

# A Framework for Seamlessly Making Object Oriented Applications Distributed

---



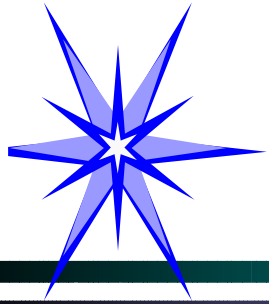
Serge Chaumette and Pierre Vignéras

Distributed Systems and Objects team

LaBRI, Université Bordeaux 1, France

*{chaumett, vigneras}@labri.fr*

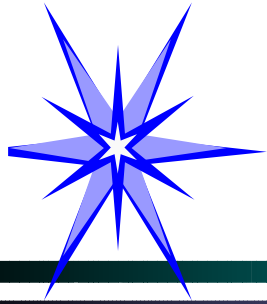
PARCO'03 Dresden, September 02-05 th



## Aim of the talk



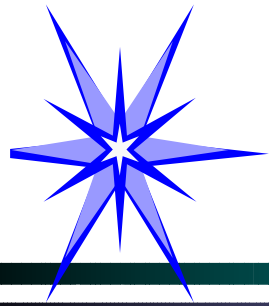
***Presents a new programming paradigm which addresses some issues commonly found in distributed programming***



# Outline



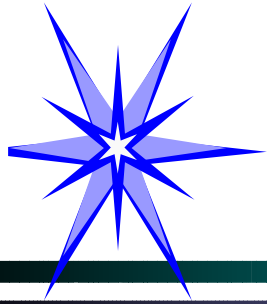
- Problem targeted : *dynamism*
- The *active container* concept proposition
- JACOb: implementation in Java of the active container concept
- Conclusion and future work



# Common framework in distributed programming



- Designing a distributed application require a clear separation between non-functional and functional property
- Main constraint imposed by today's standard (RMI/EJB, CORBA, DCOM )
  - ◆ Objects must be designed for a specific framework to be used remotely
- ➔ ... but remote access is clearly a non-functional property !



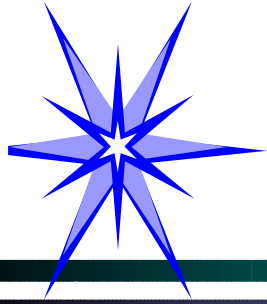
# Focusing on dynamism



Provides a solution that allows any object to be accessed remotely

## Constraints

- Objects do not have to be designed remote (no interface to implement, no class to extend)
- Provides a clear separation between objects management (deployment, migration, cloning) and object use (referencing, remote method invocation)



