Non-Functional Exceptions for Distributed and Mobile Objects

Denis Caromel and Alexandre Genoud

INRIA Sophia Antipolis, CNRS - I3S - UNSA
BP 93, 06902 Sophia Antipolis Cedex - France
First.Last@inria.fr

Abstract. While there is quite a lot of techniques to separate non functional properties from functional code, the handling of induced exceptions remains often blurred within application. This paper identifies *Non-Functional Exceptions* as exceptions related to various failures of non-functional properties (distribution, transaction or security). We propose a hierarchical framework where reified exception handlers are attached to various entities (proxies, remote objects, futures). Such handlers allow middleware and application oriented handling strategies for distributed and mobile computation. The mechanism tries to handle exceptions at non-functional level as much as possible.

1 Introduction

Distributed environments provide synchronous and asynchronous calls, remote references, migration of activities. Complex communications are subject to various failures such as the remote communication failure. It is always unclear whether the failure occurred in the communications medium or in the remote process, and the state of the system is in general uncertain. Unfortunately, the `try/catch` construction is heavy to use, and only convenient for simple communication errors.

In this article, we define *non-functional exceptions* as exceptions related to distribution. We present a hierarchical model based upon *handlers of exception*. Sets of handlers are dynamically attached to various entities (JVMs, remote and mobile objects, proxies, ...) in order to provide a generic and flexible recovery mechanism at a non functional level. This model has been implemented and bench marked in a framework for parallel, distributed and mobile computing known as ProActive[1]. As implementation remains simple, the port to other middlewares is possible.

The first section presents previous works related to exception handling in distributed architectures. Then, non-functional exceptions are defined and those related to distribution are classified. The next chapter describe a flexible model used to handle simple communication failures but also to create advanced fault-tolerance strategies. Finally, pragmatic examples are presented. Performances are discussed in the appendix.

[1] http://www.inria.fr/oasis/ProActive

2 Related Work

Exceptions have been created in ADA in the 1970s and are now a standard mechanism to report errors and failures. In distributed environments, they are raised from host to host and thus are difficult to handle. Through the development of our distributed library, we realized that standard handling mechanisms are not appropriate to distribution, as developers must define handling code for every distributed exception.

Authors of [4] highlight a critical problem that appears when several failures occur simultaneously. While communications between distant processes are broken, an unstable state is probably reached. This article suggests to gather communicating processes into a *conversation* before starting any kind of communication. Participants first save their own state ; then a set of handlers is associated to the conversation. All action participants are involved in co-operative handling of any exception raised by any action participant : the conversation is paused until the handling process is terminated. When handlers are not sufficient to recover from failure, the conversation is canceled. Every process checks possible side effects and rollbacks to its initial state. This collaborative strategy seems really promising but fails with asynchronism. As the return time of an asynchronous call is unknown, the lifespan of the conversation is also unknown. The recovery process could be maintained as long as no result is delivered.

Agents are active objects having autonomous behavior according to their environment. As mobility is one possible behavior, an agent can decide to migrate on a different virtual machine. In this context, authors define *guardians* in [5] as centralized mechanisms helping agents to handle exceptions related to distribution. Only one guardian is needed for every agents-based application. When an agent cannot handle an error, the exception is raised to the guardian which send back instructions. Of course, the handling behavior depends not only of the nature of the exception but also of the agent environment. When distant objects become unreachable, the guardian can advise to delay communication. When critical failures occur, the guardian can terminate agents. An interesting strategy to handle failures related to the migration of agents could be to find an equivalent destination using the replication strategy. This centralized model offers simplicity as it provides only one single guardian even for large distributed systems. However, many problems would occur if the guardian becomes unreachable or crashes.

3 Non-Functional Exceptions

During the conception process, we identified three majors features required for distributed handling mechanisms : flexibility, genericity and dynamicity. Considering that previous models did not meet all of these requirements, we decided to create an original model from scratch based upon a new classification of exceptions.

3.1 Functional versus Non-Functional

In recent literature, classifications of exceptions are proposed. According to [7], exceptions can be divided into *internal exceptions*, raised from and handled within a method, and *external exceptions* propagated toward other methods. This classification is not useful for distributed environments which require complete description of internal failures. In our framework, we consider the mechanism of distribution as a *non-functional property* [8]. We use this specificity to define non-functional exceptions as exceptions raised from any non-functional property.

