Robust Exception Handling in an Asynchronous Environment

Denis Caromel and Guillaume Chazarain

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

Abstract. While distributed computing is becoming more and more mainstream, it becomes apparent that error handling is an aspect that deserves more attention. We place ourselves in the context of distributed objects, and we try to hide the network behind the usual remote method call semantic.
More precisely, we concentrate on the case where method calls are asynchronous, which complicates the error handling. We start with a state of the art in this research field, and then propose our approach, detailing the problems we faced and how we solved them.
Our achievement was to provide the well-known way of handling exceptions using the `try/catch` construction to programs written using asynchronous method calls. Unfortunately, the usage is not totally transparent because of Java limitations. The main part of the approach is to build an exception mask stack following the Java one.

1 Introduction

Before the introduction of exceptions, errors were commonly returned along the same path as normal return values, it was the responsibility of the caller to check the return value. Exceptions, however, have a dedicated channel [1], it is a subpath of the returned value path but it deserves as much interest as its parent. So, the introduction of exceptions was a major milestone, but when dealing with distributed computing they are often neglected. Efforts are under way to improve this situation, for instance with [2] and [3].

We concentrate on the Java implementation of exceptions, which brings two interesting aspects: exceptions are checked thanks to the `throws` annotation, and it's possible to bypass this check using a subclass of `RuntimeException` as the exception type.

The most prominent combination of distributed computing and exceptions may be Java RMI's `RemoteException` which forces every remote method to add a `RemoteException` to its `throws` clause, and the assorted `try/catch` blocks.

These obligations have negative effects: constrained to wrap remote method calls with `try/catch` blocks, programmers sloppily tended to place empty `catch` blocks, thus swallowing the exception. So all of this care in the Java exceptions

system resulted in less robust programs. The solution from the C# camp is to totally get rid of the `throws` keyword, so all exceptions are unchecked. This difference between C# and Java is still subject to discussion, our take is that incomplete `throws` lists should be reported as a warning by the compiler, not an error. So one gets to choose between robust or toy code simply by fixing or ignoring the warning.

2 ProActive

ProActive [4] is a GRID Java library (Source code under LGPL license) for parallel, distributed, and concurrent computing, also featuring mobility and security in a uniform framework. With a reduced set of simple primitives, ProActive provides a comprehensive API allowing to simplify the programming of applications that are distributed on Local Area Network (LAN), on cluster of workstations, or on Internet Grids.

ProActive is used as a base for our implementation, so we present here the aspects needed to understand the context.

As with every library, there is always some confusion as how to call the end user, since it's actually most of the time a programmer. So we use the term programmer as the library user.

ProActive (like RMI) works with the method call abstraction, so we always have a caller and a callee. We name them using the client/server metaphor, although these roles are not static in ProActive, but it is more common in the distributed world.

These method calls are made to active objects. These are objects with a thread to serve requests from their incoming queue. When possible, ProActive method calls are asynchronous. If the method return type is `void` the call is then one-way, in that we will not wait for any reply and the call will be asynchronous in the message passing semantic.

Asynchronous method calls return a placeholder object of type compatible with the actual return type. This placeholder is called a future and is dynamically updated with the real returned result upon availability. Trying to manipulate the future object before its return will result in a wait-by-necessity, that is, the operation will block until the result arrives, at which point the operation will be resumed.

ProActive is developed with some constraints in mind, to which the presented system must adhere. These constraints are that we prevent ourselves from tampering with the bytecode/JVM or preprocessing the source code before compilation. Only pure Java code is allowed. Breaking these constraints makes the programmer's code harder to debug because the executed code is not the one he actually wrote.

2.1 Exception Handling in ProActive

Unlike most other projects presented here, ProActive does not treat all exceptions the same way. It makes a distinction between functional exceptions and non-functional ones. To put it simply, functional exceptions are thrown and

caught in the ProActive code, and non-functional ones are thrown by ProActive to signal problems impacting the ProActive layer. The latter exceptions are caught using a mechanism called NFE (Non-Functional Exceptions) [5] briefly presented in section 2.1.2.

2.1.1 Functional Exceptions

In ProActive, methods that declare exceptions in their signature are always called synchronously. The mechanism presented here is designed to overcome this limitation.