# The *active container* concept

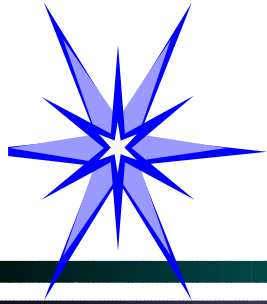


## A container of objects

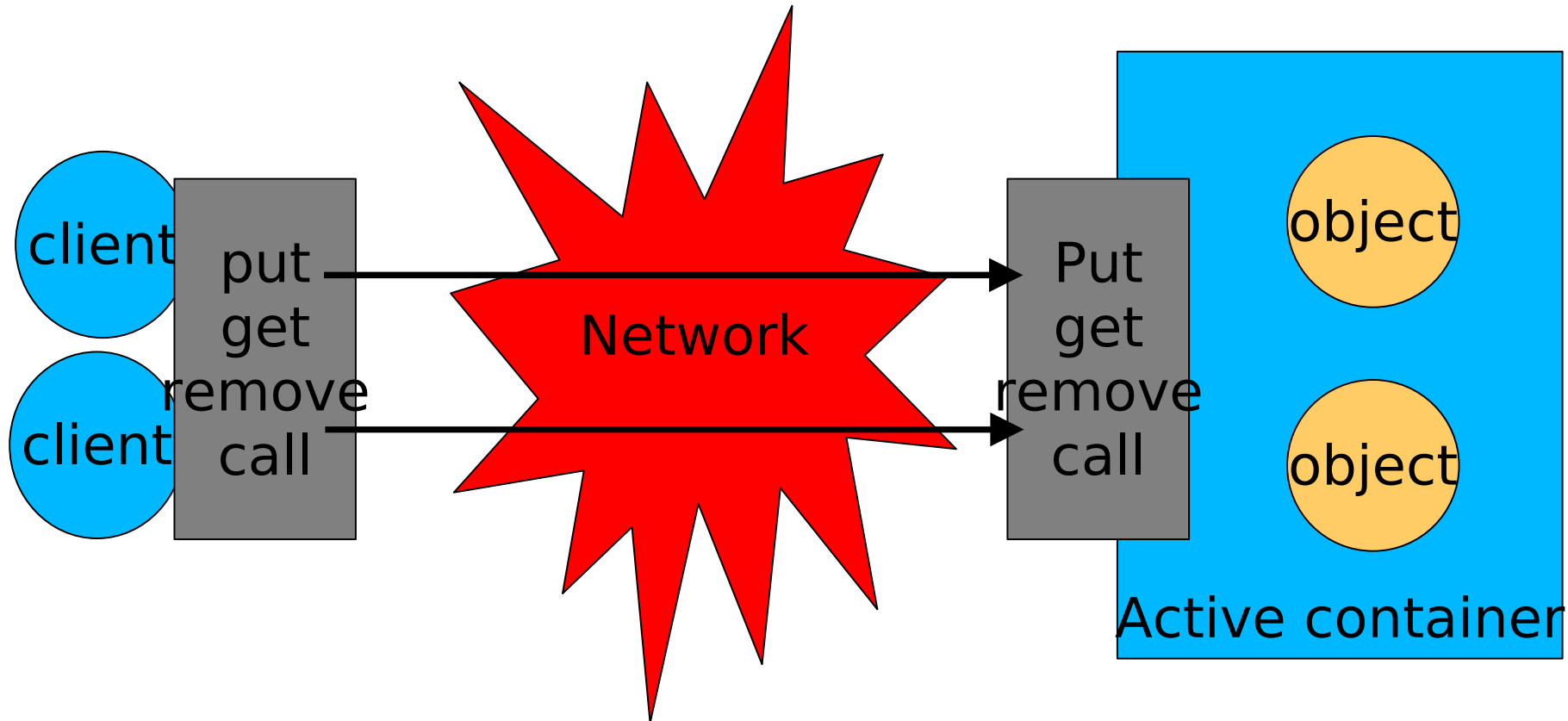
- Objects in the container are called *stored objects*

## Four methods

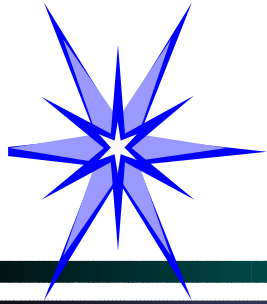
- ◆ `Object put(Object key, Object object)`  
`Object remove(Object key)`
- ◆ `Object get(Object key)`
- ◆ `void call(Object key,  
Method method,  
Object[] args,  
MethodResult result)`



# The *active container* concept



**Any object can become *remote* dynamically by being inserted into a *remote* active container**



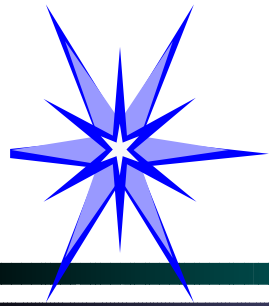
# The *active container* concept



Stored objects management through a widely used and understood Map interface (`put()`, `get()` and `remove()`)

The `call()` method is the communication channel to stored objects

*A stored object reference* is defined by the pair  
(*activeContainer*, *key*)



