ProActive:
Distributed and Mobile Objects for the GRID

Denis Caromel, et al.
www.inria.fr/oasis/ProActive

OASIS Team
INRIA -- CNRS - I3S -- Univ. of Nice Sophia-Antipolis, IUF
Vers. October 9 th 2003

- Remote Objects, Asynchronous Method Calls, Futures,
- Group Communications, Mobile Objects,
- Graphical Interface (IC2D), XML Deploiement,
- Interfaced with Globus, rsh, ssh, LSF, RMIregistry, Jini

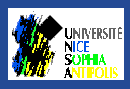


Table of Contents

1. ProActive Basic Model, Features, Architecture, and Tools
   - 1.1 Basic Model
     - 1.1.1 Active Objects, Asynchronous Calls, Futures, Sharing
     - 1.1.2 API for AO creation
     - 1.1.3 Polymorphism and Wait-by-necessity
     - 1.1.4 Intra-object synchronization
     - 1.1.5 Optimization: SharedOnRead
   - 1.2 Collective Communications: Groups
   - 1.3 Architecture: a simple MOP
   - 1.4 Meta-Objects for Distribution
   - 1.5 Abstract Deployment Model
   - 1.6 IC2D: Interactive Control & Debug for Distribution
   - 1.7 DEMO: IC2D with C3D : Collaborative 3D renderer in //

Table of Contents (2)

2. Mobility
- 2.1 Principles:
  Active Objects with: passive objects, pending requests and futures
- 2.2 API and Abstraction for mobility
- 2.3 Optimizations
- 2.4 Performance Evaluation of Mobile Agent
- 2.5 Automatic Continuations

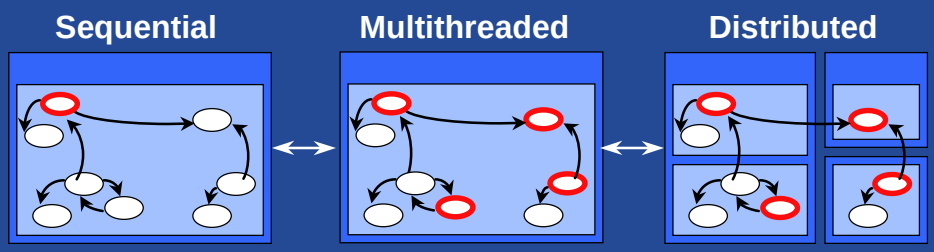
ProActive:
A Java API + Tools for Parallel, Distributed Computing

- A uniform framework: **An Active Object pattern**
- A formal model behind: **Prop. Determinism, insensitivity to deploy.**

Main features:
- **Remotely accessible Objects (Classes, not only Interfaces, Dynamic)**
- **Asynchronous Communications with synchro: automatic Futures**
- **Group Communications, Migration (mobile computations)**
- **XML Deployment Descriptors**
- **Interfaced with various protocols:** `rsh,ssh,LSF,Globus,Jini,RMIregistry`
- **Visualization and monitoring: IC2D**

In the www. ObjectWeb .org Consortium (Open Source middleware)
since April 2002 (LGPL license)

ProActive PDC
Objectives and Rationale

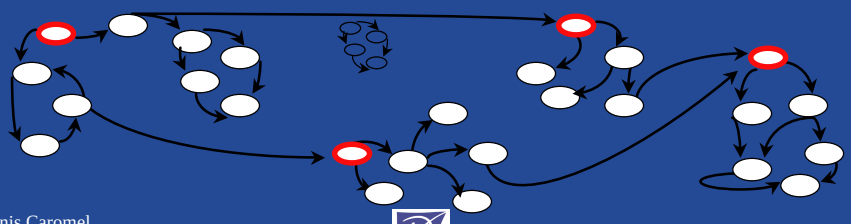


- Most of the time, activities and distribution are not known at the beginning, and change over time
- Seamless implies reuse, smooth and incremental transitions

