# Transparency and Asynchronous Method Invocation

Pierre Vignéras

GIK Institute of Engineering Sciences and Technology
Topi, N.W.F.P, 23460 Pakistan.
pierre@giki.edu.pk

**Abstract.** This article focuses on transparency in the context of asynchronous method invocation. It describes two solutions available to provide *full-transparency*: where asynchronism is entirely masked to the developer. The main contribution of this paper is to clearly present the drawbacks of this approach: exception handling and developer consciousness are two problems inherent to *full-transparency* that makes it at least, hard to use, at worst, useless. This paper defends explicit asynchronous method invocation and proposes *semi-transparency*: almost all the complexity of asynchronism is masked to the developer but the asynchronism itself.
**keywords:** transparency, asynchronous method invocation, concurrency

## 1 Introduction

Transparency is an abstract notion already used in many contexts. Intuitively, its goal is to hide the complexity of an aspect – usually a non functional one – to developers. Java/RMI [16] is a good example of transparency used to hide the complexity of the remote aspect. The syntax of a remote method invocation is almost identical as a local one. The only difference is the fact that remote methods may throw a checked exception (an instance of the class `java.rmi.RemoteException` or of one of its subclass). Anyway, the semantic of such a call is rather different than in the local case: since parameters are marshaled, the remote target of the call gets either copy or references of each original parameters depending on some of their characteristics (primitive type, serializable, implementing the `java.rmi.Remote` interface). This behavior is also used for the result transmission. As seen by the success of the Java/RMI framework, and despite the existence of many works that try to enhance it [13,15,10,17], this example clearly shows that transparency of the remote aspect eases the making of distributed applications. Distributed programming has thus clearly been simplified thanks to transparency. Is it also true in the context of concurrent programming?

Concurrency will probably be one of the major concern for developers in the next decade. Whereas SMP architectures are quite common today, next-generation processors (CMP, SMT, VMT) [1] will provide lots of low-level threads to the operating system. So, concurrency will be almost anywhere, in low-level architectures, in operating systems and in high level languages such as Java and C# which already provide a thread API to express concurrency in object oriented applications.

Asynchronous method invocation is another paradigm that allows the expression of concurrency. It extends very well the standard synchronous method invocation paradigm and also extends naturally to the remote case as remote method invocation does.

This paper focuses on transparency in the context of asynchronous method invocation. Note that the Java language has been used in our study, but many issues described in this paper will also be found in any other object oriented language. Furthermore, some solutions described in this paper is already available in the Mandala project [18].

This paper is organized as follow: section 2 deals with concurrency in the asynchronous method invocation paradigm. Section 3 presents *full-transparency*, and two solutions for its implementation. The reasons why this mechanism should be avoided are also explained in this section. We propose an alternative in section 4. We finally conclude in section 5 along with some perspectives.

## 2 Dealing with Concurrency

Since we are focusing on transparency, we distinguish the mechanism that provides transparency and the one that provides concurrency. For the latter, many abstractions may be used such as active objects [11], actors [2,3], separates [14], active containers [7,6], or asynchronous references [19].

When making an asynchronous call, the specified method may not be executed concurrently with the caller thread. The underlying abstraction may do many things before running the method while the caller thread may have reached the end of the caller method, or may have already died.

Moreover, when performing many asynchronous method invocations successively, the execution of these methods may also be sequential. This is sometimes necessary when the object on which asynchronous method invocations are made is not designed in a concurrent context (thread-safety, re-entrance). In this case, to prevent problems such as deadlocks and data corruption for example, the underlying abstraction may forbid the concurrent execution of methods using a *non-concurrent asynchronous semantic:* a single thread deal with the method invocation requests. This is the approach of the active object paradigm. On the other extreme, an abstraction may be customized to use a specific *asynchronous policy* (FIFO, one thread per call, thread pool) for the implementation of a given *asynchronous semantic* (non-concurrent or concurrent). This is the way taken by the *asynchronous reference* paradigm.

