An Exception Handling System for a Replicated Agent Environment and its Implementation in DimaX

Christophe Dony, Selma Kchir, Chouki Tibermacine
LIRMM, CNRS and Montpellier University, France
{dony,kchir,tibermacin}@lirmm.fr

Christelle Urtado, Sylvain Vauttier
LGI2P, Ecole des Mines d'Alès, Nîmes, France
{Christelle.Urtado,Sylvain.Vauttier}@mines-ales.fr

Sylvain Ductor and Zahia Guessoum
LIP6, Pierre and Marie Curie University, France
{Sylvain.Ductor,Zahia.Guessoum}@lip6.fr

Abstract

Exception handling and replication are two complementary mechanisms that increase software reliability. Exception handling helps programmers in controlling situations in which the normal execution flow of a program cannot continue. Replication handles system failures through redundancy. Combining both techniques is a first step towards building a trustworthy software engineering framework. This paper presents some of the results from the FACOMA project. It proposes the specification of an exception handling system for replicated agents as an adaptation of the SaGE proposal. It then describes its implementation in the DIMAX replicated agent environment.

1 Introduction

Exception handling and replication are two well-known mechanisms to enhance software reliability, and more particularly fault-tolerance. Replication handles system fail-stops. Exceptions enable programmers to dynamically handle situations that prevent software from running normally. These two mechanisms handle complementary causes of untrustworthiness. Associating exception handling techniques and replication can therefore be a first interesting step towards building a trustworthy software engineering framework. Such association is one of the objectives of the FACOMA project[1] where it is applied to making reliable multi-agent software.

The DimaX replicated agent platform [14, 8] is a multi-agent platform that can recover from fail-stops. To do so, the platform seamlessly manages agent replicas. It is thus able to replace an agent that failed by one of its 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 to answer messages for a given amount of time, either because network connections are lost or because the machine on which the agent ran is down. Active replication systems such as DimaX include algorithms capable of identifying the most critical agents to automatically replicate them. Messages sent to a replicated agent are transmitted to all its replicas, which process the same message in parallel. Responses are all filtered except for those from a single replica, the *leader*.

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 in the standard way but that he is still alive. The primitives offered by the exception handling system (EHS) enable agent programmers to both signal the problem and search for handlers and write handlers that propose alternative behaviors to put the system back into a coherent running state or smoothly terminate its execution.

Exception handling and replication do not apply to the same situations and are of different nature : replication is preventive and exception handling is curative. Both mechanisms are obviously very complementary. The motivation for combining them as a proposal to make agents reliable is threefold : i) combination of replication mechanisms and

[1]The FACOMA project (`http ://facoma.lip6.fr/`) is partly financed by the french national research agency (ANR).

exception handling is a new and interesting challenge for software reliability ; ii) Exception handling 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, 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 ; iii) Replication can also improve exception handling 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 [21], components [22] and active objects [5] and the DIMAX replicated agent system [8, 14] presented in Sect. 2.

The remainder of this paper is structured as follows. Section 2 sets the context of this work, describing the targeted DIMAX repliacted agent patform. Section 3 abstracts the requirements for exception handling 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 while Sect. 4.2 discusses how system-defined handlers can be defined in the replication system to integrate exception handling to the replication manager. Section 5 presents the implementation of the X-SaGE EHS within the DimaX platform and an illustrative case study. Section 6 discusses related works. Section 7 concludes with open perspectives to this work.

2 Overview of the DimaX 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 [8] and the DARX fault-tolerant middleware [14].

2.1 The DIMA 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 the *BasicCommunicatingAgent* agent base class. The control behavior of the agent is defined by the *run* method which implements a loop that is executed while the agent is alive. Each iteration of this control loop calls the *step* method that implements the decision mechanism. 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 *AgentManager* class. This enables agents to execute on top of different platforms, which can adapt their specific execution model by specializing its *AgentManager*. 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.