Concerning runtime exceptions, as we cannot predict if a method call will throw one or not we put the potential `RuntimeException` in the future object, and throw it when the future is accessed. For one-way calls, we cannot have such a mechanism (because no reply is expected), so we simply print a stack trace when a `RuntimeException` is caught, this behaviour will be improved by the proposed mechanism.

2.1.2 Non-Functional exceptions

Non-Functional exceptions in ProActive are classified within different categories and handlers are associable with these exception categories at different priority levels. These handlers run on one side of the call or the other depending on the exception type. This mechanism permits to achieve some interesting behavior like a disconnected mode for a mobile environment.

3 Related Work

We can see a trend in most attempts at asynchronous remote method invocation that tackled exceptions, which is the use of callbacks. Basically, a callback is associated with an exception, in one way or another, and when this exception is thrown the callback is called. The aforementioned ProActive NFE mechanism fits into this case. We will see the benefits of this approach, its limitations, and some ways to go past them.

3.1 Exception Callbacks

3.1.1 The JR Programming Language

The JR [6] programming language extends Java to provide a rich concurrency model, based on that of the SR [7] concurrent programming language. JR provides dynamic remote virtual machine creation, dynamic remote object creation, remote method invocation, asynchronous communication, rendezvous, and dynamic process creation.

The extension of interest to us is the `handler` [8] keyword. It is used in combination with the `send` keyword. The latter keyword is used to introduce the asynchronous calls extension. So, with these two keywords a callback is associated with the forthcoming exception at call time. As we are in an object oriented environment, it's not simply a function that is used as a callback, but a whole object implementing callback methods like the `myHandler` instance in Figure 1.

```
IOHandler myHandler = new MyHandler();
...
send readFile("/path/to/my/file") handler myHandler;
...
```

Fig. 1. Example using the `handler` keyword in JR

This annotation is mandatory for every asynchronous call introduced by the `send` primitive that can throw exceptions. The argument to `handler` is an instance of a class responsible for handling any exception that may be thrown according to the method signature. The JR compiler statically checks this. So the handler provides methods for every exception type throwable by the called method. These callbacks receive as a single argument the actual thrown exception.

In this realization, asynchrony is not transparent because it's explicitly requested by the programmer. On the other hand it can be cumbersome to annotate each asynchronous call that can throw exceptions as in the example.

3.1.2 ARMI

The ARMI [9] project covers not just exception handling but aims at providing an asynchronous remote method invocation mechanism for Java. To deal with exceptions, it proposes two solutions, one of which uses a callback mechanism.

ARMI uses future objects, like ProActive, as a way to transparently introduce asynchrony, the callback mechanism works by registering <exception-type, exception-handler> pairs with the future. With this solution they reach the same level of granularity as the JR approach, but the goal is different since asynchrony serves as an optimization here.

3.1.3 Conclusion on Callbacks Mechanisms

The first advantage of callbacks mechanisms is that it's quite easy to implement. The second one is that it gives the feeling of letting the programmer all the freedom to implement his behavior to handle the exception since he chooses the function to be called upon an exceptional event. This feeling is quite misleading because the main usage of exceptions is impossible with callbacks: unwinding the call stack up to a certain point. An asynchronous environment makes this even harder since the call stack may have gone away when the exception is thrown. But when we use asynchrony as an optimization it is important to keep the ability to unwind the stack as this is taken for granted in a synchronous environment.

The other limitation with callbacks is the context representation. When using the `try/catch` pair, the error handling code is written next to the code to protect, so it sounds normal to have access to all local variables when handling the exception. On the other hand, with callbacks this is not possible since these variables are not part of the context information given to the callback.

Modern exception implementations (like the Java one) alleviate this problem by letting the designer build exceptions as first class object, thus adding every

contextual information he finds useful to them. This is not enough since the exception thrower may not anticipate how the exception catcher will handle it. Moreover we argue that the most interesting context is the one provided by the local variables on the stack. For example, the local variables in the `try` block preceding the `catch` handling the exception can be useful in this handling. Unfortunately, this information is inaccessible to an exception callback.

Another advantage offered by the `try/catch` usage in Java is the automatic unlocking of mutexes acquired using the `synchronized` primitive.

