Exception handling in component-based systems : a first study

Frédéric Souchon* **, Christelle Urtado*, Sylvain Vauttier*, Christophe Dony **

* LGI2P - Ecole des Mines d'Alès - Parc scientifique G. Besse - 30 035 Nîmes - France - +33 4 66 38 70 00
{Frederic.Souchon, Christelle.Urtado, Sylvain.Vauttier}@site-eerie.ema.fr

** LIRMM - 161 rue Ada - 34 392 Montpellier - France - +33 4 67 41 85 33
dony@lirmm.fr

Abstract Reliability and fault-tolerance raise new issues in modern software architectures (such as component based architectures or multi-agent systems) as they aim at integrating separately developed software entities to build large-scale systems. In this paper, we study existing exception handling systems associated to the various component models that can be found in java-based platforms. This includes components having contract-based or event-based interaction schemes and synchronous or asynchronous communication schemes. We then focus on what we consider to be the most problematic issue: exception handling for asynchronously communicating components. We then present four qualities we think an EHS for a CBP should have and that we will consider as requirements for future work on designing an EHS that fulfill them.

1 Introduction

Modern ways of building applications require new means of software engineering: it is now easier to reuse existing pieces of software and integrate them to build complete applications. Component-based platforms (CBPs) [24,11,15] or multi-agents systems (MAS) [4] allow to build such applications with separately developed software entities. We believe that this reuse and integration process raises new reliability issues and that reliability becomes a more and more critical as systems get larger. In our previous work, we have studied exception handling in MASs and have proposed an exception handling system (EHS) dedicated to MASs [19]. In this paper, we address exception handling in CBPs and focus on what we consider to be the most problematic issue: exception handling for contract-based, asynchronously communicating components.

The remainder of this paper is organized as follows. Section 2 presents our study of exception handling in various existing CBPs. It introduces three categories of components and examines their particularities regarding exception handling. Section 3 focuses on a particular category of components (contract-based asynchronously communicating components) for which exception handling is particularly difficult and presents what we think to be the requirements that an EHS dedicated to such components should fulfill. Section 4 concludes and presents work in progress: how we are transposing and adapting our previous work on exception handling in MASs to provide a fully functional EHS for J2EE message-driven beans.

2 Exception handling in component-based platforms

Our study of exception handling is based on three kind of components that can be found in Java-based platforms (session or entity beans, message-driven beans and JavaBeans). They are well known, globally representative of the various kind of components available in existing operational platforms, and their open implementation makes it possible to freely experiment.

2.1 Component categories

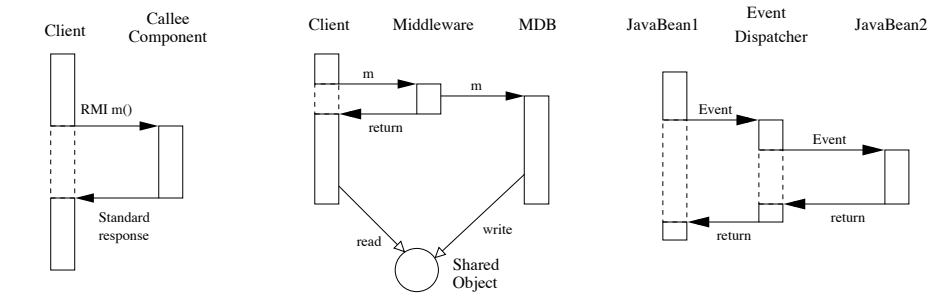
Components can be distinguished by (among other criteria non meaningful here) the way they interact and communicate.

Interaction schemes: main interaction modes are "Contract-based" and "Event-based".

- With Contract-based interactions (cf. Fig. 1a,b), components send requests to other components (acting as service providers) and expect answers in return as specified by component software contracts [13,12]. In this paper, we consider that requesting a **service** from a component triggers the execution of the corresponding **activity** in the concerned component.
- With Event-based interactions, components are more loosely coupled and interact via an event dispatcher that communicate events emitted by a component to those registered as its listeners. (cf. Fig. 1c). In this context, the emitting component does not know which components listen to its emitted events and does not expect any answer.