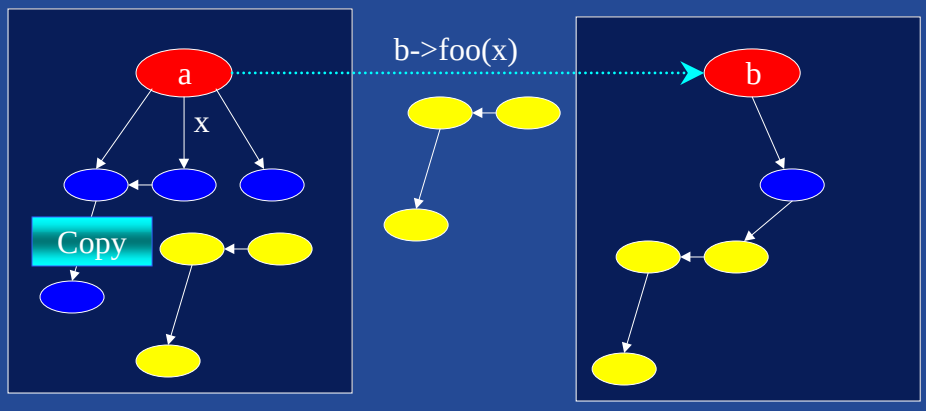
Library !

ProActive : model

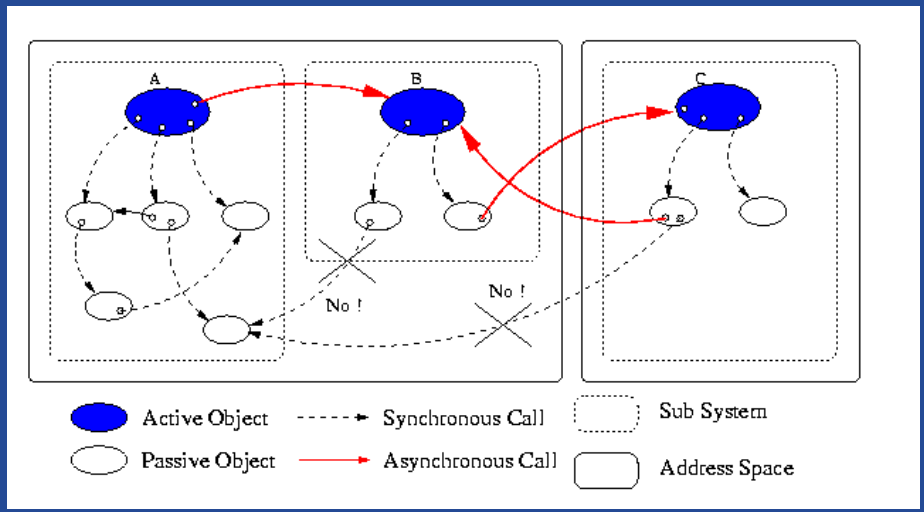
- Active objects : coarse-grained structuring entities (subsystems)
- Each active object: - possibly owns many passive objects
  - has exactly one thread.
- No shared passive objects -- Parameters are passed by deep-copy
- Asynchronous Communication between active objects
- Future objects and wait-by-necessity.
- Full control to serve incoming requests (reification)



Call between Objects



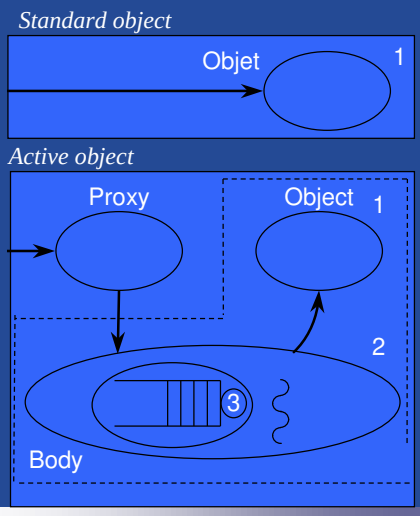
Standard system at Runtime



ProActive : Active object

An active object is composed of several objects :

- The object itself (1)
- The body: handles synchronization and the service of requests (2)
- The queue of pending requests (3)



ProActive : Creating active objects

An object created with `A a = new A (obj, 7);`
can be turned into an active and remote object:

- **Instantiation-based:**

```
A a = (A)ProActive.newActive(«A», params, node);
```

The most general case.

