# Specification of an Exception Handling System for a Replicated Agent Environment

Christophe Dony
LIRMM, UMR 5506
CNRS and Univ. Montpellier 2
Montpellier, France
Christophe.Dony@lirmm.fr

Chouki Tibermacine
LIRMM, UMR 5506
CNRS and Univ. Montpellier 2
Montpellier, France
Chouki.Tibermacine@lirmm.fr

Christelle Urtado
LGI2P, Ecole des Mines d'Alès
Nîmes, France
Christelle.Urtado@ema.fr

Sylvain Vauttier
LGI2P, Ecole des Mines d'Alès
Nîmes, France
Sylvain.Vauttier@ema.fr

## ABSTRACT

Exception handling and replication are two mechanisms that increase software reliability. Exception handling helps programmers control situations in which the normal flow of a program execution cannot continue. Replication handles system failures. Exceptions handling and replication do not apply in the same way to the same situations and thus are two complementary mechanisms to increase software reliability. The paper proposes a specification of an execution history oriented exception handling system for an agent language and middleware providing replication. This paper proposes an original signaling algorithm adapted to replicated agents and a rationale of how exception handling and replication mechanisms can combine to increase programmers' capability to achieve reliable agent-based applications.

## 1. INTRODUCTION

Exception handling (EH) and replication are two mechanisms (algorithms and architectures) dedicated to reliability and fault-tolerance that we wish to associate. Replication handles failures whereas exceptions enable programmers to dynamically handle those situations that prevent software from running normally. An agent replication system [12, 7] is able to replace an agent (provided he has been replicated) that fails by one of its (active or passive) replicas. This replacement is as seamless as possible to software users and does not require any additional code from programmers. A failure is generally detected when an agent fails answering to messages for a given amount of time, either because network connections are lost or because the machine on which the agent ran is switched off. Active replication systems include algorithms capable of identifying the most critical agents to automatically replicate them. When replication is active, messages sent to an agent are transmitted to all its replicas, which process the same message in parallel. When replication is passive, a single replica (the *leader*) processes messages and periodically sends state updates to the other replicas. Exceptions are situations in which the standard control flow of a program execution cannot continue. An exception is not a failure because it is a kind of answer from the agent. It indicates that the agent is unable to continue its task the standard way but that he is still alive. EH and replication do not apply to the same situations and of different nature: replication is preventive and EH is curative. However, both mechanisms are obviously very complementary. In the context of the FACOMA[1] project that studies the adaptive reliability of large scaled multi-agent applications, we are proposing an exception handling system (EHS) capable of working on top of a replication system. The motivation for integrating these two mechanisms is threefold:

- The combination of replication mechanisms and EH in general is a new and interesting challenge for software reliability.
- EH can improve replication. Firstly, the implementation of the replication system can be made more robust by internally using exceptions. Secondly the use of exceptions can improve replication strategies. For example, with passive replication, the signaling of a system exception by the leader can become a new situation for the replication system to replace the leader by one of its replicas.
- Replication can also improve EH by providing active copies of the computation state.

The objective of this paper is to present our study on the first of the three above points: how an EHS can be combined with a replication mechanism to increase the reliability of agent-based applications. The bases of the study are our SAGE exception handling system dedicated to agents [19], components [20] and active objects [5] and the DIMAX replicated agent system [7, 12] abstracted in Sect. 2.

[1] http://www-src.lip6.fr/homepages/facoma.officiel/


*WEH '08,* November 14, Atlanta, Georgia, USA

Our study lists and discusses issues, related to the adaptation of a computation history oriented EHS on top of such replicated agent systems (RAS). Here is a panel of the main issues:

- How to transparently exploit replication for EH?
- What should happen when an exception is raised by an agent that has one or more, active or passive, replicas?
- When and where should the replication system be able to take control when an agent signals an exception?
- Which decisions can be taken within a handler defined at the replication system level?
- How to distinguish an exception that can be interpreted as a failure from the point of view of the replication system and that cant thus entail the election of a new replica, from an exception which can be interpreted as a correct answer for the service caller? For example, signaling the "division-by-zero" exception is the normal answer from the "divide" function if its second argument is zero. In this case it is useless to activate another replica, because it will compute the same answer.
- Do standard resolution mechanisms used in distributed EH to concert exceptions apply to synthesize the results computed by different replicas of the same agent?

The remainder of this paper is structured as follows. Section 2 sets the context of this work, describing the targeted agent model and replication system. Section 3 abstracts the requirements for EH in a multi-agent world and provides the agent programmer-directed API of our X-SAGE EHS. Section 4 describes how exception handlers are searched for in a replicated agent system (RAS) while Sect. 5 discusses how system-defined handlers can be defined in the replication system to integrate exception handling to the replication manager (RM). Section 6 concludes with a short discussion on the benefits of our approach and open perspectives to this work.

