents) of the application at a certain time. Therefore, we consider using various levels of information: system level, e.g., communication load, and application/agent level, e.g., roles or plans, to estimate criticality. This report will discuss our past and current experiments using various types of information, notably communications, roles, norms, and plans.

Outline of the report

Section 2 introduces the context of this work, notably the type of faults that we consider here and the existing techniques. Section 3 introduces our approach of dynamic and adaptive control of replication. Section 4 discusses several possible metrics for estimating agent criticalities. Section 5 discusses some experiments. Section 6 discusses related work before Section 7 concludes this report.
2 Context of this Work

2.1 Model of Failure Considered

Any software/hardware component may be subject to faults resulting in output errors, which can lead to a deviation of its specified behavior, i.e. a failure. In distributed systems, and even more so in scalable environments, failures are unavoidable. A subdomain of reliability, fault tolerance aims at allowing a system to survive in spite of faults, i.e. after a fault has occurred, by means of redundancy in either hardware or software architectures.

In this work, we consider the crash type of failures, that is when a component stops producing output. It is the simplest type of failure to contend with. However, in various cases our solution allows to deal with other types of failures (omission, timing, byzantine). They are currently being investigated, but will not be considered in this report.

2.2 Types of Techniques Considered

Replication is an effective way to achieve fault tolerance for crash types of failures. A replicated software component has representations (replicas) on two or more hosts [17]. The two main types of replication protocols are: active replication, in which all replicas process concurrently all input messages; passive replication, in which only one of the replicas processes all input messages and periodically transmits its current state to the other replicas in order to maintain consistency.

Active replication strategies provide fast recovery but lead to a high overhead. Passive replication minimizes processor use by activating redundant replicas only in case of failures. Then a new replica is elected among the set of passive ones and the execution is restarted from the last saved state. This technique requires less CPU resources than the active strategy but it needs an expensive checkpoint management.

Note that when applied to agents with a deterministic behavior, active replication strategies ensure consistency between replicas, thanks to the total ordering of messages. For non deterministic agents, a (light weight) additional consistency management mechanism is required. Passive replication strategies are immune to this issue as only one replica is actually active.

2.3 Limitations of Current Replication Techniques

Many toolkits include replication facilities to build reliable applications. However, many of them are not flexible enough to implement an adaptive replication. For instance, MetaXa [15] implements in Java active and passive replication in a flexible way. Authors extended Java with a meta-level architecture. However, MetaXa relies on a modified Java interpreter. GARF [16] 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 an adaptive mechanism to apply these strategies.

3 Principles of our Approach for Dynamic Replication

To overcome the limitations of static or explicit replication, we propose an approach with automatic and dynamic control of replication. At first, we need a replication architecture which allows dynamic replication and dynamic adaptation of the replication policy (e.g., passive to active, changing the number of replicas). As discussed above, current replication toolkits rarely support such
dynamicity. Therefore, we designed a novel replication framework, named DarX, with such dynamic features.

3.1 DarX: A Framework for Dynamic Replication

DarX is a framework for designing reliable distributed applications based on adaptive replication. Here, we consider failure of processes by crashing, and this crash is permanent. With DarX, each agent can be replicated an unlimited number of times, with different replication strategies (main ones are: passive and active). One of DarX specific features is the reification of the replication strategy, so that it may be dynamically changed. DarX also includes an original failure detection service, based on a hierarchy of adaptive failure detectors [3].

DarX was designed to easily integrate various agent architectures, and the mechanisms that ensure dependability are kept as transparent as possible to the application. For portability and compatibility issues, DarX is implemented in Java.

3.2 DarX Architecture

In DarX, a replication group is an opaque entity underlying every application agent. The number of replicas and the replication strategy of a specific agent are totally hidden to the other application agents. Each replication group has exactly one leader which communicates with the other agents. The leader also checks the liveness of each replica and is responsible for reliable broadcasting. In case of failure of a leader, a new one is automatically elected among the set of remaining replicas.

![Figure 1: DarX application architecture](image)

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

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

DarX includes group membership management to dynamically add or remove replicas. It also provides atomic and ordered multi-cast for the replication groups’ internal communication. Messages between agents, that is communication external to the group, are also logged by each replica, and sequences of messages can be re-emitted for recovery purposes. See, e.g., [28] for further details about DarX.