`To get` **Class-based: a static method as a factory**

`To get a non-FIFO behavior` **(Class-based)**:

```
class pA extends A implements RunActive { … }
```

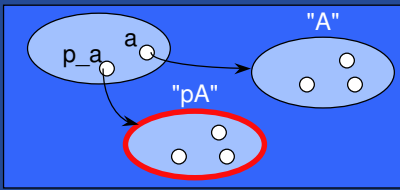
- **Object-based:**

```
A a = new A (obj, 7);
...
...
a = (A)ProActive.turnActive (a, node);
```

ProActive : Reuse and seamless

Two key features:

- Polymorphism between standard and active objects
  - Type compatibility for classes (and not only interfaces)
  - Needed and done for the future objects also
  - Dynamic mechanism (dynamically achieved if needed)




```
foo (A a)
{
  a.g (...);
  v = a.f (...);
  ...
  v.bar (...);
}
```

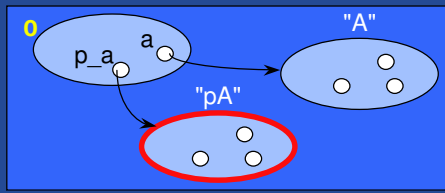
- Wait-by-necessity: inter-object synchronization
  - Systematic, implicit and transparent futures

    Ease the programming of synchronizations, and the reuse of routines

ProActive : Reuse and seamless

Two key features:

- Polymorphism between standard and active objects
  - Type compatibility for classes (and not only interfaces)
  - Needed and done for the future objects also
  - Dynamic mechanism (dynamically achieved if needed)




```
foo (A a)
{
  a.g (...);
  v = a.f (...);
  ...
  v.bar (...);
}
```

```
O.foo(a) : a.g()
and a.f() are
« local »
O.foo(p_a): a.g()
and a.f()are
«remote + Async.»
```

- Wait-by-necessity: inter-object synchronization
  - Systematic, implicit and transparent futures (“value to come”)

    Ease the programming of synchronizations, and the reuse of routines

ProActive : Intra-object synchronization

Explicit control:

Library of service routines:

- Non-blocking services,...
  - `serveOldest ();`
  - `serveOldest (f);`
- Blocking services, timed, etc.
  - `serveOldestBl ();`
  - `serveOldestTm (ms);`
- Waiting primitives
  - `waitARequest();`
  - `etc.`

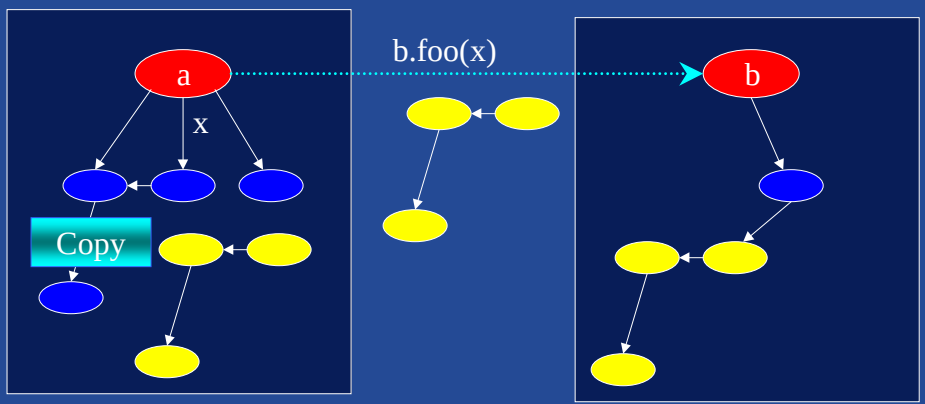
```
class BoundedBuffer extends FixedBuffer
                    implements Active
{
  live (ExplicitBody myBody)
  {
    while (...)
    {
      if (this.isFull())
        myBody.serveOldest("get");
      else if (this.isEmpty())
        myBody.serveOldest ("put");
      else myBody.serveOldest ();

      // Non-active wait
      myBody.waitARequest ();
}}}
```