# 2. CONTEXT OF THE STUDY: OVERVIEW OF A REPLICATED MULTI-AGENT SYSTEM

The context of this work is the programming of reactive, collaborating agents that are deployed over a middleware which handles agent replication. The concepts exposed in this section are derived from the DIMAX software that combines the DIMA multi-agent system [7] and the DARX fault-tolerant middleware [12]. Initial names and principles are generalized here and adapted to the agent interaction scheme we studied in our previous work [5], namely peer-to-peer service exchanges.

## 2.1 The agent model

An agent is a computation entity that executes in its own thread. This provides the agent with the properties of being active and autonomous. The behavior of a reactive agent consists of two parts: a control behavior which defines how the agent makes decisions to act, depending on its internal state and the state of its environment; several elementary behaviors that represent the actions the agent knows how to do. Figure 1 shows an abstract of the *BasicCommunicatingAgent* class, the base class in our context, used to implement reactive agents. The control behavior of the agent is defined by the *live* method. It implements a loop that is executed while the agent is alive. Each iteration of this control loop calls the *step* method. This method implements the decision mechanism that enables the agent to choose, step by step, the action it executes. A control behavior represents the existence of the agent and is executed in a separate thread, provided by the execution platform as an instance of the *AgentEngine* class. This enables agents to execute on top of different platforms, which can adapt their specific execution model to the management of an agent as an *AgentEngine* subclass. The other behaviors of the agents are represented as methods of the agent class. Some of these behaviors are executed upon the reception of a request from another agent. These behaviors are called services. Agents interact by exchanging asynchronous messages. Each agent holds a message box and a communication interface respectively to send and receive messages (*cf.* the *MessageBox* and *CommunicationComponent* classes on Fig. 1). The communication interface is provided by the execution platform and is responsible for the delivery of the messages. As described above for the *AgentEngine* subclass, specific asynchronous messaging mechanisms can be adapted to the agent model as a *CommunicationComponent* subclass. Each agent bears a unique identifier used as a logical reference, to designate messages senders or recipients. The execution platform uses name directories to convert these abstract agent identifiers to effective references, in order to deliver the messages to the agents. A specific semantics is associated to messages in order to set up a request / response interaction protocol between agents. This protocol describes peer-to-peer collaborations in which a client agent asks a server agent for a service via a request message. Conforming to a contract-based approach of software, whenever a server agent accepts a request, it commits to send back a result, either standard or exceptional, to the client agent via a response message. Response messages are correlated with request messages. When no response is received for a period of time defined by the client agent, a timeout exception is signaled. As an illustration (*cf.* Fig. 2), we use the canonical *Travel Agency* example in which a *Client* can send to a *Broker* a reservation message in order to request a bid for a travel. The contacted broker sends in turn a bid request to several travel providers and collects their responses. Then, the *Broker* selects the best offer and requests the *Client* and the selected *Provider* to contract.

## 2.2 The Replication System

Agents are executed on a middleware which provides a fault-tolerant execution context thanks to a replication mechanism [12]. The execution context consists of a set of distributed replication servers which manage the execution of tasks (*ReplicatedTask* class of Fig. 1). Every task belongs to a replication group (*ReplicationGroup* class) that identifies the set of tasks which are replicas of a same logical task. Thus, all the tasks within a replication group have the same behavior (they actually are instances of the same task class). Moreover, the middleware maintains the replication group consistent so that all the tasks are in the same state after each computation. In the same replication group, all the tasks are not exact replicas. While they have the same be-