3.2 Closures

With exceptions, we have the problem of capturing some environmental context which is handled by closures as present in functional languages. A closure can be assimilated to the association of a function and a captured environment. In Figure 2 we can see a small example of an error handler defined as a closure. Our local variable `some-value` is captured when the `error-handler` closure is created, and when the closure is executed, not only does it sees the actual value of the variable, but it can modify it. The context capture is complete, we just need to unwind the stack thanks to some continuations and we can handle exceptions using callbacks but with the good properties of the standard `try/catch`.

```
; Actually this should do serious work instead of just calling
; the error handler.
(define (try-something-with-error-handler error-handler)
     (error-handler))

; some-value is our local variable we'll have access to.
; Unlike in Java, the error handler code has a read/write
; access to the environment captured when creating the closure.
(let* ([some-value 1]
       [error-handler (lambda ()
                        (if (= some-value 2)
                            (set! some-value 3)))])
     (set! some-value 2)
     (try-something-with-error-handler error-handler)
     (display some-value))

=> 3
```

Fig. 2. Scheme example of a closure

Closures can be simulated in Java with inner classes but the dirty tricks needed to work around the `final` limitation [10] make them impractical to handle exceptions, compared to the `catch` block.

3.3 Exception in the Future

As we are concerned with asynchrony, future objects are commonly used, so another approach to asynchronous exception handling is to save the exception in the future and throw it upon access.

3.3.1 ARMI

This solution is the second one implemented in ARMI. The authors notice that the result may be accessed far away from the actual method call. With this in mind, we lose the benefits of reusing the common `try/catch` model since the exception may be thrown in a totally different `try/catch` block from the one containing the method call.

3.3.2 Java 1.5

In its 1.5 version (code named Tiger), Java features non transparent future objects [11]. One of the differences with the other approaches is that this one is absolutely user explicit. This is certainly a drawback in our context of asynchrony as an optimization, but by forcing the programmer to manually retrieve the resulting object from the future, he won't have any surprise about where his exception is thrown.

3.3.3 Mandala

The Mandala [12] project provides the RAMI [13] package which is an asynchronous RMI implementation. There are different asynchrony models more or less transparent to the programmer, but with various limitations.

It provides both exception handling techniques: callbacks usage, and encapsulation of the exception in the future. In totally transparent asynchronous mode, the exception is handled by simply setting the result in the future to `null` which will trigger a `NullPointerException` upon access to it.

3.4 Other Approaches

3.4.1 Proxyc

Proxyc [14] is another attempt at providing future objects in a transparent way thanks to some bytecode rewriting. Their solution to the exception handling problem is to make synchronous all asynchronous method calls that can throw exceptions. They suggest to use instead some wrapper method that would handle the exception, this wrapper would be an asynchronous call.

They propose another approach using the fact that they are rewriting the bytecode. As they have to insert code to get the actual result from the future object, they introduce the possibility of letting the programmer specify some exception handling code to place at these points.

3.4.2 RMIX

RMIX[15] is a communication framework for Java, based on the Remote Method Invocation (RMI) paradigm. Its feature of concern to us is asynchronous method calls, and its exception handling solution is quite original. The fact that an asynchronous method raised an exception is a sticky bit in the object, no more asynchronous calls are accepted until this bit is cleared. The ways to clear this bit are: either by handling an exception from a synchronous call, or by manually resetting it.

3.5 Summary

The conclusion we can draw after reviewing these projects are twofold. The first one is that there is a mutual exclusion between transparently providing a mechanism and keeping the bytecode intact. The second conclusion is that callbacks are not enough as a solution.

4 Explicit handling of the exception mask stack

First we provide an overview of the proposed approach, then it will be shown step by step how to use it. Its working will be detailed. The main aspect of the mechanism is to insert some wait-by-necessity in appropriate moments.

4.1 Overview

The goal is to provide an exception handling mechanism for both functional exceptions and non-functional ones. We also want to reuse the well known `try/catch/finally` construction since this is the only way in Java to manipulate (unwind) the call stack. Furthermore, nobody should be offended by an error handling code located in a `catch` block.

The mechanism is useful for asynchronously launching a method call even if its signature includes exceptions. A `try/catch` block is associated with this call, not necessarily the toplevel one at the moment of the call, but as an optimization we use the first one on the stack that can catch one of the declared exceptions. In order not to break the semantics associated with exceptions, we add two rules.

These rules are the main principle of the system:

- when accessing the future result, the potential exception is thrown,
- we do not leave the `try` block associated with the method call before the future returns.

