Non-Functional Exceptions for Distributed and Mobile Objects

Denis Caromel and Alexandre Genoud

INRIA Sophia Antipolis, CNRS - I3S - Univ. Nice Sophia Antipolis
BP 93, 06902 Sophia Antipolis Cedex - France
First.Last@inria.fr
http://www.inria.fr/oasis/ProActive

Abstract. Exception handling strategies are the key to safe applications. While there has been quite a lot of research developing models to remove non-functional properties from application code, the handling of exceptions remains often associated to functional code. Existing solutions are neither dynamic nor customizable and miss important features. Thus, we defined *Non-Functional Exceptions* : exceptions due to abnormal behavior of non-functional properties such as distribution, transaction or security. We proposed then a generic, dynamic, and flexible model based upon a hierarchy of non-functional exceptions. In the context of distributed environments, this solution lead to the handling of exceptions at non-functional level. ...

1 Introduction

Distributed environments require complex communications such as synchronous and asynchronous calls, remote references, migration of activities and even interactive tools to set up connectivity. Those communications need flexible mechanisms to handle exceptions related to distribution.

In the scope of Java for instance, RMI is an API for distributed programming. The limited `java.io.RemoteException` is provided and raised when an error occurs, whatever the error is. The lack of information is obvious as nobody knows exactly the nature of the exception despite additional encapsulated information. The standard use of `try/catch` is convenient with simple communications but not flexible enough with those presented above.

Unfortunately, existing solutions are not necessary well-adapted to large scale distributed systems. Some of them do not work with asynchronous and mobile environment while others, based upon centralized error manager, are not adapted to the potential failure of communications. Moreover, their lack of customization prevents the construction of complex fault tolerance strategies.

In this article, we first identify exceptions related to distribution as *non-functional exceptions*, different from classical *application level exceptions*. We present then a simple but powerful mechanism which provides hierarchical exception handling using some sets of *object handlers*. Handlers are exception managers attached to various entities (JVMs, remote and mobile objects, proxies,...)

providing flexible and dynamic configuration well-adapted to distributed applications.

The previous concept has been implemented and bench-marked in the framework of ProActive[1], a library for parallel, distributed and mobile computing. RMI, currently used as the transport layer, limits the control over exceptions as it does not provide a specific and efficient handling mechanism. For such distributed systems, flexibility as well as the possibility to create fault tolerance strategies would be a great improvement. Considering previous goals, the concepts presented in this paper could be used with most of the distributed environments.

The paper is organized as follow : we first present work related to exception handling in distributed architectures. Then, we define and classify non-functional exceptions, particularly those related to distribution. In the next part, we propose a model for hierarchical handling mechanisms. This model deal with non-functional exceptions and offer the possibility to create advanced handling strategies. Finally, we discuss performance issues and give practical examples. We end up with some perspectives around components.

2 Related Work and Background

Through the development of a distributed library, many difficulties are encountered among which raising of exceptions. The standard mechanism, based upon the `try/catch` control, is rather limited indeed, especially in distributed environment. Communication exceptions (i.e. remote call exception, migration exception...) as well as every exception related to distribution require specific handling code. Unfortunately, constraints are let to the developer who write handling code with every use of `try/catch`. This solution is not reliable because of the difficulty to provide an interesting and re-usable strategy this way. The well-known `printStackTrace(e)` pattern is not an interesting solution and should not be retained anymore. As existing models are not necessarily complete as we will see in the following paragraphs, RMI and other distributed libraries still miss a generic and flexible handling mechanism.

2.1 Collaborative Exception Handling

We focus now on the work of [2] which provides an interesting model as well as practical examples. In distributed environments, remote objects and other entities can simultaneously raise exceptions. The algorithm presented in this article uses the concept of *conversation* between processes. Distant processes save their current state and join a conversation when they have to communicate. A set of handlers associated with exceptions is built according to the communication. When an error occurs, the guilty process try to handle the exception alone while other processes wait passively : the conversation is paused. If the set of handlers

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

is not sufficient, the conversation is canceled. Every process check some possible side effect due to the raised exception before restoring original states.