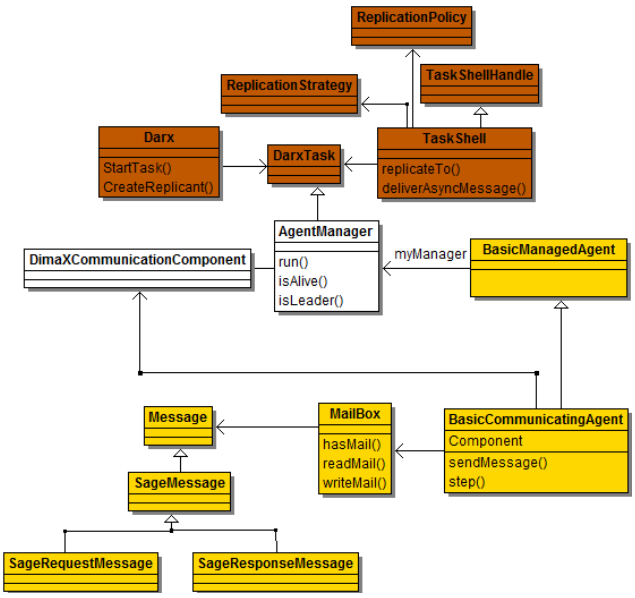

FIG. **1. Excerpt of the Class Diagram of the Replicated Agent Platform**

Agents interact by exchanging asynchronous messages. Each agent holds a message box (*MailBox* class) and a communication interface to send and receive messages.A specific semantic 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.

2.2 The Replication System

Agents are executed on a middleware which provides a fault-tolerant execution context thanks to a replication mechanism [14]. The execution context consists of a set of distributed replication servers which manage the execution of tasks (*DARXTask* class of Fig. 1). Every task belongs to a replication group that identifies the set of tasks which are replicas of a same logical task. All tasks within a replication group thus have the same behavior. They only differ by the environments in which they are running (machines of different kinds that have 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. The leader forwards the messages sent to the task to its active replicas so that they do the same computation and reach the same new state. 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 (*ReplicationPolicy* class), 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.When the leader fails, its responsibility is transferred to another 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 exception handling is combined with replication.

3 EHS from a programmer point of view

This section presents the exception handling system (X-SaGE) from the application developer point of view. It specifies the control structures for defining handlers and for signaling exceptions illustrated by an example of an e-commerce application. As replication must be transparent to the agent programmer, handling replication will intervene in the implementation of these control structures, which is detailed in Sect. 4.

3.1 Requirements for an EHS for Agent Programming

The key requirements of an exception handling system for agent-based application programming, extended from [21, 5], are :

- to enforce agent encapsulation,
- to take into account collaborative concurrent activities [19] and to provide mechanisms for their coordination and control [18],
- instead of delegating exception handling to specialized agents, to look for handlers in the runtime history and to execute handlers in their lexical definition context ; we call this caller contextualization [5] for handler definition and execution. 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 [10],
- and, to support asynchronous signaling and handler search, in order to preserve agent reactivity.

Our specification of the EHS is decomposed in four parts indicating : (1) at which program level exception handlers can be defined ? (2) how exceptions can be signaled ? (3) what can be specified within exception handlers to put the system back into a consistent state ? and (4) in which order handlers are looked up ? 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.

3.2 Signaling Exceptions and Specifying Handlers

To show how exceptions can be signaled and how they are treated by handlers at different levels of agent programs, we have considered an example of an e-commerce application. In this simple system, the user can search for flight prices in different airline companies and travel agencies as shown in Figure 2. The agent named `Client` sends initially its request to an agent called `Broker` which transmits it to travel agencies and airline companies and returns the prices to the Client.

Figure 3 shows the java code of the Broker 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. The code shows examples of service definitions (annotated by *@service*) : lines 4–10 define the *getService* service and lines 28–35 the *receivePrice* 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.

```
( 1)  public class Broker extends SaGEManagerAgent
( 2)    { ...
( 3)    // service provided by the Broker agent
( 4)   @service public void getService (Destination destination, Date date){
( 5)    sendMessage(companies, new SageRequestMessage("getPrice",destination,date,"getService")
( 6)     {  @requestHandler public void handle(Exception e){ retry(); }
( 7)     });}
( 8)
( 9)   //handler associated to getService service
(10)   @serviceHandler(serviceName="getService")
(11)    public void badParametersService(noFlightAvailableException e){
(12)      date = date.nextDay();
(13)     retry();
(14)  }
(15)   //handler associated to getService service
(16)   @serviceHandler(serviceName="getService")
(17)    public void TechnicalProblem(TemporaryTechnicalProblem e){
(18)           wait(120);
(19)          retry();
(20) }
(21)    //handler associated to Broker Agent
(22)     @agentHandler  public void handle(noFlightAvailableException e){ signal(e); }
(23)
(24)     //service provided by the broker agent
(25)    @service public void receivePrice(Object agence, int price){
(26)       sendMessage(Client,new SageResponseMessage("receiveService",agence,price,"receivePrice"){
(27) @requestHandler public void handle(Exception e) {
(28)            signal(new NoFlightForDestination(e));
(29) }
(30) });
(31)
(32)
(33)    // resolution function associated to the getService service
(34)    @serviceResolutionFunction(servicename="getService") public TooManyProvidersException concert ()
(35)       {
(36)       int failed = 0;
(37)       for (int i=0; j<subServicesInfo.size(); i++)
(38)          if ((ServiceInfo) (subServicesInfo.elementAt(i)).getRaisedException() != null)  failed++;
(39)       if (failed > 0.3*subServicesInfo.size())  return new TooManyProvidersException(numberOfProviders);
(40)       return null;
(41)       }
(42)   }
```