The second rule implies that we may have to wait for some pending futures at the end of the `try` block. Actually, this is not enough, as we take great care to ensure that the exception is caught in its corresponding `catch` block. This must be specifically enforced by the mechanism, for example in some nested `try/catch` instances, as shown in Figure 3.

```
class ParentException extends Exception {}
class ChildException extends ParentException {}

try {
    A a = ro.foo(); // throws ChildException
    /* Here we must wait for 'a' */
    try {
        a.bar();
    } catch (ParentException pe) { ... }
} catch (ChildException ce) { ... }
```

Fig. 3. `tryWithCatch()` can be a wait-by-necessity point

If in that code, the `ro.foo()` call ends up with a `ChildException`, we don't want this exception to be thrown when the future is accessed through the `a.bar()` call. Doing so would wrongly let the exception be handled by the nested `catch (ParentException pe)` block instead of the rightful `catch (ChildException ce)` one. It's important to notice that the situation would be the same if it were `ParentException` that was a subclass of `ChildException` because in this case the exception thrown by `ro.foo()` could be a `ParentException`.

In Java, the target catch of a given exception is dynamically determined, so we must take care to prevent exceptions from being handled in unexpected `catch` blocks.

In order to be activated, the mechanism must be called in the code to protect at some key points, all of them shown in Figure 4.

```
public class RemoteObject {
    public DangerousThing dangerousMethod() throws DangerousException {}

    public static void main(String[] args) {
        ProActive.tryWithCatch(DangerousException.class); // Here (1)
        try {
            DangerousThing[] dt;
            RemoteObject[] ro;

            for (int i = 0; i < dt.length; i++)
                dt[i] = ro[i].dangerousMethod();

            ProActive.endTryWithCatch(); // Here (2)
        } catch (DangerousException de) { ... }
        finally {
            ProActive.removeTryWithCatch(); // And finally here (3)
        }
    }
}
```

Fig. 4. Complete example with the mechanism

The only goal of all those added method calls is to let ProActive know the state of the exception mask stack. Since this information mostly pertains to the client, it is implemented on the client side and very little is actually transmitted on the wire.

Note that those instructions are explicitly needed because of the lack of reification of the Java exception mask. The programmer and the middleware runtime have no access to it.

4.2 Before the Block

Before every `try` willing to put at work the mechanism, ProActive must be informed about which exceptions types will be caught in this block. This is done using the `tryWithCatch()` method taking as parameter an exception class or

an array of exception classes. These are the exceptions caught in the subsequent `catch` blocks.

As we reimplement a stack unwinding based exception system, this method pushes an exception handler on the stack. As an optimization, it also recomputes the current exception mask so that we know for each ProActive call if its exceptions will be handled by the mechanism or not.

4.3 During the Block

Here, we expect to asynchronously call methods throwing exceptions corresponding to the advertised exception types, otherwise the system would be quite pointless.

If an asynchronous method call ends with an exception, it is saved in the future in order to be thrown upon access, or when for some reason we have to throw the current exception.

If the method call was synchronous because of its return type (primitive type like `int` or a `final` class), the behavior is not changed from before the mechanism: the call is synchronous and the potential exception is thrown at its end.

In the one-way asynchronous case, the only moment where we could throw the exception is when the method `endTryWithCatch()` is called. This is suboptimal because the programmer is left with a subtle dilemma concerning one-way calls: either he makes big `try/catch` blocks so that the `endTryWithCatch()` comes late and a large window is left for calls in parallel before seeing the exception, or he wants his exception soon and he has to make small `try/catch` blocks.

The proposed solution to this problem consists in two Java methods used to throw an exception if one has arrived. We provide two methods because one is blocking and the other is not. The non-blocking one (`throwArrivedException()`) simply consults the current state and throws the potential exception, whereas the blocking one (`waitForPotentialException()`) is in some sort a barrier in that it waits for all asynchronous calls that could throw an exception in their return, and then throws the potential exception too.

4.4 At the End of the Block

The last instruction in the `try` block must be a call to `endTryWithCatch()`. The job of this method is to wait for the first of those events:

- every call in the current block that could throw an exception declared in the `tryWithCatch()` parameter has normally returned,
- an exception arrives.

This method also pops the current exception mask on the stack. If an exception is thrown before this call, it will be bypassed. We'll see in a moment how we handle this situation.