This model looks really promising but imposes a collaborative strategy, and fails with an important feature required by today's distributed environments. Asynchronous calls are indeed the weakest point of the mechanism. With such calls, the return time of the result is unknown and thus the lifespan of the conversation is also unknown. The conversation is maintained as long as the result is not deliver but this specific case prevent the continuation of the program.

2.2 Centralized Error Manager

The work presented in [3] provide a model to handle exceptions in mobile agents systems. Agents are a kind of active objects having autonomous behavior according to their environment. As mobility is one possible behavior, agents decide whether or not they migrate on other virtual machines.

Authors define a *guardian* as a centralized mechanism helping agents to handle exceptions related to distribution. Only one guardian is needed for every agent-based application. When an agent cannot handle an exception by itself, it contacts the guardian which gives further instructions. Of course, instructions depend of the agent environment. When distant objects are not reachable, the guardian can advise to delay communication. When critical failures occur, the guardian can terminate agents. It can also propose various and complex handling solutions. An interesting strategy could be used when the migration of agents fails : the guardian try to find an equivalent destination.

Nevertheless, centralized models suffer some weak points. Centralization, which offers simplicity, is not scalable. The model presented here provides only one guardian even for large distributed systems. What would happen if the connection between the guardian and all agents is broken ? What would happen if the guardian crashes ? Moreover, the mechanism seems mostly static which means that he cannot be modified at runtime.

2.3 Distribution and Mobility with ProActive

As ideas and concepts introduced in this paper are implemented and benchmarked within the ProActive library [4], this section presents its main features and the global architecture.

ProActive is built on top of standard Java APIs[2], it does not require any modification to the standard Java execution environment, nor does it make use of a special compiler, pre-processor or modified virtual machine. The model of distribution and activity of ProActive is part of a larger effort to improve simplicity and reuse in the programming of distributed and mobile object systems.

A distributed application built using ProActive is composed of a number of medium-grained entities called *active objects*. An activity is composed of a single active object which is the only entry point to the activity and any number of

[2] Java RMI, the Reflection API

standard and private Java object. Each active object has its own thread of control and is granted the ability to decide in which order to serve the incoming method calls that are automatically stored in a queue of pending requests. Method calls (see figure 1) sent to active objects are always asynchronous with transparent *future objects* and synchronization is handled by a mechanism known as *wait-by-necessity*. At the beginning of each asynchronous remote call, the caller blocks until the call reaches the context of the callee (on figure 1, step 1 blocks until step 2 has completed). The ProActive library provides a way to migrate any active object from any JVM to any other one through the `migrateTo(...)` primitive which can either be called from the object itself or from another active object through a public method call.

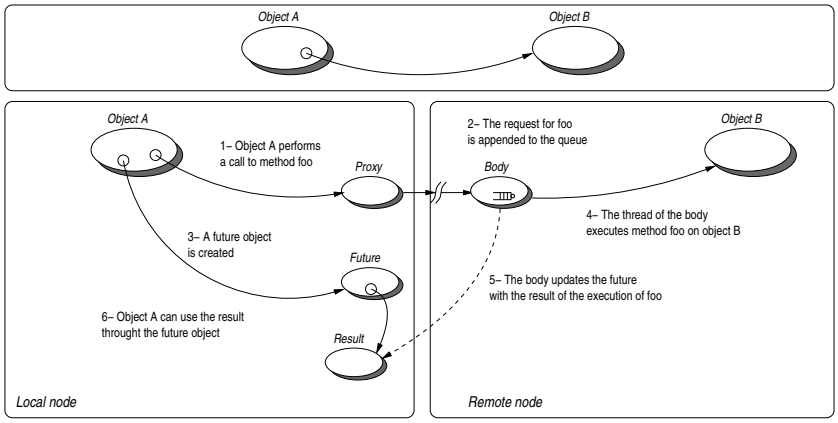

Fig. 1. Execution of Asynchronous Remote Method Call

3 Non-Functional Exceptions

Considering previous analysis, it is obvious that existing models cannot afford our initial objectives. They are efficient in specific cases but not flexible enough to cover the diversity of problems. They are neither customizable nor dynamic. Moreover, no difference is made between exceptions related to distribution and others. This is why we defined a new classification in order to achieve a suitable mechanism.

