

Mandala/RAMI User's Guide – April 28, 2004



Pierre Vign ras
“eipi”
eipiequalsnoone@users.sf.net

April 28, 2004

Abstract

This document describes the Mandala/RAMI subpackage, a library for *Reflective Asynchronous Method Invocation*. It is entirely based on the *reflexion* mechanism which is provided in the Java™¹ language.

This document deals also with *exception handling* usually neglected in asynchronous method invocation ; with the *event driven* programming scheme provided ; with the *chained asynchronism* mechanism ; with the *asynchronism semantic* and with *transparency* where the complexity involved in asynchronous method invocation is entirely masked (*total transparency*) or partially masked *semi transparency* to the end user.

keywords: reflective asynchronous method invocation, asynchronism semantic, total-transparency, semi-transparency

¹Java and all Java-based marks are trademarks or registered trademarks of *Sun Microsystems, Inc.* in the United States and other countries. The author is independent of *Sun Microsystems, Inc.*

Contents

1	Introduction	4
2	Related Works	5
3	Reflection provides dynamism	6
4	Asynchronous method invocation	8
4.1	The message passing abstraction	8
4.2	Models	8
5	Asynchronous references	10
5.1	Futures: handling the result	11
5.1.1	InvocationInfo: getting information	13
5.1.2	InvocationObserver: polling	13
5.1.3	MethodResult: getting the result	14
5.1.4	InvocationEventsWaiter: waiting for events	14
5.1.5	FutureClient: the <i>all-in-one</i> interface	14
5.2	Exception handling	15
5.2.1	On result recovery	15
5.2.2	Earliest notification	15
5.3	Callback: <i>event driven</i> programming	16
5.3.1	Global Object Scope: the <i>default</i> one	16
5.3.2	Local Thread Scope: the <i>thread specific</i> one	17
5.3.3	Local Method Scope: the <i>method specific</i> one	17
5.3.4	CallbackManager: the handler of Callback	17
5.3.5	Illustration of <i>event driven</i> programming scheme	18
5.4	Chained asynchronism	18
5.4.1	Mechanism	18
5.4.2	AsynchronousReferencePair: pair of asynchronous references	20
5.4.3	List of asynchronous references	20

6	RAMI implementation description	22
6.1	An implementation of the AsynchronousReference interface : the AsynchronousReferenceImpl class	22
6.2	Asynchronism semantics	23
6.2.1	Concurrent semantics	23
6.3	Single threaded semantic	24
6.4	Customized asynchronous semantic	25
6.5	Performance consideration	25
6.5.1	Overhead of the reflexive asynchronism provided by RAMI .	25
6.5.2	Analysis on a real example	27
7	Transparency	30
7.1	Introduction	30
7.2	The <i>Strong typing</i> problem	30
7.3	Total transparency	31
7.3.1	Inheritance	31
7.3.2	Interfaces and the java.lang.reflect.Proxy class . .	31
7.3.3	Problems of <i>total transparency</i>	33
7.4	Semi transparency	34
7.4.1	Client's view of object	34
7.4.2	The JayaCompiler class	35
7.5	Summary	36
8	Conclusion and perspectives	37

List of Figures

5.1	RAMI asynchronous reference and Java synchronous reference. . . .	10
5.2	The <code>call()</code> asynchronous mechanism.	12
6.1	Time spent to compute 1000 decimals of Pi on a quadriprocessor node using RAMI.	28

Chapter 1

Introduction

Java [?] is no more a tool for applets programmer but is a language widely used in every domain of computer science : smartcards [?], personal digital assistant [?], workstations [?], enterprise server [?], supercomputers [?].

Java is an objects oriented language [?] and uses the model of method invocation – sometimes abstracted to message passing. Its development kit [?] contains a thread API (`java.lang.Thread`) and its virtual machine specification [?] defines threads behaviour¹.

We propose a new package to extend *dynamically* method invocation to asynchronous method invocation. Related works are described in the section 2. *Dynamism* is our main objective and means that objects have not to be designed specifically to be used in an asynchronous way ensuring legacy code and separation of concerns. This property is achieved by the use of the *reflection* provided in Java and will be discussed in the section 3. Different models used to provide asynchronous method invocation are discussed in the section 4. The package we propose called RAMI for *reflective asynchronous method invocation* is based on the concept of *asynchronous references* – inspired on standard synchronous Java reference – and is described in the section 5. Our implementation of this concept is then described and analysed in the section 6. Whereas reflection provides dynamism, it raises the problem of *weak type checking* where type checking is done at runtime instead of compile time which is a real drawback for programmers. Our solution to this problem is provided by the *transparency mechanism* described in the section 7. Conclusion and perspectives are given in the section 8.

¹Which is not followed by implementation of the Java Virtual Machine which uses native threads such as HotSpot™ [?].

Chapter 2

Related Works

Several works focuses on asynchronous method invocation but they are all dedicated to the *remote* case. For example, K.Falkner and al. [?] extends the language with the keyword `asynch` to declare methods of the remote object that are called in an asynchronous way. A re-implementation of the RMI *stubs* and *skeletons* and the use of *Futures* [?] allow developers to deal with asynchronous calls.

Research is also carried out around *dynamism*. HORB [?] for instance, is a Java ORB (Object Request Broker) which does not need a declarative interface to create remote objects. Any object can be compiled by a dedicated compiler to become remote. Asynchronous remote method invocation is based on a *method naming convention*. For example, a method `foo()` of a remote object must be renamed `foo_Asynch()` and clients must use `foo_Request()` and `foo_Receive()` to achieve the asynchronous call to `foo()`. Reflective Remote Method Invocation [?] focuses on the same concern using the same reflection technique, *i.e. dynamic remote method invocation*, and provide also asynchronism but is also dedicated to remote objects.

Chapter 3

Reflection provides dynamism

As defined in the API documentation of the `java.lang.reflect` package: "Reflection allows programmatic access to information about the fields, methods and constructors of loaded classes, and the use reflected fields, methods, and constructors to operate on their underlying counterparts on objects, within security restrictions."

This allows the programmer to access some language characteristics of an object at runtime. For example, given an instance of the `java.lang.String` class containing "foo", and an instance `a` of the class `A` defines below:

```
public class A {
    public StringBuffer foo(StringBuffer sb) {
        return sb.append(sb.reverse()); // do something very usefull !
    }
}
```

An invocation of the `foo` method of the object `a` can be done in two different way:

```
// Natural way
StringBuffer sb = a.foo(new StringBuffer("Dummy"));

// Reflective way
StringBuffer sb = (StringBuffer)
    Class.forName("A") // Returns a Class object reflecting 'A'
        .getMethod("foo", // Returns a Method object reflecting 'foo'
            new Class[]{StringBuffer.class})
        .invoke(a, // Invokes the method 'foo'
            new Object[]{new StringBuffer("Dummy")});
```

As shown above, reflective method invocation is not really adapted to the development of applications. Many problems arise with reflection. The first and probably the most important of all is the *lack of type checking*. In the 'natural way', the compiler tells if the parameter given to the method does conform or not to the one specified in the signature. The same check is made on the result returned by the method. The compiler do verify also that the variable refers to an instance of the specified class (or one of its subclass) so the method called 'foo' can safely be invoked. All of this checks are not done in reflective method invocation. Moreover, reflective method invocation is *less efficient* than natural method invocation. Some works has to be done to discover the class of the object, discover the name of one of its method, and invoking it. All of this is done at runtime while it is done at compile time in the natural way.

So why focusing on the reflective approach ? Basically, the best advantage of reflection is also its best drawback : reflection provides *dynamic* programming. Although the discovering an object at runtime is mainly used in object inspectors, debuggers or class browser, it can actually provide extensibility and pluggability[?]. The *dynamic* feature provided by reflection is the focus of RAMI *i.e.* any method of any objects must be asynchronously invocable even if the object was not designed in this way. This ensure *separation of concerns*.

Chapter 4

Asynchronous method invocation

4.1 The message passing abstraction

Object oriented programming is sometime abstracted to message passing programming: invoking `a.foo(t)` may be abstract to "sending the message `foo(t)` to the object `a`". The returned result is the reply of the message. If we extend the abstraction, then we may consider two phases:

1. sending the message,
2. getting the reply.