FIG. **3. Service, handler and resolution function definitions in X-SaGE using annotations**

Signaling exceptions is performed by a classical *signal* primitive (*cf.* Fig. 3, line (28)). Signaling is possible anywhere in the code. This includes the possibility of signaling an exception from within handlers.

The underlying request / response interaction pattern of the agent model highlights 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. This is illustrated in Figure 2 by the colored squares. The position of these squares in the figure represents the order in which these handlers are invoked. More details are given in Sect. 4.

– Exception handlers can be attached to **requests** using the `@requestHandler` annotation. Such handlers can, for example, specify two distinct reactions to the occurrence of two identical exceptions raised by two invocations of the same service. Line 6 of Fig. 3 depicts how a handler can be attached to a specific request.
– Exception handlers can be attached to **services**. Such handlers manage exceptions that are raised, directly or indirectly, by some service's execution. If the service makes concurrent calls, the handler has to be able to deal with concurrent exceptions, to compose with partial results or to ignore partial failures. Lines 10–20 of Fig. 3 show the code of two handlers attached to a same service. Note that the *serviceName* attribute of the annotation `@serviceHandler` allows to identify the service which is protected by the handler.
– Finally, exception handlers can be associated to

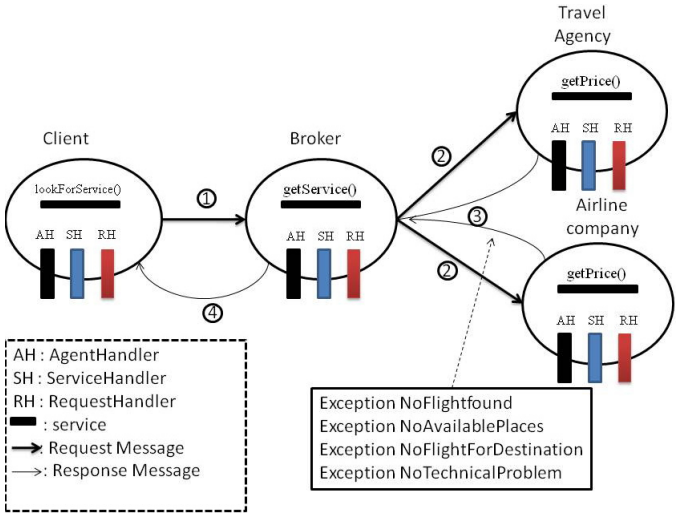

FIG. **2. Exceptions and Handlers defined in an e-commerce application example**

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 the consistency of the agent's private data. Line 22 of Fig. 3 shows how such handlers can be attached to agents using the`@agentHandler` annotation.

These capabilities are powerful enough to encompass most frequent cases the agent programmer is confronted to, and simple enough to be easy to learn and use.

3.3 Defining Handler Bodies and the Resolution Function

Exception handlers are defined by the set of exception types they can catch and by their code body (as illustrated by Fig. 3). There are three main actions a handler can classically perform :

- a handler can restore an agent's state 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.
- a handler can **signal** a new exception (generally of a higher conceptual level) or **re-signal** the same one it captures (Fig. 3, line 28).
- a handler can **retry** the execution of the program unit it is attached to (Fig. 3, lines 11–13).

A programmer can define his own exception resolution function using the *@serviceResolutionFunction* annotation as shown in the example of Fig. 3, lines 33–40. X-SaGE enables resolution functions to be defined at places where concurrent activities are launched and have to be coordinated (*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 called agents have replied, to aggregate all the exceptions that occurred into a new concerted exception. Another possible 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 [10]. A resolution function is executed each time an exception handler is searched for at the service level. This is detailed in the next section, where we present the implementation of the handler search.

