# A Framework for Seamlesly Making Object Oriented Applications Distributed

Serge Chaumette[a] and Pierre Vignéras[a].
email: {Serge.Chaumette, Pierre.Vigneras}@labri.fr

[a] LaBRI, Laboratoire Bordelais de Recherche en Informatique,
Université Bordeaux 1,
351 Cours de la Libération,
33405 TALENCE CEDEX,
FRANCE

Traditional distributed frameworks such as DCOM, CORBA or EJB do not allow any object to become remote *dynamically* even if they were not designed for it. This is a major drawback solved by the *active container* concept presented in this article. The Java™ [1] implementation of this concept, JACOb, is also presented.

**keywords:** active container, dynamism

## 1. INTRODUCTION

Distributed applications are more and more often developed using object oriented languages even if other programming techniques such as agent-oriented applications [6] or aspect oriented programming [7] emerge. Three *de facto* standard exist: DCOM™ [4] (Distributed Component Object Model) – Microsoft™, CORBA [11] (Common Object Request Broker Architecture) – OMG, and EJB™ [12] (Enterprise Java Beans) – Sun microsystems™. Even though these standard offer similar functionalities they have the same lack: objects must be written specifically as remote objects to be used remotely. Moreover, the “remote” behavior depends on the framework used. A DCOM object cannot be used in an EJB environment and vice-versa. The protocol used for communication is directly imported from the underlying framework and cannot be changed preventing one to choose between *efficient* versus *safe* or *secure* protocol. Some implementors try to solve this problem by writing some bridge between “naturally incompatible” worlds such as RMI/IIOP, but we believe this is not the good approach.

A new paradigm must be used to allow any object to become remotely accessible *dynamically*. Several challenges must be solved in order to achieve this goal: remote objects must deal with issues usually not found in the local case such as *latency*, *memory access*, *partial failure* or *concurrency* [16]. Moreover, a clear separation between remote object management and remote object communication must be found.

This paper presents the *active container* concept and its implementation in Java: JACOb. JACOb is the server-part of the Mandala project [15] which focuses on dynamic distribution of objects using the Java reflection facility. The rest of this article is organized as follows: we first propose in section 2 a new paradigm for distributed programming that we call: *active containers*. Problems related to remote objects are dealt with in section 3. Then JACOb, our Java implementation of the concept is presented in section 4. Conclusion and future works are the topic of section 5.

## 2. THE ACTIVE CONTAINER CONCEPT

An active container is a container of objects which provides a communication channel for any of its object. It supports the following main methods:

| | |
|---|---|
| `void put(Object key, Object object)` | : insert an object in the container; the inserted object becomes a *stored object*; |

[1] *Java* 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.*

| | |
|---|---|
| `void remove(Object key)` | : remove an object from the container; the object is no more a *stored object*; |
| `Object get(Object key)` | : get a copy of an object from the container; the *stored object* remains in the container; |
| `void call(Object key, String method, Object[] args, MethodResult result)` | : call a method of a *stored object*; |

The method `call()` generates the activity. It invokes the given method on the stored object that is mapped to the given `key` in the active container. To support dynamism, the method is specified as an instance of a `Method` meta-class and the implementation uses reflection to resolve it. This leads to the problem of *strong type checking*. This problem is discussed in section 4.4.1. The fact that reflection is used makes it possible to work at the level of the meta-model, reasoning about all applications that can be written using the framework, instead of dealing with a specific one. This leads to a compact model [14] in $\pi$-calculus [9] of the framework.

A separation between objects management and method invocations is *de facto* provided by the active container model: the active container is a *repository* of objects and is used for object management (deployment, upgrade, deletion); the `call()` method provides the *communication channel* with stored objects. This separation is known to be a good design and is widely used for example in the EJB framework that defines *EJB containers*. Nevertheless, these containers can only contain specific objects (*beans*) whereas in our model, stored objects are really *any* object. Javaspace [13] also uses the container abstraction but does not provide the communication channel to use stored objects remotely. Instead, objects must be retrieved locally to be used.