5 Implementation: Problems & Solutions

Trying to provide asynchronous exceptions in a way resembling to synchronous exceptions is not trouble free. We identify three main problems, more or less tied to our implementation in Java.

5.1 Source Code Modification

Java has the refinement of presenting exceptions as first class objects, thus, giving them all the features found in objects. Unfortunately, their handling (`try`, `catch`, `finally`) is totally impenetrable. That's why, the major part of the proposed mechanism is to reimplement an exception stack, side by side with Java's original.

In this situation, the main hazard to watch for is an inconsistency of the exception stacks with respect to the actual user stack. The ProActive view of the stack needs not be complete, only the subset containing asynchronous method calls is needed. Nevertheless, assuming the programmer takes care to annotate its exception stack manipulations with the aforementioned ProActive method calls, we can be out of sync as a result of a thrown exception unwinding the stack. Not only we cannot detect when an exception is thrown, but also we cannot figure out up to where did we unwind the stack.

Even when we ourselves throw an asynchronous exception, we cannot be sure at which level of the stack it will end up. This is because our view of the stack is incomplete.

Our solution to all these problems resides in asking the programmer to add a ProActive call in the `finally` block of every instrumented `try/catch` blocks. When an exception unwinds the stack, every `finally` block on the way upward will be executed, popping our stack at every step.

This problem has its origin in what is perceived as a Java shortcoming. Java does not provide methods to inspect the exception mask stack. It only provides the call stack thanks to a method, ironically, in the `Throwable` class (i.e. `getStackTrace()`). On the other hand, this shortcoming can be accepted if the goal is to hide the exception handling mechanism in order to optimize [16] it in the JIT.

Another source of desynchronization is a change in the exception handling code not propagated to the system, for example, a `catch` clause is suppressed, but the `tryWithCatch()` call continues to advertise it. We fix this problem by providing an automatic annotator. The annotations can be automatically computed given the source code, so we wrote a tool to do exactly that. This is not considered as a preprocessing but as a help to the programmer, that he is free not to use.

5.2 Throwing Exceptions

In the mechanism we rely on the possibility to launch any exception at any time, as shown by the `throwArrivedException()` method. On the other hand, we are in a Java environment, which enforces the `throws` declaration in method signatures to inform about the type of exceptions that can be thrown. This aspect has been a showstopper in many attempts at asynchronous exceptions, (as noticed by [9]).

The first point to consider in that problem is that the `throws` rules are enforced only by the compiler, so if we bypass the compiler checks, nobody else will prevent us from throwing exceptions.

The chosen approach to bypass the compiler is to dynamically create the bytecode at runtime with the help of libraries like ASM [17]. It permits to write

a method that throws any exception without checking the `throws` declaration, that we will happily leave empty. So we write our class with ASM and we load it thanks to the well known technique of calling `defineClass()` by reflection.

Now that we have our method that can throw any exception without the associated `throws` declaration, we have to call it. We cannot do it directly because as it is created dynamically it does not exist at compile time, and we would end with code that does not compile. Another solution would be to call the method using the reflection API, but the method in this API to call any method not only has a non-empty `throws` declaration but also wraps the actual exception in a `InvocationTargetException`. So the reflection way is useless there.

We had to fallback on another solution, this time using Java's dynamic binding. Let's say we have an interface containing a single method (used to throw an exception) with a, very importantly, empty `throws` declaration. The dynamically built class implements this interface, and the method actually throws the exception thanks to the lack of verification of the `throws` declaration. In the code we keep an instance of this interface, so the code compiles. Then we create an instance of the dynamically built class by reflection and we affect it to the aforementioned variable. When we want to throw an exception we simply have to call the method on this variable, it will compile since the method is in the interface, and thanks to Java's dynamic binding the dynamically built method is the one to be actually executed.

5.3 Consecutive Exceptions

Trying to implement asynchronous exceptions, side by side with Java exceptions can lead to strange situations. In the synchronous world of exceptions, when something bad happens an exception is thrown and the control flow leaves the offending stack frame, and of course the subsequent calls in this stack frame are skipped until the block with the matching `catch`.

With asynchrony, the whole point is to avoid waiting for the end of the call, so it is possible to receive two consecutive exceptions, potentially having the same cause (e.g. a network failure occured and several remote calls aborted).