In method invocation, when a message has been sent, the caller must wait the answer before continuing. Sometime, it is preferable to send a message, to do something during the time the object constructs its reply (*i.e.* during the execution of the specified method), and to get the reply a bit later¹. This leads to asynchronous message passing *i.e.* asynchronous method invocation.

4.2 Models

Method invocation is not very adapted to asynchronous communication since a method call often returns a result. The caller thread has to wait for the availability of this result. Therefore, many frameworks modify the method invocation paradigm to suite asynchronism. For example, some provide asynchronism with a *send/receive* model [?], others propose a *publish/subscribe* model [?]. Although these models are more adapted to asynchronous communication, any existing code has to be rewritten - when possible. For example, if a synchronous communication is changed for efficiency reasons to an asynchronous one, the programmer has to introduce the *send/receive*

¹It must be noticed that this description may be enlarged to the remote case where the sender and the receiver of the message do not reside on the same host.

instructions on both client and server sides². Such a change may have a really bad impact on the design of the application and is prone to the introduction of new bugs.

The use of an *inner anonymous inlined thread* such as

```
new Thread() {
    public void run() {
        // invoke the desired
        // method asynchronously
        StringBuffer sb = a.foo(new StringBuffer("Dummy"));
    }
}.start();
```

is not a good design solution: a thread object must be instantiated and result polling must be implemented. Moreover, using an explicit thread to deal with asynchronous method invocation breaks the model of the message passing paradigm. It may be preferable to deal with synchronous/asynchronous method invocation (message passing) instead of using threads as described by [?].

Hence, several projects focus on an asynchronous method invocation schema in a more or less transparent way, using so called *future objects*. Most of them are static and dedicated to asynchronous *remote* method invocation. For example, the asynchronous remote method invocation mechanism described by K.Falkner and Al. [?] needs a compilation phase to generate stubs dedicated to the asynchronous paradigm.

This document describes the architecture of the RAMI package which provides *reflective asynchronous method invocation*.

²Since objects use message passing for their communication, the name *client* is used for the object that invokes the method of another object named the *server*. This definition extends naturally to remote objects.

Chapter 5

Asynchronous references

Whereas Java defines reflection of references (`java.lang.ref`) for garbage collecting purposes, RAMI defines the notion of *asynchronous reference* for concurrency purposes. An asynchronous reference is a reference on another object which provides asynchronous method invocation. If O is an object, $AR(O)$ is an asynchronous reference on O (figure 5.1). As with standard, synchronous reference, there is only one asynchronous reference on any object in a given Java Virtual Machine. RAMI contains the `AsynchronousReference` interface which declares the following method:

```
FutureClient call(Method method, Object[] args);
```

For any Java objects, invoking one of its method asynchronously requires three steps:

1. getting an asynchronous reference on the object;
2. getting the reflection of the method to invoke;
3. and performing the asynchronous method invocation.

Hence, the asynchronous invocation of

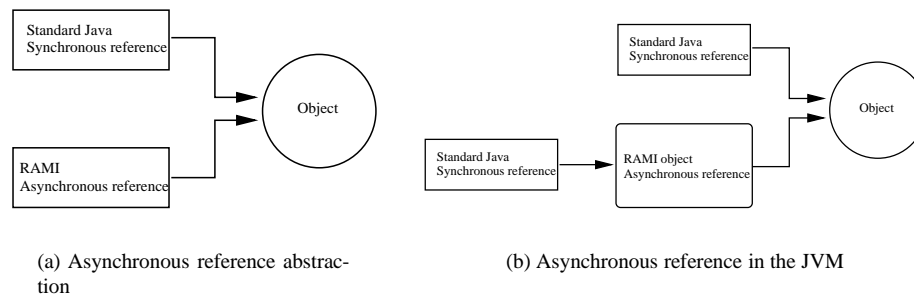


Figure 5.1: RAMI asynchronous reference and Java synchronous reference.

```
a.foo(new StringBuffer("Dummy"));
```

is done by the following code:

```
// Gets an asynchronous reference on 'a'
AsynchronousReference asynchronousReference = // Depends on implementation !
// Gets the reflection of the "foo()" method
Method fooMethod = a.getClass().getMethod("foo",
                                           new Class[]{StringBuffer.class});

// perform the asynchronous invocation of 'a.foo(new StringBuffer("Dummy"))';
asynchronousReference.call(fooMethod,
                          new Object[]{new StringBuffer("Dummy")});
```

As the reader may noticed, using reflection leads to a really unnatural and nearly impractical syntax. Solutions to this problem will be discussed in section 7. Moreover, the above code needs more explanations:

- the reader may have noticed that it will not compile since it doesn't catch the `NoSuchMethodException` the `getMethod()` method may throw, hence a `try/catch` block must be added for this code to compile. This is the main problem of the reflection: errors which are usually checked at compile time are discovered at runtime. Developers are encouraged in this case to declare their `fooMethod` in a `static final` field, and initialized in a `static` block. Hence, performance penalty of `try/catch` block is minimized and errors are discovered when the class is loaded.
- Since RAMI is Java-interface based¹, getting an `AsynchronousReference` variable depends on the implementation of the interface. Hence, it is not shown here.
- The method `toString()` returns a result which is not used in the preceding code. The next section describes mechanism RAMI provides to handle the result of an asynchronous method invocation.

5.1 Futures: handling the result

The last line of the preceding code does not use the result returned by the method `foo()`. Moreover, it does not illustrate the use of the asynchronism provided by the `call()` method of the asynchronous reference. Hence, consider the following line of code:

```
// Gets an asynchronous reference on 'a'
AsynchronousReference asynchronousReference = // Depends on implementation !
// Gets the reflection of the "foo()" method
Method fooMethod = a.getClass().getMethod("foo",
                                           new Class[]{StringBuffer.class});

// perform the asynchronous invocation of 'a.foo(new StringBuffer("Dummy"))';
asynchronousReference.call(fooMethod,
                          new Object[]{new StringBuffer("Dummy")});

doSomething();
```

¹RAMI also provides different implementations of every interfaces it defines. Some implementations of the `AsynchronousReference` interface will be discussed in the performance section 6.5

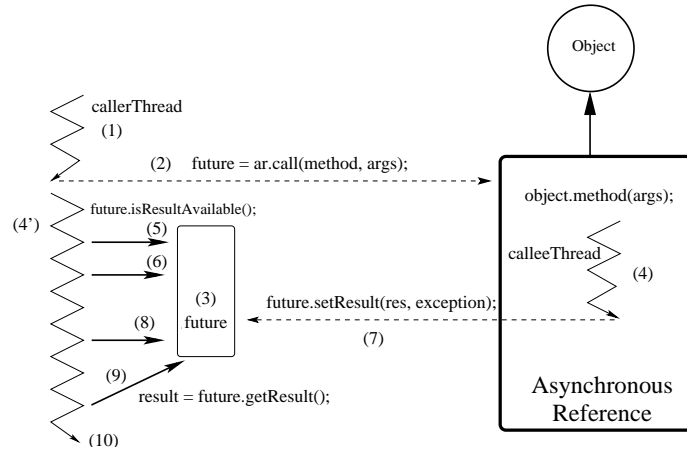


Figure 5.2: The `call()` asynchronous mechanism.

Here, the method `doSomething()` is executed in concurrency² with the `foo()` method. This is why asynchronous method invocation is useful for: the invocation of a method - the sending of a message - does not require waiting for the reply: getting the reply is just done when needed. This requires some practice to get used to: invoking methods at the earliest and getting their result at the latest.

Then, how the result of an asynchronous method invocation can be get? The idea comes with the notion of *future* [?]. When a method is invoked asynchronously thanks to `call()`, a future object is returned that handles everything related to asynchronism. The Figure 5.2 illustrates the `AsynchronousReference`'s `call()` mechanism and one use of the `FutureClient` instance returned. (1) A thread (`callerThread`) is running, preparing a reflective asynchronous method invocation. It does the `call()` invocation (2) which returns a `FutureClient` object (`future`) (3). The asynchronous reference implementation performs the actual asynchronous method invocation in the `calleeThread` (4) the other client thread continues its execution (4'). Periodically, the `callerThread` may test the availability of the result (5 and 6). When the called method terminates, the asynchronous reference implementation updates the `future` object (7). The next run by the `callerThread` is a positive availability test (8). It then gets the returned result (9) and continues its execution (10).

The `FutureClient` object is a future object which provides *result polling*, *result blocking* and *callee thread control*.