Nevertheless, the use of any *asynchronous semantic* is subject to deadlocks [19], even non-concurrent one. So even if the asynchronous method invocation may seem simpler to use (compared to thread programming), it does not solve common issues found in concurrent programming in general. Asynchronous method invocation is just a way to *express* concurrency, not a solution to problems it involves. For this purpose, a careful design of classes is still required using concurrent design principles and patterns [12].

## 3 Full-Transparency

In the case of concurrent programming, we define *full-transparency* as below:

**Definition 1 (Full-Transparency)**
*An asynchronous method invocation is* fully-transparent *if its syntax is not distinguishable from a standard, synchronous method call. Moreover, the object type used to make an asynchronous method invocation must be compatible with the one used to make a synchronous method invocation.*

Two distinct entities must be provided to ensure *full-transparency*:

- the *asynchronous proxy* [8] sends invocation requests to the underlying abstraction[1] which makes the call really asynchronous;
- the *transparent future* [20] used to recover the result.

Transparent futures may use the *wait-by-necessity* mechanism [4] provided by ProActive [5] and Mandala [18]: when a client makes an asynchronous call, a future object – subtype of the original type declared by the method – is immediately returned. When the client uses this future, it is blocked until the real result becomes truly available. The figure 1 illustrates the mechanism: a client calls a method `p.m()` on an asynchronous proxy (1). This last uses an abstraction to realize the actual asynchronous invocation (2) which may lead to the concurrent execution of the method `m()` (4'). The *future* returned by the abstraction (3) is then wrapped into a *transparent future* which is a subtype of the original result. Client can thus use the result, `r`, as usual thanks to polymorphism. When a method, say `foo()` is called on the result `r` (4), the *transparent future* uses the wrapped *future* to know the status of the asynchronous method invocation through the call `waitFotResult()`[2] (5). When the real result is available (5'), the original call `foo()` is finally invoked (6).

Note that the property about *type compatibility* may produce concurrency where it is not expressed explicitely: by the passing of a type-compatible asynchronous proxy to a library, the method invocation made by the latter on the former, while expressed synchronously, may execute concurrently. Hence, allowing the use of legacy classes in a concurrent context may ease the production of concurrent applications: this is the main goal of *full-transparency*.

[1] The abstraction and the asynchronous proxy may be the same object, but in this paper we distinguished the two, as the asynchronous proxy is the one which actually provides transparency.

[2] The `java.util.concurrent.Future.get()` method has the same functionality, but we keep `waitForResult()` in this article which is more explicit.

