Submission to DSN-2003 – Main Track: Dependable Computing and Communications

DARX - A Framework For Fault-Tolerant, Agent-Oriented Software

Olivier MARIN[1,2] **Marin BERTIER**[2] **Pierre SENS**[2]

olivier.marin@univ-lehavre.fr marin.bertier@lip6.fr pierre.sens@lip6.fr

[1]Laboratoire d'Informatique du Havre
University of Le Havre
25 rue Philippe Lebon
BP540 76058 Le Havre cedex

[2]Laboratoire d'Informatique de Paris 6
University Paris 6 - CNRS
4 place Jussieu
75252 Paris Cedex 05, France

Abstract

This paper presents DARX, our framework for building applications that provide adaptive fault tolerance. DARX relies on the fact that multi-agent platforms constitute a very strong basis for decentralized software that is both flexible and scalable, and makes the assumption that the relative importance of each agent varies during the course of the computation.

Firstly, the interwoven concerns of multi-agent systems and fault-tolerant solutions are put forward. An overview of the DARX architecture follows, as well as an evaluation of its performances. We conclude, after presenting the promising outcomes, by drawing some perspectives of evolution.

Keywords: scalable systems, multi-agent framework, adaptive replication
Submission category: Regular Paper
Word count: 6672
Contact author:

Olivier MARIN
Laboratoire d'Informatique de Paris 6
8, rue du Capitaine Scott 75015 PARIS
Phone 01.44.27.34.23
Fax 01.44 27 87 71
olivier.marin@univ-lehavre.fr

CARTER Award: YES

Nominated Student: MARIN, Olivier.
Advisor(s): Research Director, SENS, Pierre.

1 Introduction

Nowadays it barely seems necessary to emphasize the tremendous potential of decentralized software solutions. Their main advantage lies in the distributed nature of information, resources and action. One software engineering technique for building such software has lately emerged in the artificial intelligence research field, and appears to be both promising and elegant: distributed agent systems [BDC00] [MCM99] [NS00].

Intuitively, multi-agent systems appear to represent a strong basis for the construction of distributed applications. The general outline of distributed agent software consists in autonomous computational entities which interact with one another towards a common goal that is beyond their individual capabilities.

In addition, the multi-agent paradigm bears two attractive notions: flexibility and scalability. By definition, agents have the ability to adapt in order to meet new context requirements. A software consisting of multiple agents can therefore be dynamically modified: objectives of specific agents may be altered, new agents can be brought in to collaborate towards a computation, agents that have become partly useless for the application can be adapted or set aside, and so on... Moreover, multi-agent systems are based on communicating, autonomous entities; it ensues that there is no theoretical limit to the number of agents involved, nor is there any bound to the number of hosting machines. Distributing such systems over large scale networks may therefore tremendously increase their efficiency as well as their capacity.

However, large-scale distribution also brings forward the crucial necessity of applying dependability protocols. For instance, the greater the number of agents and hosts, the higher the probability that one of them will be subjected to failure. As mentioned earlier, multi-agent applications rely on the collaboration amongst agents. It follows that the failure of one of the involved agents might bring the whole computation to a dead end.

In this paper, we depict DARX, our architecture for fault-tolerant agent computing [MSBG01]. DARX uses the flexibility of multi-agent systems in order to offer adaptive fault tolerance by means of dynamic replication mechanisms: software elements can be replicated and unreplicated on the spot and it is possible to change the ongoing replication strategies on the fly. We have developed a solution to interconnect this architecture with two existing multi-agent platforms, namely MadKit[GF00] and DIMA [GB99], and in the long term to other platforms. The originality of our approach lies in two major orientations. Firstly, the choice of the fault tolerance protocol – which computational entities are to be made fault-tolerant, to which degree, and at what point of the execution – is not entirely incumbent upon the application developer; DARX offers automated observation and control functionalities to address these issues. And secondly, the overall architecture is conceived with a view to being scalable.

The paper is organized as follows. In section 2, the main existing approaches towards solving the fault tolerance problems in the multi-agent systems context are presented. Section 3 depicts the general design of our framework dedicated to bringing adaptive fault tolerance to multi-agent systems through selective replication. Section 4 reports on the issues raised by the implementation of DARX-compliant applications, and section 5 evaluates the performances of the resulting software. Finally, the conclusion and perspectives are drawn in section 6.

2 Related Work

Research on fault tolerance in multi-agent systems mainly focuses on the ability to guarantee the continuity of every agent computation. This approach includes the resolution of consistency problems amongst agent replicas. However, some solutions also address the complex problems of maintaining agent cooperation [KCL00], providing reliable migration for independent mobile agents and ensuring the exactly-once property of mobile agent executions [PS01].

Several solutions use specific entities to protect the computational elements of multi-agent systems [H96] [KIBW99] [KCL00]. The principal contribution of these approaches is in separating the control of the agents from the functionalities of the multi-agent system.

In [H96], sentinels represent the control structure of the multi-agent system. Each sentinel is specific to a functionality, handles the different agents which interact to provide the corresponding service, and monitors communications in order to react to agent failures. Adding sentinels to a multi-agent system seems to be a good approach, however the sentinels themselves represent bottle-necks as well as failure points for the system.

