Combining Exception Handling and Replication for Improving the Reliability of Agent Software

Christophe Dony[1], Sylvain Ductor[2], Zahia Guessoum[2], Selma Kchir[2],
Chouki Tibermacine[1], Christelle Urtado[3], and Sylvain Vauttier[3]

[1] LIRMM, CNRS and Montpellier University, France
{Christophe.Dony,Chouki.Tibermacine}@lirmm.fr
[2] LIP6, Pierre and Marie Curie University, France
{Sylvain.Ductor,Zahia.Guessoum,Selma.Kchir}@lip6.fr
[3] LGI2P, Ecole des Mines d'Alès, Nîmes, France
{Christelle.Urtado,Sylvain.Vauttier}@mines-ales.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. Exception handling enables 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 is a first interesting step towards building a trustworthy software engineering framework. Such association is one of the objectives of the FACOMA project[4] where it is applied to making reliable multi-agent software.

The DIMAX replicated agent platform [15,9] 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 those events signaled when the standard control flow of a running service cannot continue. An exception is a kind of answer from an agent running the service. It indicates that the agent is unable to continue its task in the standard way but that it is still alive. The primitives offered by the exception handling system (Ehs) enable agent programmers to signal the problem, to search for handlers and to write handlers that propose alternative behaviors to put the system back into a consistent 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. If studies can be found on how to combine n-versionning and exception handling [14,11,21], combining exception handling and replication appears to be an original and promising solution. This paper describes X-SaGE, a specification and concrete running implementation, achieved in the Dimax [9,15] context of a programming language running on top of a replication middleware, of such a combination. It extends, adapts and provides a running implementation to our primary specification described in [5]. The main issue raised by this combination are: i) to provide agent programmers with an exception handling system that behaves correctly in presence of seamless active replication, and ii) to improve replication strategies, on the base of information conveyed while propagating exceptions from replicas.

| Chouki | *A relire*

The remainder of this paper is structured as follows. Section 2 briefly describes the Dimax replicated agent platform which is the context of the implementation. In Section 3, we introduce the Api of our X-SaGE exception handling system and an example of a user-defined program that uses X-SaGE within Dimax. This example will support the case study discussion in section 5 and shows that replication is seamlessly managed for end-programmers. Section 4 discusses key issues of the implementation of the combination exception/replication : i) how to view a replication manager as a broadcasting agent, ii) how to adapt handlers search to the presence of replication managers and iii) how to deal with exceptions at the replication manager level. In Section 5 we present a case study that highlights the interest of exception concertation made at the replication manager level. Section 4.2 describes the key-aspects of implementation of the X-SaGE Ehs within the DimaX platform. Before concluding with open perspectives to this work in Section 7, we discuss related works in Section 6.

2   Implementation context : 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 [9] and the DARX fault-tolerant middleware [15].

A DIMAX agent is an active and autonomous computation entity that executes in its own thread. Agents interact by exchanging asynchronous messages. Each agent holds a message box and a communication interface to send and receive messages. 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 a result, either standard or exceptional, back to the client agent, via a response message. Response messages are correlated with request messages.

Agents are executed on a middleware which provides a fault-tolerant execution context thanks to a replication mechanism [15]. The execution context consists of a set of distributed replication servers which manage the execution of tasks. 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, 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. As such, agents can be replicated by the middleware and benefit from this fault-tolerance mechanism.

3   Exception programming in a multi-agent context

This section describes an exemple of exception handling programming with X-SaGE in the Dimax context that will support the following sections discussions. The rationale for the SaGE exception handling system used in this example can be found in [6,22]. Its main requirements are:

- to enforce agent encapsulation,
- to take into account collaborative concurrent activities [20] and to provide mechanisms for their coordination and control [19],
- to look for handlers in the runtime history and to execute handlers in their lexical definition context;
- to handle concurrent exceptions with resolution functions [11],
- and, to support asynchronous signaling and handler search, in order to preserve agent reactivity.