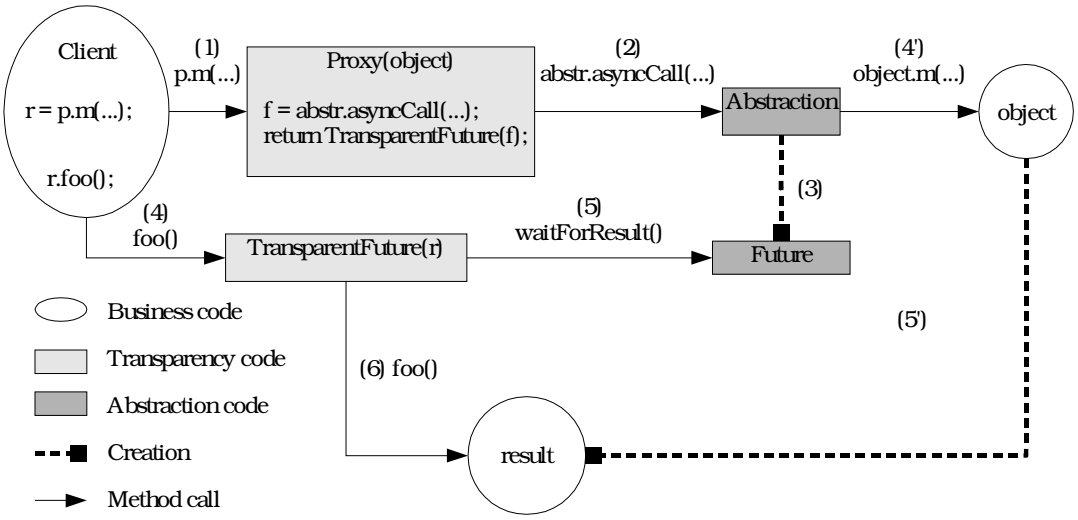


**Fig. 1.** Illustration of a *fully-transparent* asynchronous method invocation on an *asynchronous proxy*.

### 3.1 Solutions in Java

Two solutions may be provided to implement fully-transparent asynchronous method invocations in Java. They both use polymorphism in method invocation.

**Using inheritance: the ProActive approach.** The first solution, used in the ProActive framework, is based on inheritance. Basically, an asynchronous proxy is an instance of a class which extends the original object class. A transparent future is an instance of a class which extends the one returned by the original method. For instance, after an object, say `a`, instance of a class `A`, becomes active, client uses a *proxy* `p` which class `P` extends `A`. Each method of `A` is redefined in `P`. So, if `A` declares: `T m(...)`, then in the proxy class `P`, the method `m()` returns a transparent future which type is a subtype of `T`. This subclass implements the *wait-by-necessity* behavior: each call runs a test to know the availability of the result. If the test is true, the real method is invoked, else the client is blocked. When the result becomes available, every blocked threads are notified.

This solution contains many problems, at least in Java:

- `final` classes cannot be inherited preventing the use of both asynchronous proxies and transparent futures (on 3,950 classes of the JDK v1.4[3], only 57% are declared `public` on which 9% are in this case);
- `final` methods (5% of public methods of public classes) cannot be overridden preventing the implementation of both asynchronous proxy and transparent futures;
- methods returning a primitive type (27% of public methods of public classes) cannot be overridden to return a subtype used by the *wait-by-necessity* mechanism;
- clients accessing to public fields of a transparent future cannot be blocked waiting the availability of the result of the asynchronous method invocation.

The last problem may seem minor since, as a common good practice, public fields are usually also declared `static final`: accessing such fields is not a problem neither in a concurrency context nor in a distributed context. Anyway, if 99.9% of class fields in the JDK v1.4 are declared `final`, only 20% of instance fields are too: 8 non-`final` public instance fields are found for any 100 public classes found.

Since those problems are directly related to inheritance, another approach may avoid them.

[3] The Java example program, called `ClassPathAnalyser`, and released with the Mandala framework [18] has been used. Only classes with a full class name prefixed by `java` was considered.

**Using interfaces: the Mandala approach.** Java interfaces do not contain fields (or they are constant) and their methods are all declared (implicitly) `public`. Moreover, they cannot be declared `final`. Hence, the exclusive use of interfaces for full-transparency of asynchronous method invocation solves problems found in the inheritance solution.

Java provides *dynamic proxy* since the JDK v1.3 : the `java.lang.reflect.Proxy` is able to produce a class at runtime implementing a given set of interfaces. The business code of the proxy is an instance of a class which implements `java.lang.reflect.InvocationHandler`.

Using the dynamic proxy feature, it is possible to implement a fully-transparent asynchronous method invocation mechanism. First, an *asynchronous proxy* can be defined using the general code design of PROG 3.1.

```
1  public class AsynchronousProxy implements InvocationHandler {
2      // Creates concurrency
3      private Abstraction abstraction;
4
5      /**** InvocationHandler implementation ****/
6      public Object invoke(Object proxy, Method method, Object[] args)
7                   throws Throwable {
8          Class returnType = method.getReturnType();
9          Class[] resultInterfaces;
10         if (returnType.isInterface()) {
11             resultInterfaces = new Class[] {returnType};
12         }else{
13             resultInterfaces = returnType.getInterfaces();
14         }
15         Future future = abstraction.asyncCall(method, args);
16         return Proxy.newProxyInstance(resultInterfaces,
17                                       new FutureProxy(future));
18     }
19 }
```

PROG. 3.1: full-transparent implementation of an asynchronous proxy using dynamic proxies.