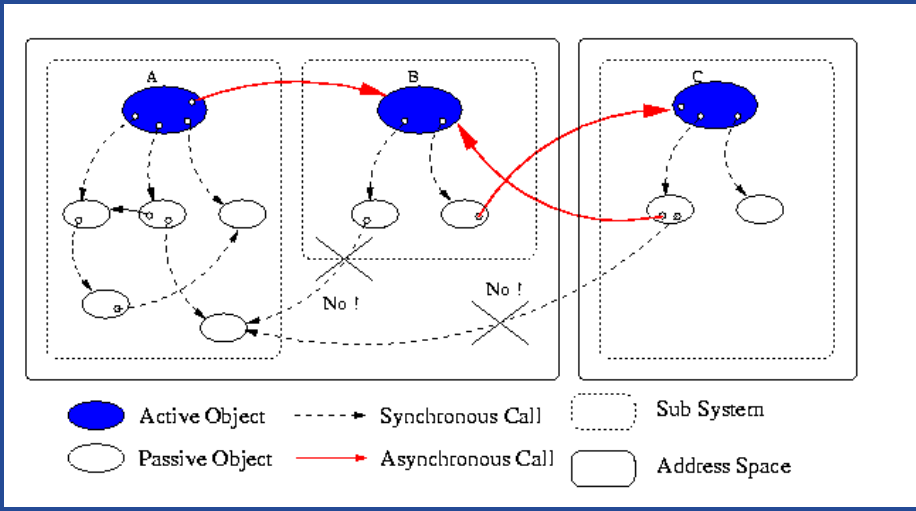
Implicit (declarative) control: library classes

```
e.g. : myBody.forbid ("put", "isFull");
```


SharedOnRead
Call between Objects



Standard system at Runtime



Optimization: SharedOnRead

- Share a passive object if same address space
- Automatic copy if required
- Implementation: Use of the Mop
- Help from the user:
  - Point out functions that make read access
  - Write access by default

SharedOnRead (2)

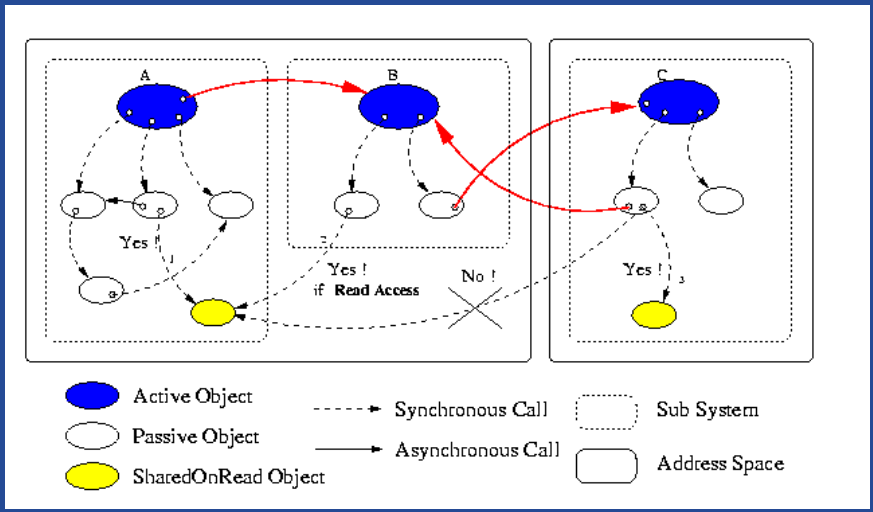
Algorithm

- If a SharedOnRead Object is send during a method call:

  If within the same VM:
  - send a reference instead of the real object
  - (reference counter)+1

- After that:
  - Read access can be freely achieved
  - Write access triggers an object copy

SharedOnRead (3)



1.2. Collective Communications: Groups

- Manipulate groups of Active Objects, in a simple and typed manner:
  - Typed and polymorphic Groups of active and remote objects
  - Dynamic generation of group of results
  - Language centric, Dot notation
- Be able to express high-level collective communications (like in MPI):
  - broadcast,
  - scatter, gather,
  - all to all

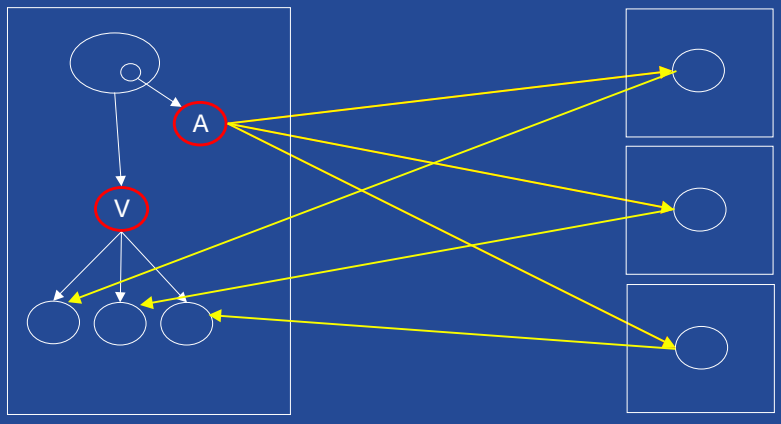
```
A ag=(A)ProActiveGroup.newActiveGroup(«A»,{{p1},...},{Nodes,..});
V v = ag.foo(param);
v.bar();
```