A similar architecture is that of the Chameleon project [KIBW99]. Chameleon 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 host daemons for handshaking with the FTM via the agents. Adaptive fault tolerance refers to the ability to dynamically adapt to the evolving fault tolerance requirements of an application. This is achieved by making the Chameleon infrastructure reconfigurable. Static reconfiguration guarantees that the components can be reused for assembling different fault tolerance strategies. Dynamic reconfiguration allows component functionalities to be extended or modified at runtime by changing component composition, and components to be added to or removed from the system without taking down other active components. Unfortunately, through its centralized FTM, this architecture suffers from the same objections as the previous approach.

[KCL00] presents a fault tolerant multi-agent architecture that regroups agents and brokers. Similarly to [H96], the agents represent the functionality of the multi-agent system and the brokers maintain links between the agents. [KCL00] proposes to organize the brokers in hierarchical teams and to allow them to exchange information and assist each other in maintaining the communications between agents. The brokerage layer thus appears to be both fault-tolerant and scalable. However, the implied overhead is tremendous and increases with the size of the system. Besides, this approach does not address the recovery of basic agent failures.

In order to solve the overhead problem, [FD02] proposes to use proxies. This approach tries to make transparent the use of agent replication; that is, computational entities are all represented in the same way, disregarding whether they are a single application agent or a group of replicas. The role of a proxy is to act as an interface between the replicas in a replicate group and the rest of the multi-agent system. It handles the control of the execution and manages the state of the replicas. To do so, all the external and internal communications of the group are redirected to the proxy. A proxy failure isn't crippling for the application as long as the replicas are still present: a new proxy can be generated. However, if the problem of the single point of failure is solved, this solution still positions the proxy as a bottle-neck in case replication is used with a view to increasing the availability of agents. To address this problem, the authors propose to build a hierarchy of proxies for each group of replicas. They also point out the specific problems which remain to be addressed: read/write consistency and resource locking, which are discussed in [SBS00] as well.

3 The architecture of the DARX framework

This section presents DARX, our Dynamic Agent Replication eXtension, and depicts its features.

3.1 System model and failure model

A distributed system is assumed, in which processes/agents communicate through messages. Communication channels are considered to be reliable. Our model follows that of partial synchrony, proposed by Chandra and Toueg in [CT96]. 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: the global stabilization time.

Processes are assumed to be fail/silent. Once a specific process is considered as having crashed, it cannot participate to the global computation anymore. Byzantine behaviours might be resolved with DARX, but are not yet integrated in the failure model.

Finally, for scalability issues, a hierarchic structure is imposed for the network topology. Sets of hosts are organized in groups. Broadly connected machines are regrouped in clusters of workstations (COWs), and a higher inter-COW level is constructed. Within each COW, a single (a set of) host(s) is elected so as to participate to the higher level.

3.2 Overview

Figure 1 gives an overview of the logical architecture of DARX.

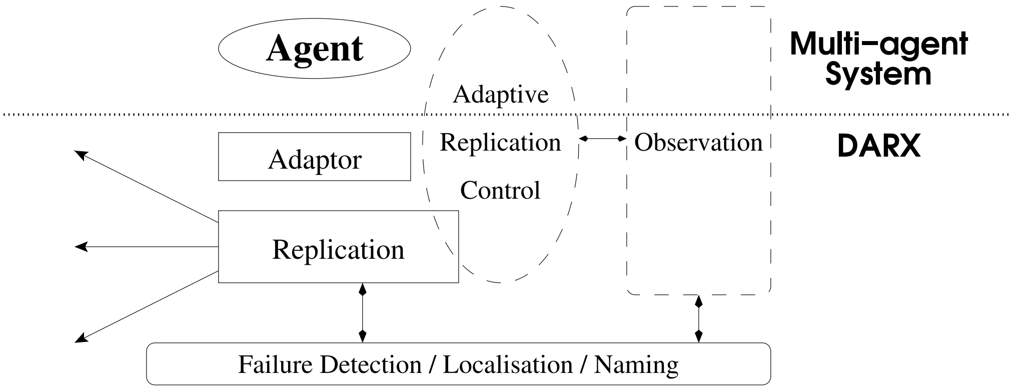


Figure 1: DARX application architecture

The fault tolerance features are brought to agents from various platforms through their corresponding adaptor by an instance of a DARX server running on every location[1]. Each DARX server implements the required replication services, backed by a common global naming/location service enhanced with failure detection (see 3.3). Concurrently, a scalable observation service (see 3.4) is in charge of monitoring the system behaviour at each level – local, intra-COW, inter-COW. The information gathered through both means is used thereafter to adapt the fault tolerance schemes on the fly: periodically, a decision module combines system-level information and application-level information to determine the *criticity*[2] of each agent, and to apply the most suitable replication strategy.

DARX includes transparent replication management. While the supported application deals with agents, DARX handles replication groups. Each of these groups consists in software entities – replicas – which represent the same agent. Thus in the event of failures, if at least one replica is still up, then the corresponding agent isn't lost to the application. A more detailed explanation of a replication group, of its internal design and of its utilization in DARX can be found in 3.5.