In this code, the concurrency is produced by a supposed `abstraction` able to invoke asynchronously a given method[4]. This invocation (supposed made by the `asyncCall()` method) must return a `Future` instance such as the one found in the `java.util.concurrent` of the new JDK v1.5. This object is then encapsulated in a `FutureProxy` instance class which is returned. This class is a subtype of the original result, thanks to the use of dynamic proxies another time. The business code of this proxy implements the *wait-by-necessity* mechanism with a code similar to PROG 3.2.

As seen, each method called on our *transparent future* waits for the actual return of the underlying asynchronous method call, and redirects the call to the business object.

The major drawback of the approach based on interfaces is the constraints that limit its use of application:

- the object used must be of a class which implements at least one interface;
- the method invoked asynchronously must be defined in an interface;
- the return type of the method must also be an interface.

Among the 57% declared `public` classes over the 3,950 found in the JDK v1.4[5], 20% are interfaces and 30% implement at least one interface. Only 5% of public methods are declared in interfaces. Few of them

[4] Reflection is used in this case.

[5] Only classes with a full class name prefixed by `java` were considered.

```
class FutureProxy implements InvocationHandler {
    final Future future;

    FutureProxy(final Future future) {
        this.future = future;
    }
    /**** InvocationHandler implementation ****/
    public Object invoke(Object proxy, Method method, Object[] args)
                  throws Throwable {
        // Object's method must not be redefined.
        if (method.getDeclaringClass().equals(Object.class)) {
            return method.invoke(this, args);
        }
        Object result = future.waitForResult();
        return method.invoke(result, args);
    }
}
```

PROG. 3.2: Implementation of the *wait-by-necessity* mechanism in the business code of a future dynamic proxy.

return a type which is either defined by an interface or a class which implements an interface: over 17,254 public methods of the JDK v1.4, only 653 (4%) conform to the previous criterion and are thus usable with a fully-transparent asynchronous proxy based on interfaces. It is then clear that this solution rarely allows the use of such proxies in applications not designed for it.

### 3.2 Inherent Problems

One of the goal of full-transparency, is to allow the use of legacy classes which were not designed in a concurrent context. Polymorphism is the core of the mechanism: the use of subtypes (either by inheritance or by interface) allows the passing of *asynchronous proxy*, and *transparent futures* to some methods which believe they are standard objects. This *may* naturally produce concurrency. This section shows that full-transparency has inherent problems which make it at best, difficult to use, at worst useless.

**Exception Handling.** When considering legacy code, the instructions given in PROG 3.3 are commonly found in a Java program. If the call `out.write(b)` is asynchronous (and of course fully-transparent), then

```
int b = ...; // byte to write
java.io.FileOutputStream out = null;
try{
    out = new java.io.FileOutputStream(...);
    out.write(b);
}catch(java.io.IOException ioe) {
    // handle any IO exception
}catch(java.lang.SecurityException se) {
    // handle security exception
}
```

PROG. 3.3: full-transparency and exceptions.

the caller thread continues its execution and may reach a point far beyond the `catch()` statements when the `write()` method actually ends.

In the case of checked exceptions, such as the `java.io.IOException`, one approach (the ProActive one) is to make these calls synchronous. This trivially prevents the problem to appear, but does not solve the case of unchecked exceptions such as the `java.lang.SecurityException`. A solution, is then to enforce a synchronization between the caller and the callee thread by using the future such as in the PROG 3.4:

```
1  java.util.List list = library.getAList();
2  ...
3  try{
4      // Asynchronous call
5      Object o = list.remove(0);
6      // *Must* wait the result (in case of a runtime exception)!
7      o.toString();
8  }catch(UnsupportedOperationException e) {
9      // This exception is a runtime exception
10     ...
11 }
```

PROG. 3.4: Explicit synchronisation.

