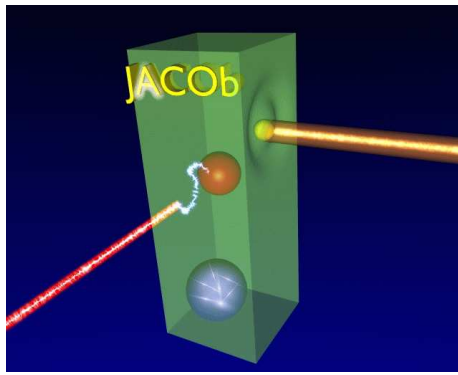


Mandala/JACOb User's Guide – February 22, 2005



Pierre Vignéras
"eipi"
eipiequalsnoone@users.sf.net

February 22, 2005

Abstract

This document describes the Mandala/JACOb subpackage, a framework for *dynamic reflective asynchronous remote method invocation* entirely written in Java™¹. *Dynamism* means that object's class do not have to be written with the remote feature in mind as in RMI to be remotely accessible in an asynchronous manner.

It is entirely based on the concept of *active container* as JACOb, *Java Active Container of Objects* is an implementation of this concept in Java. This concept implementation essentially provides the *dynamic remote aspect* of objects. JACOb uses the RAMI library which provides mechanisms related to *reflective asynchronous method invocation: chained asynchronism, exception handling, and transparency*.

keywords: reflective remote asynchronous method invocation, client-sided asynchronism, server-sided asynchronism, full-asynchronism, 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	Prerequisite	6
3	The active container concept	7
4	Dynamic remote objects	9
4.1	Introduction	9
4.2	Dynamic Aspect	10
4.3	Resource sharing	12
4.4	Summary	13
5	Dealing with the JACOb package	14
5.1	The <code>ActiveMap</code> interface	14
5.2	Dealing with remote objects	14
5.2.1	Terminology	14
5.2.2	Direct remote objects : the <code>Remote</code> interface	15
5.2.3	Direct remote active map	15
5.2.4	Manipulation of stored object: the <code>StoredObjectReference</code>	15
6	Reflective asynchronous (remote) method invocation	16
6.1	Introduction	16
6.2	Server-side reflective asynchronous (remote) method invocation	17
6.3	Client-side reflective asynchronous (remote) method invocation	18
6.4	Exception handling	22
7	Advanced techniques	24
7.1	Object caching	24
7.1.1	Security	24
7.2	Direct remote communication with stored objects using a custom protocol	24
7.2.1	Just in time security	24
7.3	Services	26
7.3.1	Monitoring	26
7.3.2	Persistence	26
7.3.3	Transaction	26

8	Conclusion and perspectives	27
8.1	Conclusion	27
8.2	Perspectives	27

List of Figures

4.1	<i>Stub</i> and <i>skeleton</i> in Java-RMI	10
4.2	<i>Stub</i> and <i>skeleton</i> in Java-RMI	11
4.3	Remote active container	11
4.4	Communication with a stored object	13
6.1	The <code>call()</code> asynchronous mechanism	18
7.1	Object caching	25

Chapter 1

Introduction

Parallel processing is no more limited to expensive supercomputers [18] but can also make use of clusters of workstations thanks to the growth of processors power and network speed. Programming applications for that kind of environment is a real challenge and software tools must be available to ease this task. Distributed applications tend to be expressed using object oriented languages even if other programming techniques emerge (design patterns [7], application frameworks and componentware [26], agent-oriented applications [1], aspect oriented programming [12], etc.). The Java™¹ language [3, 11, 15] has proven to be a generic language as many others and is not dedicated to so called applets. Even if the Java implementations suffer from performance problems due to the interpreted nature of the language, Just In Time compilers [4, 5, 13], Sun's HotSpot™ technology [24] and native compilers [25] are real effective solutions.

The use of Java in the domain of distributed computing is a reality as shown for instance by the JavaGrande Forum [2]. The Java Remote Method Invocation framework [22] allows components of distributed applications to communicate via method invocation. The communication complexity is hidden to the programmer by *fragmented objects* [17]**** TODO : VERIFY ****. Nevertheless, Java RMI does not provide efficient mechanisms for distributed computing in terms of performance and design. In fact, it was mainly designed for client/server applications, not for large distributed applications with many objects. Whereas asynchronous communication is a standard paradigm in parallel programming using MPI [9] or PVM [8] to achieve communication/calculus overlapping, it does not directly exist in standard Java² RMI specification. Reasons may be asynchronism is not well suited for remote method invocation and exceptions handling is not well defined. The main reason is probably the focus of Java on simple mechanisms.

Several projects try to improve the performance of RMI. For example Krishnaswamy and al. [14, 16, 19] present a more efficient implementation of RMI. A reimplementa-

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.

²Even if they can be implemented with threads; an asynchronous input/output Application Programming Interface will not be available before the JDK v1.4.

tion of the serialization mechanism used by RMI is proposed by Philippsen and al. [20].

Other works extend the language to provide distribution transparency or asynchronous remote method invocation. JavaParty [21], for example, adds the keyword `remote` to the language to declare remote objects. A compiler generates additional classes to transparently distribute objects which are accessed using standard method invocation. The Falkner and al. project described in [6] 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 *stubs* and *skeletons* and the use of *Futures* [27] allow developers to deal with asynchronous calls.

Research is also carried out around the automatic generation of remote objects. HORB [10] 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 *method naming convention*. For example, a `foo()` method of a remote object must be renamed by `foo_Asynch()` and clients must use `foo_Request()` and `foo_Receive()` to achieve the asynchronous call to `foo()`.