Group Communications

- **Method Call on a typed group**
- **Avoid network latency**
- **Based on the ProActive communication mechanism**
  - Replication of N ‘ single ’ communications
  - each communication is « adapted »
  - preserve the « rendez-vous »

Construction of a Result Group

```
A ag = newActiveGroup (…)
V v = ag.foo(param);
v.bar();
```



Typed Group Java or Active Object

Collective Operations : Example

```
class A {…
 V foo(P p){...}
}
class B extends A
  { ...}
A a1=PA.newAct(A,);
A a2=PA.newAct(A,);
B b1=PA.newAct(B,);
```

```
A a3=PA.newAct(A,);
// => modif. of
  group ag :
Group ga =
  ProActiveGroup.
  getGroup(ag);
ga.add(a3);
//ag is updated
```

```
// Build a group of « A »
A ag = (A)ProActiveGroup.newGroup(« A »,
  {a1,a2,b1})
V v = ag.foo(param); // foo(param) invoked
                    // on each member
// A group v of result of type V is created
```

```
v.bar();
// starts bar() on
  each member of
  the result group
  upon arrival
```

```
ProActiveGroup.waitForAll
  (v); //bloking -> all
v.bar();//Group call
```

```
V vi =
ProActiveGroup.getOne(v);
     //bloking -> on
vi.bar(); //a single call
```

Group as Result of Group Communication

Dynamicaly built and updated

```
B groupB = groupA.foo();
```

Ranking oder property

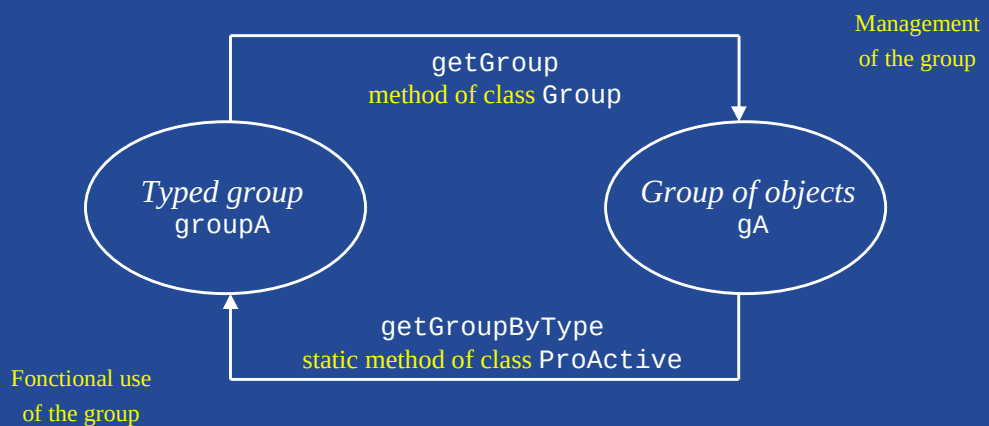
Synchronization primitive

```
ProActiveGroup.waitOne(groupB);
ProActiveGroup.waitAll(groupB);
        ... waitForAll, ...
```

Predicates:

```
noneArrived, kArrived, allArrived, ...
```


Two Representation Scheme



Two Representations (2)

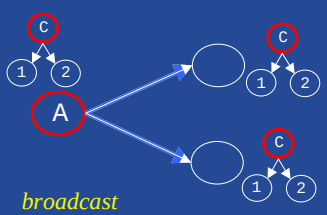
- Management operations `add`, `remove`, `size`, …
- 2 possibility : static methods, second representation
- 2 representations of a same group : Typed Group / Group of objects
- ability to switch between those 2 representations

```
Group gA = ProActiveGroup.getGroup(groupA);
gA.add(new A());
gA.add(new B()); //B herits from A
A groupA = (A) gA.getGroupeByType();
```