3.3 Need for Automatic and Adaptive Control

Provided the architecture for dynamic replication, we need a control mechanism for deciding which agents should be replicated and with what strategy (active or passive, how many replicas, where to create the replicas, etc.).\(^1\) For dynamic applications,\(^2\) a manual control is not realistic, as the application designer cannot monitor the evolution of a distributed cooperative application of a significant scale. Therefore, the control mechanism should be automatic, although it may use some information as provided by the designer of the application.

3.4 Air Traffic Control Scenario

Let us imagine an example scenario with some future application of assistance for air traffic control through assistant agents. The airspace is divided into sectors, each sector being controlled by a human controller. Each controller is assisted by an assistant agent who cooperatively monitors the air traffic to suggest decisions about traffic control (see Figure 2). Agents communicate in order to assist with collaborative procedures, e.g., hand off procedures, i.e., when a controller passes the responsibility of an airplane exiting from its supervision sector to the controller of the sector the plane is entering.

![Figure 2: Air traffic control](image)

While trying to accomplish their tasks, agents can be faced to different kinds of failures. In our work, we initially consider the crash type of failures, which can be caused by internal (operating system crashes, hardware problems) or external factors (malicious attacks, power failures, environmental disasters). Additionally, the failure of one agent can also impact on the

---

\(^1\)In this report, we only discuss the decision about which agents to replicate and with how many replicas. Other issues are addressed elsewhere, e.g., where to create the replicas in [20].

\(^2\)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.
agents who depend on it (hand off procedures). To minimize the impact of failures, agents can be replicated. It is clear that replicating every agent in every machine is not a feasible approach since not only the available resources are often limited, but also the overhead imposed by the replication could degrade performance. Thus, the problem consists in finding a replication scheme which minimizes the probability of failure of the most critical agents. This scheme must also be revised over time, considering that the contexts of tasks are dynamic and, thus, the criticalities of the agents vary at runtime.

### 3.5 Scheduling Meetings Scenario

As another (and simpler) example of scenario, that we will use for comparing strategies, let us consider a distributed multi-agent system that helps at scheduling meetings. Each user owns a personal assistant agent which manages his calendar. This assistant agent interacts with:

- the user to receive his meeting requests and the associated information (a title, a description, possible dates, participants, priority, etc.)
- the other assistant agents of the system to schedule meetings, based on preferences of their human owners.

In practice, the assistant agents use the Contract Net Protocol (CNP) [34], as following (see Figure 3):

- a call for proposals message is sent (broadcasted) to the participants by the initiator;
- the participants reply (propose or refuse) to the initiator with the proposed meeting times;
- the initiator sends accept or reject messages to participants;
- the participants which agree to the proposed meeting inform (confirm) the initiator.

![Figure 3: Contract net protocol](image)

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

The control mechanism will estimate the most critical agents of the application and this information will be regularly updated. Here we may informally define the criticality of an agent as follows: the criticality of an agent, relative to an organization of agents it belongs to, is the measure of the potential impact of the failure of that individual agent on the failure of the organization. In the following, we consider criticality of an agent as a numerical value within the interval \([0, 1]\). Various metrics to estimate the criticality of an agent will be discussed in Section 4.

3.7  Replication Control

Once we have a metric for computing (estimating) the criticality of each agent, we may compute the number of replicas \(n_{bi}\) of an agent \(a_i\). A simple initial formulation is:

\[
n_{bi} = \max(\text{rounded}(w_i \cdot R_{\text{max}} / W), r_{\text{max}}),
\]

where:

- \(w_i\) is the criticality of the agent,
- \(W\) is the sum of the domain agents' criticalities,
- \(r_{\text{max}}\) is the maximum number of replicas (usual value is 3, as it is a standard heuristics for replication strategies),
- \(R_{\text{max}}\) is the maximum global number of replicas, i.e., the available maximum resources for creating replicas.

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 [11]). Another refined mechanism takes into consideration 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. This mechanism 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 [2].

4  Metrics for Estimating the Criticality

In order to estimate the criticality of an agent, the issues are:

- What kind of information will be pertinent?
- And how can we obtain it? (explicitly stated by the application designer, inferred by external observation, e.g., amount of messages exchanged, or by internal observation, e.g., plans of an agent, etc.).

Some metrics have been proposed for identifying most critical software components (see, e.g., [10]). However, these metrics are often based on the static structure and local nature of software components, e.g., size, complexity, and dependences. Some recent directions are considering more dynamic properties, e.g., frequency, that is the number of times that a function is executed during a period of time, and mechanisms for updating them automatically (see e.g., [36]).