As seen above, RMI, like other comparable projects rely on *static* declaration of properties that say 'remotely accessible', 'asynchronously accessible' or both. JACOb focuses on *dynamic aspect* : any object can be made remote at any time, and any method can be invoked asynchronously. Using keywords is not a solution since it is a *static* solution.

This document presents JACOb, the framework for *dynamic* asynchronous remote method invocation. It is based on the concept of *active container* introduced in section 3. The *dynamic generation of remote objects* it provides is describe in section 4. Its *asynchronous remote method invocation* mechanism is described in section 6. Conclusion, future work and research directions are presented in section 8.1.

Chapter 2

Prerequisite

The reader must be familiar with Java, *reflection*, and with the RAMI package which provides *reflective asynchronous method invocation*. **** TODO : REFERENCES ****

Chapter 3

The active container concept

An active container is a container of objects which can generate activities over its stored objects. Currently, it has only a limited set of methods which are :

<code>void put(Object key, Object object)</code>	: insert an object in the container;
<code>void remove(Object key)</code>	: remove an object from the container;
<code>Object get(Object key)</code>	: get a copy of an object from the container;
<code>void call(Object key, String method, Object[] args, MethodResult result)</code>	: call a method of an object contained in the active container.

Almost all methods are standard container's method and are self-explanatory. The `call()` method is a special method which generate the activity. It invokes the specified method on the object mapped to `key` in the active container. The method is specified as a string which is why *reflection* is necessary. This leads to many problems which are solved by the notion of transparency provided by the RAMI package****
TODO : REFERENCE ****.

The simplicity of this concept eases application modeling and its genericity makes it possible to express sophisticated tasks such as code migration. The program 3.0.1 shows how object migration can be expressed.

The *JACOb (Java Active Container of Objects)* framework is an implementation of the concept of active containers in Java. It adds two other functionalities that are not explicitly specified above :

- objects can become remote dynamically;
- remote method invocation of *stored objects*¹ is asynchronous.

¹A *stored object* is an object contained in an active container.

Program 3.0.1 Active containers : migration

```
1  fromActiveContainer.call(key,
    "stopActivity",
    null,
    null);
5  MyObject object =
    (MyObject) fromActiveContainer.get(key);
    fromActiveContainer.remove(key);
    toActiveContainer.put(key, object);
10 toActiveContainer.call(key,
    "restartActivity",
    null,
    null);
```

Migration of an object from the active container `fromActiveContainer` to the active container `toActiveContainer`.

The next sections describe these two functionalities in details.

Chapter 4

Dynamic remote objects

4.1 Introduction

Distributed systems such as RMI or CORBA use the *fragmented objects* [17] paradigm****
TODO : VERIFY ****. We consider a remote object as composed of four parts ??:

the object part: it is the 'business code' of the remote object;

the proxy part: it may hold local data, achieve computations locally, and forward calls to its corresponding remote reference by encoding them into network packets (a *stub* is a special case of a proxy performing no local processing and reduced to the communication function);

the skeleton part: it decodes incoming network packets into method invocations;

the server part: it listens to the network for incoming calls, and uses the skeleton part to invoke the specified method.

When no ambiguity is possible, server and skeleton will be considered as a whole.

These parts are usually generated by a compiler. Most often, an interface must be declared (IDL for Corba, `java.rmi.Remote` in RMI, keyword `remote` in JavaParty, etc.) and implemented by remote objects. A compiler generates the proxy/skeleton pair (idl compiler, `rmic`, `HORBC`, etc.) that hides the complexity of the communication.

For example, the `rmic` compiler generates both skeleton¹ and stub classes. The stub class implements the remote interface of the object part which inherits the server part (usually `UnicastRemoteObject`). Any remote method call is sent by the stub to the remote server part which calls the skeleton part and invokes the effective method as illustrated by figure 4.2.

Each of the four distinct parts needed by an object to become remote is either :

¹Since JDK 1.2, skeletons are not necessary. Using reflection, the server part decodes the method invocation itself. The command `rmic -v1.2` generates classes for RMI v1.2 protocol without skeleton classes.

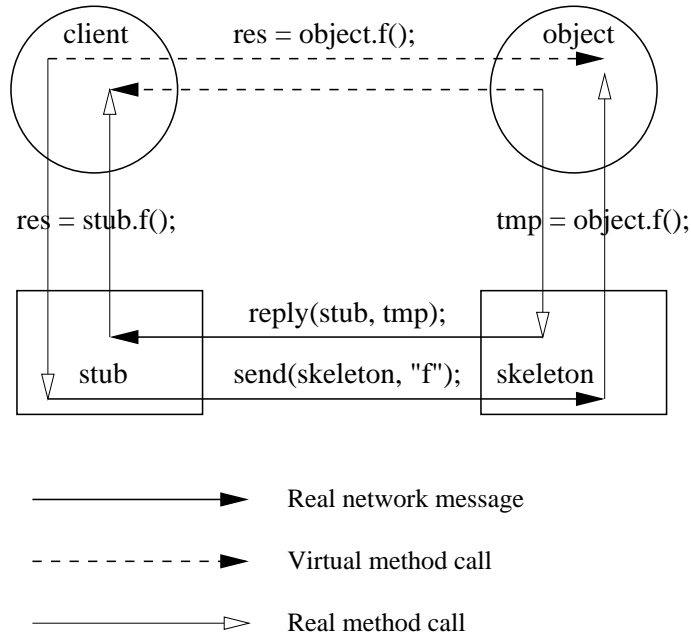


Figure 4.1: *Stub* and *skeleton* in Java-RMI
 Java Remote Method Invocation uses the *stub/skeleton* paradigm.