3.1 Functional versus Non-Functional

In recent literature, some classifications of exceptions are proposed. According to [5], exceptions can be divided into two categories. The *internal exceptions* are raised and handled from within a method while the *external exceptions* are

propagated toward other methods. In the specific context of distributed environments, this classification misses something really important as it does not represent the nature of exceptions but just gives an indication about where handling takes place.

The intrinsic nature of an exception is used to define *a non-functional exception as an exception raised from a non-functional property* [6]. We make a clear difference indeed between functional exceptions related to abnormal behavior of applications, and non-functional exceptions related to abnormal operations in non-functional properties. For instance, distribution is considered as a non-functional property when it is transparent at application level. According to the previous definition but also to [7] which claims that exceptions must be handled at Meta level, exceptions related to distribution should be considered as non-functional exceptions to achieve transparent handling. Transparent fault tolerance strategies, developed with such non-functional exceptions, will then improve soundness of applications and simplify the life of developers.

3.2 Classification of Non-Functional Exceptions for Distribution

We identified potential failures (see figure 2) which we used to gather exceptions. We built then a logical and opened *hierarchy* where developers add their own exceptions. However, this structure remains customizable as flexibility should be the most important property of every handling model. Thus, we can create new concerns if needed. This hierarchy is used to handle specific exceptions as well as groups of exceptions.

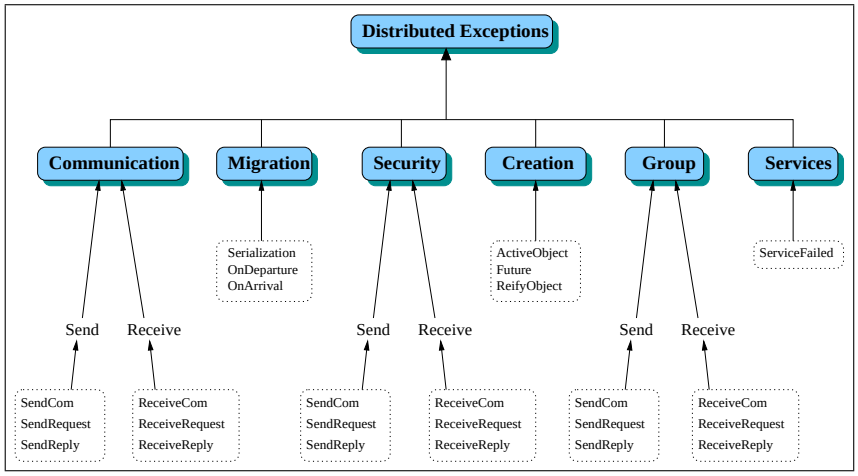

Fig. 2. Hierarchy of Non-Functional Exceptions for Distribution

3.3 Exceptions Raised from Synchronous and Asynchronous Calls

Literature provides different semantics of communication according to the needs of applications.

- *Synchronous Method Call* : Callers send reified method calls and wait until some results are returned.
- *One-way Method Call* : This asynchronous communication is used when no result is returned.
- *Asynchronous Method Call* : Callers do not wait results which are stored in future objects when available.

When remote calls fail, awaited results (which could be functional exceptions) are replaced with non-functional exceptions. The handling is obviously different according to the communication. With synchronous calls, non-functional exceptions are handled at results delivery. On the other hand, asynchronous calls lead to two different solutions. They are first synchronously placed in queues of pending requests. Exceptions are eventually handled next to this operation. Then, awaited results are stored in future objects when available. Most of the exceptions are handled in such future objects. The figure 3 summarize handling location.

3.4 Dealing with Mobility

As many distributed environments offer mobile objects, we have to take into account this particular constraints. We must provide an automatic and dynamic way to adapt handling mechanism to mobility. As explained later, handling mechanism is part of any mobile objects in order to carry on their strategy, even after a migration. Nevertheless, the mechanism can also be associated to proxy to have a specific strategy attached to remote references.