Methods of a future object can be *unrelated* or *related* to the result returned by the asynchronous method invocation. *Unrelated* methods can be used as any usual methods since their invocation does not depend on the asynchronous method invocation such as `toString()`, `equals()` or `hashCode()`. *Related* methods can be either *safe* or

²In parallel if the computer contains more than one CPU. To be really rigorous, it depends on the implementation: an implementor may provide a synchronous implementation of the `AsynchronousReference` interface! RAMI provides such an implementation used for testing (see 6.2)

unsafe regarding the validity of their returned value which depends on the *availability* of the result. The result returned by an *unsafe related* method is undefined, hence unusable, if the result of the asynchronous method invocation is not available. On the opposite, the result returned by a *safe related* method is always valid. *Unsafe related* methods are provided to allow efficient (non-blocking) programming scheme.

Unrelated methods will not be shown in this document, but the RAMI API specifies which methods are *unrelated* and which are *related*.

In the following, each method that returns an information on the asynchronous method invocation is specified *safe* or *unsafe*. If a method is specified *unsafe*, you must wait the availability of the result before considering the value returned by the method as valid.

5.1.1 InvocationInfo : getting information

Basic informations on an asynchronous method invocation such as "which method was invoked ?" or "which arguments were specified ?", are provided by the `InvocationInfo` interface. Every method of this interface is *safe related* meaning the result they return is always valid.

This interface provides methods to get informations that are available as soon as the asynchronous method invocation is made (when the `call()` is invoked). Thus informations are *static* regarding the life of the asynchronous method invocation: they do not change over time. Moreover, each methods this interface declares is *safe related*.

5.1.2 InvocationObserver : polling

When an asynchronous method invocation has been made, some information may be usefull to gather. The `InvocationObserver` interface provides methods for this purpose. For example, a method is provided to know if the result is available. The "result" is either the returned value of the method or the exception it throws. It is available if the method invocation has *terminated*. Hence, a method invocation is considered *terminated* when either the method as returned normally (with a `return` statement or when the last statement of a `void` method has been reached) or when the method throws an exception (either a checked exception, a runtime exception or an error). Polling the availability of the result of an asynchronous method invocation is provided by the method `isResultAvailable()`.

During an asynchronous method invocation, it may be usefull to have a reference on the thread which is really executing the code of the method you've specified - the *callee* thread. This thread may not be available if the invocation has not yet started. Hence, polling its availability is provided by the method `isCalleeAvailable()`. Once this method returns `true`, the `getCalleeThread()` can be used to get a reference on the callee thread making possible its interruption thanks to the method `interrupt()` for example. Developpers must be aware that the thread may have died if the method is already terminated.

This interface provides methods to get informations that are available *during* the asynchronous method invocation. Thus informations are *dynamic* regarding the life of the asynchronous method invocation: they change over time. Moreover, each methods

this interface declares is *safe related* except for the method `getCalleeThread()` which is *unsafe related* and returns an undefined value while `isCalleeAvailable()` returns `false`.

5.1.3 MethodResult : getting the result

When the execution of an asynchronous method invocation is considered *terminated*, the result is available and some informations such as "did an exception occur during the invocation of the method ?" or "What is the result of the asynchronous method invocation ?" may be get. This is the aim of the `MethodResult` interface. It provides informations on a *terminated* asynchronous method invocation. This is the reason why each of its method is *unsafe related* meaning they return an undefined value if the result is not yet available *i.e.* if the method is not considered *terminated* yet.

For example, the result of an asynchronous method invocation is provided by `getReturnedResult()`. Developpers must be aware that this method returns an `Object` and may throw the exception thrown by the invocation of the method. If the programmer is sure that no exception has been thrown (`exceptionOccured()` must return `false`), you can use `getReturnedResultTrusted()` instead which eliminates the need of a `try/catch` block and its performance penalty. Otherwise, `getException()` returns the exception that have been thrown during the asynchronous method invocation.

This interface provides methods to get informations that are available *after* the asynchronous method invocation *i.e.* when it is considered as terminated. Thus informations are *static* regarding the life of the asynchronous method invocation: they do not change over time. Moreover, each methods this interface declares is *unsafe related*: they return an undefined value while `isResultAvailable()` returns `false`.

5.1.4 InvocationEventsWaiter : waiting for events

Developpers may be interested in events occurring during an asynchronous method invocation. For example, the availability of the result, the availability of the callee are such events. Instead of polling for such events, programmers may wait until this event occurs. The `InvocationEventsWaiter` provides methods to wait fo such events. For example, waiting for the availability of the result of an asynchronous method invocation and getting it in the same time is provided by the `waitForResult()` method. Again, the result may be an exception; in such a case, this exception is rethrown by this method. Symetrically, waiting until the callee thread has been set by the underlying implementation is provided by the `waitUntilCalleeAvailable()`.

Each method this interface declares is *safe related*.

5.1.5 FutureClient : the *all-in-one* interface

Instead of dealing with many interfaces, RAMI provides for ease of use, the `FutureClient` interface which extends the four interfaces seen above : `InvocationInfo`, `InvocationObserver`, `MethodResult` and `InvocationEventsWaiter`. This interface is the one returned by the method `call()` of the `AsynchronousReference` interface which defines the behavior of any asynchronous reference.

5.2 Exception handling

One of the bad thing in asynchronous method invocation is how to handle exceptions properly ? When the method is really invoked, the caller thread is probably not in the `try/catch` statement corresponding to the one that must be used when a usual method which declares exception is invoked. Consider the following code :

```
public class A {
    public StringBuffer foo(StringBuffer sb) {
        return sb.append(sb.reverse()); // do something very usefull !
    }

    public static void main(String args[]) {
        A a = new A();

        Method fooMethod = A.class.getMethod("foo", new Class[0]);
        AsynchronousReference reference = // depends on implementation

        FutureClient future = reference.call(fooMethod,
                                           new Object[]{null});

        doSomethingElse();
    }
}
```

The invocation of `foo(null)` throws a `NullPointerException` as expected. Where should the `try/catch` statements be placed to ensure correct handling of the exception ?

RAMI provides two techniques to deal with exceptions in asynchronous method invocation.

5.2.1 On result recovery

The first and easiest technique is to deal with exceptions when recovering the result. This means that until the caller checks exceptions, it is not notified that the asynchronous method invocation has thrown an exception. The `MethodResult` interface provides the method `exceptionOccured()` to know if an exception has been thrown and the `getException()` to get the thrown exception. The method `waitForResult()` of the `InvocationEventsWaiter` may throw the exception that has been thrown during the asynchronous method invocation. Since `waitForResult()` declares `Throwable` to be thrown, it must be enclosed by a `try/catch` statement ensuring the proper handling of the thrown exception.

The program 5.2.1 is an example of *polling* using the `InvocationObserver` interface and of handling result and exceptions "by hand" while the program 5.2.2 shows the use of the `InvocationEventsWaiter` interface. The use of the *callee* thread is shown by the program 5.2.3.

5.2.2 Earliest notification

The second technique to deal with exception in asynchronous method invocation is to use an *event driven* programming scheme. Hence, the next section will describes this scheme, and the *earliest notification* technique used to deal with exception in asynchronous method invocation will be described in section 5.3.5.

Program 5.2.1 Using the `InvocationObserver` interface.

```
1 // Gets the reflection of the "foo()" method
  Method fooMethod =
    A.class.getMethod("foo",
                      new Class[]{StringBuffer.class});
5
  FutureClient future =
    asynchronousReference.call(fooMethod,
                              new Object[]{new StringBuffer("Hello")});

10 // Polling the result
   while(!future.isResultAvailable()) {
     // This method is running concurrently to "foo"
     doSomething();
   }
15 // The result is available : the method invocation is considered terminated

   if (future.exceptionOccured()) {
     // An exception occurred
     Throwable t = future.getException();
20     handleException(t);
   }else{
     // No exception occurred
     StringBuffer result = future.getReturnedResultTrusted();
25     handleResult(result);
   }
```

5.3 Callback : *event driven* programming

Whereas `future` provides *result polling* and *result waiting*, it may be more convenient to use *event driven* programming scheme. This means invoking a method asynchronously and doing something else without referring in the following code in the previously made asynchronous method invocation. Another object will be notified of the termination of the asynchronous method invocation and will use the result (or the exception thrown). The `Callback` interface provides the *event driven* programming scheme. It defines the method `done()` method which is invoked once an asynchronous method is considered terminated. Each reflective asynchronous method invocation has an associated `Callback` accessible with the method `Callback()` of the `InvocationInfo` interface.