Communication schemes: components may communicate synchronously or asynchronously.

- With synchronous communications, service callers are blocked until callees reply. Only a single execution flow is executed at a given time: there is no concurrency (Fig. 1a,c).
- With asynchronous communications, request emission is non-blocking. Thus each service request triggers this creation of a new thread to run the corresponding activity (cf. Fig. 1b). This communication means is very flexible and suits particularly well applications in which QoS policies have to be implemented (eg. «provide a result once a certain QoS [6,26] level has been reached» or «providing the best result obtained before a specified delay»).



(a) Contract–based interactions using synchronous communications

(b) Contract–based interactions using asynchronous communications

(c) Event–based interactions using synchronous communications

Figure 1. Java components interaction and communication Schemes

Regarding this classification, we found three kinds of existing components:

- **CS components**: contract-based interactions with synchronous communications,
- **CA components**: contract-based interactions with asynchronous communications,
- **E components**: event-based interactions with synchronous communications.

2.2 Examples of Java components

Java Beans [23] are E components (cf. Fig. 1c) and are conceptually similar to Microsoft Active/X.

Other examples come with J2EE:

Session and *entity beans* are CS components which communicate via middlewares (RMI or CORBA). The middleware ensures packaging, transport and unpackaging of both method calls and results exchanged between remote components.

Finally, *message-driven beans (MDBs)* are CA components which communicate via messaging protocol called JMS (*Java Messaging Service*)[21,22]. A MDB which receives a JMS message executes the corresponding activity in a dedicated thread: this allows MDBs to concurrently process various activities. When a callee needs to return something to its caller, if the caller is not a component that can receive JMS messages, they have to use a shared object (cf. Fig. 1b) as a temporary repository. Otherwise, the callee can send a new JMS message to give the answer, thus implying the creation of a new thread.

JMS also allows to broadcast messages to a set of components thanks to the notion of *topic* (to be compared to *object groups* [14] which allow to gather a group of objects which can be seen and addressed as a unique entity). In JMS, a message sent to a topic is broadcasted to all MDBs which subscribed to the given topic but the sender is aware of neither the identity nor the number of receivers.

	Contract-based interactions	Event-based execution interactions
Synchronous communications	J2EE (RMI, Corba)	JavaBeans, ActiveX
Asynchrnous communications	J2EE (JMS)	none identified

Figure 2. Categories of components

2.3 Exception handling

This subsection describes how exceptions can today be signaled and handled for the three component models. Let us briefly recall that **exception signaling** [5], is a mechanism used to notify undesirable situations that hamper the standard execution of a program to continue. When an exception occurs, reliable software is able to react appropriately in order to continue its execution or, at least, to interrupt it properly while preserving data integrity as much as possible. An **exception handling system (EHS)** [3,5,10,13,27] offers control structures enabling developers to define program units (e.g. the source code of a procedure[10], a class definition [3]...) as protected by a set of **exception handlers** in order to capture exceptional situations that may be signaled in these protected regions. Each handler is defined to capture occurrences of a given exception type. The signal of an exception provokes the interruption of what is currently being executed and the search for an adequate handler. Handler search mechanisms are mostly based on Goodenough proposal [5] (eg. in C++ and Java) in which exceptions are propagated through the execution history. From a component point of view, such a mechanism would propagate exceptions signaled by component activities to their corresponding clients which would therefore be able to handle exceptions in the context of the failed service calls.

Exception handling for CS components (eg. session beans): their specification [20] and our experiments on the JOnAS platform [1] show that exception handling is achieved through standard Java control structures (*try/catch, throw...*). The middleware (RMI or CORBA) enables to transparently use server components as Java objects: the middleware abstracts distribution (cf. section 2.2) for standard execution as well as for exception handling as components use a proxy representation of remote components. This EHS seems to satisfy developers needs as it adopts the behavior of standard C++/Java languages EHSs which propagate exceptions through execution history. Thus, context-sensitive exception handling is possible as shown in figures 3 and 4a.