## 2.1. Usage examples

There are many applications that can be developed in a straightforward manner using the framework of active containers. This section presents some of them which we believe illustrate the ease of use of the concept and its capacity to model some interesting applications in the distributed world.

### 2.1.1. Active containers as a memory model

This concept can be seen as a memory model of a distributed virtual machine where the objects seen by the end-user are always *stored objects*: `put()` provides a mechanism to create objects; `remove()` provides a mechanism to delete objects; `call()` provides a mechanism for method invocation. This model may be used to transparently distribute a local application as explained in section 3.2

### 2.1.2. Active containers and mobility

The active container concept is powerful enough to express the migration of mobile agents [5]. A `stopActivity()` method can be invoked on an object – in case strong migration is not possible. The object can then be retrieved using `get()` and moved to another location using `put()` and eventually restarted calling a dedicated method, for instance `restartActivity()` as shown in the program 1.

**Program 1** Mobile agents expressed using active containers

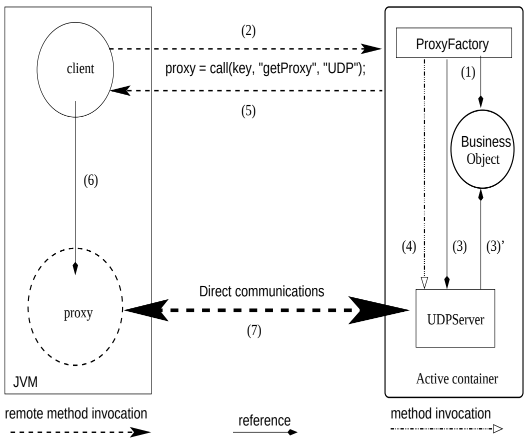
```
fromActiveContainer.call(key,
                         "stopActivity",
                         null,
                         null);
MyObject object =
    (MyObject) fromActiveContainer.get(key);
fromActiveContainer.remove(key);
toActiveContainer.put(key, object);
toActiveContainer.call(key,
  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

Note that in such a case, the object is moved twice. First, on the host which invokes the `get()` method, and then on the host where the `put()` is actually performed. If a direct move between two hosts is needed, a previously inserted `Migrator` object which contains a `migrateTo()` method may be used.

### 2.1.3. Active containers and multi-protocols remote objects

All communications with stored objects has to go through the active container they live in. This implies that the protocol used to communicate with a stored object is the protocol used to communicate with the remote active container. Using the remote active containers model, we can provide a mechanism for direct communication with stored objects with any protocol such as RMI, IIOP, TCP, UDP, etc. For example, clients outside of a LAN must use a secure, reliable internet protocol such as SSL/TCP/IP to access an object whereas others clients on the LAN would use a faster protocol such as BIP/Myrinet.

For this purpose, an object which provides new remote access to a stored object may be inserted in an active container. Consider an instance of a `ProxyFactory` class which delivers *remote proxies* to a business object it has a local reference on. Clients use its method `getProxy()` which takes one parameter: the protocol the returned proxy must use to transform method invocations into remote method invocations. The method may instantiate a server if necessary.
Then, the following figure illustrates how multi-protocol can be achieved using the active container concept. Suppose the `ProxyFactory` – which has a local reference to the business object it must return proxies on – has already been inserted (`put()`) in a remote active container (1). The method `getProxy()` is invoked remotely through the `call` method of the active container (2) by a client. The proxy factory may instantiate a new server for the given protocol (`"UDP"` here) if necessary (3) and the local reference to the business object (3)' is given to him by the factory. Then the server related proxy is returned (4) to the client. The client then has a reference on the returned proxy (5) and direct communications can take place (6) using the specified protocol.



Such a mechanism is proposed for security purpose in the JITS (Just In Time Security) [3].

## 3. DEALING WITH REMOTE OBJECTS

Distributed computing contains some well-known issues [16] we believe the active container concept solves. These problems are: *latency*, *memory access*, *partial failure* and *concurrency* and are dealt with separately below.

### 3.1. Latency

Latency is defined by the delay between a method invocation and its effective execution. The latency introduced by the network on remote method invocation is several times the order of magnitude of the local method invocation's one. If this issue is not addressed, a distributed application will poorly perform.