This may be done by an automatic tool, or at least by a checker. But even in this case, the method `write()` of the first example is declared returning `void`. So there is no way to enforce clients to use a non-existent result!

This problem is still open. Whereas, some directions are work in progress in our team, we believe it is a major drawback that makes full-transparency very hard to use when exceptions are considered.

**Developer Consciousness.** As for us, the main problem of the full-transparent mechanism is related to the implicit concurrency it seems to produce. The degree of concurrency in an application written using the asynchronous method invocation paradigm can be increased by following the general guidelines:

**Definition 2 (Guidelines for efficicency)**
*A maximum concurrency degree is achieved using an asynchronous method invocation paradigm when:*

- *method calls are made as soon as possible;*
- *result recovery are used as late as possible.*

This may seem trivial but it is not the natural way applications are written in the sequential world. Consider the following code:

```
1  // Step 1
2  MyClass myObject = new MyClass();
3  // Step 2
4  MyInfos infos = myObject.myMethod(myParameters).getInfos();
5  // Step 3
6  doSomething();
```

This code scheme is very common in Java. If the `myObject` variable references a fully-transparent asynchronous proxy, the resulting code will not be more efficient. It will even be less efficient since the handling of concurrency has a cost. Hence, to gain in concurrency, the code must be rewritten:

```
1  // Step 1
2  MyClass myObject = new MyClass();
3  // Step 2
4  MyResult result = myObject.myMethod(myParameters);
```

```
5 // Step 3
6 doingSomething();
7 // Step 4
8 MyInfos infos = result.getInfos();
```

And this is clearly not a natural sequential code in Java.

Hence, full-transparency does not allow the becoming concurrent of sequential application almost automatically. Most of the code must be rewritten – at least reordered – to gain some efficiency. These modifications may be done by a tool (more or less automatic), but in this case, why focusing on full-transparency?

We strongly believe asynchronous method invocation should be as simple as its synchronous version. But it must be *explicit* in order to ensure the writing of efficient concurrent applications.

## 4 Proposition: Semi-transparency

Following the conclusion of the previous section, we call *semi-transparency*, the mechanism which masks almost every aspect of the concurrency involved by an asynchronous method invocation (abstraction, asynchronous semantic and policy) but the asynchronism itself. This mechanism defends explicit expression of concurrency as with the `java.lang.Thread` API in opposition to the implicit concurrency provided by a fully-transparent solution.

Hence we propose the following syntax – promoted in Mandala:

**Notation 1 (Semi-transparency syntax)**
*If a public method has the following signature:*

```
T m(A1 a1,..., An an) throws E1, E2, En
```

*then its* semi-transparent asynchronous *version has the signature:*

```
Future<T> #m(A1 a1, ..., An an, Meta meta)
```

*In particular, exceptions declared in* `m()` *are no more part of the signature of* `#m()`*. The object* `meta` *may contain some informations used by the underlying abstraction such as priority, before and after methods, security informations, etc. It must at least contain an* exception handler *which will be used by the underlying abstraction when an exception occurs.*

This syntax solves many problems:

- strong typing is ensured (thanks to generics);
- the developer knows the asynchronous nature of the invocation of `#m()`[6] thanks to its signature which differs from the original;
- exceptions are always handled by a *client specific* object[7] and thus can never be ignored; anyway, exceptions may be re-thrown on the client side when retrieving the result through the return future.

The last problem to solve is where these method will be found? Which class defines them?

[6] Even if semi-transparency is provided in Mandala [18], the `'#'` character is reserved in Java and cannot be the first of an identifier (field or method). This character is replaced by the prefix `rami_` where RAMI stands for Reflective Asynchronous Method Invocation.

[7] Exceptions are always handled though. But the default handler found in the `ThreadGroup` class is not a good solution. Consider remote asynchronous method invocation as a case study.