4 Combining Exception Handling and Replication

Now, let us suppose that the `Travel Agency` and `Airline Company` agents are replicated. While deploying the application on the DimaX platform, the criticality of these agents have been evaluated and the obtained values led the platform to duplicate them[2] as shown in Fig. 4.

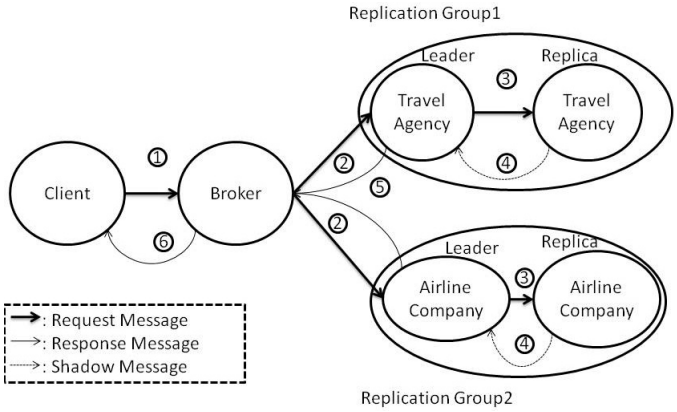

FIG. **4. The example with Replicated Agents**

With a replication group which replaces a single agent, message exchange between agents slightly changes. When the `Travel Agency` agent receives a request message from the broker, it broadcasts this message to all the replicas. Concretely, there is a replication group leader which holds all the replicas identities. The different replicas process the message and send their response to the leader. These messages are of a specific kind of response messages, called **shadow messages**. These are represented by dashed arrows in Fig. 4. The reason why we introduced another

[2]Details about agent criticality and replication can be found here [6].

kind of messages is an implementation detail, which is explained in Sect. 5.

Suppose now that an exception is raised by one of the agent replicas. What will be the behavior of the EHS ? In the following sections, we will answer with detail to this question.

4.1 Handler Search in a Replicated Multi-Agent Environment

In handler search, we monitor a tree of **service execution contexts**. Each node in this tree represents a service execution context and records the identities of the service being executed and the agent that owns the current service. Each node can have a parent node that links to the calling context of the current service. In this parent node, the request that triggered the execution of the current service is recorded. Links between nodes (callee to caller links) are used to look for handlers.

If an exception is raised within an agent's service, then the execution of the service is suspended and handler search is started. The handler search process is composed of four steps. Figure 5 shows a flowchart that synthesizes the different steps of the handler search process.
First, if there is a resolution function at the service level, it is executed. We enumerate three possible cases during resolution :

1. the exception is considered as critical for the service. The resolution function returns the exception object and handler search carries on.
2. the resolution function considers the exception under-critical and that nothing more should be done yet. The exception is thus 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 considers the exception 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.

If there is no resolution function, a handler for the exception is then searched in the list of handlers attached to the service. If a handler exists, it is executed. If no handler is found at the service level, the EHS looks for a handler at the agent owner of the service. If a corresponding 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 potential pending services called by the current service are terminated. If no handler has been found at the agent level, and if the agent is replicated, control is given to its replication manager (RM). Each RM has a special kind of a resolution function (*replication-level* or *system-level resolution function*) the goal of which is to coordinate the answers given by replicas of the agent. Contrarily to the first kind of resolution function introduced previously (*application-level resolution function*), these functions are not defined by agent programmers. They are predefined in the EHS, and their behavior is described in Sect. 4.2). At the end of the execution of this resolution function, an RM handler is invoked. This handler is responsible for propagating the exception to the caller agent.

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 context of the calling service. The list of handlers attached to the request which initiated the called service is searched first. If a handler exists, it is executed and the search stops. Then, the search proceeds by starting again at the first step of the process, executing the (application-level) resolution function associated with the service, if any, then searching the list of handlers attached 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, which displays the message embedded in the exception, its name and the stack trace.

4.2 Handling Exceptions at the RM Level

When an agent's replica raises an exception, the EHS invokes the replication manager. The RM will either, as described in the following section, put the system back into a consistent state, signal a new exception to the request caller or propagate one of those trapped by the resolution function. In this latter case, the handler search will continue as explained in Sect. 4.