Definition 1 *Non-functional Exceptions announce failures occurring in non functional properties. They are raised in non-functional code and handled, as much as possible, within it.*

We make a clear difference between functional exceptions, related to abnormal behavior of applications, and non-functional exceptions, related to failures of non-functional properties. Exceptions related to distribution should be considered as non-functional exceptions coming from the middleware. We agree with the recommendation of [9] which claims that exceptions have to be handled at meta level. It is much more simple indeed to handle exceptions directly in internal mechanisms of distribution.

3.2 Location of Non-Functional Exceptions

Distributed environments provide synchronous and asynchronous communications as describe in [1]. Failures in such communications result in non-functional exceptions as shown in 1. While in synchronous calls, those exceptions are simply handled at results delivery, asynchronous calls lead to two solutions. Exceptions are eventually handled when requests containing reified calls are synchronously queued. But non-functional exceptions have to be handled in future objects when pending requests are served or when results are stored within them.

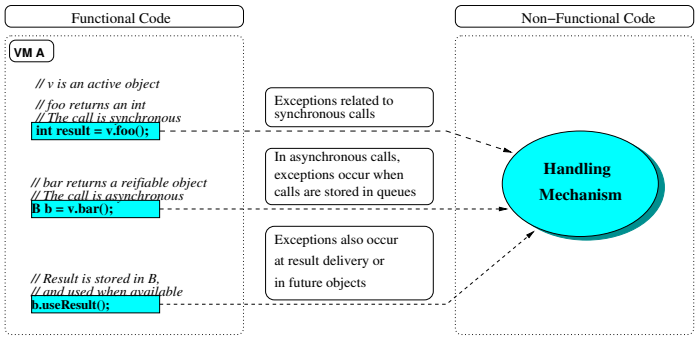

Fig. 1. Exceptions Raised from Synchronous and Asynchronous Calls

3.3 A Hierarchy of Distributed Exceptions

We first identified and classified potential failures (figure 2) of distributed environments. Then, we built a *hierarchy of potential failures*, opened to developers who can add new failures and topics. We kept this structure customizable as flexibility is the most important feature of recovery mechanism. Finally, we associated non-functional exception to every failure. This hierarchy is used to define handling strategies for specific exceptions as well as for groups of exceptions.

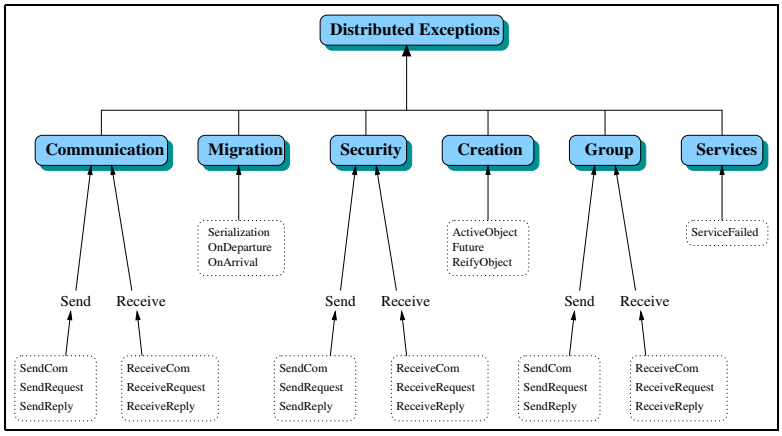

Fig. 2. Hierarchy of Failures Encountered in Distributed Environments

4 A Hierarchical and Dynamic Handling Mechanism

The hierarchy of failures described above is used in the construction of hierarchical handlers, working indifferently at functional or non-functional levels.

Definition 2 *Handler of exceptions handle non-functional exceptions as well as groups of such exceptions, thanks to object inheritance.*

For instance, a handler can be associated to `SendRequestGroupException` or to every member of `GroupException` (see [10] for detail about group communications). Handlers provide basic strategies in non-functional code, but application-specific strategies are also possible. They reify the `try/catch` construction to support both genericity and flexibility required by any handling mechanism. Handlers implement a common interface and provide functional as well as non-functional treatments of non-functional exceptions.