3.5 Non-Functional Exceptions at Functional Level

Sometimes, non-functional exceptions have to be handled at functional level. Online banking applications give interesting examples. Clients connect to centralized server with any browser and make different operations on their account. If the server is not reachable anymore, money transfers could be transparently delayed while statements concerning the account cannot be obtained. In this special case, the only solution consists to inform the client that the request failed. The treatment is thus at functional application level.

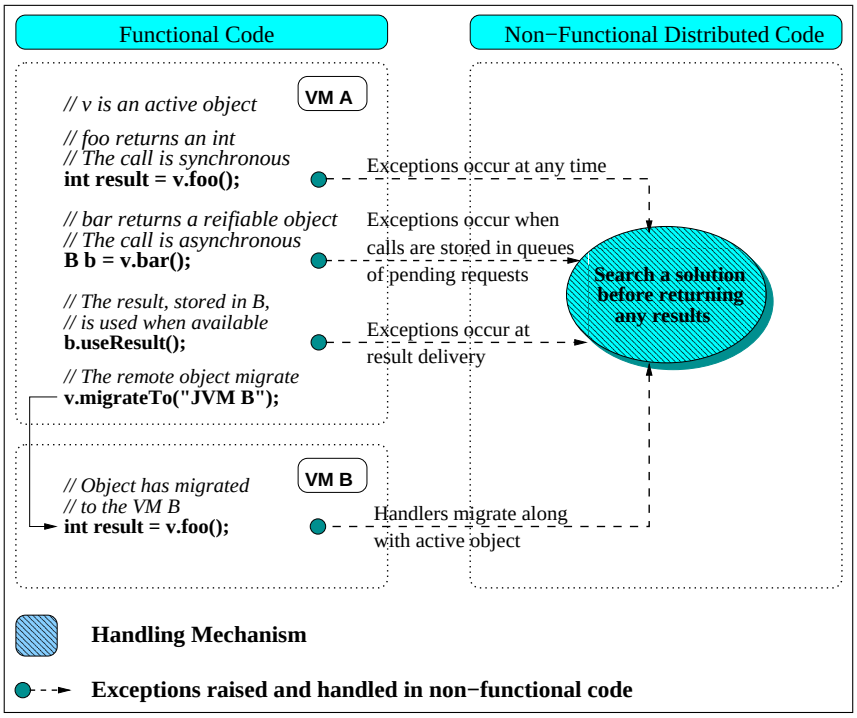

Fig. 3. Various Contexts Lead to Different Handling Location

4 Hierarchical and Dynamic Handling

To achieve transparent handling, we suggest to deal with non-functional exceptions at a non-functional level. Intensive use of the previous hierarchy should simplify the construction of the handling mechanism and improve its flexibility. Considering the models presented above as well as the current needs of distributed environments, some features are highly desirable. We identify the following characteristics as being needed.

1. Handling of non-functional exceptions is achieved in non-functional code.
2. Configuration of handling mechanism must be both dynamic and flexible.
3. Setting occurs at various infrastructure levels.
   - Middleware : static configuration in the code
   - Virtual Machine
4. Setting also occurs at various programming entity levels.
   - Remotely accessible entity
   - Mobile entity
   - Proxy
   - Future object (with asynchronous calls)
5. Both centralized and collaborative mechanisms remain possible.
6. When needed, non-functional exceptions can be handled in functional code.

4.1 Reified Exception Handlers

Handlers are dedicated exception managers applied to non-functional exceptions. To handle exceptions outside the functional code, the classical `try/catch` language construct has to be *reified* into a dedicated object. A handler object have to implement a specific interface called `IHandler`.

```
Interface IHandler {

    // Is the exception managed by the current handler ?
  public boolean isHandling(Exception e);

    // Provide a treatment for handled exception(s)
  public void handle(Exception e);
}
```

Handlers are built on the hierarchy of non-functional exceptions in order to deal with exceptions (for example, `SendRequestGroupException`) as well as groups of exceptions (every group exceptions, see [8] for further information about group). Following this paragraph, we present a handler class implementing the classical `printStackTrace` behavior for every exception. The optional `System.exit(0)` can be removed to continue the execution of applications.

```
Class HPrintStackTrace implements IHandler {

  public boolean isHandling(Exception e) {
    return (e instanceOf Exception);
  }

  public void handle(Exception e) {
    e.printStackTrace();
    System.exit(0);
  }
}
```