This problem can be solved by an object oriented program analysis (either static, dynamic or both) which maps instances to active containers (on distinct hosts) in such a way that the overall communication cost becomes minimal. Some work in this direction is currently done in our research team using the notion of *separable objects* [2].

### 3.2. Memory access

The memory access problem is the fact that a remote access is not exactly the same as a local access. For example, when a method invocation occurs, parameters must be marshaled in order to be transfered to the remote host. This implies a *call by copy* schema which is different from the regular *call by reference*. Using the active container concept, and defining a *stored object reference* as the pair $(activeContainer, key)$ it is possible to enforce developers to express which schema they want for their method invocations:

- every reference used in the program must be a *stored object reference*[2] and communication between

[2] This can be verified by a compiler.

*stored objects* must be done through the `call()` method exclusively; this ensure a *call by reference* mechanism;

- when a *call by copy* is needed, the `put()` method of active containers must be used – in the same way the `clone()` method is used in the Java language for the same purpose in the local programming case.

### 3.3. Partial failure

Partial failure is one of the main characteristic of distributed computing. In the local case, failure is either total or detectable (by the operating system for example). In the remote case, a component (network link, processor) may fail while others continue to work properly. Moreover, there is no central component that can detect the failure and inform others, no global state which describes what exactly occurred. For example, if a network link is down, a previous remote method invocation may have completed successfully while it is probably not the case if the processor is broken.

Nevertheless, the concept does not specify how remote failure must be handled (since, in fact, the concept don't specify that active containers are remote objects). Hence, it is up to the implementation to provide a remote failure handling mechanism. We will see our proposition in section 4.1.

### 3.4. Concurrency

Remote objects must usually deal with concurrent method invocation and this prevents non *thread-safe* objects from becoming remote without care. As in the partial failure case, the concept does not specify the semantic of method invocation on the active container nor on its stored objects. The implementation is free to provide some mechanism that enable non *thread-safe* objects to be inserted into remote active container and thus to be accessed remotely. The process used in our implementation will be discussed in section 4.2.

## 4. THE JACOb FRAMEWORK FOR JAVA DISTRIBUTED APPLICATIONS

JACOb (Java Active Container of Objects) is a Java implementation of the above concept. Active containers are defined by the `ActiveMap` interface which inherits `java.util.Map` and defines the `call()` method.

Several implementations of this API are available. A local implementation allows the active container concept to be used locally, stored objects are accessed from the virtual machine it is running on. Remote implementations support distributed applications providing remote access to the objects that are inside the container. Active containers make it straightforward to provide this feature: it is enough to make the container usable remotely, and this directly provides remote access to stored objects.

From the server side of view, the design of the framework – which uses the *design by interfaces* scheme intensively – makes straightforward the implementation of a new transport layer such as CORBA, or Myrinet [10]. Currently, three protocols are implemented: UDP, TCP and RMI. A JToe [1] based implementation is planned.

From the client point of view, the use of the JNDI [8] API prevents the user from being dependent of a particular transport layer. Hence, for the end-user, using a JACOb remote `ActiveMap` is as easy as using a conventional `Map` where objects are stored by value[3]

### 4.1. Remote failure handling in JACOb

As mentioned in section 3.3, partial failure is one of the main characteristic of distributed applications and dealing with this issue must be done at the interface level [16].