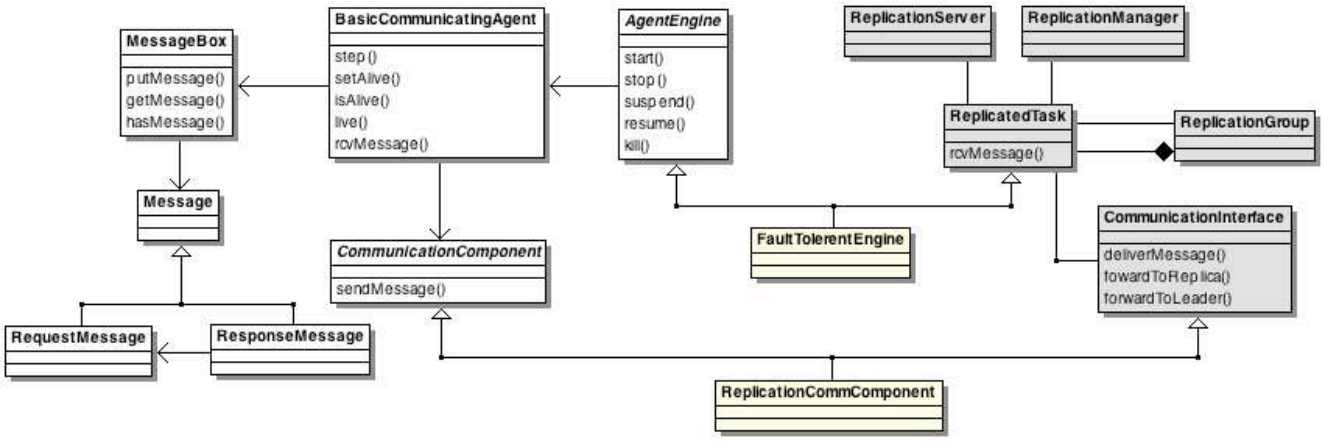


**Figure 1: Excerpt from the agent model**

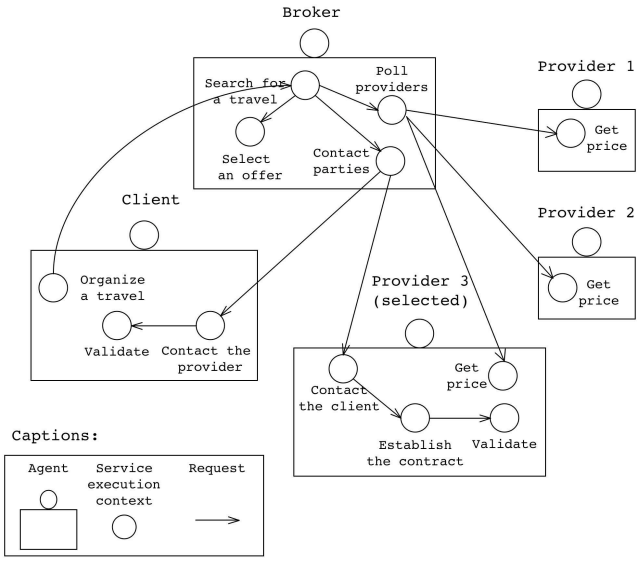


**Figure 2: Execution resulting from a request to a travel agency**

havior, these tasks could have different environments (they could run on machines of different kinds, with their own resources). Logical tasks are identified by logical names that are used to send them messages. The replication middleware is in charge of the location and delivery of messages to the corresponding replicas. More precisely, messages are delivered first to the leader of the corresponding replication group. The leader is a replica which has the specific role to control the replication group. For this purpose, it holds an RM which monitors the messages sent to or by the replicas in the replication group. The replication manager maintains status information about the replicas and executes group management operations (creation, destruction of replicas, etc.). The RM distinguishes two kinds of replicas. Active replicas effectively execute treatments. The leader is necessarily an active replica. The leader forwards the messages sent to the task to the other active replicas so that they do the same computation and reach the same new state. Passive replicas only perform state updates. When the leader completes the computation, its new state is serialized and sent to all the passive replicas. After their update, the passive replicas are in the same state as the leader (and supposedly as other active replicas). Conversely, all messages sent by replicas are filtered by the replication middleware. Only messages sent by the leader are actually delivered to other tasks. This makes replication transparent to other tasks. Whatever the number of replicas of a task is, a unique message is sent to invoke a computation and a unique message is received as a response. The number and type of replicas is determined by the replication policy, regarding the criticality of the task and the availability of resources (memory, CPU). In case of failures, new replicas can be dynamically created in order to maintain the redundancy required to provide an expected level of fault-tolerance. The type of replica (active, passive) can be changed to adapt resource consumption to criticality and risk. When the leader fails, its responsibility is transferred to another replica. When a passive replica is chosen to become the leader, its status is changed to active. The state of the task (meaning the state of all the replicas of the corresponding replication group) is thus rolled back to the state of the new leader (which represents the previous consistent state of the task, backed up in a passive replica). If no replica still exists, the task has finally been destroyed by the failure. Every agent executes inside a task (*cf.* Fig. 1). As such, agents can be replicated by the middleware and benefit from this fault-tolerance mechanism. The following sections explain how EH is combined with replication.

## 3. CONTROL STRUCTURES FOR EH WITH REPLICATED AGENTS

This section motivates and presents the first part of our proposal: the X-SaGE control structures for exception handling designed for agent programmers. These control structure only slightly differ from those of the SaGE system. Indeed, they are programmer-directed and replication mechanisms do not interfere in any programmer-directed capability as replication must be transparent to the agent programmer. Handling replication will intervene in the implementation of these control structures in Sect. 4 and 5.