You have three different ways to specify the `Callback` to use once your asynchronous method invocation terminates.

5.3.1 Global Object Scope : the *default* one

Each `AsynchronousReference` contains a `Callback` instance with a *global object scope*. This means that the `Callback` instance is related to its `AsynchronousReference` instance. If a thread modify the global object scope `Callback` instance, then any other thread will see the same `Callback` instance. This global object scope instance is the *default* one meaning that in absence of any other type of `Callback` scope (see below), it is used when an asynchronous method invocation terminates.

Program 5.2.2 Using the `InvocationEventsWaiter` interface.

```
1 // Gets the reflection of the "foo" method
  Method fooMethod =
    MyObject.class.getMethod("foo",
                             new Class[]{StringBuffer.class});
5
  FutureClient future =
    asynchronousReference.call(fooMethod,
                             new Object[]{new StringBuffer("Hello")});

10 // This method is running concurrently to "foo"
    doSomething();

    try{
      StringBuffer result = future.waitForResult();
15   handleResult(result);
    }catch(Throwable t) {
      handleException(t);
    }
```

Program 5.2.3 using the *callee* thread.

5.3.2 Local Thread Scope : the *thread specific* one

Each `AsynchronousReference` contains many `Callback` instances with a *local thread scope*. This means that each `Callback` instance is *not only related* to its asynchronous reference instance but also to the thread which makes asynchronous method invocation (via the `call()` method). Hence, if a thread modify its local thread scope `Callback` instance, then other threads are not affected by this modification and will continue to see their own local thread scope `Callback` instance if they have one or the global object scope one otherwise. When a thread has an associated local thread scope `Callback` instance, it is used instead of the global object one.

5.3.3 Local Method Scope : the *method specific* one

You may specify a `Callback` to use instead of the global object scope default or the local thread scope ones. Hence, when doing an asynchronous method invocation, you use the overloaded `call()` method which takes a supplementary parameter : a `Callback` which is used instead of the default and the thread specific one.

5.3.4 `CallbackManager` : the handler of `Callback`

Handling global object scope and local thread scope requires pair of `Get/Set` methods - accessors which are defined in the `CallbackManager` interface. Hence, the `AsynchronousReference` interface contains the `getCallbackManager()` which returns the instance used to manage `Callback`.

5.3.5 Illustration of *event driven* programming scheme

This section presents three implementations of the `Callback` interface to illustrate the benefits of the *event driven* programming scheme.

CallbackListeners: separate normal from abnormal termination

This implementation separates the real result of an asynchronous method invocation and the exception it may have thrown. It takes `Callback` has events listeners and when the result arrives, dispatches it to exception event listeners if an exception occurred or to result listeners otherwise.

InterrupterCallback: the interrupter one

As describes in the section 5.2.2, waiting for the caller to check for exception is not always acceptable. It may be preferable to be notified as soon as possible to prevent a thread for computing something unused if the asynchronous method invocation as thrown an exception. Hence, as soon as the asynchronous method invocation is considered terminated, a notification event must be handled. The `InterrupterCallback` implementation of the `Callback` interface interrupts the caller thread when the method is terminated. So, the caller thread may test its *interrupt* flag providing a mechanism called the *earliest notification*. The program 5.3.1 shows such a mechanism.

5.4 Chained asynchronism

What is the benefits and the signification of having an asynchronous reference on another asynchronous reference on an object ? Even if this idea seems really strange, it may be considered that the RAMI package can be used by framework designed to distributed computing as the JACOb framework. In such a situation, where objects are remotely accessible, an asynchronous method invocation may be implemented in two distincts way :

- the asynchronism may be implemented by the proxy of the remote objects ;
- the asynchronism may be implemented by the remote objects themselves.

In this last situation, invoking a remote method is *semi asynchronous* since the server side of the remote object must be contacted first - which is a synchronous operation. Then asynchronism may be added on the proxy of the remote object to provide a *client-side* asynchronism leading to *full-asynchronism*. Moreover, chained asynchronism provides a way to add services to some methods of an object. This is why having chained asynchronous reference is considered.

5.4.1 Mechanism

In a chain of asynchronous references, the following notation is adopted:

$$(AR(AR(AR(O))))$$

Program 5.3.1 The *earliest notification* mechanism.

```
1 // Gets the reflection of the "foo" method
  Method fooMethod =
    MyObject.class.getMethod("foo",
                             new Class[]{StringBuffer.class});
5
  FutureClient future =
    asynchronousReference.call(fooMethod,
                              new Object[]{new StringBuffer("Coucou")},
                              new InterrupterTerminatedCallHandler());
10
  // Polling with exception handling
  while(!future.isResultAvailable()){
    doSomeLittleWork();
    if (Thread.isInterrupted() &&
15      result.isResultAvailable()){
      // The interrupter has
      // interrupted us. The remote
      // method invocation has thrown
      // an exception.
      break;
20    }else{
      // Maybe someone else
      // has interrupted us ?
    }
25    endWork();
  }
  try{
    // Get the result. If an exception
    // has been thrown by the asynchronous method
    // invocation, it is thrown here.
30    MyResultObject myResult =
      (MyResultObject)future.getReturnedResult();
  }catch(Throwable t){
    // The asynchronous method invocation has thrown an exception,
    // handles it.
35    handleException(t);
  }
```

which means the last (left-sided) asynchronous reference is on another asynchronous reference on another asynchronous reference on the object O. Thus, invoking the method foo of the object O needs a chain of call () invocations such as :

```
MyClass object = new MyClass();

// Gets an AsynchronousReference on an AsynchronousReference on an
// AsynchronousReference on MyClass
AsynchronousReference reference = //depends on implementation

// Gets the reflection of the "foo()" method
Method fooMethod =
  MyClass.class.getMethod("foo",
                           new Class[]{StringBuffer.class});

// Gets the reflection of the "call" method
Method callMethod =
  AsynchronousReference.class.getMethod("call",
                                         new Class[]{Method.class,
                                                         Object[].class});

// Parameter of the the "foo" method
Object[] fooParam = new Object[] {new StringBuffer("Dummy")};
```

```
// Parameter of the last call
Object[] lastCallParam = new Object[] {fooMethod, fooParam};

// Parameter of the mid call
Object[] midCallParam = new Object[] {callMethod, lastCallParam};

FutureClient future = asynchronousReference.call(callMethod, midCallParam);
```

This is practically unusable ! So RAMI provides an elegant way to solve this problem.

5.4.2 AsynchronousReferencePair : pair of asynchronous references

The `AsynchronousReferencePair` takes two asynchronous references called the *head* and the *tail* where the head is referencing the tail and makes a new `AsynchronousReference` which simulates an asynchronous reference on the object referenced by the tail. So, if $AR(AR(O))$ is a chain of asynchronous reference on the object O , you can make an `AsynchronousReferencePair` referencing O which is also a standard `AsynchronousReference` (it implements the same interface) : $ARP(O)$. Then invoking a method `foo()` on O do not require a chained invocation of the `call()` method but only one - as with any `AsynchronousReference`. The `AsynchronousReferencePair` implementation must provides the same semantic on the `call()` invocation as the `AsynchronousReference`.

5.4.3 List of asynchronous references

Now that we have pairs of asynchronous references that can be considered as standard `AsynchronousReference`, making list of `AsynchronousReference` is fairly simple : $AR(AR(AR(O)))$ can be arranged into $AR(ARP(O))$ and since an `AsynchronousReferencePair` can be considered as an `AsynchronousReference` (*i.e.* $ARP \equiv AR$), we have $AR(AR(O))$ which can be transformed into $ARP(O)$ which gives $AR(O)$. Hence, writing it shorter:

$$AR(AR(AR(O))) \Rightarrow AR(ARP(O)) \equiv AR(AR(O)) \Rightarrow ARP(O) \equiv AR(O)$$

So doing an asynchronous method invocation on the object O do not depends on the number of asynchronous references chained to it. The last example which is really unusable can be written - thanks to `AsynchronousReferencePair` :

```
MyObject myObject = new MyObject();

// Constructs an AsynchronousReference with the help of
// AsynchronousReferencePair wich constructs a chained of AsynchronousReference
// on MyObject
AsynchronousReference reference = //depends on implementation

// Gets the reflection of the "foo" method
Method fooMethod =
    MyObject.class.getMethod("foo",
        new Class[]{StringBuffer.class});

// Usual asynchronous method invocation while it does a chained invocations of
```

```
// call() methods.  
FutureClient future =  
    asynchronousReference.call(fooMethod,  
        new Object[] {new StringBuffer("Blah")});
```

Chapter 6

RAMI implementation description

This section focus on the RAMI own implementation of its specification.