For portability and compatibility issues, DARX is Java-based. Indeed, the Java language and more specifically the JVM provide – relative – hardware independence, an invaluable feature for large-scale distributed systems. Moreover, a great number of the existing multi-agent platforms are implemented in Java. In addi-

[1]A location is an abstraction of a physical location. It hosts resources and processes, and possesses its own unique identifier. DARX uses a URL and a port number to identify each location that hosts a DARX server.

[2]The *criticity* of a process defines its importance with respect to the rest of the application. Obviously, its value is subjective and evolves over time. For example, towards the end of a distributed computation, a single agent in charge of federating the results should have a very high *criticity*; whereas at the application launch, the *criticity* of that same agent may have a much lower value.

tion to all this, the remote method invocation (RMI) facility offers many useful high-level abstractions for the elaboration of distributed solutions.

3.3 Failure detection and naming service

As part of the means to supply adequate support for large-scale agent applications, the DARX platform includes a hierarchical, fault-tolerant naming service. This distributed service is deployed over an asynchronous network enhanced with a failure detection service based on an adaptable implementation of the unreliable failure detector [BMS02].

The failure detection and naming layer serves a major goal: to maintain dynamic lists of the valid sites and of the valid agents, as well as their casual replicas, participating to the application. Specific agents can thus be localized through this service. Failure detectors are organized in groups. They exchange heartbeats and maintain a list of the processes which are suspected of having crashed. Therefore, in an asynchronous context, failures can be recovered more efficiently. For instance, the failure of a process can be detected before the impossibility to establish contact arises within the course of the supported computation.

The service aims at detecting both hardware and software failures. Each DARX server integrates an independent thread which acts as failure detector/name server. Software failure is detected by monitoring the running processes on each server. Hardware failures are suspected by exchanging heartbeats among groups of servers. For large-scale integration purposes, this structure comprises two levels: a local and a global one. Local groups of servers are bound together by a global group. Every local group elects exactly one representative which will participate to the global group. At the global level, each name server maintains a list of the known agents within the application – the replication group leaders (see 3.5) in the DARX context. This information is shared and kept up-to-date through a consensus algorithm implying all the global name servers. When a new agent is created, it is locally registered as well as by the name server of the group representative; likewise in the case of an unregistration. At the local level, the naming servers maintain the list of all the replicas supported in their local group, disregarding whether these are leaders or not.

In this architecture, the ability to provide different qualities of service to the local and the global detectors is a major asset of our implementation. Thus on the global level, failure suspicion can be loosened with respect to the local level. This distinction is important, since a failure does not have the same interpretation in the local context as in the global one. A local failure corresponds to the crash of an agent or of a host, whereas in the global context a failure represents the crash of an entire COW.

The naming service makes use of the failure detection to convey its communications. The information is exchanged between name servers via piggybacking on the failure detection heartbeats. The local lists of processes which are suspected to be faulty are directly reused to maintain the global view of the application. With respect to DARX, this means that the list of running agents is systematically updated. When a DARX server is considered as having crashed, all the agents it hosted are removed from the list and replaced by replicas located on other hosts. The election of a new leader within an agent replication group is initiated by a failure notification from the naming service.

3.4 Observation service

DARX aims at providing decision-making support so as to fine-tune the fault tolerance for each agent with respect to its evolution and that of its context. Decisions of this type may only be reached through a fairly good knowledge of the dynamic characteristics of the application and of the environment. In order to obtain such knowledge, a scalable observation service has been designed and implemented, yet remains to be integrated in DARX.

Similarly to the naming service, the observation service piggy-backs its communications on the existing flow created by the regular heartbeat emissions of the failure detection service. Moreover, it is also hierarchic; it distinguishes local and global levels.

The data collected at the local level consists in instantaneous information such as the current memory load of a host, the overall execution time of an agent since it was created, the number of messages exchanged between two agents, ... This type of data is shared within local groups; broadcasting it or enabling subscription to it on a large scale does not appear worthwhile. Indeed, the validity of such information over a long period of time is highly questionable. Besides, its diffusion on a great number of distant locations bears a heavy cost, even though it would be diluted in the failure detection flow. Nonetheless, it may be needed to gain instantaneous information on a specific machine outside the local COW. For example, it may be necessary to determine the feasibility of a replication to a remote COW. The observation service therefore allows for point-to-point subscription to data collection on distant hosts.

Statistical information, however, possesses a longer lifespan in the DARX context. Such material encompasses all the data derived by processing the local information: the average CPU load of a host over a long period of time, the failure rate of a host or of a local group, their average network load, their meantime between failures, ... Hence it is shared at the global level. Every local group elects a member responsible for the aggregation of the statistical information, as well as for its diffusion at the global level. Statistical information about other groups can thus be retrieved at the elected local workstation.