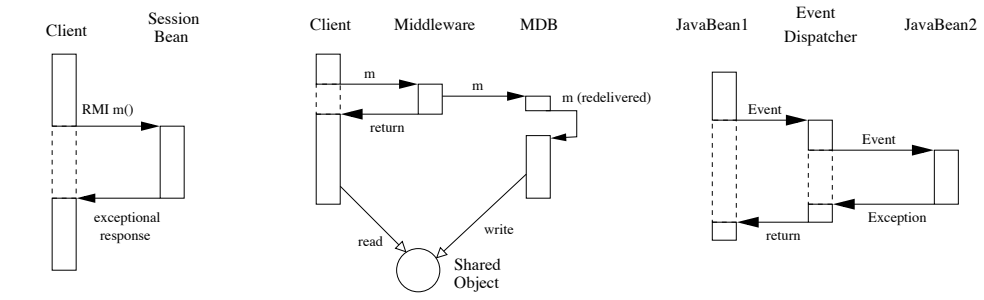
```
...
// A component invoking the buy method
through RMI
try {
  utx.begin(); // Starts a first transaction
  t1.buy(10); //request on the bean
  utx.commit();  //Commits the transaction
  }
catch (LimitedStockException exc) {
  int n =  exc.getMessage();
  println("Buying only" + n + "units");
  t1.buy(n);
  }
...
```

```
...
// Remote Business method implementation.
public void buy(int s) {
  if (stock>=s) {
    newtotal = newtotal + s;
    return;
    }
  else
    throw(new LimitedStockException(stock);)
}
...
```

Figure 3. Example of method invocation with RMI

Exception handling for E components (eg. Javabeans): If an event emitted by a component (*Javabean1*) leads to the signal of an exception in another component that cannot catch it, the exception is signaled to the event dispatcher (cf. Figure 4c). This event dispatcher can only generically handle the exception without propagating it to *Javabean1* (it just prints a stack trace) [23]. Thus, the exception cannot be propagated to the emitter's context. Such isolation of components concerning exception handling is coherent with the interaction scheme of these components: when notifying an event E components expect no response, either normal or exceptional (cf. subsection 2.1).

Exception handling for CA components (eg. MDBs): According to JMS specification [20] and to our experiments on the JOnAS platform, exception handling support in MDBs is limited: an error during a JMS message send can be notified to the sender but an exception signaled during the execution of the corresponding activity cannot be propagated to its caller. Indeed, an



(a) Contract–based interactions using synchronous communications

(b) Contract–based interactions using asynchronous communications

(c) Event–based interactions using synchronous communications

Figure 4. Communications in presence of exceptions

exception signaled in one the activity of a MDB without being handled locally is not propagated to the calling context. The only mechanism provided to allow the programmer to detect these situations is that the message that initiated the activity is redelivered (with a flag set to true) to the MDB in which it is executed (cf. Fig. 4b). Thus, to differentiate received messages from exception notifications, the developer has to test (for each message delivery) if the message is redelivered or not (cf. Figure 5). From our point of view, the handling of exceptions in the CA components we studied is too poor because:

1. Exceptions are not propagated through the execution history. It is therefore not possible to define context-sensitive treatments in handlers. Handler can only be generic and be used, for example, to display error messages (cf. Fig. 5). This issue will be developed in section 3.1.
2. There is no means to coordinate concurrent activities of components, and, thus, no exception coordination. For example, when an activity which requested services signals an exception, it should be possible to terminate the pending services it requested. This issue will be developed in section 3.2.
3. There is no means to concert exceptions that occur concurrently. This lack makes it impossible to correctly manage situations where multiple exception signals reveal a unique problem. It also hampers the programmer's capability to provide QoS policies (for example, he/she cannot distinguish under critical from critical exceptions). This issue will be developed in section 3.3.

```
public void onMessage(Message message) {

// Exceptional execution
  try {
  // The programmer tests if the message has been redelivered
  if (message.getJMSRedelivered()) {
    // The programmer handles the exception without information
    // about the execution context in which the exception has been signaled
    System.out.println("Error while handling" + message);
    return;
    }
  }
  catch(Exception ex) { System.err.println(ex.toString());}

//Standard execution
  try { ... }
  catch(HandledExceptionType ex) { System.err.println(ex.toString());}
}
```