6.1 An implementation of the `AsynchronousReference` interface : the `AsynchronousReferenceImpl` class

This class implements the `AsynchronousReference` interface in a highly configurable way :

- logging is provided by a `Syslog` class instance¹ which provides debugging facilities such as thread trace;
- the class implements a variant of the *singleton* design pattern [?]: only one instance of this class referencing a given object can exists in any Java Virtual Machine²;
- this class uses the *factory* design pattern for futures creation (class `FutureFactory`) enabling their customization (remote futures for example);
- this class uses the *strategy* design pattern to allow the customization of the asynchronism (class `MethodInvoker`); three semantics are available as described in 6.2.

¹The next release will use the new `java.util.logging` package of the JDK 1.4.

²This property is also ensured on deserialization of an instance of this class.

6.2 Asynchronism semantics

Synchronous semantic

This semantic ensure that a method invocation on an asynchronous reference (`AsynchronousReference.call()`) do not return until the method has been terminated. This semantic is in opposition with the specification of the `AsynchronousReference` but the implementation provided by the `SynchronousMethodInvoker` class is used essentially for testing and debugging purpose: a programmer can develop with the synchronous semantic to find non asynchronous related bugs before switching into an asynchronous semantic one.

6.2.1 Concurrent semantics

This semantic ensure that sequential asynchronous methods invocation (sequence of `AsynchronousReference.call()` invocations) result in concurrent execution of those methods. Two classes are provided that implement this semantic.

Threaded concurrency semantic

Each asynchronous method invocation creates a dedicated thread for the execution of the specified method. While this implementation (`ThreadedMethodInvoker`) may have a little overhead it does not have the deadlock problem of the thread pooled concurrency semantic implementation (see below).

Threadpooled concurrency semantic

This implementation (`ThreadPooledMethodInvoker`) uses a highly customizable `ThreadPool` to prevent threads creation overhead. This threads pool has some specific property:

- the class `ThreadPool` extends the class `java.lang.ThreadGroup` ensuring a “grouping” semantic of the pre-allocated threads in the pool (global interruption for example);
- priority of the daemon status of the pre-allocated threads can be specified;
- the threads pool has a lower and a upper bounds of pre-allocated threads that can be fixed.

The behaviour of the threadpooled concurrency mechanism is as follow:

When an asynchronous method invocation is made, the method is enqueued, while each pre-allocated thread of the thread pool try to dequeue a method to execute it. If the number of methods enqueued is greater than the number of pre-allocated threads then new threads are created so their number will not exceed the upper bound. Then, if some pre-allocated threads are idle (there are no more methods in the queue to execute) and if their number is greater than the lower bound, some pre-allocated threads die so their number is not smaller than the lower bound. To prevent threads allocation and

deallocation *yoyo effect*, deallocation of threads is an event that occurs less frequently (by a customizable factor) than allocation. Hence, setting the lower bound to 0 and the upper bound to an infinite value (`java.lang.Integer.MAX_VALUE`) correspond to the threaded concurrency semantic.

The major drawbacks of threads pools is that they are subject to deadlock: consider a generic threads pool (like the class `ThreadPool` described above) with an `execute()` method which enqueue *tasks* (Such as method invocation). Suppose a is enqueueing lots of tasks, each of them also enqueueing other tasks in the same threads pool and waiting on their termination. The number of threads will grow up to the upper bound limit of pre-allocated threads in the pool (either the user fixed limit, the system fixed limit, or the limit imposed by the available memory in the system). Every thread is waiting the termination of a task still not dequeued since there is no more idle pre-allocated threads that can execute it. This situation leads to a deadlock!

Hence, with a threads pool, developers should never enqueue tasks that enqueue task on the same threads pool and wait on their termination. RAMI consider this situation as rare (or bad design of the application which is using the threads pool) and choose the threadpooled concurrency semantic as the default concurrency semantic implementation when creating a new `AsynchronousReferenceImpl` since it is the best choice for performance.

6.3 Single threaded semantic

The concurrency semantic ensure that methods runs concurrently but this behaviour may not be acceptable when asynchronous method invocation is made on *non-threadsafe* instances. Hence, the single threaded semantic ensure that only one thread runs the methods of the object the asynchronous reference is referencing. Currently two implementation is provided:

Fifo single threaded semantic

This implementation (`FifoMethodInvoker`) preserve the order of sequential invocations of the method `AsynchronousReferenceImpl.call()`. The first invocation will be executed by the attached thread firstly. This semantic is rather similar to the one provided in ProActive [?] and has the same interesting properties. It must be pointed out that this semantic can lead to deadlock in some situations such as the following:

A instance `a` of a class `A` contains a method `ResultF f()` which invokes asynchronously the method `ResultG g()` of an instance `b` of a class `B` and waits for its termination. Consider also that the method `ResultG g()` invokes asynchronously the method `f()` of the object `a` (indirect recursion) and also waits for its termination. If the asynchronism semantic is the *fifo single threaded semantic*, then this leads to a deadlock since the invocation of `f()` in the method `b.g()` enqueue the request (the method `f()` to execute) in the object `a` which has only one thread currently waiting for the termination of `b.g()`. Hence, this semantic must be use with care.

Random single threaded semantic

This implementation (`RandomMethodInvoker`) randomly execute a method within the list of previous asynchronous method invocations (`AsynchronousReferenceImpl.call()`). This semantic may leads to deadlock as in the fifo single threaded semantic.

6.4 Customized asynchronous semantic

The semantic of the asynchronism provided by the method `AsynchronousReferenceImpl.call()` is defined by an instance of a class which must implements the interface `MethodInvoker`. Hence, it is possible to customize the semantic of the asynchronous method invocations on an object by implementing this interface. This feature allows the developer to specialize its asynchronous method invocation to ensure some specific properties or to enhanced the performance.

6.5 Performance consideration

Analysing performance of an asynchronous method invocation is not easy. Synchronous and asynchronous method invocation are in opposition: they do the same thing (invoking a method of an object) but in rather distinct ways. Hence comparing them is meaningless. Moreover, reflection adds an overhead to the method invocation itself but provides dynamism which is the main concern of RAMI: any public method of any object can be called asynchronously.

6.5.1 Overhead of the reflexive asynchronism provided by RAMI

In this section, we measure the overhead added by the handling of the reflexive asynchronism on empty calls. The table 6.1 shows the average time spent to invoke an empty method with the RAMI framework using the `AsynchronousReferenceImpl` with different `MethodInvoker` implementations on different jdks. The test was performed 100,000 times on a GNU/Linux 2.4/Bi-Pentium III 450 mhz system. The jdk1.2 uses a classic JVM whereas jdk1.3 and jdk1.4 uses the HotSpot [?] technology. The test invokes the empty method a hundred of times before the real benchmark begins to prevent cache effect.

The first thing to notice is how the RAMI framework adds a significant overhead to a standard synchronous empty method invocation. The table do not mention the average time for these calls because they are near 0 ms/call (minimum is 3 ms/100000 calls for the JDK1.2 while the maximum is 38 ms/100000 calls for the JDK1.4 -client). It must be noticed that empty method can be inlined in NOP (No Operation) by Just In Time compilers (either classic JVM or HotSpot VM), while the amount of work to do in RAMI invocations to provide reflexivity and asynchronism is really non negligible. `FutureClient` are created on each invocation of the `AsynchronousReferenceImpl.call()` method using a `FutureFactory` implementation. Moreover, creating lots of short lived objects such as futures increase the garbage collector work which decreases the performance. This fact is really easy

	Synchronous	Concurrency		SingleThreaded	
		Threaded	ThreadPooled	Fifo	Random
JDK 1.2	137.22	60.263	26.837	23.953	23.861
JDK 1.3 -client	89.12	911.833	48.130	20.684	20.746
JDK 1.3 -server	95.65	905.551	50.837	21.221	21.398
JDK 1.4 -client	87.43	72.499	24.950	22.260	22.791
JDK 1.4 -server	77.62	74.788	24.299	22.210	22.748

Table 6.1: RAMI overhead in ns/call

to observe by increasing the stack size of the virtual machine by the `-Xms` and `-Xmx` of the command line: running the same “Synchronous” test with `java -Xms256m -Xmx512mx` command line (JDK1.4 -client) gives a 35% speedup with an average of 56.72 ns/call. In the next RAMI release, we will experiment with two different techniques to prevent short lived objects creation: implicit and explicit reuse of objects. Both techniques will be new `FutureFactory` implementations and will not break the existing framework.