We believe that, because of the very dynamic nature of multi-agent systems (agents changing roles, non deterministic behavior) as well as thanks to the higher conceptual level (organizational and cognitive levels of knowledge), more appropriate metrics should be explored.
In the following of the report, we introduce several alternative metrics. We will also discuss strategies to compute them. Some metrics are completely general and use basic information (references, messages). Some metrics make some assumption of higher-level abstractions (performatives, roles, norms, plans) which may or not be supported by a given multi-agent architecture.

4.1 Static Dependences

The first metric that we studied is based on the concept of dependence (between agents). Intuitively, the more an agent has other agents depending on it, the more it is critical in the organization. Interdependence graphs [7] were introduced as a way to specify interdependences between agents. But as we want control to be as much automatic as possible, we would like to estimate and infer such dependences. A first estimation of dependences may be done statically by using message sending instructions from the code of agents.

Starting from the code of the whole multi-agent system, we automatically extract message sending instructions. In current implementation, we made the assumption that the behavior of an agent is structured through <condition,action> transition rules.\(^3\) Thus, the code extractor can use the causality information about condition (message reception) and action (message sending).

Then, an algorithm automatically computes the graph of communication dependences. An example of resulting communication dependences graph is shown at the left side of Figure 4, where each node is a communication expression (\(x \rightarrow y\) means: agent \(x\) sends a message to agent \(y\)).

![Communication Dependences Graph](image)

![Agent Dependences Graph](image)

![Replication level ordered list](image)

Figure 4: Static extraction of dependences

The complexity of that first algorithm is \(O(N^p)\), where \(N\) is the aggregated number of transitions, \(p\) the maximum number of transitions matching a specific message sending. \(p\) is at most \(N - 1\), but in practice with a lower bound, so the algorithm is polynomial. A second algorithm transforms the communication dependences graph into an agent dependences graph, where each node corresponds to an agent and a label representing how many agents depend on it (can receive messages from it). It is shown at the middle of Figure 4. Last, from that second graph, we can extract, in a standard way, its connex parts, minimal covering trees, and finally an ordered list, as a guide for ordering criticality of agents (shown at the right side of Figure 4). Further details have been described in [27].

\(^3\)Such as used in the DIMA multi-agent architecture [18]. Meanwhile, this technique is actually more general and the extraction algorithm (code parser) could be adapted to other kinds of structurations (agent architectures), as long as at least message sending instructions are made accessible.
4.2 Dynamic Dependences

A limitation of the previous metric is that it is static and based on communication expressions, thus only capturing potential communications. Also, complex multi-agent systems are characterized by emergent structures, which thus cannot be always statically defined by the designer. Therefore, we introduce an alternative dynamic version of that metric. In order to compute it, we explicitly represent dependences between agents as a weighted graph. Then we provide a mechanism to automatically update its respective weights. The criticality of an agent is computed as the aggregation of the weights representing dependences of other agents on it.

![Figure 5: Example of interdependence graph](image)

We consider the interdependence graph as a labeled oriented graph (see Figure 5), where each node represents a domain agent and each labeled arc between two nodes represents a dependence between the associated agents. The label of an arc (oriented) is a real number which reflects the importance of the dependence (oriented) between the associated agents. The interdependence graph is dynamic as it can be modified when a new domain agent is added, or disappears, or when interaction patterns evolve.

At design time, the interdependence graph is initialized by the designer. It is then dynamically and automatically adapted at run time. Several parameters may be used to update the interdependences between agents. Our primary updating strategy is using communication load (number of messages) as the parameter. The adaptation algorithm updates the interdependence graph, based on local information (communication load) and on global information, which is defined as an aggregation of the local information of the various agents and hosts.

The algorithm is very simple: only the number of messages is considered, independent of their contents, thus the cost of monitoring is very low. We also proposed and experimented with an extension of this algorithm, using performatives as additional input information (e.g., request has a weight greater than cancel) [20]. Note that monitoring of communication is implemented by a general monitoring distributed architecture (see details in [11]) which can also be used by other metrics (e.g., for monitoring roles, see next section).

4.3 Roles

An alternative metric that we studied is based on the concept of role. A role, within an organization, represents a pattern of services, activities and

---

4Note that it can actually be automatically initialized by another metric, e.g., based on static analysis of dependences (see in Section 4.1).
relations. As an example, in some e-commerce organization, roles are: service provider, client, broker, etc. A role will be fulfilled (played) by one or more agents, and the same agent may simultaneously play several roles in different organizations.