4.1 Prioritized Levels of Handling

Our mechanism is based upon a default and static level, created during the middleware initialization, and some dynamic levels set during execution. Each

structure can provide a specific fault tolerance strategy created from an appropriate set of handlers. Every non-functional exception is associated to one handler in the default level. The default strategy is basic but always present while complexes strategies appear occasionally in higher levels. We defined six different levels, associated to constants within the implementation and presented below from lower to higher priority.

1. *Default level* is static and initialized in core of applications. This level provide a basic handling strategy for every non-functional exception.
2. *Virtual Machine level* is the first level that can be created dynamically. It offers the possibility to define a general handling behavior for every VM.
3. *Remote and Mobile Object level* is used to bind handlers to remote objects. Handlers associated to mobile entities migrate along with them.
4. *Proxy level* is used to define strategies for references to active objects. When reference are passed to other VMs, handlers are passed also.
5. *Future level* is attached to the results of asynchronous calls.
6. *Code level* allows temporary handlers to be set in the code.

As describe above, the default level provides a basic handling strategy, defined during the initialization of middleware. Virtual machine level and higher ones are set dynamically to improve this strategy. Dynamic handlers are created at runtime and added to an appropriate level (VM, remote object, proxy, future or code levels).

4.2 Presentation of the API

The API is both used for middleware adaptation (e.g. wireless oriented) and for distributed application. It consists in two major static functions which offer settings and configurations of handlers into appropriate levels. The five dynamic levels are defined with constants.

```
    // Binds one handler to a class of exception at the specified level.
void setExceptionHandler(level, Handler, Exception, Target);

    // Removes handler associated to a class of exception at specified
    // level. Target is different from null when level is object-related.
Handler unsetExceptionHandler(Level, Exception, Target);
```

The following example show how to protect an application from communication failures. We add a handler with the *setExceptionHandler* primitive. Communication failures are thus handled for that object.

```
    // Creation of a remote and mobile object with handlers
RO ro = (RO) ProActive.newActive("RO", "//io.inria.fr/VM1");

    // A communication handler is dynamically associated
    // to the remote object trough its proxy.
setExceptionHandler(ProxyLevel,
                    "CommunicationHandler",
                    "CommunicationException",
                    ro);
```

4.3 Dealing with Mobility

Most of the distributed environments offer remote and mobile objects. Such objects can migrate from host to host. This additional constraint can break the continuation of the handling mechanism. The migration process must be modified to take into account the migration of mobile object handlers. As explained later, mobile objects and their associated levels remain always gathered. Handling mechanism can be associated to proxy also in order to attach a specific strategy to remote references.

4.4 Implementation

As explained before, the handling strategy is built upon one static level improved occasionally with dynamic levels. Handlers are searched with the following dedicated function.

```
    // Searches through prioritized levels the handler associated
    // to the given class of exception
Handler searchExceptionHandler(Exception, Target);
```

The following code is part of the middleware and describe how to activate the handling mechanism. Instead of providing a treatment directly in the *try/catch* block, we use the *searchExceptionHandler* primitive.

```
try {
        // Send the reified method call
    sendRequest(methodCall, null);

} catch (NonFunctionalException e) {

        // Looks for an appropriate handler and
        // use the handler if possible
    Handler handler = searchExceptionHandler(e);
    if (handler) handler.handle(e);
}
```

We tried to keep implementation as simple as possible but performance issues were also considered. Levels are implemented with hashmap to provide fast access to handlers. Considering the memory available in modern computer, we support time complexity instead of space complexity even if migration increase memory requirements because of levels associated to mobile objects. The cost is proportional to the number of handlers contained in the object level.

Reflexion is used to search handlers for a specific class of exception or for the mother class of a group of exceptions. The algorithm supports generic handlers of higher levels instead of specific handlers from lower level ; Levels have precedence over the type of exceptions. For instance, on figure 3, the most suitable handler for exceptions related to class 02 is found in the highest level. When no handler is available at remote object level, the search continue in VM an lower level. This choice, which seems more natural, can be invert.