The table is interesting to see the behaviour of different JVM when using RAMI.

The “Synchronous” row test (using the contradictory *synchronous asynchronous semantic* of `SynchronousMethodInvoker` class) shows the direct RAMI overhead since no threads are involved.

The “Threaded” and “ThreadPooled” rows shows the overhead of the *concurrency semantic* (`ThreadedMethodInvoker` and `ThreadPooledMethodInvoker` classes). As expected, creating a thread for each method invocation add a large overhead to the reflexive asynchronous method invocation.

Concerning the *threaded concurrency semantic*, we can observe several things: the JDK1.4 test uses 30% of the CPU in system space and 46% in user space while an average of 1500 threads/secondes were created. We guess that the time spent in system space concern the context switch of threads while in the user space, the CPU execute the RAMI code, and probably most of the garbage collector code. In the JDK1.3 test, only 3% of its CPU time was into system space and 5% in user space while an average of 200 threads/secondes were created. We assume that the HotSpot VM of the JDK1.3 is a really immature version of the HotSpot technology where threads, locks and memory management is really inefficient.

The “Fifo” and the “Random” rows shows the overhead of the *single threaded semantic* (`FifoMethodInvoker` and `RandomMethodInvoker` classes): invocations are enqueued and this require synchronization.

It must be noticed that the HotSpot Virtual Machine delivered with the JDK1.3 is really inefficient.

Using the RAMI framework in the way described above is really not a good idea ! First, empty methods are really rare in common programs ! But, worse, using an asynchronous method invocation to invoke a method synchronously is a real bad design practice: it adds a significant overhead and gives absolutely no advantages since the asynchronism which ususally provides better reactivity (unblocked I/O operation) and/or speedup on SMP computers, is ignored ! Hence to measure the advantage of the

RAMI framework, we need a real example.

6.5.2 Analysis on a real example

We have tested the RAMI library to compute the decimals of the number π . The algorithm used is the one found by Bailey, Borwein and Plouffe [?, ?]. The formula used for this test is:

$$\pi = \sum_{i=0}^{\infty} \frac{1}{16^i} \left(\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6} \right)$$

We implement the above formula in the `PiComputer` class which contains the method

```
BigDecimal compute(int start, int end);
```

Hence, to compute the n first decimals of π , it suffice to create an instance `computer` of the `PiComputer` class and to invoke:

```
computer.compute(0, n);
```

To illustrate the benefits of RAMI, the following code has been implemented: instead of invoking `computer.compute(0, n)`, we cut the original n sum into p subsums where p is a parameter which usually depends on the number of processors in the computer. The `main()` method invokes the `compute()` method p times in a reflexive asynchronous way using the RAMI framework. Hence, if two or more processors are available, each subsum can be computed in parallel. Before each computation, dummy calls are generated to prevent caches effect (Just In Time compiler optimizations for example) and the method `System.gc()` is invoked to try garbage collecting as much objects as possible before the computing start. Moreover, memory parameters are specified to the JVM to have a minimum stack (option `-Xms`) of 128 Mb and a maximum (option `-Xmx`) of 256 Mb. Since each parameter affect the behaviour of the Java Virtual Machine, results give a general overview of the performance one can obtain with RAMI.

The test was performed on an AIX IBM R6000 SP3 375Mhz quadri-processor node to compute 1000 decimals. On the AIX system, the shell environment variable `AIXTHREAD_MNRATIO` specify the number of user threads per kernel threads to be used. Since all threads involved in this example are cpu consuming, we specify a ration of 1:1.

Notice that the direct invocation of `PiComputer.compute(0, 1000)` take an average of 3:18.20 (3 minutes and 18.20 secondes, average on 7 execution) to terminate. The chart 6.1(a) shows the time spent to compute 1000 decimals of π in function of the parameter p of subsums.

Several things must be noticed:

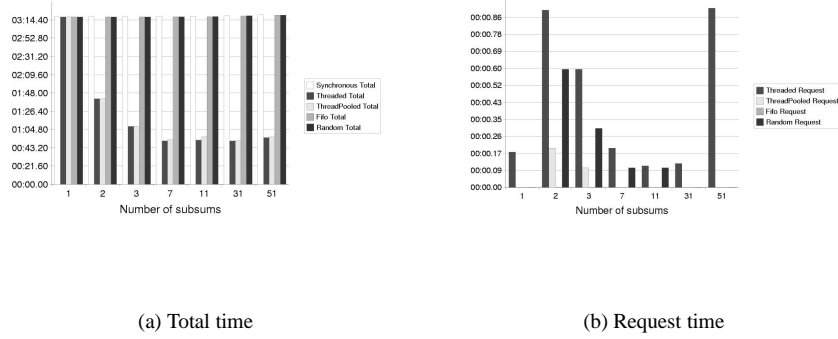


Figure 6.1: Time spent to compute 1000 decimals of Pi on a quadriprocessor node using RAMI.

- the best time is achieved with $p > 3$ as expected (4 processors node) resulting in a speedup³ of

$$Speedup = \frac{BestSequentialTime}{BestParallelTime} = \frac{3 : 18.20}{0 : 51.24} = 3.87 \approx 4$$

and an efficiency of

$$Efficiency = \frac{Speedup}{ProcessorsNumber} = \frac{3.87}{4} = 0.9675 \approx 1$$

- the time spent with any *asynchronous semantic* (including the contradictory *synchronous* one) is sometimes better than a direct invocation ;
- the *threaded concurrency semantic* provides the best performance.

The first point shows that the *concurrency asynchronous semantic* is the most efficient as expected.

The second point tends to prove that RAMI do not introduce a significant overhead when used properly but is suspicious: using RAMI implies many operation such as *futures* handling and reflexive invocation. While we had suspected the Just In Time compiler of the JVM to introduce “noises” in our benchmark, executing it in interpreted mode (`-Xint`) shows the same result. We are currently analysing this strange behaviour.

The last point shows that the *threaded concurrency semantic* tends to be the most efficient in terms of computation time. Nevertheless, when measuring the time spent for the request to be acknowledged⁴ as shown in the chart 6.1(b), the *threaded concurrency semantic* is the less efficient as expected (the *synchronous semantic* request time

³According to the definition of the speedup where the best sequential time is the time of the direct invocation of the method `compute()` without the RAMI overhead

⁴i.e. on the returned of the `call()` method where the caller thread is no longer blocked

is nonsense as it is the same as the computed time !). Creating a thread on each asynchronous method invocation has an overhead. The *threadpooled concurrency semantic* is a best choice when the time spent to do the request is more critical than the time spent to execute the method.

Chapter 7

Transparency

7.1 Introduction

As mentionned above, RAMI focused on *dynamic aspect*. To achieve its goal, RAMI uses the *reflection* mechanism provided by Java. So, the `call` method need a `Method` and an array for its arguments. This is a major drawback. First, it is really annoying to give such information for a simple method invocation (even if in fact, it is asynchronous). Second, and probably most importantly, this syntax avoid compiler checking, what is called *strong typing*.

A solution to both problems is the notion of *transparency*, that is the possibility to perform an asynchronous method invocation as a synchronous local method invocation.

7.2 The *Strong typing* problem

Strong typing is the mechanism allowing the compiler to verify a bit more than the syntax of a call : its arguments and the variable in which the result is to be set is also checked using the notion of type. Typing is *strong* because it is known when the programmer writes its application.

The problem arise in reflective method invocation because methods are specified with a `String` which is not checked at runtime. The compiler doesn't know that this string represents a method name. Also, parameters are specified in a generic array of `Objects`. As with `String`, the compiler cannot suppose that the array represents method parameters. How can strong typing be ensured in such a situation ?