As an illustration, we use the canonical *Travel Agency* example (*cf.* Figures 1 and 2) 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 airline companies and collects their responses. Then, the *Broker* selects the best offer and requests the *Client* and the selected *Provider* to contract.
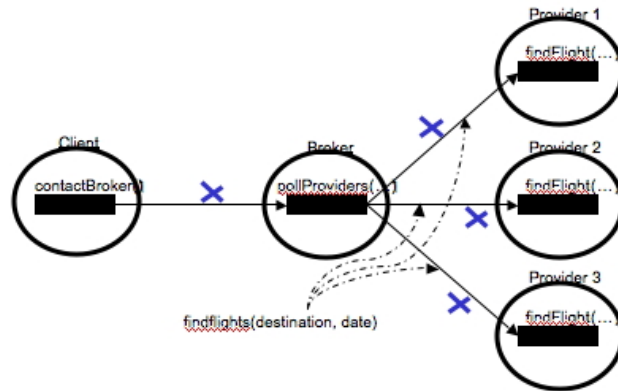

Fig. 1. An e-commerce application example with 3 agents : contractual requests and responses.

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. The system makes it possible to define handlers at the request, service and agent level and to define resolution functions at the service and agent groups
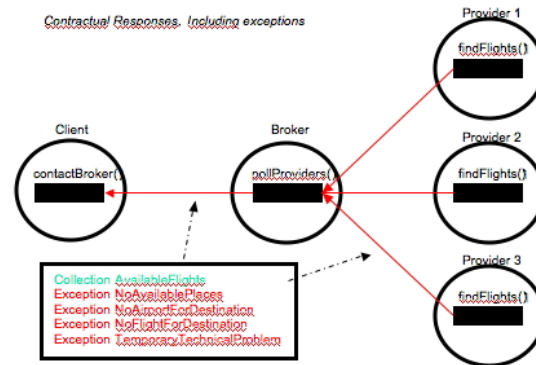
Fig. 2. An e-commerce application example with 3 agents : contractual responses.

level. This is illustrated in Figure 3 by the colored squares. The position of these squares in the figure represents the order in which these handlers are searched for. The details of handler search, augmented by the adaptation required to take replication into account, are given in Section 4.

Figure 4 shows some parts of the java code of the three agents that contain various exampls of handler definitions.

– 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.
  A handler is attached to a request message within the service `contractBroker` of class `Client` (lines 4–6). It states that, for this spcific request, if there is no place left at the given date, it is possible to change the date.
– Exception handlers can be attached to **services**. Such handlers can exceptions that are raised in the dynamic scope of the service's execution.
  As an example, a handler is attached to the `pollProvider` service within the `Broker` class. It explicitly propagates to the client for him to adapt his request, the exception `NoAirportForDestination` raised by providers.
– 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)
  A programmer can define exception resolution functions (based on a model inspired by [11]) using the *@serviceResolutionFunction* annotation as shown in the example of Figure 4, lines 20-27. such a function is called, is it exists, before any service level handler, each time an exception is trapped at a service level. Its standard behavior is, once all called agents have replied, to aggregate all the exceptions that occurred into a new concerted excep-
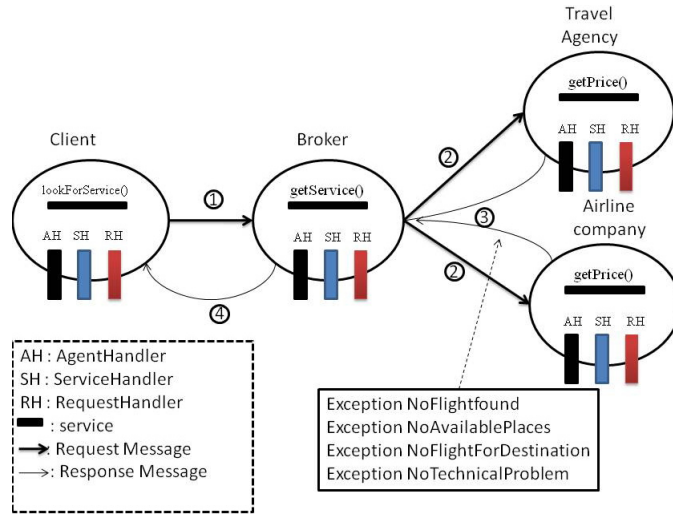
Fig. 3. Agents are potentially equiped with three kind of handlers