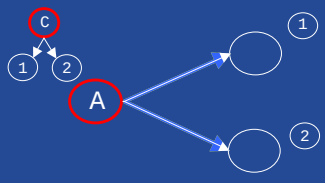

Broadcast or Scatter for Parameters

- **Broadcast is the default behavior**
- **Scatter is also possible**

```
groupA.bar(groupC);    // broadcast groupC
ProActiveGroup.setScatterGroup(groupC);
groupA.bar(groupC);    // scatter groupC
```



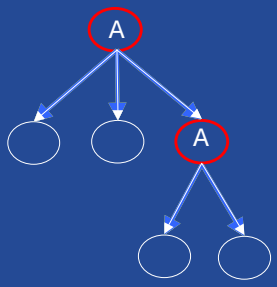
broadcast



scatter

Hierarchical Groups

- Add a typed group into a typed group
- Benefit of the network structure



Implementation

MOP generates Stub

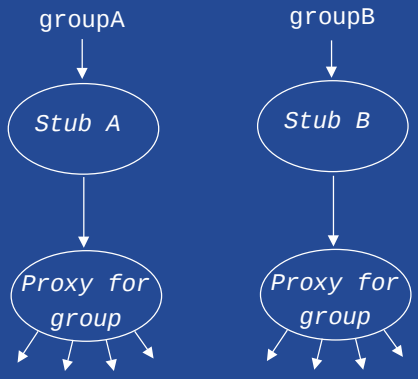
Stub inherits from the class of object

Stub connects a proxy

special proxy for group

result is stub+proxy

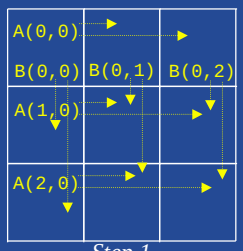
```
B groupB = groupA.foo();
```



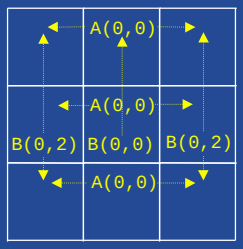
Example : Matrix multiplication

Matrix code : Broadcast to Broadcast approach

- more than 20 lines of Java code



Step 1



Step 2

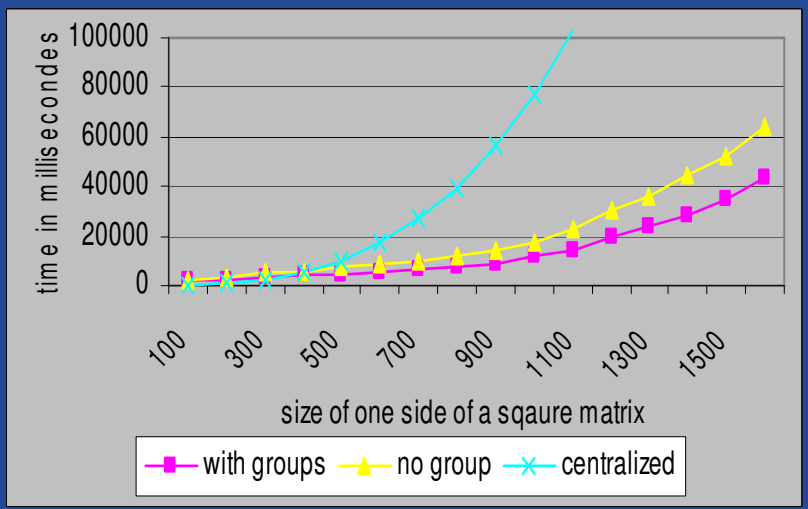
...