One solution is to declare an interface and to give client a proxy which implements this interface. When the client invoke a method of this proxy, the real reflective method invocation is performed. Since the proxy may be generated by compilation, *strong typing* is ensured. The reader may wonder how usefull is reflective method invocation if a compiler must generate a proxy to ensure *strong typing*. The reader may see a contradiction here and yes it is ! But imagine that the proxy resides on a different JVM (host) that the object it redirect method invocation on. In such a way, this mechanism ensure remote method invocation with *strong typing*. Using interfaces, clients invokes

method naturally and this is what is called *transparency*. RMI and Corba uses a similar mechanism to implement *transparency of remote method invocation*.

But, RAMI provides *reflective asynchronous method invocation* and the asynchronism leads to some more problems.

7.3 Total transparency

As seen in the previous section, the notion of interface resolve both problems of *strong typing* and of *transparency of reflective method invocation*. How this solution can be extended to provide both *strong typing* and *transparency of asynchronous method invocation* ?

One solution is the use of the *wait-by-necessity* mechanism. The idea is to provide *total transparency* of asynchronous method invocation. When a method is called, a *future result* is returned immediatly¹ allowing the thread to continue its execution during the real method invocation. When the thread use the result, it is blocked. Thus, for the caller thread, the asynchronous layer is transparent, it is not seen at all.

7.3.1 Inheritance

As done in [?], each method of an interface implemented by a proxy returns a *future object*. This object is an instance of a class which extends the original declared returned class in the (remote) interface. Then, each method of the returned object is a blocking method : if the result of the related call is achieved, the method returns immediatly, otherwise, the caller is blocked until the method returns.

This solution arise lots of problems related to inheritance :

- final classes cannot be inherited, thus, such classes cannot be inherited to create *future objects*;
- final methods cannot be overridden to implement the *wait-by-necessity* mechanism in *future objects*;
- public fields access cannot be blocked : the *wait-by-necessity* mechanism cannot be implemented for fields.

Moreover, for asynchronous *remote* method invocation, public fields access arises a coherency problem : when such a field is used by a remote client, the proxy is modified instead of the related remote objects.

For all those reasons, another implementation is provided in RAMI.

7.3.2 Interfaces and the `java.lang.reflect.Proxy` class

The *wait-by-necessity* mechanism is implemented only for objects which implements interfaces. Since interface declares only public methods, fields are not concerning

¹*i.e.* note that some cpu cycles may be consumed to manage the asynchronism, but the real method has not yet start running.

avoiding the problems describes above. Moreover, this solution avoid the problem of final classes or methods arises by the inheritance solution.

The `AsynchronousProxyInvocationHandler` class

Java, since the jdk v1.3 provides the `Proxy` class which can implements any given class. To get an instance of this class, a user must give the list of interfaces to implement, and a `java.lang.reflect.InvocationHandler` which whill handle invocations on the returned proxy instance.

RAMI provides the `AsynchronousProxyInvocationHandler` which implements the *wait-by-necessity* mechanism.

The program 7.3.1 is an example of a *total transparent asynchronous method invocation* using this implementation.

Program 7.3.1 An example of a *total transparent asynchronous method invocation* using `AsynchronousProxyInvocationHandler`.

```
1 // Instanciate an object implementing a given interface
  MyInterface object = new MyImpl();

  // Gets its remoteReference
5 AsynchronousReference asynchronousReference = // depends on implementation

  // Gets the invocation handler
  InvocationHandler proxyHandler =
    AsynchronousProxyInvocationHandler.getInstance(asynchronousReference);
10

  // Gets the total transparent proxy
  MyInterface proxy = (MyInterface)
    Proxy.newProxyInstance(this.getClass().getClassLoader(),
                          new Class[] {MyInterface.class},
15                          proxyHandler);

  // Total transparent asynchronous method invocation
  // 'result' is a wait-by-necessity futur implementation.
  ResultInterface result = proxy.foo("fooArg");
20

  // If the asynchronous invocation of 'foo' has terminated, this methods returns
  // the result. Otherwise, the caller is blocked until the result is available.
  Object o = result.method(arg);
```

Problems related to interfaces solution

Even if these solution avoid the problems of fields access, and of final classes and methods, it has its drawback : it can only be used for classes which implements interfaces - which limits legacy code.

For example, if a method returns a `Long` object, only the method `compareTo` can be invoked with this mechanism : `Long` do not implements another interface than `Comparable` and thus, any other method must be called with the raw mechanism provided by RAMI: the `call` method preventing *transparency*.

7.3.3 Problems of *total transparency*

Many problems arise with the *total transparency* mechanism.

Exceptions handling

The *total transparent asynchronous method invocation* mechanism is not well suited for exception handling. What happens when the method declares an exception and throws one? Remember that the caller thread did continue its execution and is not in a standard `try/catch` block statement.

Perhaps a solution is to catch the exception behind it, and to throw it again on result access thanks to the *wait-by-necessity* mechanism. But, access to the result object are methods, which do not necessarily declare the thrown exception declared in the invoked method.

Another solution, as done by [?] is to limit asynchronous method invocation to method which do not declare any exception. Hence, any method which declares exception to be thrown, would be invoked synchronously.

Those solutions do not cover the problem of undeclared exception, subclasses of `RuntimeException`.

Developer consciousness

The *total transparent asynchronous method invocation* allows programmer to forget the asynchronism and to develop as in usual synchronous method invocation. This provides legacy code since already existing code can be used with very little modifications to ensure asynchronism, hence concurrency or parallelism (depending on number of CPUs considered or hosts in case of remote method invocation).

Consequently, this mechanism, conjugated with *remote method invocation*, allows programs to run more efficiently since they use parallelism transparently.

Programmer who have already developed parallel application using standard parallel library (PVM or MPI), can see the benefit of this mechanism.

Nevertheless, this mechanism has the *developer consciousness* problem.

To be efficient, *total transparent asynchronous method invocation* must follow a simple rule :

- method invocation must be done earliest;
- result access must be done latest.

Consider the code fragment below :

```
doSomething();  
doSomethingElse(object.method());  
continue();
```

which is the "natural way" method invocation are done. If the *total transparent asynchronous method invocation* mechanism is used, this fragment will probably run *less efficiently* than without it. In fact, the mechanism has a cost (handling threads and future object creation), which must be considered. The previous fragment does not use the concurrency involved in the mechanism (parallelism in distributed computing). So, to be more efficient, the code fragment should be rewritten using the simple rule :

```
// Total transparent asynchronous method invocation
MyResult result = object.method()

// Do something concurrently
doSomething();

// Use the result (may be blocked on result method
// invocation if object.method() did not returned yet.)
doSomethingElse(result);

continue();
```

Programmer must be conscious of the asynchronism of their method invocation, and of the *total transparent* mechanism to write efficient code.

7.4 Semi transparency

The *semi transparency* mechanism try to solve problems linked to *total transparency*. First, to avoid the developer consciousness problem, asynchronous method invocation need to be "special call". Second, *strong typing* must be ensure, and the notion of interface seems a good solution to provide a sort of transparency in *asynchronous method invocation*.

7.4.1 Client's view of object

RAMI distinguish the object which is referenced by an `AsynchronousReference` and the client's view of it. In fact, two clients may have distincts view of the same objects. This is the notion of interfaces. An object may implements many interfaces, each of which represents distincts view of him.

But, RAMI do not require objects to implement interfaces. Remenber that RAMI uses *reflection* to provides dynamism. Here the dilemna.

Why objects should they declare the interface they implements ? The answer is *strong typing* : an interface is a *type* and the compiler can check several things with it. But, RAMI consider that every objects implements a whole set of interfaces even if they do not declare them.

How can this be true ?

Here an extract of a run of a well known program called `javap`²:

```
shell> javap java.lang.Object
Compiled from Object.java
public class java.lang.Object {
    public java.lang.Object();
    public final native java.lang.Class getClass();
    public native int hashCode();
    public boolean equals(java.lang.Object);
    [...]
    public java.lang.String toString();
    public final native void notify();
    public final native void notifyAll();
    static {};
}
```

²This as been executed on a Linux machine but results must be similar on any platform supporting Java

What does this mean? `java.lang.Object` implements its own interface which are all of its `public` fields³. Moreover, it implements many more interfaces which are all the combinations of its `public` fields.