- inherited² by the object like the server part (`UnicastRemoteObject`) of RMI objects,
- generated by a compiler like the proxy and skeleton parts of RMI objects created by `rmic`³.

These solutions are static. To provide dynamic aspect, a solution based on *remote active container* is proposed.

4.2 Dynamic Aspect

In JACoB, active containers can be *remotely accessible*. In such a case, an active container is in some sens the server *and* the skeleton part of any of the objects it contains. The remote reference of the contained object is composed of its *key* and the proxy of the active container. The `call()` method of the active container uses reflection to encode/decode method invocation and can be considered as the replacement of the skeleton part as illustrated in figure 4.3.

Hence, to be remotely accessible, an object *has to be in* an active container. Two mechanisms can be used for this purpose. An object may be instantiated locally and

²Composition is used instead of inheritance by Corba's components.

³RMI v1.1.

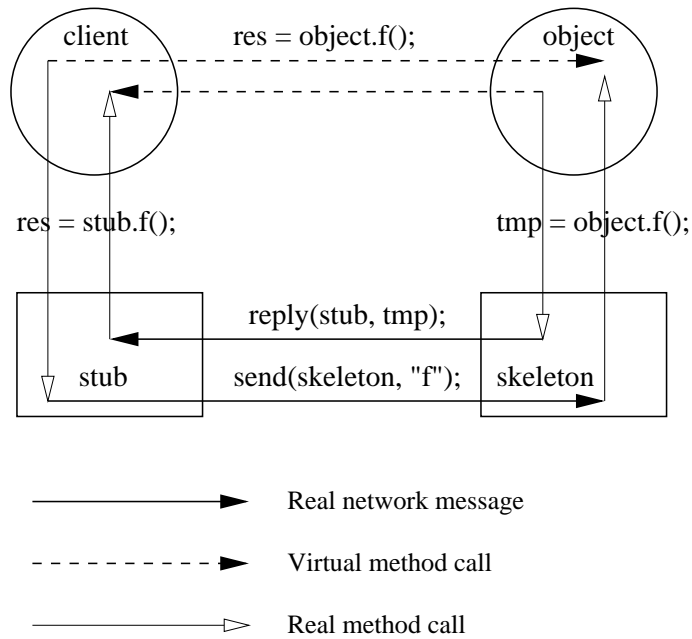


Figure 4.2: *Stub* and *skeleton* in Java-RMI
 Java Remote Method Invocation uses the *stub/skeleton* paradigm.

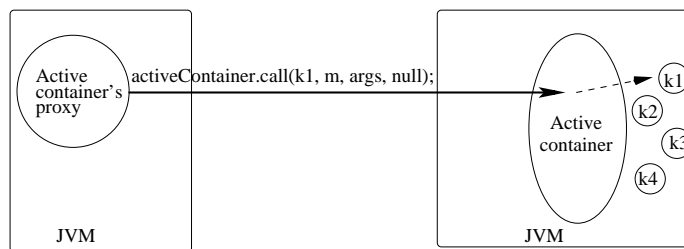


Figure 4.3: Remote active container
 A remote *active container* is the server and skeleton part of the objects it contains.

inserted in an active container with its remote `put ()` method. Since the object must be transferred through the network, it has to be serializable. An object may be instantiated directly in the active container with a local `put ()` call. In this case, the object does not have to be serializable.

Any object can then be remote, and the declaration of interfaces is not needed for remote objects in JACOb. Moreover, remote objects do not have to inherit a special class. Any Java class can be remote and still be a subclass of any class. This is very important for building object systems using design patterns [7]. If a remote class had to be a subclass of a specific class, it could not be a subclass of a pattern since Java does not support multiple inheritance.

JACOb neither requires compilation nor the generation of classes designed for remote access and which cannot be used in any other context. JACOb also provides separation of concerns : programmers do not have to worry about the remote feature of their objects.

4.3 Resource sharing

As described in 4.2, a remote object usually has an associated server with an open listening socket, one or more input/output streams, one or more dedicated threads, etc. So, remote objects consume a lot of resources even if they probably do not need all of them at the same time. Since real applications may hold a large number of remote objects, they are limited on some operating systems by one or more of the maximum number of : threads⁴ per process, threads in the system, file descriptors⁵ per process, file descriptors in the system.

JACOb provides a mechanism to share such resources since an active container is the server *and* the skeleton part of any of the objects it contains. The remote reference of a stored object is the pair

$$(activeContainer, key)$$

where *activeContainer* is the active container that contains the object and *key* is its associated key. To communicate with a stored object, a client acquires a local reference on the *proxy* of the active container and invokes its `call ()` method as shown figure 4.4.

Active container's objects share sockets and communication threads allowing as many number of remote objects as the available memory makes it possible. Thus, JACOb is said to be *resource friendly*.

⁴Threads limitation occurs if the Java Virtual Machine is not implemented on user space threads library such as the green threads. Since JDK v1.3, Sun delivers a virtual machine based on the HotSpot technology which does not support green threads. Moreover, almost all native threads implementations of the Java Virtual Machine make use of kernel threads.

⁵Since sockets use file descriptors.