4.2.1 Typology of exceptions

The first global task performed by the replication-level resolution function 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. This function distinguishes two kinds of exceptions, replica-specific (examples of this include exceptions raised when some resources specific to a given replica are unavailable) and replica-independent (for example, 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

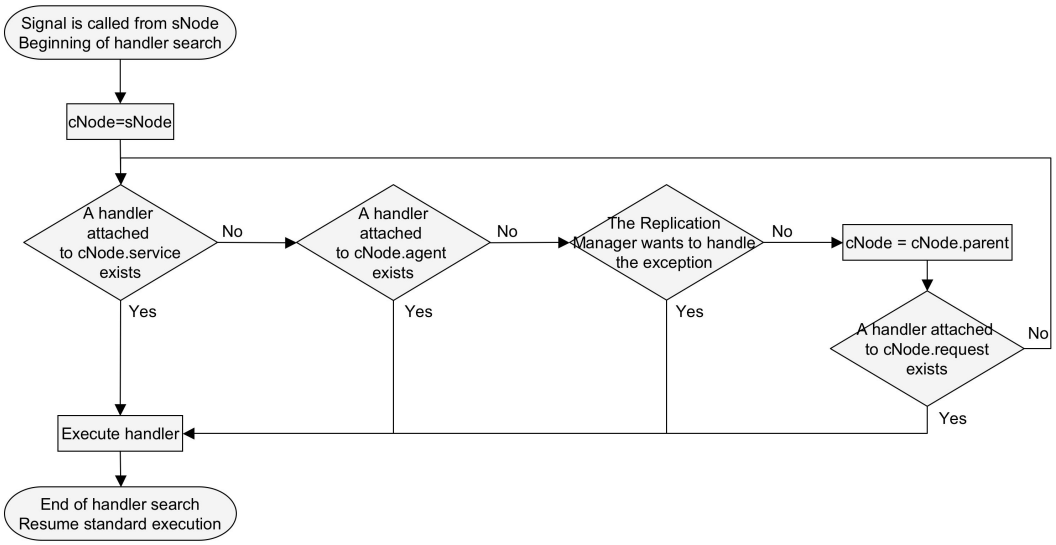

FIG. **5. Flowchart for handler search**

another replica retrying the same computation. This would affect the performance of the program execution.
We thus have designed our algorithms based on a classification of exceptions. Goodenough's seminal paper [7] has proposed a first classification in *domain*, *range* and *monitoring* exceptions. The classification criteria here is "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 replica-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 relies on exception semantics and distinguishes *business* (also called *domain* or *applicative*) exceptions from *system* or *resource* exceptions. System exceptions are raised by the runtime environment and are likely to reflect a specific communication or resource lack problem. In our context, system exceptions can be raised by the Java virtual machine, by the agent language interpretor or by the replication middleware. They can be considered as replica-specific. Business exceptions are direct consequences of a programmer's code. Under the assumption that all the replicas of an agent have the same deterministic behavior, exceptions identified as business exception can be considered as replica-independent : they must be raised by all replicas of a given agent or by no replica. The question of knowing how to distinguish exceptions has been resolved statically, by typing exception classes with two distinct interfaces.

Beyond this classification, the strategies of the exception handler of the replication manager also take into account the composition of the replication group. Three strategies are described in the following sections.

4.2.2 Controlling a set of 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 check if 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 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.

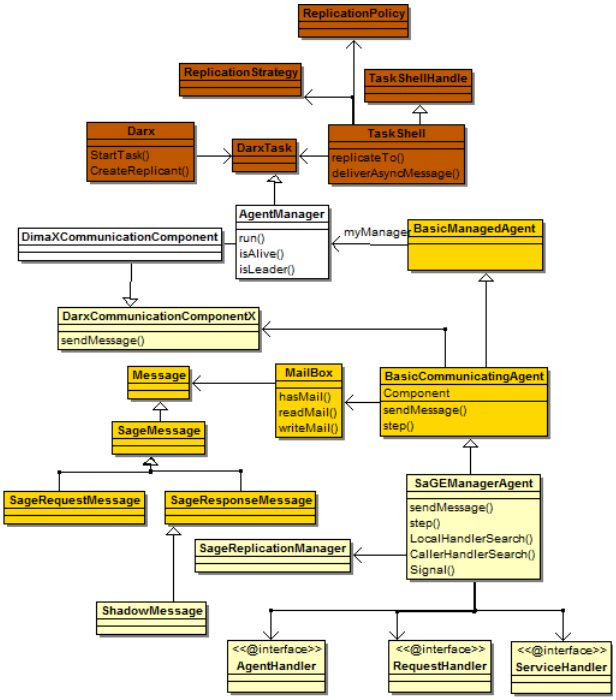

FIG. **6. EHS integrated to DimaX**

5 Implementation and Experimentation of the Exception Handling System

This section presents first the implementation of the EHS in the Dimax platform, and then exposes a case study as an experiment we conducted on this implementation.