For example, RAMI considers that `java.lang.Object` implements also the interface below :

```
public interface Anonymous1 {
    public native int hashCode();
    public boolean equals(java.lang.Object);
    public java.lang.String toString();
}
```

and also this one :

```
public interface Anonymous2 {
    public java.lang.String toString();
}
```

It is evident that `java.lang.Object` do not declare the interfaces `Anonymous1` and `Anonymous2`.

A client can communicate with an object if it has a knowledge of methods the object implements *i.e.* if it has an interface.

7.4.2 The JayaCompiler class

RAMI provides the `jyac` compiler which is in fact based on the `JayaCompiler` class. This class creates a Java class in pure Java source code which has the same name as the one given in argument, but which resides in a package named the same as the one of the given class prefixed by 'jaya'. Hence, if you compile the `java.lang.Object` class, then the `jaya` compiler will generate the `jaya.java.lang.Object`. Moreover, any class or interface, which the given class extends or implements is also compiled. Then, if you compile the class `java.lang.String`, then the `jaya` compiler will generate the `jaya.java.lang.String` class along with `jaya.java.lang.Object`, `jaya.java.io.Serializable`, `jaya.java.lang.Comparable`, and `jaya.java.lang.CharSequence`.

The `jaya` generated classes (called *jaya classes*) contains all the `public` methods declared in their related "standard classes" prefixed by `rami_` and changed to return a `FutureClient` instead of their original returned class.

For example, consider the class :

```
public class MyClass {
    public StringBuffer foo(String s) {
        ...
    }
}
```

Its *jaya related classes* is:

```
package jaya;

public class MyClass {
    [...]
    public static final Method fooMethod;
```

³ `javap` gives more information than just `public` fields as `protected` and `static` fields for example, but you can have just `public` fields with `javap -public`

```

        static{
            // Initialisation of the fooMethod field
            [...]
        }
    [...]
    public FutureClient rami_foo(String s) {
        Object args[] = new Object[]{s};
        return asynchronousReference.call(fooMethod, args);
    }
    [...]
}

```

This class will be the client's proxy of a given instance of MyClass :

```

[...]
```

```

MyClass myObject = new MyClass();
AsynchronousReference asynchronousReference = // depends on implementation

// Gets a semi transparent asynchronous proxy
jaya.MyClass ar_myObject = // gets the instance

// Semi transparent reflective asynchronous method invocation
FutureClient future = ar_myObject.rami_foo("Hello World");

[...]
```

```

System.out.println(future.waitForResult());

[...]
```

Each methods which start with `rami_` is a *semi transparent asynchronous method invocation*. Thus, this mechanism allows *strong typing* but resolve both problems of *exception handling* and of *developer consciousness* : since methods return *future* objects, the developer cannot forget the asynchronism property of its call. Thus, he can program more efficiently doing asynchronous call earlier, and result recovery later.

7.5 Summary

This chapter has focused on *transparency*. Two distincts features provided in RAMI has been described :

- total transparent asynchronous method invocation which uses the *wait-by-necessity* mechanism but which arise two problems : exception handling and developer consciousness.
- semi transparent asynchronous method invocation which uses the notion of client's view of a remote object by proxy java source generation.

Chapter 8

Conclusion and perspectives

This document has presented RAMI a framework that provides *reflective asynchronous method invocation*. It focused on *dynamic aspect* by being entirely based on the reflection mechanism. Objects in RAMI do not have to implement a special interface or inherit a special class or even to be compiled. Any object can be used asynchronously by being referenced by an *asynchronous reference*. In RAMI, such references are instances of class that implements the `AsynchronousReference` interface.

RAMI uses *futures* to handle the result of (or the exception thrown) by an asynchronous method invocation. RAMI permits developpers to use an *event-driven* programming scheme by the use of the `Callback` interface. *Chained asynchronism* specify the semantic of an asynchronous reference on another asynchronous reference in the `AsynchronousReferencePair`.

RAMI proposes an implementation of its specification which provides several properties such as unicity of asynchronous references, customizable future factories and customizable asynchronous semantic.

While RAMI focuses on *dynamism* using reflection, it has its drawback: this mechanism do not ensure *strong typing*. Two distincts solutions is provided based on *transparency* of *asynchronous method invocation*: *total transparent* and *semi transparent* reflective asynchronous method invocation.

Whereas RAMI is not dedicated for asynchronous *remote* method invocation, its design allows anyone to implements a remote asynchronous reference. This is already done by our team with the JACOb [?, ?] framework which provides *dynamic reflective remote method invocation* where any object can become remote at any time and where any methods of such a remote object can be invoked asynchronously using a specific implementation of the `AsynchronousReference` interface.

Bibliography

- [1] Pi net site. Web page. <http://www.cecm.sfu.ca/pi/pi.html>.
- [2] The javagrande homepage, January 2001.
<http://www.javagrande.org/>.
- [3] Ken Arnold and James Gosling. *The Java programming language*. Addison-Wesley, 1996.
- [4] David Bailey, Peter Borwein, and Simon Plouffe. On the rapid computation of various polylogarithmic constants. Web page.
<http://www.lacim.uqam.ca/plouffe/articles/BaileyBorweinPlouffe.pdf>.
- [5] Denis Caromel, Wilfried Klauser, and Julien Vayssière. Towards seamless computing and metacomputing in Java. In Geoffrey C.Fox, editor, *Concurrency: practice and experience*, volume 10, pages 1043–1061. Wiley and Sons, Ltd., September–November 1998.
- [6] Katrina E. Kerry Falkner, Paul D. Coddington, and Michael J. Oudshoorn. Implementing Asynchronous Remote Method Invocation in Java. Technical report, Distributed High Performance Computing Group, July 1999.
- [7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994. ISBN : 0-201-63361-2.
- [8] S. Hirano. HORB: Distributed Execution of Java Programs. In *Proc. WWCA'97*, volume Vol. 1274 of *LNCS*, pages 29–42. Springer Verlag, Berlin, 1997.
<http://ring.etl.go.jp/openlab/horb/>.
- [9] Allen Holub. If i were kink: A proposal for fixing the java programming language's. HTML page, October 2000. <http://www-4.ibm.com/software/developer/library/j-king.html?dwzone=java>.
- [10] Gosling James, Joy Bill, and Steele Guy. *The Java Language Specification*. Addison Wesley, 1996.
- [11] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, 1999.

- [12] Sun microsystems. Java 2 enterprise edition web site. Web page.
<http://java.sun.com/products/j2ee/>.
- [13] Sun microsystems. Java 2 micro edition web site. Web page.
<http://java.sun.com/products/j2me/>.
- [14] Sun microsystems. Java 2 standard edition web site. Web page.
<http://java.sun.com/products/j2se/>.
- [15] Sun microsystems. Javacard web site. Web page.
<http://java.sun.com/products/javacard/>.
- [16] Sun microsystems. Javatm 2 platform, standard edition, v 1.4.0 api specification web site. Web page. <http://java.sun.com/j2se/1.4/docs/api/index.html>.
- [17] Sun microsystems. Java messaging services specifications.
<http://java.sun.com/products/jms/>, November 1999.
- [18] Sun microsystems. The Java HotSpot performance engine architecture. white paper, April 1999.
- [19] G. K. Thiruvathukal, L. S. Thomas, and A. T. Korczynski. Reflective remote method invocation. *Concurrency: Practice and Experience*, 10(11–13):911–925, 1998.
- [20] Paul Tremblett. Java reflection : not just for tool developers. Dr. Dobb’s journal, January 1998. <http://www.ddj.com/documents/s=924/ddj9801c/9801c.htm>.
- [21] Pierre Vigneras. Jacob : un support Java pour la distribution et le calcul. *CPRSR, Réseaux et Systèmes Répartis, Calculateurs Parallèles*, 12(5-6):537–563, décembre 2000.
- [22] Pierre Vigneras. Jacob : a software framework to support the development of e-services, and its comparison to Enterprise JavaBeans. In *Perspectives for Commercial and Scientific Environments*, pages 11–12, University of Technology Munich, 19-20 April 2001. International Workshop on Performance-Oriented Application Development for Distributed Architectures (PADDA).
- [23] E Walker, R Floyd, and P Neves. Asynchronous remote operation execution in distributed systems. In *International Conference on Distributed Computing Systems*, number 10, pages 253–259, Paris, France, May/June 1990.