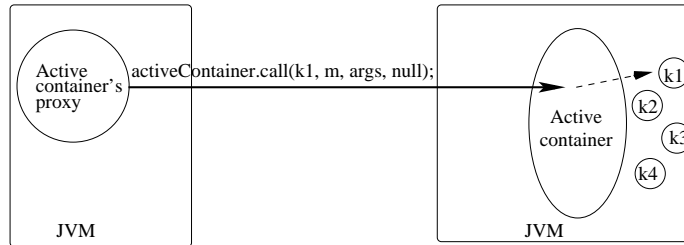


Figure 4.4: Communication with a stored object
 A client uses an active container's proxy and a key to communicate with a stored object.

4.4 Summary

This chapter has presented the *dynamic* feature of JACOb which is based on *remote active containers* : any object can be accessed remotely through the `call()` method of a remote active container without any *a priori* declaration or any compilation. JACOb is *resource friendly* since active container is the *shared skeleton and server* part of its stored objects.

It must be pointed out that even if RMI remote objects share their communication resources (threads and sockets), the high resources consumption problem of RMI objects still remains in the deployment of distributed applications. In fact, when an RMI object is running, its resources cannot be shared by a newly created RMI object easily. Two solutions can be used. The currently running remote object is stopped and a new `main` is coded to instantiate both the old object and the new remote object. This solution is not well suited for high dispoibility object server used currently in the real world. The other solution which avoids the last problem is the creation of another Java Virtual Machine process which runs the newly created object. This solution rises the high resources consumption problem since each process may allocate its own pool of threads and sockets. JACOb offers a simpler mechanism : newly created object can be inserted in the already running active container allowing its resources to be shared. Thus, only one active container has to run on a given machine to provide distributed application deployment.

Chapter 5

Dealing with the JACOb package

5.1 The ActiveMap interface

While JACOb is based on the *active containers* concept described in chapter 3, it is written in Java which provides the `java.util.Map` class - an *inactive* container interface ! Hence, JACOb extends this interface in the `ActiveMap` class which defines the method `call()` :

```
public interface ActiveMap extends Map {  
    void call(Object key, FutureServer future);  
}
```

The `call()` method is not exactly the one presented in the active container concept 3. The `FutureServer` class contains - among other things - the method to invoke, the arguments of the method and an equivalent to the `MethodResult` class. See the section 6 for details.

The `ActiveMap` interface specifies the behaviour of a JACOb *active* container. Since `ActiveMap` inherits `java.util.Map`, everywhere you used a `Map`, you may use an `ActiveMap` instead providing another way to deal with your stored objects.

5.2 Dealing with remote objects

5.2.1 Terminology

We must concentrate on terms used in the following :

a stored object : is an object which resides in an active map wether this map is remote or not;

a remote object : is any object (stored or not) accessible remotely;

a remote stored object : is a stored object accessible remotely through a remote active map;

a direct remote object : is an object accessible remotely by another mechanism than a remote active map.

Note that a stored object can be remote *and* direct remote at the same time. We will see this feature in 7.

5.2.2 Direct remote objects : the Remote interface

Direct remote objects must have a defined behaviour to be used correctly. For example, exceptions must be handled in a uniform manner. Hence, JACoB provides the Remote interface which defines the behaviour of a *direct* remote object. In fact, only the behaviour of exception handling is specified in this time, but, if other specifications must be defined for direct remote objects in the future, only one class would have to be redesigned.

5.2.3 Direct remote active map

JACoB provides some *direct remote* implementations of the ActiveMap interface. When you've instantiated such an implementation, you may consider the instance as an ActiveMap object hiding the remote feature of your object. Thus, local active map and remote active map can be interchanged easing distributed objects applications deployment.

5.2.4 Manipulation of stored object: the StoredObjectReference

The JACoB reference of a stored object is a pair

$$(activeMap, key)$$

and is wrapped by the StoredObjectReference class. Instead of using the active map reference to deal with a stored object, an instance of this class represents the pair (activeMap, key). Hence, removing a stored object do not require the key and the active map anymore: if sor is an instance of the StoredObjectReference class, doing sor.remove() remove the object mapped to the key sor.getKey() in the active map sor.getActiveMap().

Notice that an StoredObjectReference can be invalid if the key sor.getKey() is not mapped in the active map sor.getActiveMap(). The state of a stored object reference is returned by sor.isValid(). If it returns false, the only action that can be done is invoking its method put() which insert an object into the active map and mapped it to the key.

Moreover, the StoredObjectReference implements the AsynchronousReference of the RAMI package which defines the call() method providing *reflective asynchronous (remote) method invocation*. Hence, the call() method of the ActiveMap interface is not aimed to be used by the end-user of a stored object.

Chapter 6

Reflective asynchronous (remote) method invocation

6.1 Introduction

Method invocation is not very adapted to asynchronous communication since a method call often returns a result. The caller thread may wait the availability of the result. Therefore, many frameworks modify the method invocation paradigm to suite asynchronism. For example, HORB [10] provides asynchronism with a *send/receive* model while Java Messaging Service [23] proposes a *publish/subscribe* model. While both models are more adapted to asynchronous communication, any existing code has to be rewritten - when possible - to use it. For example, if a synchronous communication is transformed for efficiency reasons into an asynchronous one, the programmer has to introduce the send/receive instructions on both client and server side¹. Such a change may have a really bad impact on the design of the application.

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

```
new Thread(new Runnable() {
    public void run() {
        // invoke the desired
        // method asynchronously
        f();
    }
}).start();
```

is not a good design solution: a thread object must be instantiated and result polling must be implemented.

Several projects try to provide 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 Falkner and al. [6] needs a compilation phase to generate stubs dedicated to the asynchronous paradigm.

¹Since 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.

JACOb uses the RAMI package to provide *reflective asynchronous method invocation*. Since an active container may be remote, then JACOb may provide *reflective asynchronous remote method invocation*.