Each local DARX server integrates an observation module. It comprises three elements: a data collection module (DCM), a data processing module (DPM) and a data exchange module (DEM). The DCM extracts the information available from the operating system, such as the CPU load or the swap activity, therefore it is chosen to be host-compliant. The DPM is Java-based and gathers application-level information; the state of an agent, for example. The DPM also interfaces with the DCM to recover system-level data, and renders it into a directly usable format for the DARX platform. On a periodic basis, the DEM broadcasts the accumulated instantaneous information to the DPMs of its local group, and participates to the diffusion of the statistical information at the global level if it belongs to a leading observation module.

3.5 Replication management

DARX provides fault tolerance through software replication. It is designed in order to be agent-dependent, as opposed to location-dependent. That is, if a machine crashes, the DARX server it hosted can be restarted but no server information needs to be recovered. However, the state of the agent replicas which were present on the server may be recovered. The reason for this choice comes from the fundamental assumption that the *criticity* of an agent evolves over time; therefore, at any given moment of the computation, all agents do not require to benefit from the same fault tolerant mechanisms, if any at all. Hence on every server, some agent replicas will be kept consistent with pessimistic strategies, others with optimistic ones, while some others will not be replicated at all.

In DARX, agent-dependent fault tolerance is enabled by the notion of replication group (RG): the set of all the replicas which correspond to a same agent. It follows that a RG contains at least one active replica. Within the RG, replicas must be kept consistent so as to ensure recovery in case of failures. To allow for this, several replication strategies are made available by the DARX framework. The strategies offered can be classified in two main types: (1) **active**, where all replicas process the input messages concurrently, and (2) **passive**, in which only one replica is in charge of the computation while periodically transmitting its state to the other replicas.

One of the innovative aspects of DARX is that several strategies may coexist inside the same RG. As long as one of the replicas is active, meaning that it executes the associated agent code and participates to the application communications, there is no restriction on the activity of the other replicas. Indistinctly, those other replicas may be backups – passive strategy – or followers – semi-active strategy – of the active replica, or even equally active replicas. Furthermore, it is possible to switch from a strategy to another with respect to a replica: a follower may become a backup, and so on . . .

Hence a considerable amount of information is necessary to describe a replication group:

- the *criticity* of its associated agent,
- its replication degree – the number of replicas it contains –,
- the list of these replicas, ordered by potential of leadership[3],
- the list of the replication strategies applied inside the group,
- the mapping between replicas and strategies.

[3]The potential of leadership is the capacity of a replica to represent its RG. The strategy used to keep a replica consistent is the main parameter in the calculation of this potential; the more pessimistic the strategy, the higher its potential. The other parameters emanate from the observation service; they include the load of the host, the date of creation of the replica, the latency in the communications with the other replicas of the group, . . .

The sum of these pieces of information constitutes the replication policy of a RG. A replication policy must be reevaluated in two cases: (1) when a failure inside the RG occurs, and (2) when the *criticity* value of the associated agent changes. It seems obvious that all the replicas of a group must have a consistent and up-to-date version of their policy. However, since the replication policy may be reassessed frequently, it appears reasonable to centralize this decision process. A leader is elected within the RG for this purpose. Its objective is to adapt the replication policy to the *criticity* of the associated agent as a function of the characteristics of its context – the information obtained through the observation service. As mentioned earlier, DARX allows for dynamic modifications of the replication policy. Replicas and strategies can be added to or removed from a group during the course of the computation, and it is possible to switch from a strategy to another on the fly. For example, the recovery of a missing active replica may be decided by activating the most suitable backup inside the RG; or if a backup crashes, a new replication can be initiated to maintain the level of reliability within the group; or if the *criticity* of the associated agent decreases, it is possible either to suppress a replica or to switch the strategy attached to a replica from an active form to a passive one.