Figure 5. Example of exception handling in a MDB

	Contract-based interactions	Event-based execution interactions
Synchronous communications	Typical EHS : satisfying	Generic EHS: isolated components
Asynchrnous communications	dedicated EHS : unsatisfying	none identified

Figure 6. The 3 components categories

3 Requirements for exception handling in CA components

As shown in the previous section (cf. Fig 6) CA components (such as MDBs) are the most problematic when exception handling is concerned because of asynchronous calls. This section aims at presenting the requirements concerning what we think should happen (and which extra concepts will have to be correctly managed) in situations in which an asynchronously invoked service cannot be fulfilled because of the occurrence of an exception.

3.1 Need for contextualization respect

When a called activity signals an exception it cannot handle, **its calling activity is the best place where to handle this exception properly**: it is the only place where the objective targeted by the corresponding service call is known and thus the best place to program what should be done in case of a defect. We thus believe that the software contract is to be respected in the case of exceptional responses as it is for normal ones in order to enable context-sensitive treatments of exceptions. Unfortunately, this behavior is not always adopted for CA components (cf. section 2.3) even if CBPs provide dedicated EHSs.

Alternative propositions[25,9] exist that do not address this problematic because of their centralized aspects. For example, [2] proposes a transposition of the *supervisor* model which its authors previously proposed in the framework of MASs [8,9]: it suggests the use of an EHS using *sentinel components* to detect and handle components failures. Propagating exceptions to such dedicated entities does not provide the capability to the write context-sensitive handlers and, thus, limits treatments in handlers to generic context-independent reactions (eg. error message display). Moreover, the centralization of the handling of exceptional events may affect QoS because it may cause bottlenecks which can slow down the whole system and decrease its reliability [19].

3.2 Need for coordination tools

Efficiently handling exceptions in systems that use asynchronous communications implies the definition of means to manage activities cooperation. [18,16] provides a classification of three types of concurrency and studies their impact on exception handling:

- Disjoint concurrency appears in systems that provide no coordination of concurrent activities. For example, there is no system level means to coordinate MDB activities when exceptions occur (no global activity is considered for such components).
- Competitive concurrency appears in systems that provide mechanisms to avoid inconsistencies caused by concurrent uses of system resources (generally thanks to lock-based mechanisms). Such mechanisms exist and can optionally be used with MDBs if *JTA (Java Transaction API)* or *JTS (Java Transaction Service)* is used to delimit distributed transactions.
- Cooperative concurrency is the kind of concurrency used by systems that provide some support to manage collaborations between active entities. For example, when an activity terminates, either normally or exceptionally, it must be possible to kill all non-terminated pending activities it initiated. [17] claims that such a cooperative concurrency management requires an execution model that enables collective activities to be explicitly represented. From our point of view, such a representation would make it possible **to associate handlers to the representants of such collective activities.** It would therefore be possible to globally manage the impact of the failure of either a single participant or a set of them.

To summarize, CBPs should provide support for cooperative concurrency in order to enable the coordination of components activities.

3.3 Need for exception concertation support

Once activity coordination is supported and integrates an exception handling mechanism, a means to collect exceptions occurring concurrently among participants of a collective activity must be provided. An activity which concurrently sends requests to a set of components and which receives one or more exceptional responses from them:

- must not react immediately when under-critical exceptions (which do not hamper its standard execution) are signaled,
- must be able to take into account that exceptions received concurrently may have no pertinence individually but may be meaningful collectively (result from an unique problem or a poor QoS level).