6.2 Server-side reflective asynchronous (remote) method invocation

Communication with the stored objects of an active map is achieved through the method `call()` of the `StoredObjectReference`². Since `StoredObjectReference` implements the `AsynchronousReference` of the RAMI package, asynchronism is provided within this method: the method specified in it is executed in a dedicated thread. The `call()` method is defined as follow :

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

where

`method`: is the method to invoke asynchronously;

`args`: represents the arguments of the specified method;

`FutureClient`: is the class the method returned an instance of to handle the asynchronous (remote) method invocation.

The Figure 6.1 illustrates the `StoredObjectReference`'s `call()` mechanism and the use of the `FutureClient` instance returned. (1) A thread (`callerThread`) is running, preparing a reflective asynchronous (remote) method invocation. It does the `call()` invocation (2) which returns a `FutureClient` object (`future`) (3). The (remote) active map generates activity by allocating a thread (`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 active map 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*.

Notice that asynchronism is done on the *server-side* instead of *client-side*. The reader may wonder why this aspect is specified here since it may be considered as a server implementation problem which may be masked to the client. The originality of this concept is the possibility to access the *callee* remote thread through our `FutureClient` future object. The caller of a remote method invocation has a *direct* remote reference to the thread which runs its method allowing controls such as interrupting, joining, and so on. Thus, interrupting an asynchronous (remote) method invocation is naturally done with a call to `future.getCalleeThread().interrupt()`.

²In fact, the `call()` method of the active map is used, but end-users are strongly discouraged to use it

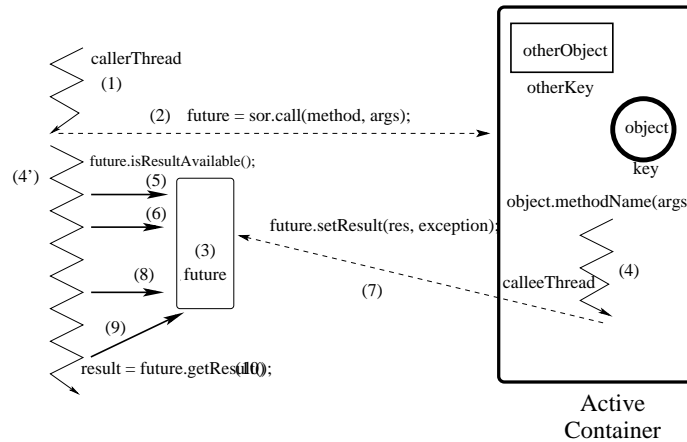


Figure 6.1: The `call()` asynchronous mechanism
 JACoB active map's `call()` method is asynchronous.

6.3 Client-side reflective asynchronous (remote) method invocation

The previous section describes the *server-side* reflective asynchronous (remote) mechanism based on the method `call()` of the active map³. When active maps are local, it is a sufficient mechanism for asynchronous communication between objects. But when they are remote, since the remote access point to an active map is its local *proxy*, any call to a method of the proxy (`put()`, `get()`, `remove()` or `call()`) requires that a message is sent to the corresponding active map over the network. This leads the caller to be blocked during the sending phase⁴.

Hence, when a client invokes `call()`, three threads are involved in the process :

- the client *caller* thread that invokes the `call()` method of the proxy of the remote active map;
- the server thread of the remote active map which polls the network and creates the *callee* thread;
- the remote active map's allocated *callee* thread which runs the requested method of the stored object.

So the active map's `call()` method is not *fully* asynchronous. It creates the *callee* thread, but does not avoid the *caller* thread to be blocked when invoking `call()` on the proxy which is sending the message to its related active map through the network. This method is *partially* asynchronous since the *caller* thread is blocked during the sending phase, but is not blocked during the execution of the called method.

³Or on the `call()` method of the `StoredObjectReference` class which is the same thing since the latter uses the former

⁴Since asynchronous input/output is only available in the new jdk 1.4

Using *client-side* asynchronism provides a solution where the *caller* thread is never blocked during a remote method invocation. In this section we shortly describe this mechanism which uses the RAMI package.

As we have seen in section 6.2, the RAMI package contains the `AsynchronousReference` interface which defines the method `call()`:

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

An asynchronous reference on an object provides reflective asynchronous method invocation on this object through the `call()` method. Adopting the following notation :

`O` : is a standard reference on a Java object,

`AR(O)` is an asynchronous reference on the object `O`.

Usually, implementation of the `AsynchronousReference` interface, provides *client-side* reflective asynchronous method invocation. The exception is the `StoredObjectReference` class which implements the `AsynchronousReference` in a *server-side* manner as described in section 6.2.

Consider now a usual *client-side* reflective asynchronous method invocation implementation. If `activeMap` is an active map remote proxy, then `AR(activeMap)` is an asynchronous reference on `activeMap`. This asynchronous reference provides *client-side* reflective asynchronous method invocation for all the methods defined in the `ActiveMap` interface (`call()`) and all of its subclasses (`java.util.Map`). For example, you may invoke the method `Collection values()` (defined in the `java.util.Map` interface asynchronously. If the map contains a lot of stored objects, this method may take some time to return the result, thus invoking it asynchronously allows the caller thread to invoke it as soon as possible while continuing its execution before really getting the result as shown in the program ??.

To deal with stored objects manipulation, the `StoredObjectReference` should be used and the following notation is adopted:

`SOR()` is an *invalid* stored object reference;

`SOR(O)` is a stored object reference on the object `O`. This implies that `O` is mapped to `key` in `activeMap` where `key` and `activeMap` are wrapped by the stored object reference *i.e.* the stored object reference is a *valid* reference.