4.2 Prioritized Levels of Handling

Handlers are associated into static and dynamic levels. A static level is been created at JVM initialization while a dynamic level is created at runtime. Each of these levels provides a specific fault tolerance strategy built with a set of handlers. Thus, we propose a basic strategy at default level while we offer more complex ones at higher levels. Every non-functional exceptions has at least a reliable handler at default level to ensure the security of the mechanism. The following levels (presented from lower to higher priority) are associated to constants within the implementation :

1. *Default* : Default level is static and initialized in core code. Static handlers provide basic handling with every distributed exception.
2. *Virtual Machine* : Such dynamic handlers allow the configuration of a general handling behavior for every VM.

3. *Remote and Mobile Object* : Such objects can have their own handlers.
4. *Proxy* : References to active object can also have their own handlers.
5. *Future* : Results of asynchronous calls can require a specific treatment.
6. *Code* : Sometimes handlers are set punctually in the code.

Default handlers provide a basic handling strategy which avoid intensive use of dynamic handlers. Construction and configuration happen during initialization of ProActive. Default level contains only static handlers which are provided for every class of distributed exception.

VM and higher levels contain dynamic handlers to improve and adapt fault tolerance strategies defined at default level. This is particularly useful when different versions of a distributed middleware (Peer to Peer, Client/Server or Desktop/Mobile applications) derive from a common source. Such dynamic handlers are created at runtime and added to appropriate level (VM, active object, proxy, future or temporary code levels). *A standard strategy, common to every distributed application, is provided with the default level while more specific strategies are achieved with dynamic levels.* Of course, as configuration change from application to application, or even from execution ton execution, dynamic handling offers obviously much more possibilities than static handling. Handlers are modified or exchanged according to the context of environment.

4.3 Presentation of the API

We defined an API consisting of three major static functions to set, configure and use handler properties.

```
   // Binds an handler to a given class of exception at specific level.
void setExceptionHandler(Level, IHandler, Exception);

   // Removes the handler associated to a given class at specific level.
IHandler unsetExceptionHandler(Level, Exception);

   // Searches the handler associated to the class of the raised
   // exception through the prioritized levels.
IHandler searchExceptionHandler(Exception);
```

This simple example show how to protect an application against potential communication failures. Every communication exception of the given remote object is from now managed by the new handler.

```
    // Creation of a remote and mobile object
RO ro = (RO) ProActive.newActive("RO", params);

    // Set a specific handler for communication
setExceptionHandler(ro.remoteObjectLevel,
                    \"CommunicationHandler\",
                    \"CommunicationException\");

   // Remote method calls are protected
ro.foo();
```

We set up handlers statically and dynamically in order to build exception handling strategies. The following code is part of a middleware and shows how to use an exception handling mechanism at non-functional level.