Roles are usually defined relatively to some organization, but they may also be defined relatively to some protocol. An example is the contract net protocol which is used in the scheduling meetings scenario (see Section 3.5). It considers two roles: manager/initiator and bidder/participant. In fact, protocol roles can be considered as some specific case of organizational role, where an organization is created dynamically during the scope of the protocol activation.

The notion of role captures some information about the relative importance of roles and their interdependences. Thus we thought that a role is a pertinent concept for estimating criticality. We ask the designer to grade the various roles along their criticality (relative importance). In the scenario of scheduling meetings, introduced in Section 3.5, two roles are considered: Initiator and Participant. Their respective weights could be set by the application designer to e.g., 0.7 and 0.4. The current criticality of an agent is computed as the weight of the role it is currently playing. In the case that an agent is playing simultaneously more than one role, its current criticality is computed as the aggregation of the weights of the roles it is currently playing.

In order to monitor roles, we must assume if agents signal explicitly when they play a role (role-taking and then role-leaving). Signaling explicitly when an agent starts (and stops) playing a role is often the case for organizational roles, where organizational actions are usually made public to the organization. For protocol roles, if agents use FIPA ACL (agent communication language) and specify explicitly the protocol used (within the messages), that information can then also be used.

Meanwhile, as we want our role-based metric to be general, we also considered the case where agents do not necessarily signal their roles. We only suppose that they communicate with some minimal agent communication language. We thus designed a role monitoring mechanism. It is based on a description language for specifying protocols, stored in a library, and a recognition algorithm for recognizing protocols and their roles by monitoring sequences of messages (see more details in [19]). Monitoring messages is implemented by the same general monitoring distributed architecture used for the metrics based on dynamic dependences (see previous section).

### 4.4 Norms

It is possible to extend the previous role-based metrics with two kinds of additional informations: time outs and norms. The underlying assumption is that time outs and norms (permissions, obligations, prohibitions) also capture some indication of criticality. We start from a description language (with its associated control architecture), named XMLaw, for law-based governance of multi-agent systems [6]. In XMLaw, we specify an interaction protocol (with its transitions), and a set of norms and clocks (see [6] for details on XMLaw).

Let’s take a simple example of e-commerce, with a seller and a customer, as shown in Figure 6. The customer requests for products through a call for proposal (cfp message, based on CNP, as in Section 4.3) sent to various sellers. When a seller proposes a product (propose message), a clock (time out) is activated in order to check that the customer answers (accept or refuse the offer) within a specific time frame (which then deactivates the clock). If the customer accepts the offer in time, then the seller sends the bill to the customer (inform (bill) message). A norm - to be more precise, an obligation - is then activated to ensure that the customer sends a proof of payment to the seller (which deactivates the norm). The specification of the norm in XMLaw, with its associated activation and deactivation events, is shown below. It is a
fragment of the whole XMLaw specification of the example.

```xml
<Norm type="obligation" id="obligation_customer_to_pay">
  <Assignee role-ref="customer" role-instance="$customer.instance"/>
  <Activations>
    <Element ref="customer-receive-bill" event-type="transition_activation"/>
  </Activations>
  <Deactivations>
    <Element ref="customer-send-proof-payment" event-type="transition_activation"/>
  </Deactivations>
</Norm>
```

These specifications about role taking/leaving, clock activation/deactivation, and norm activation/deactivation, are used as inputs to automatically adjust the criticality along the various steps of the protocol, as shown in Figure 6. These three contributions (role, clock, norm) to the change of criticality are then aggregated to produce the estimation.

![Figure 6: Evolution of criticality for customer role](image)

The designer may also have a finer grain control explicit of the weights associated to each activation or deactivation event. Further details of that metrics and related strategy may be found in [14].

### 4.5 Plans

The last metric that we discuss in this report uses the plans of an agent, i.e., the actions that the agent has planned to execute in the near future. In our model, we consider that each agent of the system knows which sequence of actions (plan) must be executed in order to accomplish its current goal.\(^6\) We assume that at each given instant of time, the agent is executing at most one action.

Using the same approach established by [22], we represent the plan of an agent as a directed acyclic AND/OR graph where each node represents an action. The nodes are connected by AND or OR edges.

In the example of Figure 7, after performing the action $A$, Agent1 needs to have both actions $B$ and $C$ executed in order to accomplish its plan. However,

\(^5\)The use of roles is analog to role-based metrics described in Section 4.3.