5.1 Implementation of the EHS

To implement our EHS, we have created a new class `SaGEManagerAgent`. All agents of our example application are instances of this class (see Figure 3). All the annotations introduced in Sect. 3 (`@service`, `@serviceHandler`, `@agentHandler`, `@requestHandler` and `@serviceResolutionFunction`) are implemented as traditional JAVA annotation interfaces.

Agents communicate by exchanging messages. If we want to handle answers sent by replicas, we have to distinguish request messages from response ones. To implement this, we have specialized the method `sendMessage()` of SaGEManagerAgent to send directly request messages through Darx classes to agent addresses and to redirect response messages to the class `DarxCommunicationComponentX` where they will be handled before passing through Darx classes. In DarxCommunicationComponentX, response messages sent by replicas are encapsulated in a kind of response messages that we call shadow messages. `SageShadowMessage` is a subclass of `SageResponseMessage`. Once replicas responses have been encapsulated in shadow messages, they are registered in a mailbox of the leader using the method `receiveAsyncMessage(m)` of the class `AgentManager`.

To summarize, all responses computed by replicas of an agent, either normal responses or exceptions, are sent back to their leader agent as shadow messages. The leader is then able to record, filter, dismiss or concert responses so as to send a unique response to the client (if needed).

When an exception is signaled by the service of a replica, searching for a relevant handler is done thanks to the class `SaGEManagerAgent` where the method `localHandlerSearch(e)` is defined. If there is no handler found, the search process is delegated to the Replication Manager of the leader class represented by the class `SageReplicationManager`. This class is instanciated each time a new agent is created, in other words, each leader of a replication group has its own replication manager. Depending on the type of the exception, Replica-Specific or Replica-Independant, the exception is logged or is signaled to the caller thanks to the method `callerHandlerSearch(e)` of the class `SaGEManagerAgent`.

For further details about the implementation, the reader is invited to visit the following link :
`http ://www.lirmm.fr/~kchir/DimaxX/`

5.2 Case Study : Search for Flights

We used our exception handling system in the implementation of the example introduced in the previous section (the “search for flights" application). The execution of the application passes through the Darx layer to create a name server and to launch the server on local and remote machines. Darx passes then the control to Dimax for agent creation on the local machines. Each agent is encapsulated in a `TaskShell` which contains specific informations to each agent (identifier, task name, ...) and can manage replication groups.
When an agent of a specific task is created for the first time, it creates a replication group and it registers itself as a leader of this task. As mentioned previously in Sect. 4, the agents `Travel agency` and `Airline company` are duplicated.

Once all agents (leaders and replicas) have been created, the service `lookForService()` of the `Client` is executed and sends a request message to the service `getService()` of the `Broker`. The latter transmits the client’s request to the travel agency and airline company agents and invoke their `getPrice()`

service. As those agents are replicated, they forward the message they received from the broker to their replicas (method `deliverAsyncMessage()` of `ActiveReplicationStrategy`). Each replica executes its service and returns its response to its leader encapsulated in a shadow message. While executing its service, one of the replicas of the agent Travel Agency signaled the exception `NoFlightAvailableException` which implements the interface `ReplicaIndependant`. The execution of the service `getPrice()` is so suspended and a handler is searched in the agent class. As no handler has been found locally, the search process continues as described in Sect. 3. The resolution function detects that the type of the exception is `ReplicaIndependant` and the handler of the replication manager sends the message *callerHandlerSearch(e)* to the broker which traps the exception in its request handler.