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

} catch (NonFunctionalException e) {

     // Search a handler
  IHandler handler = searchExceptionHandler(e);
  handler.handle(e);
}
```

We summarize important stages to customize dynamic handler mechanism.

1. Define a `Distributed Exception` (see figure 2) within the hierarchy of non-functional exceptions.
2. Create a reliable handler from scratch or from an existing one.
3. Register handler during initialization or at runtime (*setExceptionHandler*).
4. Adapt `try/catch` construction with *searchExceptionHandler*.

4.4 implementation

We tried to keep implementation as simple as possible. Performance issues were also highly considered. Levels are represented with hashmap structures to provide instant access to handlers. We decided to minimize time complexity instead of space. This decision seems obvious when considering the huge size of memory available in our computer. However, dynamic levels can be associated to active objects and made the choice difficult : migration implies indeed both active objects and their dynamic levels.

Reflexion is highly used to search reliable handlers according to their class or mother class. Levels have precedence over the type of exception. It means that the search algorithm supports handlers of higher levels instead of more specific handlers from lower level. For instance, on the figure 4, the handler HO1 is chosen instead of HO3 because it belongs to an higher level. This option seems to be more natural but it is still possible to reverse the priority when needed.

4.5 Performance Issues

Space Complexity Every JVM contains at least default and VM levels, each one consisting of some handlers associated in a hashtable. In comparison with available resources of modern computers, memory requirements are rather poor. Of course, higher levels add weight to entities associated with them. Such dynamic handlers are created on the fly and associated, for example, to active objects through the active object level. During migration, dynamic levels move along with their associated active objects. We cannot ignore the cost which

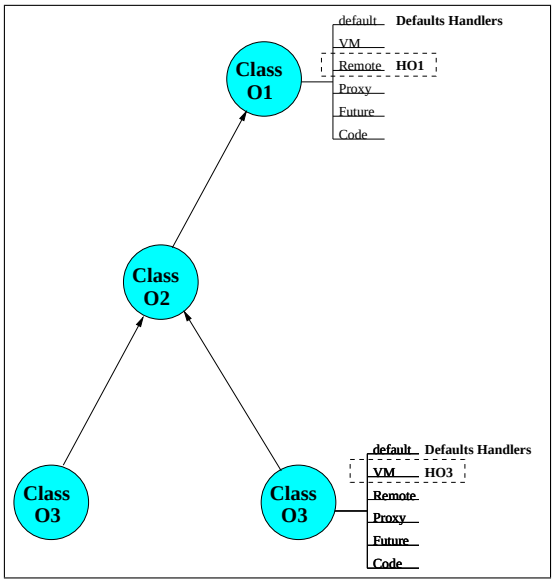

Fig. 4. Levels have Precedence over Exception Type

mostly depends of the number of handlers. We serialized active objects to compute the overall size of handlers and levels. The empty size of the *No handler strategy* could be reduced to 0 with the singleton pattern creating the data structure upon the first handler setting. The table 1 summarizes some typical space requirements.