## 3.1 Requirements for an Agent Programmer-directed EHS

The key requirements of the X-SaGE exception handling system, extended from [19, 5], are:

- to enforce agent encapsulation,
- to provide a representation for collaborative concurrent activities [17] so that they can be coordinated and controlled [16],
- to look for handlers in the history of computation, instead of delegating exception handling to specialized agents, and to execute handlers in their lexical definition context; we call this caller contextualization [5] for handler definition and execution. When encapsulation and decoupling are enforced, non lexical handlers do not have access to the execution contexts where the exceptions are signaled (the agents which execute the faulty services). They can only use generic management operations (such as service or agent termination) to cope with the signaled exception.
- to handle concurrent exceptions with resolution functions [9],
- and, to support asynchronous signaling and handler search so as to maintain agent reactivity.

Our specification comes in four steps indicating: (1) to which program code units exception handlers can be attached, (2) how exceptions can be signaled, (3) what can be written within the code of exception handlers to put the system back into a coherent state and, (4) in which order handlers are searched for. The following two subsections are dedicated to items 1 to 3. Item number 4 involves interfacing with the replication mechanism; it is discussed in Sect. 4 and 5.

## 3.2 Signaling Exceptions and Attaching Handlers

Figure 3 shows the java code of an X-SaGE agent that defines services and various exception handlers.
X-SaGE takes advantage of the java annotations to make EH for agents as seamless as possible. It shows examples of service definitions (annotated by *@service*): lines 6–7 define the *pollProviders* service and lines 17–30 the *contactParties* service. It also illustrates (lines 20–28) how a message can be sent by (a service of) a client agent to request a server agent to provide him with some (sub-) service. Signaling exceptions is done by the means of a classical *signal* primitive (*cf.* Fig. 3, line 11). Signaling is possible anywhere in the code. This includes the possibility of signaling an exception from within handlers. Steps of the request / response interaction pattern highlight the role of three key entities: the request, the service and the active agent. They are the three program code units to which exception handlers can be attached:

- Exception handlers can be attached to **requests**. Such handlers can, for example, specify two distinct reactions to the occurrence of two identical exceptions raised by two invocations of the same service. Lines 23–27 of Fig. 3 shows how a handler can be attached to a specific request.
- Exception handlers can be attached to **services**. Such handlers treat exceptions that are raised, directly or indirectly, by some service's execution. If the service is complex, the handler has to be able to deal with concurrent exceptions, to compose with partial results or to ignore partial failures. Lines 10–14 of Fig. 3 shows the code of two handlers attached to a same service (*@serviceHandler* annotation). Note that the *service-name* attribute of the annotation allows to identify the service the handler protects.
- Finally, exception handlers can be attached to **agents**. Such handlers act as if they were repeatedly attached to all of the agent's services. They can be used, for example, to uniformly maintain in the consistency of the agent's private data. Lines 3–4 of Fig. 3 shows how such handlers can be associated to agents using the*@agentHandler* annotation.

Our experience with SaGE showed us that these capabilities are powerful enough to encompass most cases the agent programmer will be confronted to and simple enough to be easy to learn and use.

## 3.3 Defining EH and Resolution Functions

Exception handlers are classically defined by the set of exception types they can catch and by their code body (as illustrated by Fig. 3, lines 23–27, for example). There are three main actions a handler can classically have:

- A handler can restore whatever should be, to put back data into a consistent state, and can **return** a value that becomes the value of the expression the handler is associated to. In case of a message sending expression (standard or broadcast), the value returned by the handler is the value of the expression. In case of a handler attached to a service, the value becomes the result of the service execution. In case of a handler attached to an agent, the value becomes the result of the execution of the service that raised the exception.
- A handler can **signal** a new exception (generally of a higher conceptual level) or **re-signal** the original exception (Fig. 3, line 11). Of course, handlers cannot protect themselves from the exceptions they signal.
- A handler can **retry** the execution of the program unit it is attached to (Fig. 3, line 27). Retry amounts to entirely re-execute the program unit it is attached to, generally after having modified the local environment, but in the same historical context. In case of handlers attached to agents, retrying means re-executing the service that signaled the exception.