# The *active container* concept



The active container have been modeled in pi-calculus

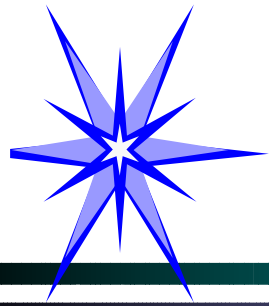
The model is expressive

agent.migrate(destination) can be modeled has:

```
from.call(agentID, stopActivityMethod, null, null);  
Agent agent = (Agent) from.remove(agentID);  
to.put(agentID, agent);  
to.call(agentID, startActivityMethod, null, null);
```

Note that the agent is transfered two times on `remove()` and on `put()`

Use a special `Migrator` stored object to address this problem



# Issues related do distributed objects



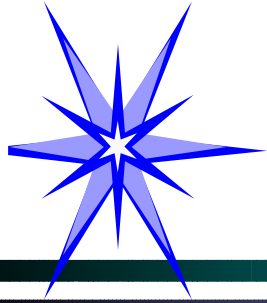
## Latency

- Remote method invocation is much slower than local method invocation

## Poor performance

Use static or dynamic analysis (or both) to map communicant objects on the same active container

- ➔ Easy to achieve even at *runtime* since any object can become remote *dynamically*
  - Load balancing by objects move/copy



# Issues related do distributed objects



## Memory access

- Method invocation semantic is usually *call by reference* in the local case while it is often *call by copy* in the remote case

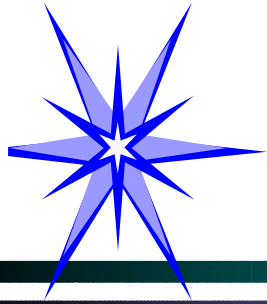
## Difficult to use

Force developers to use *stored object references* exclusively

- ➔ Easy to achieve by a compiler

*Call by reference* both in local and remote case

Use `put()` as `clone()` in the local case when *call by copy* is required



# Issues related do distributed objects



## Partial failure

- Network link failure is indistinguishable from a processor failure from a remote client point of view.

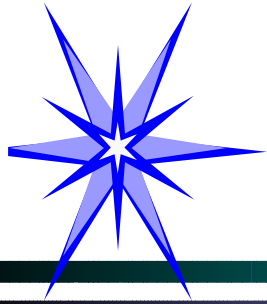
## Fault tolerance difficult to achieve

Remote nature of objects must be seen at the interface level, it is not just an implementation detail

- ➔ Example of a local interface extended to the remote case  
: NFS

*soft* versus *hard* mount

None are satisfactory, *soft mount* is almost never used, *hard mount* is responsible of hang-up



# Issues related do distributed objects



## Concurrency

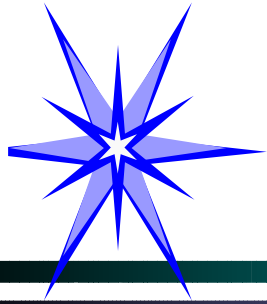
- Remote objects must deal with concurrent access

## Some objects are not *thread-safe*

- A "sequentializer" proxy that ensures only one thread can execute the code of a *non-thread safe* object can be inserted into the active container instead of the *non-thread safe* object itself.

Easy to achieve by extending the object class for example (PRO-Active technique).

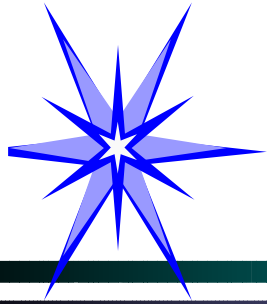
- ➔ Inserting the proxy into the remote active container makes it remote and its "business" object too.