The best we can do in this case is to report the first exception, and discard the subsequent exceptions. In the synchronous case, the second call would not have been launched, so its exception is useless. The problem is that we may end up in an incoherent state by launching calls even though the previous ones threw exceptions. We rely on the wait-by-necessity semantic between futures which produces a dependency graph in some sort to gain some correctness. Our solution is to assume that the dependency between values is the same as the one between exceptions, which seems natural.

Another considered approach is to throw the exception as soon as possible. That is to say, each time a ProActive method is called, if there is a pending exception we throw it. This is technically feasible, but can be problematic for the programmer since exceptions can then be thrown in unpredictable ways. So this approach is still left aside as experience is gained on the subject.

6 Conclusion

By turning asynchronous the method call mechanism, error handling becomes tricky. The most common solution is the usage of callbacks, but as we have seen it is not as powerful as it seems. Our approach is to trade a little bit of asynchrony in exchange of some control about where exceptions are caught. Of course this solution has a cost that needs to be evaluated. This cost is application dependent, and as before, the code needs some thinking to benefit the most from asynchrony. Typically, the result from a method call will be used in the same `try` block so as to avoid testing the result validity, this way it's not the end of the `try` block that will cause the wait-by-necessity but the return value usage.

References

1. Goodenough, J.B.: Exception handling: issues and a proposed notation. Commun. ACM **18** (1975) 683–696
2. Souchon, F., Dony, C., Urtado, C., Vauttier, S.: A proposition for exception handling in multi-agent systems. In: Proceedings of the 2nd international workshop on Software Engineering for Large-Scale Multi-Agent Systems, Portland, Oregon, USA (2003) 136–143
3. Romanovsky, A., Xu, J., Randell, B.: Exception handling and resolution in distributed object-oriented systems. In: ICDCS '96: Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96), Washington, DC, USA, IEEE Computer Society (1996) 545
4. Caromel, D., Klauser, W., Vayssière, J.: Towards seamless computing and metacomputing in Java. Concurrency: Practice and Experience **10** (1998) 1043–1061
5. Caromel, D., Genoud, A.: Non-functional exceptions for distributed and mobile objects. (In Romanovsky, A., Dony, C., Knudsen, J., Tripathi, A., eds.: Proceedings of the Exception Handling in Object-Oriented Systems workshop at ECOOP 2003)
6. Keen, A.W., Ge, T., Maris, J.T., Olsson, R.A.: Jr: Flexible distributed programming in an extended java. ACM Trans. Program. Lang. Syst. **26** (2004) 578–608
7. Olsen, R.A., Andrews, G.R., Coffin, M.H., Townsend, G.M.: Sr: A language for parallel and distributed programming. Technical Report TR 92-09, University of Arizona (1992)
8. Keen, A.W., Olsson, R.A.: Exception handling during asynchronous method invocation (research note). In: Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing, London, UK, (Springer-Verlag)
9. Raje, R.R., Williams, J.I., Boyles, M.: Asynchronous Remote Method Invocation (ARMI) mechanism for Java. j-CPE **9** (1997) 1207–1211
10. Logan, P.: Closures that work around final limitation (cunningham & cunningham wiki). (`http://c2.com/cgi/wiki?ClosuresThatWorkAroundFinalLimitation`)
11. Lea, D.: Concurrency utilities. (`http://www.jcp.org/en/jsr/detail?id=166`)
12. Vignéras, P.: Vers une programmation locale et distribuée unifiée au travers de l'utilisation de conteneurs actifs et de références asynchrones. PhD thesis, Université de Bordeaux 1, LaBRI (2004)
13. Vignéras, P.: Rami user's guide. (`http://mandala.sf.net/docs/rami.pdf`)
14. Pratikakis, P., Spacco, J., Hicks, M.: Transparent proxies for java futures. In: OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2004) 206–223

15. Kurzyniec, D., Sunderam, V.S.: Semantic aspects of asynchronous RMI: The RMIX approach. In: Proc. of 6th International Workshop on Java for Parallel and Distributed Computing, in conjunction with IPDPS 2004, Santa Fe, New Mexico, USA, IEEE Computer Society (2004)
16. Suganuma, T., Yasue, T., Nakatani, T.: A region-based compilation technique for a java just-in-time compiler. In: PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, New York, NY, USA, ACM Press (2003) 312–323
17. Bruneton, E., Lenglet, R., Coupaye, T.: Asm: a code manipulation tool to implement adaptable systems. Adaptable and extensible component systems (2002)