Time Complexity The *Time complexity* is interesting. Adding and removing handlers are just punctual operations which do not break performance. On the other hand, searching handlers is almost complexity less due to some specific hashtable properties. Existing `try/catch` required by distributed methods are overloaded with more complex strategies based upon our handling mechanism. Overall performance are still efficient as the mechanism is only used when exceptions occur. We did some benchmarks where we raised huge number of non-functional exceptions and measured how long each handling mechanisms did to find the reliable handler. Note that handling time is null as handlers do nothing. Results are presented on the figure 5. We expected our handling mechanism to be a few times slower than the standard `try/catch` mechanism due to its hierarchical and flexible architecture. But significant differences appear only after thousand successive raised and handled exceptions. The performance ratio between the two mechanisms is approximately of 1:4 which means that for regular use, the dynamic handler mechanism is competitive and can be compared to other ones.

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

Table 1. Space Requirements Depends of the Number of Handlers

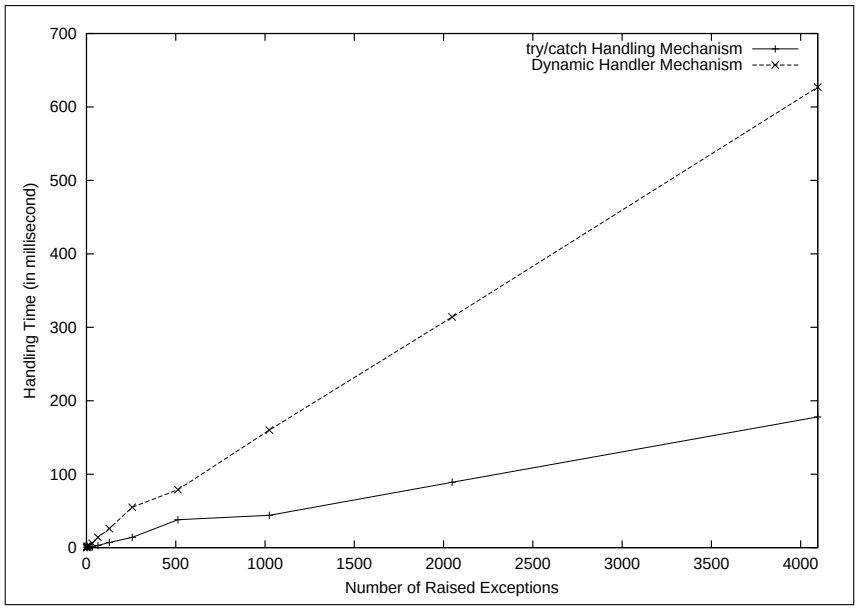

Fig. 5. Time Complexity

5 Canonical Examples

We present now some interesting properties of the model such as simplicity, flexibility or robustness with the use of canonical examples.

5.1 Simulating a Centralized Error Manager

We demonstrate that our handling mechanism can easily be configured into a centralized error manager as presented in [3]. We achieve a similar goal by creating an active object containing plenty of prioritized handlers on one single JVM. This object is known from every other JVM and from every active object. When an exception occurs somewhere, the right handler is not searched in the internal scope from where it has been raised but the exception is sent to this centralized error manager. This manager searches the appropriate handler and send it back to the caller in order to handle the exception. We don't avoid typical problems common to centralized error manager but we offer at least an equivalent handling mechanism.

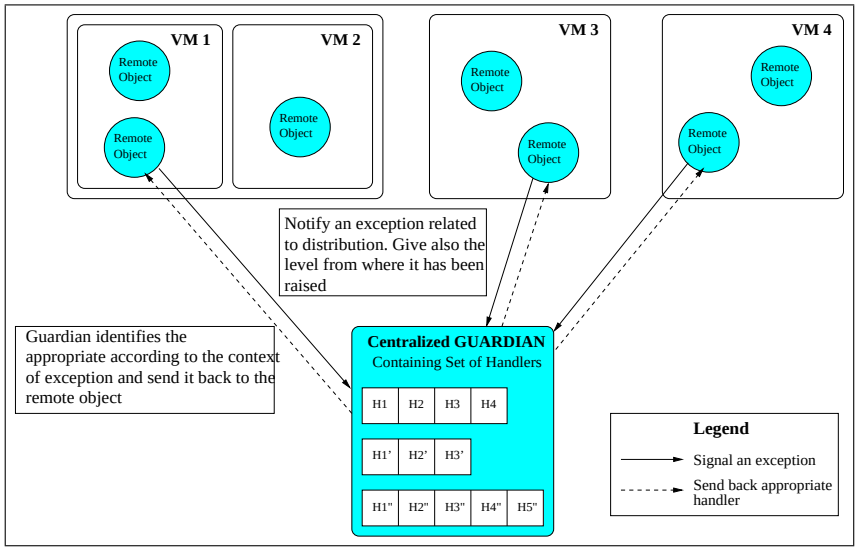

Fig. 6. Centralized Error Managers are simple to implement

5.2 Handling Exceptions with PDA

Personal Digital Assistants are mobile by definition. Distributed applications running on such computers must provide unconnected mode to handle at least `CommunicationException` due to broken connections. We created specific handlers storing the pending requests raised when a PDA is no longer reachable.

Time by time, a dedicated thread check if the connection has been restored in order to deliver the queued requests. As shown on 7, handling can be adapted according to the PDA : B and C do not have specific behavior while A stores its own requests.

PDA handlers have the following scheme (code is simplified).