X-SaGE provides exception resolution support integrated to the handler search. It enables resolution functions to be defined at places where concurrent activities are launched and have to be co-ordinated (*i.e.,* at the service level). There is no need for a resolution function either at the request level, because requests are atomic, or at the agent level because all semantically sound activities of agents, that need to be coordinated, are accessible via services. The default behavior of the resolution function associated to a service is, once all recipients have replied, to aggregate all the exceptions that occurred into a new concerted exception. Another possible

```
( 1)  public class Broker extends X_SaGEAgent
( 2)    {
( 3)    // handler associated to the Broker agent
( 4)    @agentHandler public void handle (GlobalNetworkException exc)  { ... }
( 5)
( 6)    // service provided by the Broker agent
( 7)    @service public void pollProviders ()  { ... }
( 8)
( 9)    // handler associated to the PollProviders service
(10)    @serviceHandler(servicename=pollProviders) public void handle (BadParameterException exc)
(11)       { signal (new NoAirportInDestinationException ( ... ); }
(12)
(13)    // handler associated to the PollProviders service
(14)    @serviceHandler(servicename=pollProviders) public void handle (NoProviderException exc) { ... }
(15)
(16)    // service provided by the Broker agent
(17)    @service public void contactParties ()
(18)       {
(19)       ...
(20)       sendMessage (new RequestMessage (aServerAgent, "ContactSelectedProvider")
(21)          {
(22)          // handler associated to a request
(23)          @requestHandler public void handle (OffLineException exc)
(24)             {
(25)             wait(120);
(26)             retry();
(27)             }
(28)          });
(29)       ...
(30)       }
(31)
(32)    // resolution function associated to the pollProviders service
(33)    @serviceResolutionFunction(servicename=pollProviders) public TooManyProvidersException concert ()
(34)       {
(35)       int failed = 0;
(36)       for (int i=0; j<subServicesInfo.size(); i++)
(37)          if ((ServiceInfo) (subServicesInfo.elementAt(i)).getRaisedException() != null)  failed++;
(38)       if (failed > 0.3*subServicesInfo.size())  return new TooManyProvidersException(numberOfProviders);
(39)       return null;
(40)       }
(41)   }
```

**Figure 3: Service, handler and resolution function definitions in X-SaGE using annotations**

behavior is to transmit a response as soon as it arrives without waiting for others. Such a use of resolution for concerted exception slightly differs from the original work of [9]. A resolution function is executed each time an exception handler is searched for at the service level, this makes our system reactive, because our resolution function evaluates the situation each time an exception is signaled. Of course, a programmer can define his own exception resolution function using the *@serviceResolutionFunction* annotation as shown in the example of Fig. 3, lines 33-40.

## 4. HANDLER SEARCH IN AN RAS

Handler search requires that a tree of **service execution contexts** be monitored. Each node represents a service execution context and records the identities of the service being executed and the agent that owns the current service (*cf.* Fig. 5). Each node can optionally have a parent node that links to the calling context of the current service. In this parent node, the request that triggered the current service is recorded. Links between nodes (callee to caller links) are used to look for handlers. Figure 2 shows the service execution context tree that results from the services executed in the travel agency example.

If an exception is raised within an agent service, then the execution of the service is suspended and handler search is launched. The handler search process decomposes into four steps. Figure 4 then shows the organigram that synthesizes the different steps of the handler search process.
First, if there is a resolution function at the service level, it is executed. During resolution, three cases are possible:

1. the exception is critical for the service. The resolution function returns the exception object and handler search carries on.

2. the resolution function evaluates that the exception is under-critical and that nothing more should be done yet. The exception is logged, the resolution function returns null and the handler search process stops. The collective activity is not affected. The only service that is terminated is the defective sub-service.

3. the resolution function evaluates that the exception is under-critical but that there is a need to signal something, for example because too many under-critical exceptions have been logged. The resolution function returns a special exception that reflects the situation and handler search carries on.

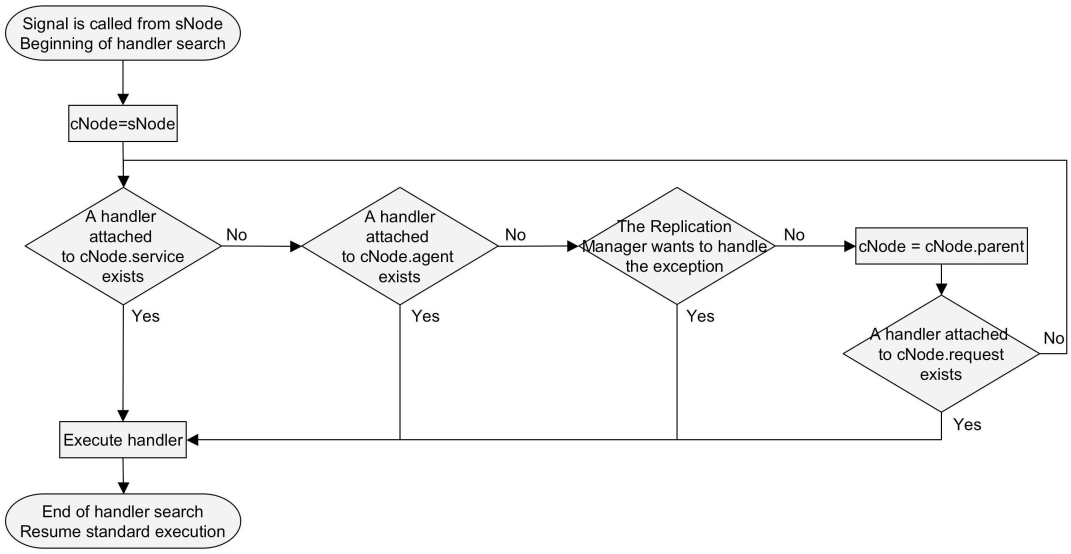


**Figure 4: Organigram for handler search**

If there is no resolution function or in resolution cases 1 & 3, a handler for the exception is then searched in the list of handlers associated to the service. If a handler is found, it is executed. If no handler has been found at the service level,