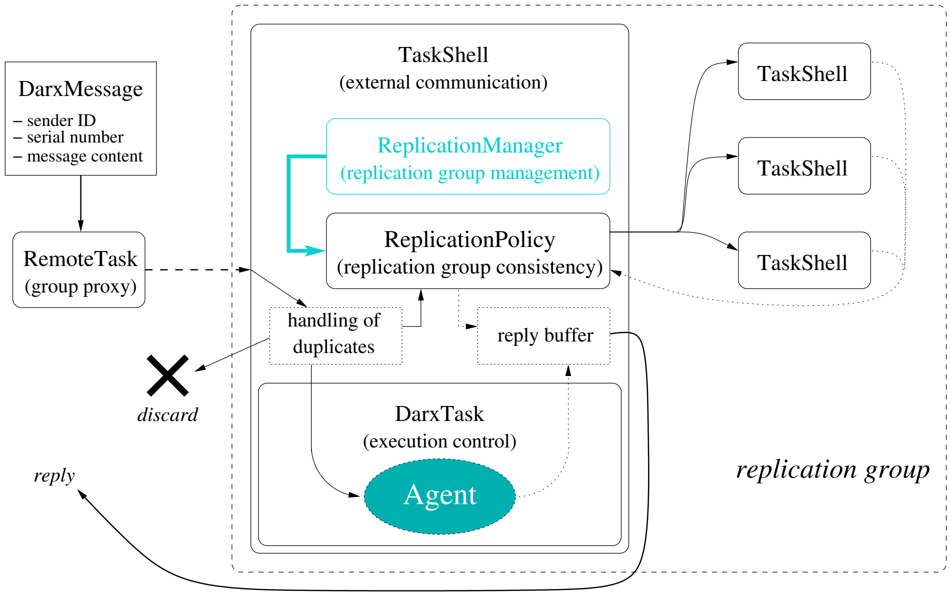


Figure 2: Replication management scheme

Figure 2 depicts the composition of a replica. In order to benefit from fault tolerance abilities, each agent gets to inherit the functionalities of a `DarxTask` object, enabling DARX to control the agent execution. Each task is itself wrapped into a `TaskShell`, which handles the agent inputs/outputs. Hence DARX can act as an intermediary for the agent, committed to deciding when an agent replica should really be started, stopped, suspended or resumed, and exactly when and which message receptions should take effect. Leaders are wrapped in enhanced shells, comprising an additional `ReplicationManager`. This manager exchanges information with the observation module (see 3.4) and performs the periodical reassessment of the replication policy. It also maintains the group consistency by sending the relevant information to the other replicas, following the policy requirements.

Communication between agents passes through proxies implemented by the `RemoteTask` interface. These proxies reference replication groups; it is the naming service which keeps track of every replica to be referenced, and provides the corresponding `RemoteTask`.

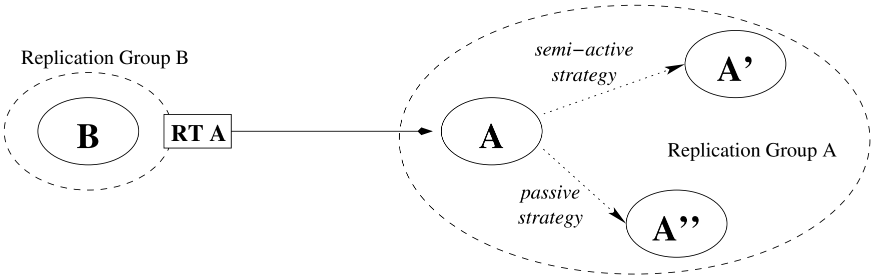


Figure 3: A simple agent application example

Figure 3 shows a tiny agent application as seen in the DARX context. An emitter, agent B, sends messages to be processed by a receiver, agent A. At the moment of the represented snapshot, the value of the *criticity* of agent B is minimal; therefore the RG which represents it contains a single active replica only. The momentary value of the *criticity* of agent A, however, is higher. The corresponding RG comprises three replicas: (1) an active replica A elected as the leader, (2) a follower A' to which incoming messages are forwarded, and (3) a backup A" which receives periodical state updates from A.

In order to transmit messages to A, B requested the relevant `RemoteTask` RTA from the naming service. Since replication group A contains only one active replica, RTA references replica A and no other.

4 Application building with DARX

This section describes how a multi-agent application may be built over DARX, and hence benefit from its fault tolerance features.

Analysis of the original agent source code might provide the required information to enable DARX support without further modifications of the original program [BGCAMS02]. However, at this point of our research, the application developer must respect a few guidelines and constraints which are given thereafter. As a Java framework, DARX includes several generic classes which assist the developer through the process of implementing a reliable multi-agent application. The choice of those generic classes comes from the study of the OMG MASIF [MASIF98] specifications, as well as that of the most recurrent aspects of various MAS, therefore DARX-compliant application building is very close to most agent developing environments.

Every agent class must extend a `DarxTask` for several reasons. Although it is not the only factor, the state of an agent is essential in determining its *criticity*. Any number of states can be defined for the agent; each of those states ought to be mapped to a corresponding static *criticity* in the code of the `ReplicationManager`. A static *criticity* is the importance of an agent taken out of its computation context. At runtime, a dynamic *criticity* will be evaluated in conjunction with the characteristics of the environment. Also, the `DarxTask` provides a boolean for active/semi-active strategy differentiation: indeed, some methods may deviate in each case. For example, in non-deterministic programs some state switches should be initiated by the active replicas only. Finally, the `DarxTask` is the point where DARX handles the execution of an agent: application-specific control methods to start, stop, suspend and resume the agent have to be defined for this purpose. Such methods would be very hard to implement in a general context, where the application developer would not have to intervene, without modifying the JVM and the efficiency loss would be drastic.

Since DARX overrides the localization and naming services of the agent platforms it supports, communications between agents must be taken care of. Communicating agents must instantiate a `DarxCommunicationInterface`; messages to other agents are emitted through this interface, built around the

RemoteTask reference of the destinations. Messages sent to a group by means of a RemoteTask are thus rerouted to the group leaders, where duplicates are discarded and ordering is guaranteed. Additionally, this scheme allows tracking of the message flows by the observation service.

Also, fault tolerance protocols that are specific to the application can be developed. DARX provides a generic ReplicationStrategy class which may be extended to fulfill the needs of the programmer. Basic methods allow to define the consistency information within the group, as well as the way this information ought to be propagated in different cases, such as synchronous or asynchronous messages for example. A few simple strategies are already built in DARX; others are undergoing research, like quorum-based strategies for instance.