```
Class HCommunicationExceptionPDA implements IHandler {

  public boolean isHandling(Exception e) {
    return (e instanceOf CommunicationException);
  }

  public void handle(Exception e) {

       // The first time an exception occur a
       // thread testing connectivity is created
       // This thread will deliver the queued requests
    if (firstUse) {
      connectivityThread = new ConnectivityThread();
    }

       // Then the reified method call is stored
       // Exceptions are not propagated
    queue.store(e.getReifiedMethodCall());
  }
}
```

5.3 Peer-to-Peer Fault-Tolerance Strategy

Peer-to-Peer computing running on large scale distributed systems is a source of errors. In the case of SETI@home[3] clients first connect to a centralized server, download and compute some amount of data and finally upload results to the server. Each of these stages is subject to errors. With more advanced Peer-to-Peer applications such as eDonkey[4], the model is slightly different but the same difficulties along with new ones appear. Of course, many solutions can deal with complex environments but we are convinced that our mechanism is not only reliable with large systems but also lead to easier managements of the handling strategy. Each participant has an appropriate set of handlers corresponding to a specific fault-tolerance strategy. Raised exceptions are handled according to the participant and its particular context.

Nevertheless, this proposition should be seen as a possible perspective as further work is required to offer sound Peer-to-Peer applications.

[3] http://setiathome.ssl.berkeley.edu/
[4] http://www.edonkey2000.com

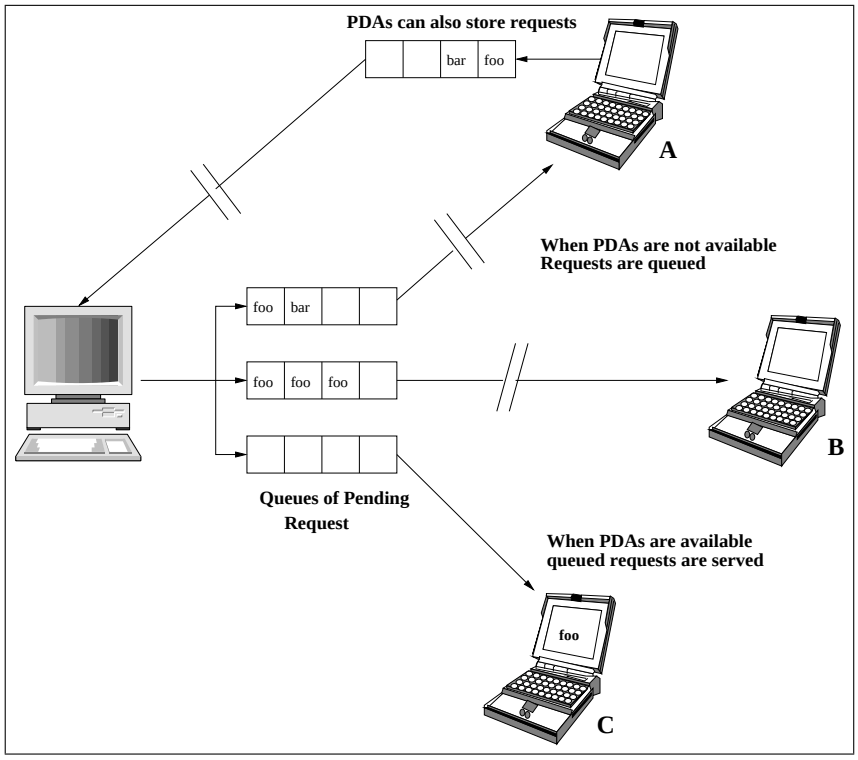

Fig. 7. Handlers Store Requests and Check Connectivity

6 Conclusion and Perspectives

We have developed a generic handling model for non-functional exceptions and more specifically for exceptions related to distribution. As implementation use the classical `try/catch` language construct, the model is reliable for a large panel of programming language such as C++, Java or even the recent C#.

We defined non-functional exceptions along with their classification and described where and when handling takes place. Then, we built a hierarchy of prioritized levels to provide hierarchical fault tolerance strategies. As seen in the canonical examples, our model is not only generic but also flexible. Every handling strategy can be adapted to the needs of specific distributed applications.

As future work, it would be interesting to develop exception handling components based upon the model presented in this paper. We could plug such entities directly in applications in order to provide specific handling strategies. More generally, we could develop distributed and other non-functional strategies for programming model such as component programming.

References

1. Valerie Issarny. *Concurrent Exception Handling.* Advances in Exception Handling Techniques 2000: 111-127. Inria Rocquencourt.
2. 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.
3. Arnand Tripathi and Robert Miller . *Exception Handling in Agent-Oriented Systems.* Advances in Exception Handling Techniques, Springer-Verlag LNCS 2022, March 2001.
4. Denis Caromel, W. Klauser, 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
5. 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.
6. Kiczales, Lamping, Mendhekara, Maeda, Lopes, Loingtier, Irwin. *Aspect-Oriented Programming.* Proceedings of ECOOP 97, n 1241 LNCS, Springer-Verlag, June 1997, p. 220-242.
7. 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.
8. 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.
9. 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.