Regarding the agent Airline Company, one of its replicas signals a `ReplicaSpecific` exception : `TemporaryTechnicalProblem`. The search for a handler passes through the steps described previously. The resolution function detects that the exception is `ReplicaSpecific`, log it and wait for other answers of the replicas. All replicas have signaled a Replica-Specific exception then the handler of the replication manager is called. The handler of this RM sends the message *callerHandlerSearch(e)* to the broker like in the previous case (`e` is the exception the most signaled by replicas, which is `TemporaryTechnicalProblem`).

Without the EHS, the exception signaled by the replica of the airline company could not be trapped and logged. Also, the agent travel agency could not send a response to the broker. Unknown behavior in case of exceptions generates thus a failure of the system, while it was possible to avoid this situation.

6 Related Work

Multi-agent systems are inherently concurrent, because of the autonomy of agents [23]. Specific EHS have been designed to handle these concurrency situations when multiple exceptions are asynchronously signaled [2, 11, 3, 9, 15, 16, 17, 1]. Most of these EHS are based on a dedicated entity (called a monitor, guardian, action, ...) which monitors a group of active concurrent entities and handles the exceptions signaled by any member of the group This kind of EHS is designed to handle exceptions that impact collectively a group of entities which share an execution context. Fewer works study the integration of EHS to asynchronous service (method) invocation schemes. This kind of EHS are best suitable for managing applicative (business) exceptions pertaining to peer-to-peer collaborations. However, they are basic EHSs provided by middlewares which lack concurrent exception resolution and propagation. Our EHS can be considered as an original hybrid of these two approaches, able to seamlessly manage the execution of complex collaborations spawning over groups of agents, while preserving agent behavior encapsulation and reactivity thanks to the support for contextualized exceptions handlers defined as part of the applicative behavior of agents.

The work presented in this paper is an extension of Sage which addresses replicated agents. Replication is a special case of n-versionning, which consists in executing multiple versions of a given functionality for robustness and fault-tolerance. All the results returned by the multiple versions are collected and compared. The effective result of the execution is resolved as a majority vote. [13, 10, 20] have studied exception handling in n-versionning frameworks. A main difference in our work is that the responses returned by a group of agent replicas are not resolved by votes. As all the replicas are identical in a group, they should return the same responses, as far as their executions are predictable (are not interrupted by contextual exceptional situations). The responses of the group leader are thus returned primarily. When the leader undergoes a failure (returns a contextual runtime exception), the other replicas are used as failovers. The first replica which succeeds in returning a valid response (whether a result or an expected business exception) becomes the leader. This resolution strategy promotes execution performance. A perspective is to compare all the valid responses of the replicas in order to detect byzantine faults [12] and also address robustness issues.

7 Conclusion and Discussion

In this paper, we have presented an exception handling system combined with a replicated agent environment for improving software reliability. This EHS firstly offers agent programmers an exception signaling and handling mechanism that works transparently with replicated agents. It second provides to the replication system programmers the capability to control internal exceptions raised by the replication algorithms, as proposed in [13]. This last point is not developed in this paper. We have proposed an original and lightweight programming API, using java annotations to define handlers and resolution functions. We detailed then a handler search and a handler invocation algorithms that relies on service execution traces. This works asynchronously to improve agent reactivity. All the propositions have been implemented on the DimaX platform and instrumented through a case study.

As a future work, we wish to further analyze the interactions between the replication mechanism and the exception handling system in order to refine replication strategies (to deal with passive and semi-active replication). Besides, we will consider comparing the responses (normal responses or

exceptions) to a same request given by several replicas of an agent in order to learn some agent's execution behavior. 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. This allows us to dynamically distinguish between replica-specific and replica-independent exceptions, without typing statically exception classes. We plan at last 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 (ANR) that supported 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, O. Marin and J.-F. Perrot — for fruitful and inspiring discussions.

Références

[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] N. Faci, Z. Guessoum, and O. Marin. Dimax : A fault-tolerant multi-agent platform. In *EUMAS*, 2006.

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

[8] 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.

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

[10] 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.

[11] 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.

[12] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva : Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4) :1–39, 2009.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

[19] 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.

[20] A. Romanovsky. An exception handling framework for n-version programming in object-oriented systems. In *Proc. of ISORC'00*, pages 226–233. IEEE CS, 2000.

[21] 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.

[22] F. Souchon, C. Urtado, S. Vauttier, and C. Dony. Exception handling in component-based systems : a first study. In *Proc. of Workshop at ECOOP'03*, pages 84–91, 2003.

[23] M. Wooldridge. *An Introduction to MultiAgent Systems - Second Edition*. 2009.