5 Performances

This section presents performance evaluations established with DARX. Measures were obtained using JRE 1.4.1 on the Distributed ASCI Supercomputer 2 (DAS-2). DAS-2 is a wide-area distributed computer of 200 Dual Pentium-III nodes. The machine is built out of clusters of workstations, which are interconnected by SurfNet, the Dutch university Internet backbone for wide-area communication, whereas Myrinet, a popular multi-Gigabit LAN, is used for local communication.

5.1 Agent-oriented dining philosophers example

A first experiment aims at checking that there is indeed something to be gained out of adaptive fault tolerance. For this purpose, an agent-oriented version of the classic dining philosophers problem [H85] has been implemented over DARX.

In this application, the table as well as the philosophers are agents; the corresponding classes inherit from DarxTask. The table agent is unique and runs on a specific machine, whereas the philosopher agents are launched on several distinct hosts. Figure 4 represents the different states in which philosopher agents can be found. The agent states in this implementation aim at representing typical situations which occur in

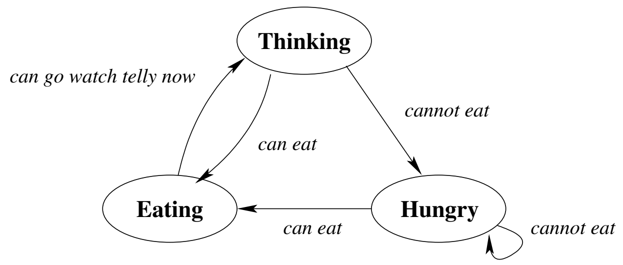


Figure 4: Dining philosophers over DARX: state diagram

distributed agent systems:

- **Thinking**: the agent processes data which isn't relevant to the rest of the application,
- **Hungry**: the agent has notified the rest of the application that it requires resources, and is waiting for their availability in order to resume its computation,
- **Eating**: data which will be useful for the application is being treated and the agent monopolizes global resources – symbolized by the chop-sticks.

In order to switch states, a philosopher sends a request to the table. The table, in charge of the global resources, processes the requests concurrently in order to send a reply. Depending on the reply it receives, a philosopher may or may not switch states; the content of the reply as well as the current state determine which state will be next. It is arguable that this architecture may be problematic in a distributed context. For a great number of philosophers, the table will become a bottleneck and the application performances will degrade consequently. Nevertheless, the goal of this experimentation is to compare the benefits of adaptive fault tolerance with respect to fixed strategies. It seems unlikely that this comparison would suffer from such a design. Besides, the experimentation protocol was built with these considerations in mind.

Since the table is the most important element throughout the application, the policy of the associated RG is constant and pessimistic – semi-active. The group consists of a leader and a follower. The RGs corresponding to philosophers, however, have adaptive policies which depend on their states. Table 1 shows the mapping between the state of a philosopher agent and the replication policy in use within the corresponding RG.

Agent state	Replication degree	Replication policy
Thinking	1	Single active leader
Hungry	2	Active leader replicated passively
Eating	2	Active leader replicated semi-actively

Table 1: Dining philosophers over DARX: replication policies

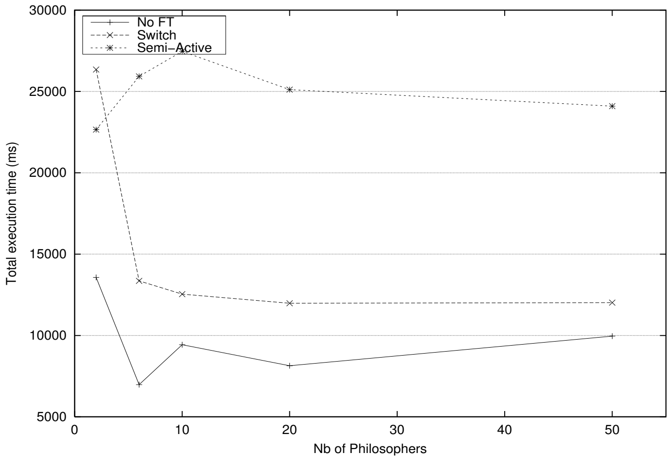
5.2 Results analysis

The experimentation protocol is the following. Eight of the DAS-2 nodes have been reserved, with one DARX server hosted on every node. The leading table replica and its follower each run on their own server. In order to determine where each philosopher leader is launched, a round robin strategy is used on the six remaining servers. The measure can start once all the philosophers have been launched and registered at the table.