\(^6\)Since unexpected events may occur in dynamic environments, agents usually interleave planning and execution. Consequently, their plans are established just for the short term.
Figure 7: Examples of plans

after C, only one of D or E needs to be performed so that Agent1 accomplishes its plan. We call an external action an action belonging to the plan of an agent which will be executed by other agents. For example, consider the action C belonging to the plan of Agent1 in Figure 7. Since this action is performed by Agent2, it is an external action in the current plan of Agent1. A terminal action is an action after which no other known action will be performed.

In order to calculate the criticality of an action, we distinguish its absolute criticality from its relative criticality. The absolute criticality (AC) of an action is defined without taking into account the current plans of the agents. It is given a priori by the system designer and can be determined in function of a number of factors: number of agents capable of performing the action, duration of the action, resources required for the execution of the action, application dependent information.

The relative criticality (RC) of an action belonging to the plan of an agent is proportional to the criticality of the agent when it is executing the action or waiting for some other agent to execute it. As a consequence, the relative criticality of an action may vary depending on the agent plan it belongs to. The relative criticality is calculated as follows:

- For an external action, it is equal to the local relative criticality (LRC). The LRC is obtained using the AND-aggregation function if the action is connected to its children by means of AND edges or the OR-aggregation function if it is connected by OR edges. If the action has only one child, its LRC is equal to the relative criticality of its child. If the action is terminal (i.e. it has no child), its local relative criticality is equal to zero.
- For a non-external action a, its relative criticality is equal to its absolute criticality plus the sum of the local relative criticalities of a in each plan to which it belongs.

We have also refined this initial strategy by considering the expected duration of actions. We compute the estimated starting time of the actions using a topological sorting in the graph (top-down), considering the elapsed times of the antecedents and siblings' actions. Another issue is the possible dynamicity of plans of agents, because of, e.g.: lack of resources, failed commitments, etc. We proposed two main types of strategies to update criticality: time-driven strategies and event-driven strategies (e.g., action completion, failure). More details of the strategies are presented in [1].

Note that one expected advantage of this plan-based metric is that we can estimate not only the immediate criticality of an agent, but also estimate its future criticality.

4.6 Discussion

Each metric has its pros and cons: static or dynamic, cost, and nature of assumptions of abstractions available (messages, roles, norms, plans). The
last metric (plan-based) has the advantage of estimating future criticality and not just instantaneous one.

Note that current metrics consider agents individually and thus do not directly address the robustness of a group of agents as a whole. Although several metrics implicitly capture some indication of cooperation between agents (e.g., correlation of the number of messages mutually exchanged, shared plans), we are planning to introduce explicit knowledge about organizations and coordination in order to consider more explicitly this issue.

Also note that the various metrics that we proposed are mostly bottom-up, as they use or infer information from the program elements or from execution, to estimate criticality of agents. We are also studying a dual direction, top-down, based on first analysis and specifications of general dependability requirements, and then in using that information to guide replication control. Some directions are in using a dependability risk-driven approach [5] or dependability cases [35], see [13].

5 Experiments

In order to assess our approach and to compare the different metrics, we implemented the integration of a flexible multi-agent platform, named DIMA [18], and the DarX replication architecture. We then implemented the various metrics, some of them using a common distributed monitoring architecture. The resulting framework is named DIMAX [11].

We are conducting some experiments to compare metrics. In this report, we summarize some of them, based on the scenario of meetings scheduling (see Section 3.5). They were carried out on twenty machines with Intel(R) Pentium(R) 4 CPU at 2 GHz and 526 Mb of RAM. Active replication was chosen (in this scenario, agents are deterministic). To compare accuracy of strategies, we used a fault simulator which randomly chooses an agent and stops its thread. If the killed agent was playing the role of an initiator, then its associated meeting scheduling negotiation (protocol) fails, unless the agent has been replicated. Thus, that experiment provides some measure of the accuracy of the metrics to identify most critical agents and protect them.

We considered a multi-agent system with 200 agents distributed on 10 machines. We run each experiment 10 minutes and we introduced 100 faults. We repeated several times the experiment with a variable number of extra resources $R_{max}$ (from 0 to 20). Remember (see Section 3.7) that $R_{max}$ defines the number of extra replicas that can be used by the whole multi-agent system.

This experiment measures the rate of succeeded simulations $SR$ which is defined as follows: $SR = \frac{NSS}{TNS}$, where $NSS$ is the number of succeeded simulations and $TNS$ is the total number of simulations.