Java-RMI uses a *per-method basis* for this purpose: each method handled by a remote object must declare the `RemoteException` in its `throws` clause. This limits code reuse since declared exceptions are taken into account in Java by the compiler: an object with interface `A` cannot become remote easily. Composition must be used since the object has not be developed to be remote. The interface `B` of the remote object declares each method of `A` with `RemoteException` added to the `throws` clause. Hence, a client which previously used `A` cannot use `B` without being rewritten.

We believe an *event-driven based* approach is more convenient since usually remote failure are handled the same way: contacting the administrator, hiding the problem by contacting a mirror server, etc...

In JACOb, each remote object[4] has a related `ExceptionHandler` which handle exceptions thrown by the transport layer. This handler may resolve the problem transparently re-invoking the failed called method

[3]The serialization mechanism is used. One may claim that *dynamism* is broken since *any* objects can't be inserted into a remote active container, only serializable objects can.

[4]`ActiveMap` is one among many interfaces – such as `Map` or `Collection` – JACOb also provides a remote implementation.

on a mirror for example (the caller will never notice that an exception occurred), contact an administrator or let JACOb throws a `RuntimeException` to the caller.

### 4.2. Concurrency in JACOb

As described in section 3.4, remote objects must deal with concurrent access. Since JACOb allows any object to be inserted in a remote active container, any object may become remote, even non *thread-safe* objects. As mentioned in the introduction, JACOb is the server part of the Mandala project. The client part of it – called RAMI– solves this problem. RAMI stands for Reflective Asynchronous Method Invocation and defines *asynchronous references* on which method invocation are always asynchronous. RAMI provides different *asynchronous semantics* to be used with *asynchronous references* such as *concurrent semantic* – where method invocations are executed concurrently – and *single threaded semantic* – where method invocations are never executed concurrently. JACOb defines the `StoredObjectReference` class which is in fact a RAMI asynchronous reference implementation: any method on a stored object may be invoked asynchronously (and remotely if the active map containing the object is remote). Hence, when a *single threaded* semantic is attached to a non *thread-safe* object by its instanciator, the object can be accessed asynchronously safely and thus can be inserted into a remote active container.

### 4.3. JACOb advantages

#### 4.3.1. Dynamism

Objects can dynamically become remotely accessible just by being inserted into an active container. This feature implicitly provides *legacy code reuse* since classes dot not have to implement or extend anything to be remotely accessible and *separation of concerns* since objects do not have to be designed with the remote concern in mind, developers can then focus on the “business” parts of their objects.

#### 4.3.2. Protocol independence

Using a remote object in JACOb does not depend on the underlying protocol, since communication is handled at the level of the active container. Hence, some protocol may be used for efficiency reasons (such as UDP or Myrinet) while others may be used for their guarantees (TCP, RMI or HTTPS).

#### 4.3.3. Asynchronism

The `ActiveMap.call()` method provides *server-sided* asynchronous remote method invocation[5]. Applications can then take advantage of their intrinsic parallelism.

### 4.4. JACOb drawbacks

#### 4.4.1. Strong type checking

If the method `call()` of the active container model is used directly to invoke methods of objects it contains, the compiler cannot check types since the method is a *meta-object*: the link between the method and the object it is invoked on is computed at runtime.

Nevertheless this is not the natural way of dealing with remote method invocation in JACOb. The client part of Mandala– the RAMI package which provides reflective asynchronous method invocation – must be used instead of the `ActiveMap.call()` method which is only used by implementors. Hence, end-users use a natural syntax to invoke their method remotely and asynchronously solving the strong typing problem. Describing the RAMI framework is far beyond the scope of this article.