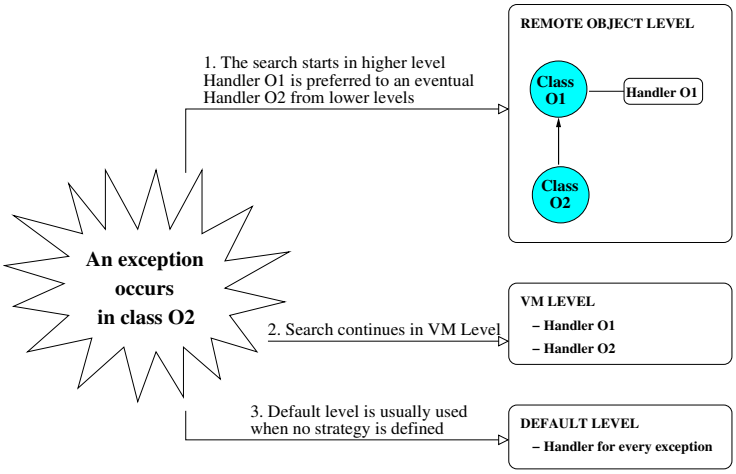

Fig. 3. Levels have Precedence over Exception Type when Searching Handlers

5 Canonical Examples

We present in this section two applications which use our handling mechanism.

5.1 Handling Exceptions in Unconnected Mode (e.g. wireless PDA)

Distributed applications for *Personal Digital Assistants* should provide an unconnected mode to handle at least communication exceptions due to broken connections. We defined a strategy where handlers store requests sent to unreachable PDAs in a queue. Time by time, a thread checks if the connection is restored in order to deliver requests. The point is not to define a sophisticated strategy, but to show how easily it can be activated. Here is the scheme of such a *PDA-Handler*.

```
Class PDACommunicationHandler implements Handler {
    public boolean isHandling(Exception e) {
        return (e instanceOf CommunicationException);
    }
    public void handle(Exception e) {

            // A thread testing connectivity is created
        if (firstUse) {
            connectivityThread = new ConnectivityThread();
        }
            // Then reified method calls are stored in the
            // queue and exceptions are not propagated anymore
        queue.store(e.getReifiedMethodCall());
    }
}
```

Imagine now that an entity is about to create a mobile object that migrate on some wireless PDA.

```
    // Creation of a remote and mobile object with handlers
RO ro = (RO) ProActive.newActive("RO", "//io.inria.fr/VM1");

    // A communication handler is dynamically associated
setExceptionHandler(ProxyLevel,
                    "PDACommunicationHandler",
                    "CommunicationException",
                    ro);

    // The mobile object can now migrate safely
ro.migrateTo("//pagode.inria.fr/VM2");
```

5.2 Simulating a Centralized Error Manager

The handling mechanism can easily be configured into a centralized error manager similar to the one presented in [5]. We create first a remote object containing a complete set of prioritized handlers. This object is located on one virtual machine but is known from every active object of the application. Non-functional exceptions reporting failure are not handled directly in the active object but are raised to the centralized error manager instead. A handler corresponding to the failure is sent back to handle the exception. This strategy does not avoid the typical problems common to every centralized error manager but offers at least an efficient centralized handling mechanism, easy to configure.

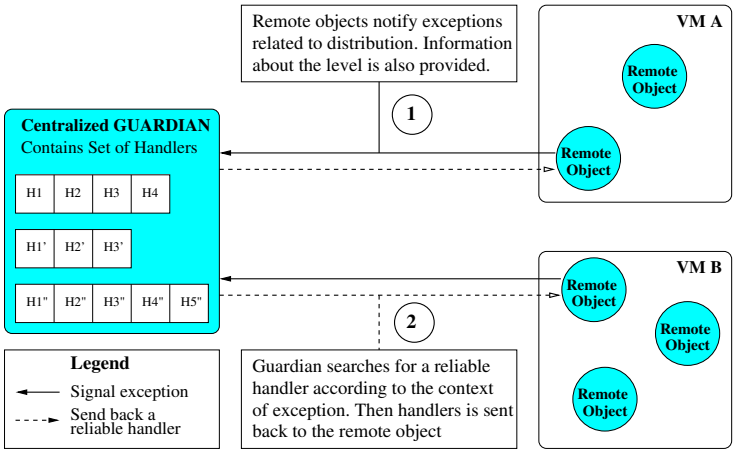

Fig. 4. Centralized Error Managers are Easy to Implement

6 Conclusion and Perspectives

We defined a dynamic, flexible and generic model to handle non-functional exceptions. We proposed a classification for non-functional exceptions along with a hierarchy of prioritized levels. As implementation use the classical `try/catch` language construct, the model is reliable for a large panel of modern, object-oriented, programming languages.