tion that conceptually abstract them. Other possible behavior and uses are
depicted in [6].
– Finally, exception handlers can be associated to **agents** (see Figure 4, lines
32–38). 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.

We think that 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. The overall contribution of the present is that this example
will continue to compute correctly and transparently for the users in presence of
replication. The following sections discuss this.

4 Combining Exception Handling and Replication

EHSs leverage the elements of execution models to provide adapted control struc-
tures for managing exceptions. The previous section has presented the X-Sage
exception handling system, initially designed to manage exceptions in multi-
agent systems. It provides means to associate exception handlers to the main
elements of the execution model of agents (the requests, the services and the
agents themselves). To support replication, X-Sage has been extended to take
into account replication groups as a new kind of elements in the execution model.
The exception signaling scheme is modified so that the exceptions signaled by
replicas are sent to the managers of the replication groups they belong to. A
handler and a resolution function are added to the definition of the replication
group managers so that they are able to concert and handle exceptions which

```
( 1) public class Client extends SaGEManagerAgent{ ...
( 2)
( 3)     @service public void contactBroker (...) {...
( 4)       sendMessage (broker, new SageRequestMessage (pollProvider, destination, date){
( 5)            @requestHandler public void handle (NoAvailablePlaces exc)
( 6)                    { date = date.nextDay(); retry();}}); }
( 7)     ...
( 8) }
( 9)
(10) public class Broker extends SaGEManagerAgent{ ...
(11)
(12)     @service public void pollProvider (Destination destination, Date date){ ...
(13)       sendMessage(companies, new SageRequestMessage("getPrice", destination, date)
(14)         ... }
(15)
(16)     @serviceHandler(serviceName="pollProvider")
(17)        public void handle (NoAirportForDestination e) { signal(e); }
(18)
(19)     // resolution function associated to the pollProvider service
(20)     @serviceResolutionFunction(servicename="pollProvider")
(21)        public TooManyProvidersException concert() {
(22)            int failed = 0;
(23)            for (int i=0; i<subServicesInfo.size(); i++)
(24)               if (subServicesInfo.get(i).getRaisedException() != null)  failed++;
(25)            if (failed > 0.3*subServicesInfo.size())
(26)               return new TooManyProvidersException (numberOfProviders);
(27)            return null;}
(28)     ... }
(29)
(30) public class Provider extends SaGEManagerAgent{
(31)     ...
(32)     @agentHandler
(33)        public void handle(NetworkConnectionException e){
(34)            signal(new TemporaryTechnicalProblem(...));}
(35)
(36)     @agentHandler
(37)        public void handle (DatabaseConnectionException e){
(38)            signal(new TemporaryTechnicalProblem(...));}
(39)     ...
(40) }
```

Fig. 4. Examples of request, service, agent-level handlers and resolution function definitions in X-SaGE

reflect the status of whole groups of replicas. Handling exceptions at replication manager level enhances both the dependability and performance of applications.

| Chouki | *Je pense qu'il faut preciser en quoi cela ameliore les performances.*