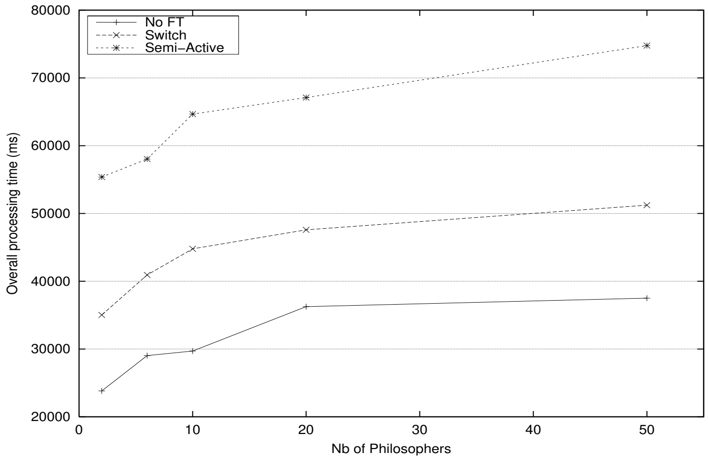
Two values are being measured. The first is the total execution time: the time it takes to consume a fixed number of meals (100) over all the application. The second is the total processing time: the time spent processing data by all the active replicas of the application. Although the number of meals is fixed, the number of philosophers isn't: it varies from two to fifty. Also, the adaptive fault tolerance protocol is compared to two others. In the first one the philosophers are not replicated at all, whereas in the second one the philosophers are replicated semi-actively with a replication degree of two – one leader and one follower in every RG.

Every experiment with the same parameter values is run six times in a row. Executions where failures have occurred are discarded since the protocol without replication does not always enable to end the application. The results shown here are the averages of the resulting measures.

Figure 5(a) shows the total execution times obtained. At first glance it demonstrates that adaptive fault tolerance may benefit to distributed agent applications in terms of performances. Indeed the results are quite close to those obtained with no fault tolerance involved, and are globally much better than those of the semi-active version. In the experiments with two philosophers only, the cost of adapting the replication policy is prohibitive indeed. But this expense becomes minor when the number of philosophers – and hence



(a) Comparison of the total execution times



(b) Comparison of the total processing times

Figure 5: Performance comparison with various fault tolerance protocols

the distribution of the application – increases. Distribution may also justify the notch in the plot for the experiments with the unreplicated version of the application: with six philosophers there is exactly one replica per server, so each processor is dedicated to its execution. In the case of the semi-active replication protocol, the cost of the communications within every RG, as well as the increasing processor loads, explain the poor performances.

It is important to note that, in the case where the strategies inside RGs are switched, failures will not forbid the termination of the application. As long as there is at least one philosopher to keep consuming meals, the application will finish without deadlock. Besides it is possible to restart philosophers which weren't replicated without risk, since these replicas had no impact on the rest of the application: no chop-sticks in use, no request for chop-sticks recorded. This is not true in the unreplicated version of the application.

Figure 5(b) accounts for the measured values of the total processing time in each situation. Those results also concur to show that adaptive fault tolerance is a valuable protocol. Of course, the measured times are not as good as in the unreplicated version. But in comparison, the semi-active version induces a lot more processor activity. It ought to be reminded that in this particular application, the switch version is as good as the semi-active version in terms of fault tolerance. However, the semi-active version obviously implies that the average recovery delays will be much shorter in the event of failures. In such situations, the follower

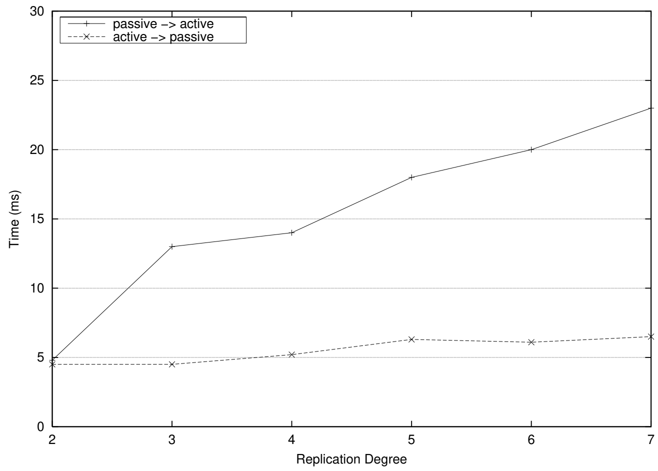


Figure 6: Strategy switching cost as a function of the replication degree

can directly take over. Whereas with the adaptive protocol, the recovery delay depends on the strategy in use: unreplicated philosophers will have to be restarted from scratch and passive backups will have to be activated before taking over.

5.3 Replication policy switching

Finally, we evaluate the times required to switch replication strategies: Figure 6 shows the times measured when changing, at different replication degrees, from an active strategy to a passive one, and vice versa. As expected, in the former case, the costs are very low as nothing much has to be performed except the strategy modification. Whereas in the latter case, each replica has to be updated in order to enable the change: a time-expensive operation even though the replicated task carries little overload – 100 kilobytes.

6 Conclusion and Perspectives

The framework presented in this paper enables the building of fault-tolerant distributed multi-agent systems. The resulting software is flexible: it possesses the ability to decide which parts of the computation are more critical than the others, and hence should be made to bypass failures through replication. DARX offers control over the way the application safeguards its components, enabling the fault tolerance of the computation to be automatically fine-tuned on the fly. Moreover, the architecture of the middleware is designed to be scalable.

However, there are still some issues left unsolved as of today: for instance, the observation service mentioned in section 3.4 remains to be integrated in the framework. It works as a stand-alone application, and the API for exchanging commands and data with DARX is set. But the modifications of the DARX classes which shall make use of the observation service are waiting to be coded.