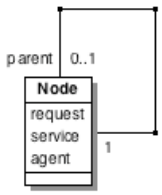


**Figure 5: Node of a service execution context tree**

one is searched at the agent owner of the service. If a suitable handler is found, it is executed and its execution terminates the execution of the service. The agent is of course still alive. Along with the execution of the handler, all pending services called by the current service, if any, are terminated. If no handler has been found at the agent level, and if the agent is replicated, control is given to its RM. Each RM has a handler that traps all exceptions and acts as a resolution function the goal of which is to coordinate the answers given by replicas of the agent. The behavior of this RM handler is described in Sect. 5). If the RM does not want to handle the exception or if it propagates it, search proceeds in the calling context. First, the caller service is suspended and the search for a handler is initiated in the calling service's context. The list of handlers associated to the request which initiated the called service is searched first. If a handler is found, it is executed and the search stops. Then, the search proceeds by starting again at step 1, executing the resolution function associated with the service, if any, then searching the list of handlers associated to the current service, then, those associated to the owner agent of the current service, etc. The same four steps are repeated until an adequate handler is found and executed, following callee to caller links in the service execution context graph. If no handler has been found when the root of the service execution context tree is reached, a default top-level handler is executed.

# 5. HANDLING EXCEPTIONS AT THE RM LEVEL

With replication, we face the following global issues: (1) how to trap an exception raised by the leading replica of an agent before the exception is propagated to the caller? (2) What to do when it has been trapped? Solving issue 1 is done by invoking the replication manager during handler search as explained in the preceding section. We have added in each RM a resolution function and an associated handler that traps all exceptions. Concerning issue 2, the RM handler will either, as described in the following section, put the system back into a coherent state, signal a new exception to the request caller or propagate one of those it has trapped. In this latter case, the handler search will continue as explained in Sect. 4.

## 5.1 Typology of exceptions

The first global question for the RM handler of an agent when one of its replicas raises an exception is to know whether the same exception will also be raised by the others. Which exception is replica-specific (examples of this include exceptions raised when some resources specific to a given replica are unavailable) and which ones are replica-independent (an example is bad parameter in the request sent to the agent (and thus to all its replicas), leading for example to a division by zero)? In the worst case, it could be considered that all exceptions are replica-specific. It would mean that when a replica signals an exception, we could systematically have another replica retry the same computation. This would significantly slow down program execution.
We thus have conceived our algorithms based on a classification of exceptions. Goodenough's seminal paper [6] has proposed a classification in *domain*, *range* and *monitoring* exceptions that highlights the reason why an exception is raised. It however appears that we have no way to know

whether a *range* exception (for example) is replica-specific or independent. A classification in terms of *Error* (serious problem, should not be handled) and *Exception* (business problem, can be handled) as in Java, inherited from the *Flavors* system, highlights the exception gravity but cannot again be applied to our problem.
A more appropriate classification classically relies on exception semantics and distinguishes *business* (also called *domain* or *applicative*) exceptions from *system* or *resource* exceptions. System exceptions are raised by the computing environment and are likely to reflect a specific communication or resource lack problem. They can be considered to be replica-specific. Business exceptions are direct consequences of a programmer's code. Under the hypothesis that all the replicas of an agent have the same deterministic behavior, exceptions identified as business exception can be considered as replica-independent: they will be raised by all replicas of a given agent. The question of knowing how to detect at run-time whether an exception is a system or business is left open at this point of the study. In future work, we will consider comparing the responses (normal responses or exceptions) to a same request given by several replicas of an agent as a possible answer to this question. Indeed, if an exception is repeatedly raised by replicas of a given agent, it might probably indicate that the exception is a business exception. In such a situation, after a determined number of occurrences of the same exception, the system might consider it is useless to wait for other replicas' responses. Beyond this classification, the strategies of the exception handler of the replication manager also takes into account the composition of the replication group. Three strategies are described in the following sections.