#### 4.4.2. Inefficiency

Since JACOb uses the reflection mechanism, method invocation may be quite slower than invoking a method locally as usual. But since active containers are usually remote, the cost of the reflection is negligible compared to the cost of communication. Furthermore, RMI itself uses reflection since JDK 1.2 and thus this overhead is unfortunately common over the different frameworks. Moreover, the performance of a remote method invocation depends on the underlying transport layer. The UDP implementation is several times more efficient than the RMI one, but dot not offers the same guarantees.

Note anyway that invoking a method in JACOb is asynchronous (on the server-side) what reduces the impact of the reflection overhead on the invoker.

[5] Mandala defi nes *server-side* and *client-side* remote method invocation. This fi rst is when the client is blocked until the invoked method is guaranteed to start its execution, the latter is when the client is not blocked at all.

## 5. CONCLUSION AND FUTURE WORK

The concept of *active container* provides an abstraction that can be used as a programming paradigm for distributed object applications. Remote active containers provide a clear separation between *stored object* management and communication channels. Moreover stored objects can be *any* object which enable *dynamism*. The concept solves issues traditionally found in distributed computing.

Our implementation of this concept in Java – called JACOb– is the server-part of the Mandala project which provides *reflective asynchronous remote method invocation*. Any object can *dynamically* become remote and be accessed asynchronously. Several protocols may be used in JACOb, and other implementations are easy to achieve.

The model is underspecified (remote active containers, partial failure, method invocation semantics (copy, reference, asynchronism)) allowing implementations to provide “goodies”. Perhaps shall it be more specified to be really usable for proof of program behavior. Further study on the model may answer this question.

Future works include integration of security features into JACOb, development of services, customized distributed garbage collector and group communications.

We also plan to produce measurements for a set of applications in order to give potential users a good idea of what they can expect from our framework.

## REFERENCES

[1] Serge Chaumette, Benoît Métrot, Pascal Grange, and Vignéras Pierre. Jtoe: a Java API for Object Exchange. To appear in Proc. of PARCO’03, September 2003.

[2] Serge Chaumette and Grange Pascal. Optimizing the execution of a distributed object oriented application by combining static and dynamic information. International Parallel and Distributed Processing Symposium, IPDPS 03, April 2003. Nice, France. Poster.

[3] Serge Chaumette and Pierre Vignéras. Extensible and Customizable Just-In-Time-Security (jits) management of Client-Server Communication in Java. To appear in Proc. of PARCO’03, September 2003.

[4] "Microsoft Corporation". Dcom technical overview. Web page, November 1996. `http://msdn.microsoft.com/library/en-us/dndcom/html/msdn_dcomtec.asp`.

[5] Carlo Ghezzi and Giovanni Vigna. Mobile code paradigms and technologies: A case study. In *Proceedings of the First International Workshop on Mobile Agents*, Berlin, Germany, 1997.

[6] N. Jennings and M. Wooldridge. *Agent-Oriented Software Engineering*. Handbook of Agent Technology. AAAI/MIT Press, J. Bradshaw, edition, 2000.

[7] 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*. Springer-Verlad, 1997.

[8] Sun Microsystems. Java naming and directory interface (jndi). Web page, 2003. `http://java.sun.com/products/jndi/`.

[9] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes (parts i and ii). Technical report, Laboratory for Foundation of Computer Science, Computer Science Department, Edinburgh University, June 1989.

[10]Myricom. Myrinet link and routine specification, 1995. `http://www.myri.com/myricom/document.html`.

[11]Jon Siegel. *CORBA, Fundamental and Programming*. Wiley, 1996.

[12]Sun microsystems. *Enterprise JavaBeans Specification*, Juin 2000. Version 2.0, Public Draft.

[13]Sun Microsystems. Javaspace - web server. `http://java.sun.com/products/javaspaces/`, 2000.

[14]Pierre Vignéras. Agents mobiles: une implémentation en Java et sa modélisation, Juin 1998. Mémoire de DEA.

[15]Vigneras, Pierre. The mandala web site. Web page, November 2002. `http://mandala.sf.net/`.

[16]Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. In *Mobile Object Systems: Towards the Programmable Internet*, pages 49–64. Springer-Verlag: Heidelberg, Germany, 1997.