Another field of investigation is the analysis of the dynamic *criticity* of agents and the adaptation of the replication policy. The heuristics used up to now are simplistic, mainly because of the lack of observation system. Once confronted to the real characteristics of the hosts and of the network, those heuristics will be enhanced for further efficiency and adequateness.

Undergoing study is the possibility to use OpenSSH in order to ensure secure communications between agents, and also within replication groups.

In order to validate the work achieved up until now, applications are currently being developed. Those include a basic crisis management system destined to test the viability and the utility of our architecture in terms of such software.

References

[BDC00] H. Boukachour, C. Duvallet and A. Cardon, "Multiagent systems to prevent technological risks" In *Proceedings of IEA/AIE'2000*, Springer Verlag 2000.

[BGCAMS02] J.-P. Briot, Z. Guessoum, S. Charpentier, S. Aknine, O. Marin and P. Sens "Dynamic Adaptation of Replication Strategies for Reliable Agents" In *Proc. 2nd Symposium on Adaptive Agents and Multi-Agent Systems (AAMAS-2)*, London, UK, April 2002.

[BMS02] M. Bertier, O. Marin and P. Sens, "Implementation and performance evaluation of an adaptable failure detector" In *Proc. of the International Conference on Dependable Systems and Networks*, Washington, DC, USA, 2002.

[CT96] T. D. Chandra and S. Toueg "Unreliable Failure Detectors for Reliable Distributed Systems" In *Journal of the ACM*, 43:2, March 1996, pp. 225-267.

[FD02] A. Fedoruk and R. Deters, "Improving Fault-Tolerance by Replicating Agents", In *Proceedings of 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems*, Bologna, Italy, July 2002.

[GB99] Z. Guessoum and J.-P. Briot, "From active objects to autonomous agents" In *Special Series on Actors and Agents*, edited by Dennis Kafura and Jean-Pierre Briot, IEEE Concurrency, 7(3):68-76, July-September 1999.

[GF00] O. Gutknecht and J.Ferber, "The MadKit agent platform architecture", In *1st Workshop on Infrastructure for Scalable Multi-Agent Systems*, Barcelona, Spain, June 2000.

[GS97] R. Guerraoui and A. Schiper, "Software-Based Replication For Fault Tolerance" In *IEEE Computer*, 30(4):68-74, 1997.

[H96] Hägg S., "A Sentinel Approach to Fault Handling in Multi-Agent Systems", in *Proceedings of the 2nd Australian Workshop on Distributed AI, 4th Pacific Rim Int'al Conf. on A.I. (PRICAI'96)*, Cairns, Australia, August 27, 1996.

[H85] C. A. R. Hoare, "Communicating Sequential Processes", Prentice Hall, 1985.

[KCL00] Kumar S., Cohen P. R., Levesque H. J., The Adaptive AgentArchitecture: Achieving Fault-Tolerance Using Persistent Broker Teams", *4th International Conference on Multi-Agent Systems (ICMAS 2000)*, Boston MA, USA, July 2000.

[KIBW99] Z. Kalbarczyk, R. K. Iyer, S. Bagchi, K. Whisnant, "Chameleon: A Software Infrastructure for Adaptive Fault Tolerance", *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no.6, June 1999, pp. 560-579

[MASIF98] D. Milojicic et al, "MASIF: The OMG Mobile Agent System Interoperability Facility" In *Proc. of the 2nd Int. Workshop on Mobile Agents*, LNCS 1477, 1998, pp. 50-67.

[MCM99] D. Martin, A. Cheyer and D. Moran "The Open Agent Architecture: A Framework for Building Distributed Software Systems" In *Applied Artificial Intelligence*, 13(1-2):91-128, January-March 1999.

[MSBG01] O. Marin, P. Sens, J.-P. Briot and Z. Guessoum "Towards Adaptive Fault-Tolerance for Distributed Multi-Agent Systems" In *Proceedings of ERSADS'2001*, pp.195-201, Bertinoro, Italy, May 2001.

[NS00] Niranjan Suri et al.,, "An Overview of the NOMADS Mobile Agent System" In *Proceedings of ECOOP'2000*, Nice, France, 2000.

[PS01] Stefan Pleisch and André Schiper, "Fatomas - a fault-tolerant mobile agent system based on the agent-dependent approach", In *Proc. of the IEEE Int. Conf. on Dependable Systems and Networks*, Goteborg, Sweden, July 2001.

[RBS96] R. van Renesse, K. Birman, and S. Maffeis, "Horus: A flexible group communication system", In *Communications of the ACM*, 39(4):76–83, April 1996.

[SBS99] M. Strasser, J. Baumann, and M. Schwehm, "An Agent-based Framework for the Transparent Distribution of Computations" In *H. Arabnia (ed.), Proc. of PDPTA'1999*, Vol I:376-382, Las Vegas, USA, 1999.

[SBS00] L. Silva, V. Batista, and J. Silva, "Fault-tolerant execution of mobile agents", In *Proc. of the International Conference on Dependable Systems and Networks*, pp. 135-143, New York, June 2000.