4.1   Handler Search in a Replicated Multi-Agent Environment

The handler search process for replicated agents is presented on figure 6. As replication is transparent, a replica is a standard agent. So, when an exception signaled to a replica, the handler search process begins with the standard Sage exception handling schemes proposed for agents (see Section 3). A request then a service and finally agent-level handlers are sought in the replica. When the exception is not caught by the handlers defined in the replica, the exception is propagated by the EHS to the manager of the replication group. The handling
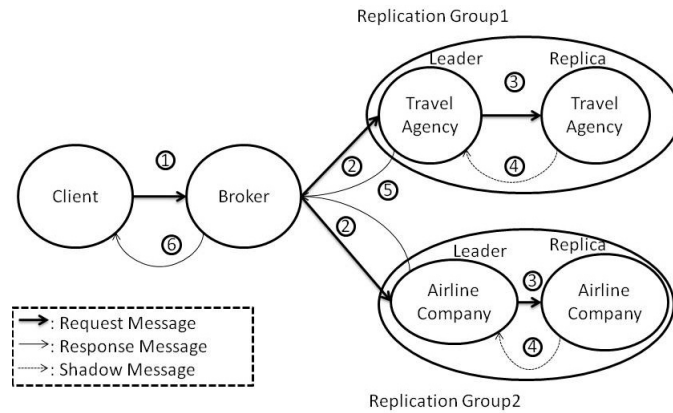
Fig. 5. The example with Replicated Agents

of the exception carries on by the execution of the resolution function associated with the replication manager. The resolution function determines whether it is worth waiting for the results of other replicas before signaling an exception to the client of the replicated agent. Indeed, the failure of one replica does not imply the failure of the whole replication group. This depends on the contextual nature of the exception, as explained in the following subsection. The resolved exception is caught by a generic handler defined in the replication manager. Its role is to signal the resolved exception first to the client of the replicated agent but also to trigger the replication strategies that determine how the organization of the replication group must evolve to maintain the dependability of the replicated agent, depending on the signaled exceptions. Without such a resolution, the exceptions signaled by the leader of the replication group would be immediately sent to the client, taking no advantage of the other replicas to provide a more dependable agent.

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 previously.

Typology of Exceptions. In order to adapt its behavior, the replication-level resolution function of an agent has to determine whether an exception raised by one of the agent's replicas will also be raised by the others. Two categories of exceptions thus have to be identified: replica-specific exceptions (*i.e.,* exceptions raised when some resources specific to a given replica become unavailable) and replica-independent exceptions (*i.e.,* a bad parameter in the request sent to the
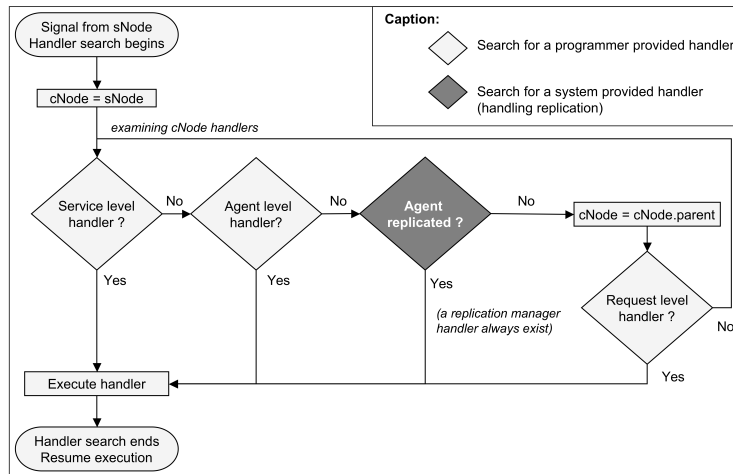
Fig. 6. Flowchart for handler search

agent that leads to a division by zero).

We thus have designed our algorithms based on a classification of exceptions. Many of such classifications exist. Goodenough's seminal paper [8] has proposed a first classification in *domain*, *range* and *monitoring* exceptions. This classification is based on "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 that should not be handled) and *Exception* (business problem that can be handled) as in Java, inherited from the *Flavors* system, highlights exception gravity but does not answer our question.