## 5.2 Controlling one active replica with multiple passive ones

The passive-replication strategy uses a single active replica (the leader) and a set of passive ones. When the active replica (leader) raises a business exception, it is immediately propagated to the client agent. The leader is considered to be in a coherent state as a business exception is part of the behavior designed by the programmer of the agent. Moreover, this exception should be raised by any replica executing the same request message. It is then useless to discard the current leader and to activate another replica to retry and execute the request.
When the active replica raises a system exception, it is handled as a failure of the leader. The leader, which is left in an undefined, potentially inconsistent and harmful state, is destroyed. One of the passive replicas is activated (it becomes the new leader) and is asked to retry the interpretation of the requested message. If this new leader raises a system exception too, another passive replica is used until their number runs out. If the last replica fails, the exception is finally signaled to the client agent.

## 5.3 Controlling a set of active replicas

When a business exception is raised by a replica, it is immediately propagated to the client agent since all other replicas are expected to raise the same exception. The RM handler does not stop the execution of the request in the other replicas but filters the exceptions they raise (in order not to send the same exception to the client agent several times). This enables to determine when the execution of the request is achieved for all replicas, to verify that they have raised the same exception and thus are in the same consistent state.
When a system exception is raised by a replica, it is recorded by the resolution function of the RM, until all the active replicas have achieved the execution of the request and have sent a response. Meanwhile, if a normal response is computed by a replica, it is forwarded to the client agent. The other subsequent normal responses are discarded by the RM. When the replicas have sent a response, the RM destroys all the faulty replicas. If the leader is destroyed, a new leader is chosen among remaining replicas. If all replicas are destroyed, an exception is then signaled to the client agent.

## 5.4 Controlling a mix of active and passive replicas

In the case where active and passive replicas are mixed (the most general case), the handler first behaves as if there were only active replicas (as in Sect. 5.3). If system exceptions are successively signaled by all active replicas and some are destroyed, it is possible to activate passive replicas, whether to augment the number of active replicas for the next request or to retry and execute the current request. It is to be noticed that the creation of new replicas is not part of the behavior of RMs (which are specific to each replication group) because it must be arbitrated between the different replication groups, according to the criticality of the tasks and the availability of computing resources. Replica creation is thus managed by another replication middleware mechanism based on the observation of task termination.

# 6. CONCLUSION AND RELATED WORKS

In this paper, we have proposed the specification of the X-SaGe exception handling system for replicated agents. X-SaGe firstly offers agent programmers an EHS that works transparently with replicated agents. It secondly offers to the replication system programmers the capacity to implement new replication strategies based on the signaling of programmers' code related exception by replicas. We have described such possible strategies for passive and active replicas. It can finally offer to the replication system programmers the capacity to internally control internal exceptions raised by replication algorithms, as proposed in [11]. This last point is not developed in this paper. We have proposed an original and light programming Api, using java annotations to define handlers and resolution functions. We propose a handler search and a handler invocation algorithms that relies on service execution history and, when possible, work asynchronously to improve agent reactivity. The implementation of our specification in the context of the Dimax [7] software which does not yet provide EH capabilities, is in progress.
Concerning related works, there are few studies on mixing exception handling and replication and as far as we know, no other in the agent context. [11] has proposed internal strategies to enhance a majority voting algorithm for replicated processes thanks to the handling of sequencing exceptions or hardware failures via an EHS. What is done for hardware failure has partially influenced our strategies for replication in presence of exceptions. The system is also able to report exceptions to callers. [9] has proposed an initial study to combine distributed object-oriented programming and N-version programming and [18] proposes an Ada framework

for the same purpose. Our handler at the RM level globally plays the same role for exceptions than the "exception adjudicator" of [18]. One main difference with our proposal is that the control of the consistency of versions is more complex with N-version since versions are defined by different programmers whereas replication simply duplicate agents that run the same code on different processors. For this reason we have been able to propose different strategies to return responses to clients as soon as possible without waiting all responses from replicas.

As replication is transparent to programmers, our exception handling system can easily be compared with other existing EHSs. Various proposals address the issues related to exception handling for active objects integrating asynchronous communication [2, 10, 3, 8, 13, 14, 15, 1]. Our solution is original in that it combines the following features: handling of request / response interactions between agents, handling of agent replicas, encapsulation and reactivity, ability to write context-dependent dynamic scope handlers (caller-contextualization), ability to coordinate and control groups of active agents collaborating to a common task, ability to configure the exception propagation policy by defining *exception resolution functions* at the service level.

These specification and implementation are first steps and we wish to develop many points in future works. In a first step, the interactions between the replication mechanism and the exception handling system have to be further analyzed for system-level and application-level exceptions to refine replication strategies. We then plan to enhance DIMAX capabilities using the exception handling system as a "last chance" mechanism to signal failures when the DARX replication system has failed. We also look forward to use replication as a support to give the core implementation of an EHS that supports a resumption policy. Indeed, even if handler search is stack destructive, as in most systems, a replica of an agent could restart the computation where it has been stopped in the original one.