`SOR(activeMap, key)` is a stored object reference which wrapped the pair `activeMap, key`; this stored object reference may be *invalid*.

Consider `sor` an instance of the `StoredObjectReference` wrapping the pair `(activeMap, key)`. The methods `sor.put()`, `sor.remove()` and `sor.get()` are all synchronous whereas `sor.call()` is *server-side* asynchronous.

Hence, if you want to insert an object asynchronously, you use the RAMI package as usual: making an asynchronous reference on an *invalid* stored object reference. So, you have an `AR(SOR())` object. Then, doing an asynchronous insertion of an instance of a `MyObject` class consists of the lines (ommiting exception handling which is not yet the focus point):

```

// Gets an invalid stored object reference: 'key' is not already mapped into
// 'activeMap'.
StoredObjectReference sor = StoredObjectReference.getInstance(activeMap, key);

// Gets an asynchronous reference on 'sor'
AsynchronousReference ar = // depends on implementation

// RAMI provides a better way to do that : see the notion of transparency in
// the RAMI documentation
Method putMethod = StoredObjectReference.class.getMethod("put",
    new Class[]{Object.class});

MyObject object = new MyObject();
// Asynchronous invocation of 'sor.put(object)'
FutureClient future = ar.call(put, object);

// This part is done concurrently to the invocation of 'sor.put(object)'
doSomethingElse();

// Wait for the termination of the asynchronous insertion
future.waitFor();

// 'object' had been inserted, we can continue.
continue();

```

Three distinct threads are involved in a *client-side* asynchronous remote call:

the client *caller* thread that invokes the *client-side* asynchronous method;

the client *sender* thread that runs the blocking method (here it is a method of a remote active map proxy⁵);

the active map's *server* thread that replies to incoming requests.

The client caller thread should know the status of its request as run by the sender thread : request still not processed, request being processed or request processed.

Having an AR(SOR(O)) provides asynchronous manipulation (put(), get(), remove()) of stored object. But what about the call() method which is already *server-side* asynchronous ?

The *client-side* asynchronous mechanism provided by AR(SOR(O)) extends the *server-side asynchronous* call() method to *full* asynchronism. One more thread is involved in *full* asynchronous remote method invocation : the active map's allocated *callee* thread that runs the method specified in the call() method. Thus, the client caller thread may control both the client sender thread and the active map's callee thread. For example, the caller may ensure the message has been sent by the client sender thread even if the callee thread is not created yet. This way, both sides of asynchronism (*client-side* and *server-side*) may be controlled by any client.

To invoke *fully asynchronously* the method f() of stored object O referenced by a stored object reference SOR, you must have a *client-side* asynchronous reference on the *server-side* asynchronous reference SOR(O): hence, an AR(SOR(O)). Then, the following lines:

```

// Gets a valid stored object reference: 'key' is already mapped into
// 'activeMap' to a stored object
StoredObjectReference sor = StoredObjectReference.getInstance(activeMap,

```

⁵Or a method of an instance of the class StoredObjectReference which uses the methods of a remote active map proxy

```

key);

// Gets an asynchronous reference on 'sor'
AsynchronousReference ar = // depends on implementation

Method fMethod = MyObject.class.getMethod("f",
    new Class[0]);

Method callMethod = StoredObjectReference.class.getMethod("call",
    new
    Class[]{Method.class,
    Object[].class});

// Full asynchronous invocation of 'f()' Client-side asynchronous invocation of
// 'sor.call("f", new Object[0])'
FutureClient future = ar.call(callMethod,
    new Object[]{fMethod, new Object[0]});

// This part is done concurrently to the invocation of 'sor.call("f", new
// Object[0])'
doSomethingElse();

// Wait for the termination of the client-side asynchronous method invocation
// It returns another 'FutureClient' object representing the server-side
// invocation of 'f()'
future = (FutureClient) future.waitForResult();

MyResult result = (MyResult) future.waitForResult();

// Use the result of the invocation of 'f()'
continue(result);

```

Writing full asynchronous remote method invocation is really inconvenient. Doing imbricated `call()` is non natural and error prone. Fortunately, RAMI provides *chained asynchronism* which solve this problem. An `AsynchronousReferencePair` transform a pair of asynchronous reference into an asynchronous reference: if you have an `AR(AR(O))` asynchronous reference, then applying an `AsynchronousReferencePair` noted `ARP` gives `ARP(AR(AR(O)))` and can be considered as an `AR(O)`. Since a stored object reference is an asynchronous reference, *i.e.* an `SOR` can be considered as an `AR`, having an `AR(SOR(O))` can be given to an `AsynchronousReferencePair` to have an `ARP(AR(SOR(O)))` which gives an `AR(O)`. The following lines shows how to do this:

```

// Gets a valid stored object reference: 'key' is already mapped into
// 'activeMap' to a stored object 'O' : 'SOR(O)'
StoredObjectReference sor = StoredObjectReference.getInstance(activeMap,
key);

// Gets an asynchronous reference on 'sor' : 'AR(SOR(O))'
AsynchronousReference ar = // depends on implementation

// Gets an asynchronous reference pair on 'ar' : 'ARP(AR(SOR(O))) -> AR(O)'
AsynchronousReferencePair arp = // depends on implementation

// RAMI provides a better way to do that : see the notion of transparency in
// the RAMI documentation
Method fMethod = MyObject.class.getMethod("f",
    new Class[0]);

// Full asynchronous invocation of 'f()'
FutureClient future = arp.call(f, new Object[0]);

```

```

// This part is done concurrently to the invocation of 'f()'
doSomethingElse();

// Wait for the termination of the fully asynchronous method invocation
MyResult result = (MyResult) future.waitForResult();

// Use the result of the invocation of 'f()'
continue(result);

```

For more details on the `AsynchronousReferencePair` functionality, see the RAMI documentation.