# JACOb overview



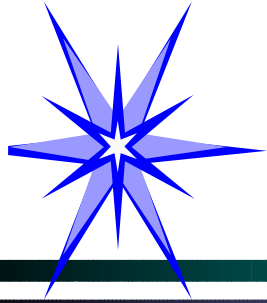
- Implementation in Java of the *active container* concept
  - ◆ JACOb: Java Active Container of Objects
  - ◆ Active containers are represented by the `ActiveMap` interface which extends `java.util.Map`
    - Easy to learn
- The `call()` method is asynchronous
  - Distribution and asynchronism provides parallelism



# JACOb overview



- Server view
  - several implementations of `ActiveMap`  
local, RMI, TCP, UDP
  - New transport layer implementation is straightforward to implement (*designed by interface*)
- Client view of `ActiveMap`
  - Use JNDI to retrieve `RemoteActiveMap`
  - Do not depends on the transport layer
  - Used for objects management **only**
  - Stored objects method invocation must be handled by an upper layer



# JACOb: remote failure handling



- Remote failure must appear at the interface level  
*Per method basis: (Java-RMI approach)*

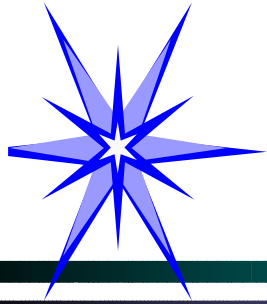
each method declares a remote exception may occurred

- Break polymorphism
- Incompatible with *dynamism*

- *Event-driven basis: (JACOb approach)*

Any remote object has an `ExceptionHandler` instance used to handle transport exceptions

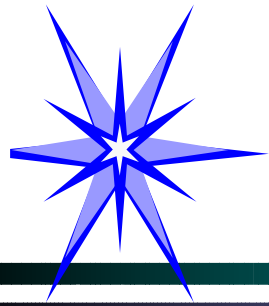
- Can silently recover the state (using a mirror for example)
- Can tell JACOb to raise an exception to the caller



# JACOb position in the Mandala project



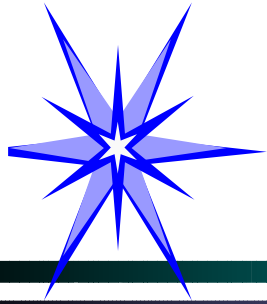
- *Server-part* of the **Mandala** project which provides *reflective asynchronous remote method invocation*
  - Defines `ActiveMap`
  - Provides transports layers
- Clients uses the *client-part* of Mandala (RAMI)
  - Defines *asynchronous references*  
`AsynchronousReference` interface
    - JACOb implements this interface with the `StoredObjectReference` class
  - Defines *asynchronous semantics*  
Concurrent (Threaded, ThreadPooled): performance  
**Single threaded**(Fifo, Random) : **for non *thread-safe* objects**



# JACOb position in the Mandala project



- Clear separation between
  - objects management: `ActiveMap`
  - Remote method invocation: `StoredObjectReference`
  
- Solves the *strong typing problem*
  - The method to invoke in `ActiveMap.call()` is specified via the `java.lang.reflect.Method`
  - ➔ `ActiveMap.call()` is intended for implementors only
  - ➔ Clients call their methods asynchronously and remotely with a natural syntax thanks to *transparency*



# Transparency

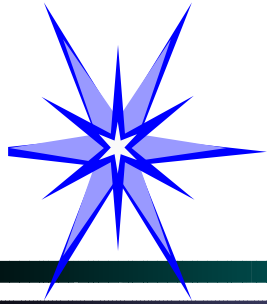


- A **jayac** compiler produces *asynchronous proxy*
  - ➔ Provides asynchronism through the use of an *asynchronous reference*
  - ➔ Provides strong typing

```
package p;  
public class A  
    extends B  
    implements C {  
  
    int f(String s) {  
        ...  
    }  
}
```

**jayac** →

```
package jaya.p;  
public class A extends jaya.p.B  
    implements jaya.p.C {  
  
    AsynchronousReference ar;  
  
    FutureClient rami_f(String s) {  
        // use ar to invoke f()  
        // asynchronously  
    }  
}
```