**Acknowledgments.** Authors wish to thank the French Research Agency[2] (ANR) that supported part of this work through the FACOMA project of the SetIn 2006 program. They also want to thank A. Romanovsky for his help on the bibliography and all colleagues from the FACOMA project — J.-P. Briot, Z. Guessoum, O. Marin and J.-F. Perrot — for fruitful and inspiring discussions.

## 7. ADDITIONAL AUTHORS

## 8. REFERENCES

[1] N. Cacho, K. Damasceno, A. F. Garcia, A. Romanovsky, and C. J. P. de Lucena. Exception handling in context-aware agent systems: A case study. In *Proc. of SELMAS'06*, pages 57–76, 2006.

[2] R. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE TSE*, SE-12 number 8(8):811–826, August 1986.

[3] R. Carlsson, B. Gustavsson, and P. Nyblom. Erlang: Exception handling revisited. In *Proc. of the 3rd ACM SIGPLAN Erlang Workshop*, 2004.

[4] C. Dony, J. Knudsen, A. Romanovsky, and A. Tripathi, editors. *Advanced Topics in Exception Handling Techniques*. LNCS, vol. 4119. Springer, 2006.

[5] C. Dony, C. Urtado, and S. Vauttier. Exception handling and asynchronous active objects: Issues and proposal. In Dony et al. [4], chapter 5, pages 81–101.

[6] J. B. Goodenough. Exception handling: Issues and a proposed notation. *In CACM*, 18(12):683–696, 1975.

[7] Z. Guessoum, N. Faci, and J.-P. Briot. Adaptive replication of large-scale multi-agent systems: towards a fault-tolerant multi-agent platform. In *Proc. of SELMAS'06*. Vol. 3914 LNCS, Springer, 2006.

[8] A. Iliasov and A. Romanovsky. Exception handling in coordination-based mobile environments. In *Proc. of COMPSAC'05*, pages 341–350, Edinburgh, Scotland, UK, 2005.

[9] V. Issarny. An exception handling model for parallel programming and its verification. In *Proc. of the ACM SIGSOFT'91 Conf. on Software for Critical Systems*, pages 92–100, New Orleans, LA, USA, 1991.

[10] A. W. Keen and R. A. Olsson. Exception handling during asynchronous method invocation. In B. Monien and R. Feldmann, editors, *Proc. of Euro-Par 2002*, LNCS, pages 656–660. Springer, 2002.

[11] L. Mancini and S. Shrivastava. Exception handling in replicated systems with voting. In *Digest of papers, Fault Tol. Comp. Symp-16*, pages 384–389, 1986.

[12] O. Marin, M. Bertier, and P. Sens. Darx—a framework for the fault-tolerant support of agent software. In *Proc. of ISSRE'03*, page 406. IEEE CS, 2003.

[13] R. Miller and A. Tripathi. The guardian model and primitives for exception handling in distributed systems. *IEEE TSE*, 30(12):1008–1022, 2004.

[14] S. Mostinckx, J. Dedecker, E. G. Boix, T. V. Cutsem, and W. D. Meuter. Ambient-oriented exception handling. In Dony et al. [4], pages 141–160.

[15] E. Platon, N. Sabouret, and S. Honiden. A definition of exceptions in agent-oriented computing. In *Proc. of ESAW*, pages 161–174, 2006.

[16] B. Randell, A. Romanovsky, C. Rubira-Calsavara, R. Stroud, Z. Wu, and J. Xu. From recovery blocks to concurrent atomic actions. In *Predictably Dependable Computing Systems*, pages 87–101, 1995.

[17] A. Romanovksy and J. Kienzle. *Advances in Exception Handling Techniques:*, volume 2022 of *LNCS*, chapter Action-Oriented Exception Handling in Cooperative and Competitive Concurrent Object-Oriented Systems, pages 147–164. Springer, 2001.

[18] A. Romanovsky. An exception handling framework for n-version programming in object-oriented systems. In *Proc. of ISORC'00, 15-17 Mar. 2000, Newport Beach, CA, USA*, pages 226–233. IEEE CS, 2000.

[19] F. Souchon, C. Dony, C. Urtado, and S. Vauttier. Improving exception handling in multi-agent systems. In *Software engineering for multi-agent systems II, Research issues and practical applications*, number 2940 in LNCS, pages 167–188. Springer, 2004.

[20] F. Souchon, C. Urtado, S. Vauttier, and C. Dony. Exception handling in component-based systems: a first study. In *Exception Handling in Object Oriented Systems: towards Emerging Application Areas and New Programming Paradigms Workshop (at ECOOP'03 international conference) proceedings*, pages 84–91, 2003.

[2] `http://www.agence-nationale-recherche.fr/`.