6.4 Exception handling

We consider two kinds of exceptions in a remote method invocation:

Remote method invocation exceptions are generated on the server side of the invocation, in the remote active map by the specified method; thus, they are handled in a per call basis as standard exceptions depending on the stored object's methods : an exception thrown by a matrix product method may not be handled as an exception thrown by a database query method;

others exceptions are exceptions which are not specific to the specified method; they are handled in a generic manner. For example a network exception may result in an attempt to contact an administrator.

JACOB provides two distinct mechanisms to handle both classes of exceptions.

Remote method invocation exceptions

These exceptions are generated on the server side, in a remote active map. Since this class of exception is related to *method invocation*, the mechanism used to handle them will be presented in the section ??.

Others exceptions

These exceptions are not specific to the specified method. Thus the caller may prefer a general exception handler. For example, when an active map refuse a method invocation (either, the server is down, or it may be overloaded, or for security reasons), the action to do may be to interrupt the caller (which may be waiting on the result or not), and to specify it to contact another active container. Sometimes, it may be useful to ignore them ! Network related exceptions for example, are ignored by JavaParty [21] (remote methods do not throw `RemoteException` as specified by RMI) because it is specifically designed for clusters where network is considered reliable.

This class of exceptions is problematic in *client-side* asynchronism only. In our framework, if the programmer uses only the *server-side* asynchronous `call()` method, any exceptions, other than remote method invocation related exception, are handled by the active container's *proxy*. Methods of the *proxy*'s may throw exceptions and must be enclosed in a standard *try/catch* statement.

Program 6.3.1 Asynchronous invocation of `java.util.Map.values()`

```
1 // Gets an activeMap (potentially remote)
  ActiveMap activeMap = // depends on implementation

  // Suppose 'useMap()' insert many objects into 'activeMap'
5 useMap(activeMap);

  // We must invoke 'activeMap.values()' soon. Since 'activeMap' contains many
  // objects, the method will probably some non neglectable times so we may
10 // invoke this method asynchronously

  // Gets an asynchronous reference on 'activeMap'
  AsynchronousReference asynchronousReference = // depends on implementation

15 // Prepare the asynchronous method invocation.

  // This can be done more easily : see the notion of transparency in the RAMI
  // documentation
  try{
20     Method valuesMethod = Map.class.getMethod("values", new Class[0]);
  }catch(NoSuchMethodException e) {
    e.printStackTrace();
    System.exit(1); // Assertion !! (May use 'assert' in jdk1.4)
  }
25 // Invoke the method 'values()' asynchronously.
  FutureClient future = asynchronousReference.call(valuesMethod, new Object[0]);

  // This part is done concurrently to the invocation of 'activeMap.value()'
30 doSomethingElse();

  // We need the result now !
  try{
35     Collection values = (Collection) future.waitForResult();
    useValues(values);
  }catch(Throwable t) {
    // Since 'Map.values()' do not declare any exception to be thrown, we may
    // consider that if an exception occurs during the asynchronous method
    // invocation, it is a RuntimeException. This can be verified with the
    // instanceof operator
40     if (!t instanceof RuntimeException) {
        t.printStackTrace();
        System.err.println("How a non RuntimeException " +
45             "instance had been thrown here ?");
        System.exit(1); // Assertion !! (May use 'assert' in jdk1.4)
    }
    handleRuntimeException(t);
  }
}
```

An asynchronous reference on an active map allows the asynchronous invocation of its `values()` method.

Chapter 7

Advanced techniques

7.1 Object caching

Caching of remote objects is a powerful mechanism that increases the performance of systems that range from file systems to distributed shared memories. If there is locality of access, caching a remote object at the client site can improve application performance because methods invoked on the object can be executed locally. As mentioned earlier, RMI does not provide such a functionality directly. Of course, a programmer can develop its own stub which can do object caching. But again, this is a static mechanism. The programmer must create its objects with that concern in mind.

Caching of objects is straightforward to achieve in JACOb with the method `get ()` of active containers as illustrated in figure 7.1.

Any object may benefit from object caching in JACOb. Moreover, programmers do not have to adapt their objects to use this service.

7.1.1 Security

**** TODO : Write something ! ****

7.2 Direct remote communication with stored objects using a custom protocol

**** TODO : Write something ! ****

7.2.1 Just in time security

**** TODO : Write something ! ****

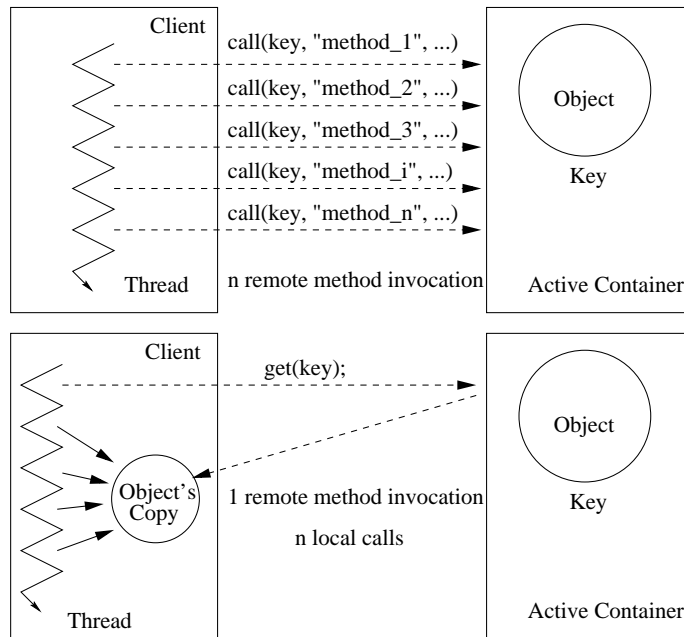


Figure 7.1: Object caching
Benefits of object caching provided by the `get ()` method of JACOb's active containers.