# JACOb sample code



- Getting a remote active map proxy

```
ActiveMap activeMap = null;
```

```
String URL = "rmi://host/RMIActiveMap"; // may be LDAP
```

```
// JNDI part
```

```
try{
```

```
    javax.naming.InitialContext context =
```

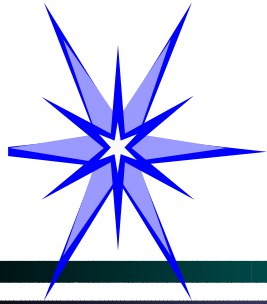
```
        new javax.naming.InitialContext();
```

```
    activeMap = (RemoteActiveMap) context.lookup(URL);
```

```
}catch(NamingException ne) {
```

```
    ...
```

```
}
```

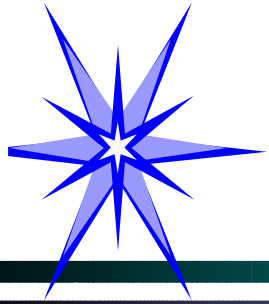


# JACOb sample code



## ➤ Remote exception handling in JACOb

```
activeMap.setExceptionHandler(new ExceptionHandler() {  
    public HandledException handleException(Object info) {  
        ExceptionInfo ei = (ExceptionInfo) info;  
  
        // Get informations on the exception raised  
        Method m = ei.getMethod();  
  
        // Use this information: if it is the method 'remove()' that  
        // failed then return 'null' and resume the caller  
        // Otherwise, raise an exception in the caller thread  
        return (m.equals(RemoteMap.removeMethod)) ?  
            new HandledException(null) :  
            HandledException.RAISE;  
    }  
}
```



# JACOb: sample code



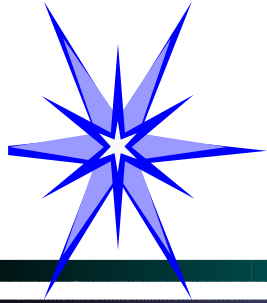
- Creating a stored object transparently

```
// The Mandala framework will create StoredObjectReference in  
// 'activeMap' using the given asynchronous reference factory  
Framework.setFactory(new SORFactory(activeMap));
```

```
// Create a stored object transparently  
// (A new key is automatically created)  
jaya.java.math.BigInteger i = new  
jaya.java.math.BigInteger("1234567890");
```

```
// 'i' is now a stored object in 'activeMap'  
StoredObjectReference sor = i.getAsynchronousReference();  
assert activeMap.containsKey(sor.getKey());
```

```
// Our BigInteger is remote whereas it has not be designed for it!  
assert activeMap instanceof RemoteActiveMap;
```



# JACOb: sample code



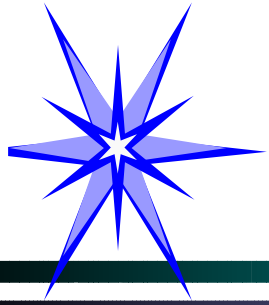
- Creating a stored object manually  
(when you are not the instancier for example)

```
// A library returns an object I want to use remotely  
java.math.BigInteger bi = library.foo();
```

```
// Insert 'bi' in a remote active map  
Object key = StoredObjectReference.getGlobalUniqueKey();  
activeMap.put(key, bi);
```

```
// Get a stored object reference  
StoredObjectReference sor =  
    StoredObjectReference.getInstance(activeMap,  
key);
```

```
// Get an asynchronous proxy on this stored object  
jaya.java.math.BigInteger i =  
jaya.java.math.BigInteger.getInstance(sor);
```



# JACOb: sample code



- Using an asynchronous proxy (and performing asynchronous remote method invocation)

```
jaya.java.math.BigInteger i = ... // As in preceding examples

// Invoke a method remotely and asynchronously
FutureClient future = i.isProbablePrime(CERTAINTY);

doSomethingElse();

try{
    boolean isPrime = future.waitForResult();
} catch (TransportException te) {
    // Handle if necessary
}
```