### 4.1 Asynchronous views

**Definition 3 (Asynchronous View)**
*The* asynchronous view *of a class* `C` *– noted* `view(C)` *– defines, for each public method* `m()` *in* `C`*, its semi-transparent asynchronous version* `#m()`*. If* `C` *is an interface, then for each method* `m()` declared *in* `C`*, its semi-transparent asynchronous version* `#m()` *is also declared in* `view(C)`*.*

*If* `B` *is a supertype of* `C`*, then* `view(B)` *is also a supertype of* `view(C)`*. Anyway,* `view(C)` ***is not*** *a subtype of* `C`*.*

*Furthermore, an instance of an asynchronous view is called a* semi-transparent asynchronous proxy*.*

As for the naming of asynchronous view, we propose a mirroring of the standard Java class naming: suppose a full class name is `p.s.C`, then using a prefix, `jaya`[8], the full asynchronous view name (a class) is: `jaya.p.s.C`. This enforce the developer to be conscious of its use of asynchronous views (since they are really distinct classes) and so, to follow the guidelines given in definition 2. Note that using a naming convention which is just based on the class name, such as `p.s.Async_C` or similar, prevents its use in the standard Java language: some packages may be *sealed* preventing the addition of new classes.

The generation of asynchronous views leads naturally to a hierarchy of types which is symmetric from the original. As an example, consider the asynchronous view generation of the standard class `java.io.FileWriter`. The figure 2 presents the UML class diagram[9]. The left part are the asynchronous views hierarchy which clearly mirrors the type hierarchy of their related class on the right. Hence, the `java.lang.Object` superclass, has its asynchronous view symmetric called `jaya.java.lang.Object`. This view plays an identical particular role: it is a supertype of any other asynchronous view.

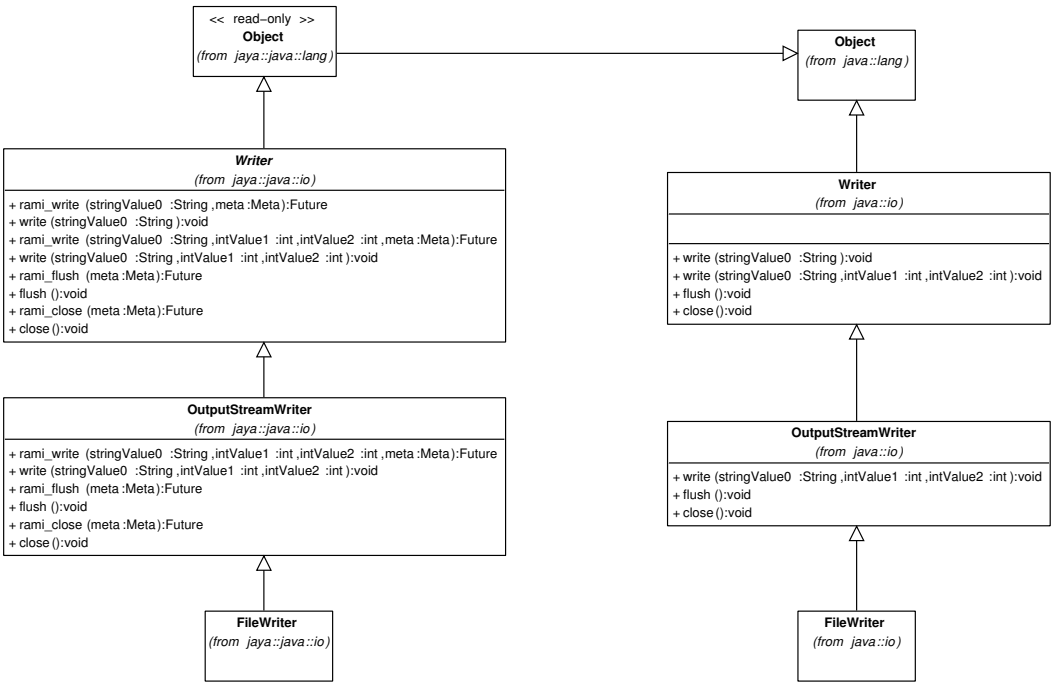


**Fig. 2.** Type hierarchy of the asynchronous view generation of the `java.io.FileWriter` class.

[8] Since the prefix is arbitrary, the name `jaya` was chosen for its meaning in Sanskrit ("Victory") where the Mandala name also comes from. Moreover, the asynchronous views generator of Mandala is called `jayac` which sounds like `javac`.