In Figure 8, we compare four metrics: 1) random, 2) roles with roles explicitly signaled, 3) roles with role monitoring, 4) dependences. For each metrics, we display the success rate $SR$ as a function of the number of extra replicas.

First, it shows that all metrics show better results than random. The metrics with roles explicitly signaled is the most accurate (actually it is also the less costly). This can be explained for the example scenario by the importance of the initiator in the negotiation. For application domains where the roles have similar importance, the metrics based on dependences may lead to better results. We are currently conducting further measures on different types of applications. The objective is to try to empirically identify possible features of applications, correlated to the relative accuracy of different metrics.
6 Related Work

Decker et al. [9] propose dynamic cloning of specific agents in multi-agent systems. But their motivation is different to ours, as their 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.

Chameleon [24] is an adaptive fault tolerance system using reliable mobile agents. The methods and techniques are embodied in a set of specialized agents supported by a fault tolerance manager (FTM) and by host daemons for handshaking with the FTM via the agents. Adaptive fault tolerance is achieved by making the Chameleon infrastructure reconfigurable. Static reconfiguration guarantees that the components can be reused for assembling different fault tolerance strategies. Dynamic reconfiguration allows component functionalities to be extended or modified at runtime by changing component composition, and components to be added to or removed from the system without taking down other active components. 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 [31] is a middleware built above Ensemble [33]. 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.

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 dif-
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 oc-
curs. 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 [12] propose to use proxies to make transparent the 
use of agent replication, i.e. enabling the replicas of an agent to act as a same 
entity regarding the other agents. A proxy manages the state of the replicas. 
All external and internal communications of the group are redirected to the 
proxy. But this increases the workload of the proxy which is a quasi central 
entity. To make it reliable, they propose to build a hierarchy of proxies for 
each group of replicas. Their approach lacks some flexibility and reusability, 
in particular concerning replication control. Replication is indeed set up by 
the designer before run time.

Kaminka et al. [25] adopted a monitoring approach in order to detect 
and recover faults. They use models of relations between mental states of 
agents. They adopt a procedural plan-recognition based approach to identify 
the inconsistencies. However, the adaptation is only structural, the relation 
models may change but the contents of plans are static. Main hypothesis is 
that any failure comes from incompleteness of beliefs. Thus, the behavior of 
agent cannot be adaptive and the system cannot be open.

Horling et al. [23] presented a distributed system of diagnosis. The faults 
can directly or indirectly be observed in the form of symptoms by using a 
failure model. The diagnosis process modifies the relations between tasks, 
in order to avoid inefficiencies. The adaptation is only structural because 
they do not consider the internal structure of tasks. However, a problem 
of performances can occur in this approach because the global performance 
improvement is based on a local performance improvement.

The work by Kraus et al. [26] 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.

We may mention that DarX has also been used at Free University of Am- 
sterdam, in the AgentScape project, of the Intelligent Interactive Distributed 
Systems Group of F. Brazier [30]. Their AgentScape agent architecture sep-
arrates the public part of an agent, which is immutable and may be freely 
replicated, from its private part which should be kept unique. AgentScape 
does not address yet the issue of adaptive replication control.

Last, a related and much more general project is the Autonomic Computing 
Program of IBM. They propose a general blueprint architecture (monitor, 
analyze, plan, execute). A prototype architecture, named ABLE (Agent 
Building and Learning Environment) [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 part, the ABLE architecture by itself does 
not solve the problem, but the blueprint guidelines are an interesting direction.

7 Conclusion

Large-scale multi-agent systems are often distributed and must run without 
any interruption. To make these systems reliable, we proposed an architecture 
(DarX) for dynamic replication and its control. In this report, we discussed 
several metrics for estimating criticality of agents, inferred automatically from 
various kinds of information (references, messages, roles, norms, plans). The 
agent criticality is then used to replicate agents in order to maximize their
reliability and availability based on available resources. We believe that our current results are promising. Meanwhile, more experiments are needed to better evaluate our approach, various metrics and strategies, and classify their respective classes of applications.

Acknowledgments  The authors thank Marin Bertier, Alain Cardon, Sébastien Charpentier, Athmane Hamel, Guillaume Lacôte and Jakob Zimmerman, for their past contributions to the project. We also acknowledge support from the following research programs: ANR Sécurité & Informatique, EuroControl CARE INO III, CAPES-COFECUB and CNPq.

References