Based on these ideas, [7] suggests the use of a concertation mechanism: when entities which participate to a collective activity concurrently signal exceptions to their caller, a *resolution function* considers the set of signaled exceptions in order to evaluate a unique exception (a *concerted exception*) which reflects the global state of the collective activity. This ***concerted exception* is used in place of the whole set of individual exceptions signaled by requested services** to search for handlers in the collective activity.

3.4 Need for a specific exception handling policy for broadcasted requests

In section 3.2, we noticed the need to coordinate collective concurrent activities to properly handle exceptions. In JMS (cf. section 2.2), such a quality can partially be reached by activity coordination and must be completed by a proper handling of broadcasted requests. It must therefore be possible, for the programmer, to associate handlers to topics. As topics are not components (they are implicit notions hidden in the J2EE container), we propose to embody them in a dedicated component to which programmers can associate handlers and where they can configure concertation. This component is used to broadcast received requests to its registered components (just like topics did) and has the additional capability **to aggregate standard and exceptional responses in an unique meaningful response** transmitted to the service caller.

3.5 Synthesis

Figure 7 synthesizes what are the capabilities and the needs, in terms of exception handling, of the three categories of components we studied in this paper. It focuses on the four qualities presented in sections 3.1, 3.2, 3.3 and 3.4. It shows that exception handling in CS and E components is adequate as the EHS capabilities match the qualities needed. On the contrary, the analysis of exception handling in CA components shows that work must be done to match EHS capabilities and programmers expectations.

	CS components		CA components		E components	
Quality	**Exist ?**	**Needed ?**	**Exist ?**	**Needed ?**	**Exist ?**	**Needed ?**
Contextualisation respect	yes	yes	no	yes	no	no
Coordination support	yes	yes	partial: transactions	yes	no	no
Exceptions concertation support	no	no	no	yes	no	no
Support for collective requests	no	no	partial: forwarding	yes	partial: event dispatcher	no

Figure 7. Potential improvements

4 Conclusion and further work

In this paper, we studied available categories of components and their EHSs. On the one hand, we believe that two out of three categories of components have adequate EHSs. On the other hand, we highlighted lacks in EHSs provided with the third components category (components which interact using a contract-based scheme and asynchronous communications).

In particular, we think that an EHS for such components must have these four qualities:

- Contextualization must be respected in order to enable the writing of context-sensitive handlers because a service demander is the best entity to handle an exception that occurred in a requested service.

- Coordination of components activities must be achieved by enabling their explicit representation and by defining means to control their execution.
- Programmers must be able to configure the exception propagation policy by defining *resolution functions* to immediately handle exceptions that are really critical for the execution while logging under-critical exceptions until their conjunction enables to diagnose a unique problem represented by a *concerted exception*.
- The EHS must manage broadcasted requests in a pertinent way.

In previous work, we designed and implemented an EHS providing these qualities for MASs (the SaGE EHS[19]). The study presented in this paper is our first step towards the adaptation of the design and implementation of this work in the context of CBPs which we plan to integrate to the JOnAS platform.