[9] This diagram has been produced from a real code generation thanks to the `jayac` asynchronous view generator of the Mandala framework. This is the reason why asynchronous methods are prefixed by '`rami_`' instead of the character '`#`'.

An asynchronous view also provides synchronous methods. Consider a method `m()` defined in a class `p.C`. Then, it also exists in `jaya.p.C`. But the semantic of methods `C.m()` and `jaya.p.C.m()` are really different: the latter **must be** a shortcut for `jaya.p.C.#m().waitForResult()`. The reason is that each call made on an asynchronous proxy – an instance of a view – must have reached the underlying abstraction used. Consider an abstraction which provides both concurrent and remote aspect as a case study[10]. Synchronous version of methods are clearly important to prevent developers from mixing standard, synchronous classes with asynchronous views. As a side effect, the class naming convention prevents even more this mixing. The fact that asynchronous views and standard classes short names are homonyms forbid developers to use both without care. For example, the following instructions are ambiguous and does not compile:

```
1 import java.io.*;
2 import jaya.java.io.*;
3 ...
4     Writer writer = new FileWriter("foo.txt");
5           writer = new FileWriter("bar.txt");
```

So the developer is enforced to use a full class name. If most of the code is synchronous, he would write:

```
1 import java.io.*; // Use shortcuts for synchronous class only
2 ...
3     Writer writer = new FileWriter("foo.txt");
4     jaya.java.io.Writer writerProxy =
5                 new jaya.java.io.FileWriter("bar.txt");
```

On the opposite, when most of the code is asynchronous, he would prefer the following form:

```
1 import jaya.java.io.*; // Use shortcuts for asynchronous view only
2 ...
3     java.io.Writer writer = new java.io.FileWriter("foo.txt");
4     Writer writerProxy = new FileWriter("bar.txt");
```

This leads to a naturally cleaner code where the developer knows it is using an asynchronous proxy. Hence, it enforces him to follow the guidelines 2, and allows him to enhance the overall concurrency degree of the whole application.

## 5 Conclusion

This article focuses on mechanisms used to hide the complexity of asynchronous method invocations. We have seen that this paradigm may use many underlying abstractions to handle concurrency: active objects, actors, separates, active containers, asynchronous references are several options among others. We show that *full-transparency* may be provided using two solutions: inheritance or interfaces. The former contains several problems the latter solves by imposing very high constraints. Finally, while the main advantage of full-transparency is its possible application with legacy code, it has two inherent problems that make its use very complex as far as exceptions is concerned or useless as far as efficiency is concerned. So we defend an explicit expression model which masks the most of the asynchronism mechanism, and abstractions in particular, but the asynchronism itself. This model is called *semi-transparency*. It provides a solution to the exception problem. It also helps the developer to focus on the concurrent aspect of its application. This enforce the following of guidelines given in definition 2 which seems necessary to gain the most of concurrency.

[10] As provided by the *stored object reference* [6], an extension of the *asynchronous reference* paradigm [19], that uses the *active container* concept [7] to provide the remote aspect.

Transparency, both *full* and *semi* is proposed in the Mandala framework [18] which helps the development of concurrent (and eventually distributed) Java applications. As such, the framework must be extended to use the new *java.util.concurrent* package. Furthermore, exceptions handling in the context of both full- and semi-transparent asynchronous method invocation must be further studied.