A more appropriate classification relies on exception semantics and distinguishes *business* (also called *domain* or *applicative*) exceptions from *system* (or *resource*) exceptions. Business exceptions are direct consequences of a programmer's code. Assuming all replicas of a given agent have the same deterministic behavior, exceptions identified as business exceptions can be considered replica-independent: they are raised by either all replicas of a given agent or by none of them. System exceptions are raised by the runtime environment and are likely to reflect a specific communication or resource access problem. In our context, system exceptions can be raised by the Java virtual machine, the agent language interpretor or the replication middleware. They can be considered as replica-specific.

In our system, the question of knowing how to distinguish these two exception categories has been resolved statically, by typing exception classes with two distinct interfaces.

Controlling a Set of Replicas. Once the category of the exception is known, the RM reacts adequately.

Handling a replica-independent exception. When a replica-independent 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 strategy makes it possible to check afterwards (when all requests are processed) if all replicas have actually raised the same exception and thus are in the same consistent state.

Handling a replica-specific exception. When a replica-specific exception is raised by a replica, it is logged by the resolution function of the RM. All remaining replicas pursue their computation. If a normal response is computed by any replica, it is forwarded to the client agent. The other subsequent normal responses are discarded. When all replicas have processed the request (responded either with a normal value or with a replica-specific exception), the RM destroys all the faulty (logged) replicas. If the leader is destroyed, a new leader is chosen among the remaining replicas. If all replicas are destroyed (they all ended with an exception), a service failure exception is signaled to the client agent.

Overview of the Implementation. To implement our EHS, we created a new `SaGEManagerAgent` class. All agents of our example application are instances of this class (see Figure 4). All the annotations introduced in Section 3 (`@service`, `@serviceHandler`, `@agentHandler`, `@requestHandler` and `@serviceResolutionFunction`) are implemented as Java annotation interfaces. Figure 7 depicts a class diagram of this implementation.

To distinguish request messages from response messages, we have specialized the `sendMessage()` method of the `SaGEManagerAgent` class to directly send request messages through DARX classes to agent addresses and to redirect response messages to the `DarxCommunicationComponentX` class where they will be handled before passing through DARX classes. In the `DarxCommunicationComponentX` class, response messages sent by replicas are encapsulated in a kind of response messages that we call **shadow messages**. `SageShadowMessage` is a subclass of the `SageResponseMessage` class. Once replicas' responses have been encapsulated in shadow messages, they are transfered to the leader's mailbox using the `receiveAsyncMessage(m)` method from the `AgentManager` class.

To summarize, all responses computed by an agent's replicas, either they be normal or exceptional, 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 a service of a replica, searching for a relevant handler is done thanks to the `SaGEManagerAgent` class for which the `localHandlerSearch(e)` method is defined. If no handler is found, the search process is delegated to the Replication Manager of the leader (represented by an instance of the `SageReplicationManager` class). Depending on the excep-

tion category (it being replica-specific or replica-independant), the exception is logged or signaled to the client agent thanks to the `callerHandlerSearch(e)` method of the `SaGEManagerAgent` class[5].

5 Case Study: Search for Flights

Fig. 7. EHS integrated to DimaX

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 Section **??**, the `Travel agency` and `Airline company` agents are duplicated.

---

[5] Further details on the implementation are available at the URL: http://www.lirmm.fr/~tibermacin/facoma/DimaxX/

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 Section 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 [2,12,3,10,16,17,18,1]. 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. [14,11,21] 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 [13] 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 [14]. 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.