References

1. D. Caromel, W. Klauser and J. Vayssiere. *Toward Seamless Computing and Metacomputing in Java.* Concurrency Practice and Experience (September-November 1998) p. 1043-1061 Editor Geoffrey C. Fox, published by Wiley & Sons
2. E. F. Walker, R. Floyd, P. Neves *Asynchronous Remote Operation Execution in Distributed Systems.* In Proc. of the Tenth International Conference on Distributed Computing Systems, May/June 1990.
3. F. Baude, D. Caromel, F. Huet and J. Vayssiere, *Communicating Mobile Active Objects in Java* HPCN Europe 2000, Amsterdam - The Netherlands, May 2000
4. Jie Xu, Alexander B. Romanovsky and Brian Randell. *Coordinated Exception Handling in Distributed Object Oriented System (Revision and Correction).* Department of Computing Science, University of Newcastle upon Tyne, Newcastle upon Tyne, UK.
5. Arnand Tripathi and Robert Miller . *Exception Handling in Agent-Oriented Systems.* Advances in Exception Handling Techniques, Springer-Verlag LNCS 2022, March 2001.
6. Valerie Issarny. *Concurrent Exception Handling.* Advances in Exception Handling Techniques 2000: 111-127. Inria Rocquencourt.
7. Alessandro F. Garcia, Cecilia M. F. Rubira, Alexander Romanovsky and Jie Xu. *A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software.* Journal of Systems and Software, Elsevier, Vol. 59, Issue 2, November 2001, p. 197-222.
8. Kiczales, Lamping, Mendhekara, Maeda, Lopes, Loingtier, Irwin. *Aspect-Oriented Programming.* Proceedings of ECOOP 97, n 1241 LNCS, Springer-Verlag, June 1997, p. 220-242.
9. Ian S. Welch, Robert J. Stroud and Alexander Romanovsky. *Aspects of Exceptions at the Meta-Level (Position Paper).* Department of Computing, University of Newcastle upon Tyne.
10. Laurent Baduel, Françoise Baude, Denis Caromel. *Efficient, Flexible, and Typed Group Communications in Java.* Proceedings of the Joint ACM Java Grande - ISCOPE 2002 Conference. Nov. 2002.
11. Anh Nguyen-Tuong. *Integrating Fault-Tolerance Techniques in Grid Applications.* Partial Fullfillment of the Requirements for the Degree Doctor of Computer Science. University of Virginia.

Appendix: Time and Space Performances

Space Complexity : Each system contains at least default and virtual machine levels : some handlers contained by two hashtables.

Strategy	Description	Number of Handlers	Size in Byte
No Handler	No handler is provided. We just pay the cost of an empty level based upon Hashtable	0	82
Minimal	One global and generic handler achieve application soundness	1	209
Per Group	One handler is provided for each group of non-functional exception (see 2)	7	1561
Per Communication	Every communication exception has 2 handlers : remote object level and VM level	2 * 6 = 12	2833

Table 1. Space Requirements Depends of the Number of Handlers

Time Complexity : Adding and removing handlers do not break overall performance of the system. Research of handlers is complexity-less, thanks to hashtable properties. We raised a huge number of exceptions and measured time to find handlers. The ratio is 1:4.

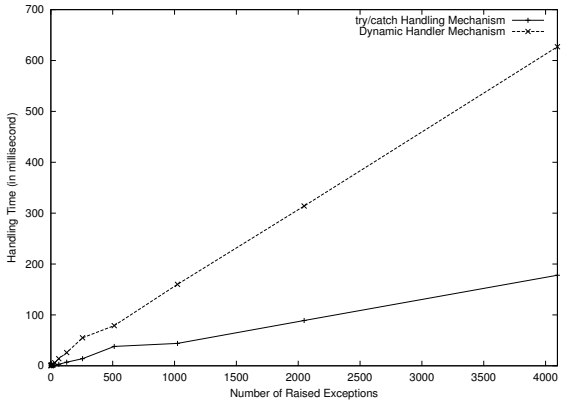

Fig. 5. Time Complexity