## References


1. TLP and the Return of KISS . Web page, January 2004. http://www.aceshardware.com/read.jsp?id=60000312.
2. AGHA, G. *Actors: A Model Of Concurrent Computation In Distributed Systems*. PhD thesis, University of Michigan, 1986.
3. AGHA, G., AND HEWITT, C. Concurrent programming using actors: Exploiting large-scale parallelism. In *Readings in Distributed Artificial Intelligence*, A. H. Bond and L. Gasser, Eds. Kaufmann, San Mateo, CA, 1988, pp. 398–407.
4. CAROMEL, D. Toward a method of object-oriented concurrent programming. *Communications of the ACM 36*, 9 (1993), 90–102.
5. CAROMEL, D., KLAUSER, W., AND VAYSSIÈRE, J. Towards seamless computing and metacomputing in Java. In *Concurrency: practice and experience* (Sept.-Nov. 1998), G. C. Fox, Ed., vol. 10, Wiley and Sons, Ltd., pp. 1043–1061.
6. CHAUMETTE, S., AND VIGNÉRAS, P. A framework for seamlesly making object oriented applications distributed. In Joubert et al. [9], pp. 305–312.
7. CHAUMETTE, S., AND VIGNÉRAS, P. Behavior model of mobile agent systems. In *FCS'05 - The 2005 International Conference on Foundations of Computer Science* (Las Vegas, USA, june, 27–30 2005). to appear.
8. GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN : 0-201-63361-2.
9. JOUBERT, G. R., NAGEL, W. E., PETERS, F. J., AND WALTER, W. V., Eds. *Parallel Computing: Software Technology, Algorithms, Architectures and Applications, PARCO 2003, Dresden, Germany* (2004), vol. 13 of *Advances in Parallel Computing*, Elsevier.
10. KURZYNIEC, D., WRZOSEK, T., SUNDERAM, V., AND SLOMIŃSKI, A. RMIX: A multiprotocol RMI framework for java. In *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS'03)* (Nice, France, Apr. 2003), IEEE Computer Society, pp. 140–146.
11. LAVENDER, R. G., AND SCHMIDT, D. C. Active object: an object behavioral pattern for concurrent programming. *Proc.Pattern Languages of Programs,* (1995).
12. LEA, D. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., 1999.
13. LYON, D. CentiJ: An RMI Code Generator. *Journal of Object Technology 1*, 5 (November-December 2002), 117–148. http://www.jot.fm/issues/issue_2002_11/article2.
14. MEYER, B. Systematic concurrent object-oriented programming. *Communications of the ACM (special issue, Concurrent Object-Oriented Programming, B. Meyer, editor) 36*, 9 (1993), 56–80.
15. NESTER, C., PILIPPSEN, M., AND HAUMACHER, B. A more efficient RMI for Java. In *Proceedings of Java Grande Conference* (San Francisco, California, june 1999), ACM, pp. 152–157.
16. SUN MICROSYSTEMS. *Java Remote Method Invocation Specification*, 1998. http://java.sun.com/products/jdk/1.1/docs/guide/rmi/.
17. THIRUVATHUKAL, G. K., THOMAS, L. S., AND KORCZYNSKI, A. T. Reflective remote method invocation. *Concurrency: Practice and Experience 10*, 11–13 (1998), 911–925.
18. VIGNÉRAS, P. Mandala. Web page, August 2004. http://mandala.sf.net/.
19. VIGNÉRAS, P. *Vers une programmation locale et distribuée unifiée au travers de l'utilisation de conteneurs actifs et de références asynchrones.*
    *Rapporteurs : Doug Lea, Françoise Baude, Michel Riveill.*
    *Jury : Françoise Baude, Serge Chaumette, Olivier Coulaud, Mohamed Mosbah, Alexis Moussine-Pouchkine, Michel Riveill.* PhD thesis, Université de Bordeaux 1, LaBRI, november, 8th 2004. http://mandala.sf.net/docs/thesis.pdf.
20. WALKER, E., FLOYD, R., AND NEVES, P. Asynchronous Remote Operation Execution In Distributed Systems. In *International Conference on Distributed Computing Systems* (Paris, France, May/June 1990), no. 10, pp. 253–259.