References

1. BullSoft. *Jonas*. http://www.objectweb.org/jonas/current/doc/JOnASWP.html.
2. Chrysantos Dellarocas. Toward exception handling infrastructures for component-based software. In *Proceedings of the International Workshop on Component-based Software Engineering, 20th International Conference on Software Engineering (ICSE), Kyoto, Japan, April 25-26, 1998*, 1998.
3. Christophe Dony. Exception handling and object-oriented programming : towards a synthesis. *ACM SIGPLAN Notices*, 25(10):322–330, October 1990. *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).
4. Jacques Ferber. *Les systemes multi-agents, vers une intelligence artificielle distribuée*. InterEditions, 1995.
5. John B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, December 1975.
6. Jun He, Matti A. Hiltunen, Mohan Rajagopalan, and Richard D. Schlichting. Providing qos customization in distributed object systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001), Lecture Notes in Computer Science 2218*, 2001.
7. Valerie Issarny. Concurrent exception handling. In A. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, editors, *Advances in Exception Handling Techniques*, LNCS (Lecture Notes in Computer Science) 2022. Springer-Verlag, 2001.
8. Mark Klein and Chrysanthos Dellarocas. Exception handling in agent systems. In Oren Etzioni, Jörg P. Müller, and Jeffrey M. Bradshaw, editors, *Proceedings of the Third Annual Conference on Autonomous Agents (AGENTS-99)*, pages 62–68, New York, May 1–5 1999. ACM Press.
9. Mark Klein and Chrysanthos Dellarocas. Using domain-independent exception handling services to enable robust open multi-agent systems: The case of agent death. *Journal for Autonomous Agents and Multi-Agent Systems*, 7(1/2), 2003.
10. A. R. Koenig and B. Stroustrup. Exception handling for C++. In *Proceedings "C++ at Work" Conference*, pages 322–330, November 1989.
11. David H. Lorenz and Predrag Petkovic. Design-time assembly of runtime containment components. In Institute of Electrical and Electronics Engineers, editors, *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*, 2000.
12. B. Meyer. Applying 'design by contract'. *IEEE Computer*, October 1992.
13. Bertrand Meyer. Disciplined exceptions. Technical report tr-ei-22/ex, Interactive Software Engineering, Goleta, CA, 1988.
14. M. Patino-Martinez, R. Jimenez-Peris, and S. Arevalo. Exception Handling in Transactional Object Groups. In *Advances in Exception Handling, LNCS-2022*, pages 165–180. Springer, 2001.
15. F. Peschanski. Comet: A reflective middleware architecture for adaptive component-based distributed systems, http://ads.computer.org/dsonline/0107/features/pes0107.htm, 2001.
16. A. Romanovsky, C. Dony, J. L. Knudsen, and A.Tripathi. *Advances in Exception Handling Techniques*. LNCS (Lecture Notes in Computer Science) 2022. Springer-Verlag, 2001.
17. Alexander Romanovsky. Exception handling in component-based system development. In *Proceedings of 25th Annual International Computer Software and Applications Conference (COMPSAC'01)*, 2001.
18. Alexander B. Romanovsky and Jorg Kienzle. Action-oriented exception handling in cooperative and competitive concurrent object-oriented systems. In *Advances in Exception Handling Techniques*, pages 147–164, 2000.
19. F. Souchon, C. Dony, C. Urtado, and S. Vauttier. A proposition for exception handling in multi-agent systems. In *SELMAS'03 proceedings*, 2003.

20. Sun Microsystems, Mountain View, Calif. *Enterprise JavaBeans (EJB).* http://java.sun.com/products/ejb/.
21. Sun Microsystems, Mountain View, Calif. *Java 2 Platform, Enterprise Edition (J2EE).* http://java.sun.com/j2ee.
22. Sun Microsystems, Mountain View, Calif. *Java 2 Platform, Standard Edition (J2SE).* http://java.sun.com/j2Se.
23. Sun Microsystems, Mountain View, Calif. *JavaSoft JavaBeans API Specification*, 1.01 edition, July 1997. http://javasoft.com/beans/docs/spec.htm.
24. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley, New York, NY, 1998.
25. Anand Tripathi and Robert Miller. Exception handling in agent oriented systems. In A. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, editors, *Advances in Exception Handling Techniques*, LNCS (Lecture Notes in Computer Science) 2022. Springer-Verlag, 2000.
26. N. Wang, D. Schmidt, K. Parameswaran, and M. Kircher. Towards a reflective middleware framework for qos-enabled corba component model applications, ieee distributed systems online special issue on reflective middleware, 2001.
27. Daniel L. Weinreb. Signalling and handling conditions. Technical report, Symbolics, Inc., Cambridge, MA, January 1983.