7.3 Services

**** TODO : Write something ! ****

7.3.1 Monitoring

**** TODO : Write something ! ****

7.3.2 Persistence

**** TODO : Write something ! ****

7.3.3 Transaction

**** TODO : Write something ! ****

Chapter 8

Conclusion and perspectives

8.1 Conclusion

This document has presented *JACOb* a framework that provides asynchronous invocation of dynamic remote object's method. It focused on *dynamic aspect* by being entirely based on the concept of *active container*. Remote objects in *JACOb* do not have to implement a special interface or inherit a special class or even to be compiled. Any object can become remote by being inserted into an active container. Any public object's method can be called in a *fully-asynchronous* way with *full* handling of exceptions.

8.2 Perspectives

A remote implementation of the *JACOb ActiveMap* class has been implemented in a straightforward manner with RMI. A new version based on a lower level protocol must be provided for efficiency.

A CORBA version will be implemented soon. Next, an UDP version may be tried as well as a TCP one. In fact, any *JACOb* is independent of the underlying protocol so any other protocol might be used such as BIP over Myrinet for high efficient communication.

An implementation of a security service is currently implemented. Much more is to be done such as persistence, transaction, naming, concurrency, messaging...

Bibliography

- [1] *Agent-Oriented Software Engineering*. Handbook of Agent Technology. AAAI/MIT Press, 2000.
- [2] The javagrande homepage, January 2001.
<http://www.javagrande.org/>.
- [3] Ken Arnold and James Gosling. *The Java programming language*. Addison-Wesley, 1996.
- [4] J. Chen and B. Leupen. Improving instruction locality with just-in-time code layout. In *USENIX Windows NT Workshop*, pages 25–32, August 1997.
- [5] M. Cierniak and W. Li. Just-in-time optimization for high-performance Java programs. In *Concurrency, Pract. Exp. (UK)*, volume 9 of *Java for Computational Science and Engineering - Simulation and Modeling II*, pages 1063–73, Las Vegas, NV, June 21 1997.
- [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] Al Geist, Adam Beguelin, Jack Dongarra, Jiang Weicheng, Robert Manchek, and Vaidy Sunderan. *PVM : Parallel Virtual Machine : A Users' Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Computation Series. The MIT Press, janusz kowalik edition, 1994.
- [9] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.
- [10] S. Hirano. HORB: Distributed Execution of Java Programs. In *Proc. WWCA'97*, volume Vol. 1274 of *LNCS*, pages 29–42. Berlin, springer edition, 1997.
<http://ring.etl.go.jp/openlab/horb/>.
- [11] Gosling James, Joy Bill, and Steele Guy. *The Java Language Specification*. Addison Wesley, 1996.

- [12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Videira Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Springer-Verlad, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of LNCS.
- [13] A. Krall. Efficient JavaVM just-in-time compilation. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 54–61, 1998.
- [14] Vijaykumar Krishnaswamy, Dan Walther, Sumeer Bhola, Ethendranath Bommaiah, George Riley, Brad Topol, and Mustaque Ahamad. Efficient implementations of Java Remote Method Invocation. In *4th USENIX Conference on ObjectOriented Technologies and Systems (COOTS'98)*, 1998.
- [15] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, 1999.
- [16] J. Maassen, R. Nieuwpoort, R. Veldema, H. Bal, and A. Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, July 1999.
- [17] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Narzul, and Marc Shapiro. Fragmented objects for distributed abstractions. In *Advances in Distributed Systems*. IEEE, 1992.
- [18] Hans Meuer, Erich Strohmaier, Jack Dongarra, and D. Simon Horst. Top500 super computers, January 2001.
<http://www.top500.org/>.
- [19] Christian Nester, Michael Pilippsen, and Bernhard Haumacher. A more efficient RMI for Java. In *Proceedings of Java Grande Conference*, pages 152–157, San Francisco, California, June 1999. ACM.
- [20] Michael Philippsen and Bernard Haumacher. More efficient object serialization. In *Parallel and Distributed Processing*, number 1586 in LNCS, pages 718–732, San Juan, Puerto Rico, April 1999. International Workshop on Java for Parallel and Distributed Computing.
- [21] Michael Pilippsen and Matthias Zenger. Javaparty - transparent remote objects in java. In *Concurrency: practice and experience*, volume 9, pages 1225–1242, 1997.
- [22] Sun microsystems. *Java Remote Method Invocation Specification*, 1998.
- [23] Sun microsystems. Java messaging services specifications.
<http://java.sun.com/products/jms/>, November 1999.
- [24] Sun microsystems. The Java HotSpot performance engine architecture. white paper, April 1999.
- [25] Tower Technology Corporation. Towerj - web server, January 2001.
<http://www.towerj.com/>.

- [26] J. Udell. Componentware. BYTE, pp.46-56, May 1994. Bibliography 167.
- [27] 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.