References

1. Cacho, N., Damasceno, K., Garcia, A.F., Romanovsky, A., de Lucena, C.J.P.: Exception handling in context-aware agent systems: A case study. In: Proc. of SELMAS'06. pp. 57–76 (2006)
2. Campbell, R., Randell, B.: Error recovery in asynchronous systems. IEEE TSE SE-12 number 8(8), 811–826 (August 1986)
3. Carlsson, R., Gustavsson, B., Nyblom, P.: Erlang: Exception handling revisited. In: Proc. of the 3rd ACM SIGPLAN Erlang Workshop (2004)
4. Dony, C., Knudsen, J., Romanovsky, A., Tripathi, A. (eds.): Advanced Topics in Exception Handling Techniques. LNCS, vol. 4119, Springer (2006)
5. Dony, C., Tibermacine, C., Urtado, C., Vauttier, S.: Specification of an exception handling system for a replicated agent environment. In: Proceedings of WEH '08, the 4th international workshop on Exception handling - Atlanta, Georgia. pp. 24–31. ACM (2008), `http://www.lirmm.fr/~dony/postcript/exc-weh08.pdf`
6. Dony, C., Urtado, C., Vauttier, S.: Exception handling and asynchronous active objects: Issues and proposal. In: Dony et al. [4], chap. 5, pp. 81–101
7. Faci, N., Guessoum, Z., Marin, O.: Dimax: A fault-tolerant multi-agent platform. In: EUMAS (2006)
8. Goodenough, J.B.: Exception handling: Issues and a proposed notation. In CACM 18(12), 683–696 (1975)
9. Guessoum, Z., Faci, N., Briot, J.P.: 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)
10. Iliasov, A., Romanovsky, A.: Exception handling in coordination-based mobile environments. In: Proc. of COMPSAC'05. pp. 341–350 (2005)
11. Issarny, V.: An exception handling model for parallel programming and its verification. In: Proc. of the ACM SIGSOFT'91 Conf. on Software for Critical Systems. pp. 92–100. New Orleans, LA, USA (1991)
12. Keen, A.W., Olsson, R.A.: Exception handling during asynchronous method invocation. In: Monien, B., Feldmann, R. (eds.) Proc. of Euro-Par 2002, pp. 656–660. LNCS, Springer (2002)
13. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: Speculative byzantine fault tolerance. ACM Trans. Comput. Syst. 27(4), 1–39 (2009)
14. Mancini, L., Shrivastava, S.: Exception handling in replicated systems with voting. In: Digest of papers, Fault Tol. Comp. Symp-16. pp. 384–389 (1986)
15. Marin, O., Bertier, M., Sens, P.: Darx—a framework for the fault-tolerant support of agent software. In: Proc. of ISSRE'03. p. 406. IEEE CS (2003)

16. Miller, R., Tripathi, A.: The guardian model and primitives for exception handling in distributed systems. IEEE TSE 30(12), 1008–1022 (2004)
17. Mostinckx, S., Dedecker, J., Boix, E.G., Cutsem, T.V., Meuter, W.D.: Ambient-oriented exception handling. In: Dony et al. [4], pp. 141–160
18. Platon, E., Sabouret, N., Honiden, S.: A definition of exceptions in agent-oriented computing. In: Proc. of ESAW. pp. 161–174 (2006)
19. Randell, B., Romanovsky, A., Rubira-Calsavara, C., Stroud, R., Wu, Z., Xu, J.: From recovery blocks to concurrent atomic actions. In: Predictably Dependable Computing Systems. pp. 87–101 (1995)
20. Romanovksy, A., Kienzle, J.: Advances in Exception Handling Techniques:, LNCS, vol. 2022, chap. Action-Oriented Exception Handling in Cooperative and Competitive Concurrent Object-Oriented Systems, pp. 147–164. Springer (2001)
21. Romanovsky, A.: An exception handling framework for n-version programming in object-oriented systems. In: Proc. of ISORC'00. pp. 226–233. IEEE CS (2000)
22. Souchon, F., Dony, C., Urtado, C., Vauttier, S.: A proposition for exception handling in multi-agent systems. In: in Proc. of SELMAS'03 : 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems, Portland, Oregon (2003)
23. Wooldridge, M.: An Introduction to MultiAgent Systems - Second Edition. John Wiley & Sons (2009)