Step 3

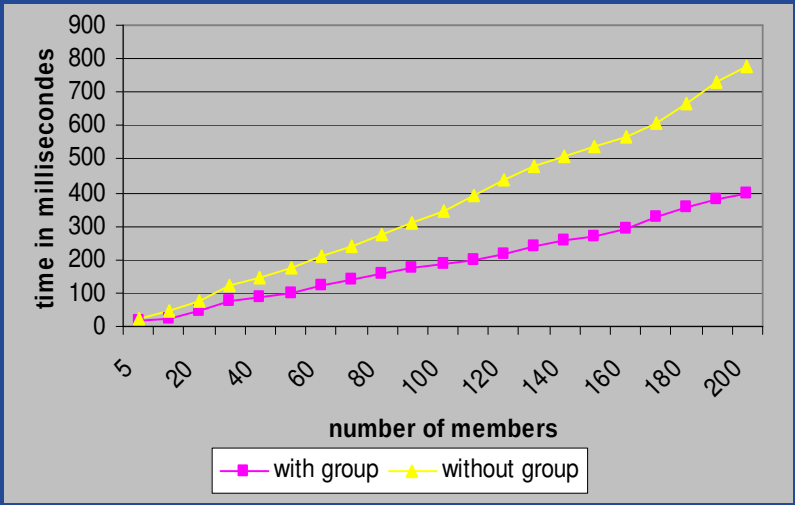
- 2 lines with ProActive groups

```
for (int i = 0 ; i<P ; i++)
  A.grouprow(i).multiply(B.groupcolumn(i));
```


Measurement : Matrix Multiplication



Measurement : Method Call



Other features for Groups

Optimization

- Parallel calls within a group (latency hiding)
- Treatment in the result order (if needed)
- Scatter (a group as a parameter to be dispatched in Group. Com.)
- A single serialization of parameters

Conceptuel : Active Group

- A group becomes remotely accessible so: updatable and consistent

  ---> migration

  ---> Dynamic changes

Perspective: using network multicast

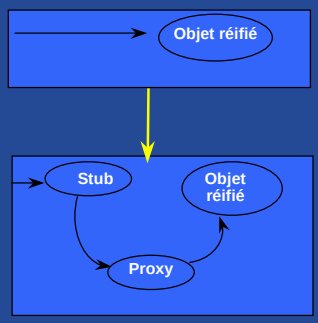
1.3. ProActive architecture: a simple MOP

- MOP (Meta-Object Protocol)
  - Runtime reflection (for dynamicity)
  - New semantics for method and constructor calls
  - Uses the Java Reflection API
- ProActive
  - Implemented on top of the MOP
  - Other models can be built on top of ProActive or on top of the MOP

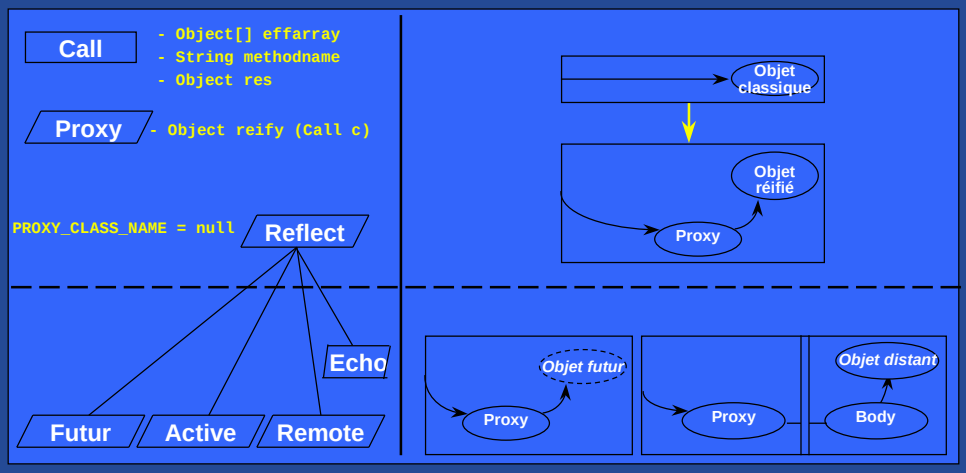
MOP principles

Static or dynamic generation of stubs:

- Take place of a reified object
- Reification of all method calls
- Sub-class of the reified object: type compatible
- Stub only depends of the reified object type,
  not of the proxy
- Any call will trigger the creation of an object **`Call`** that represents the invocation.
- It will be passed to the proxy: has the semantics to achieve



The MOP - principle



All interfaces that inherit Reflect are marker interfaces for reflexion

User Interface of the MOP

Instantiation of reified objects with static method newInstance of the MOP class

- Programming *class par class* with interfaces deriving from **Reflect**

```
Vector v = (Vector) MOP.newInstance ( <name of reified class (impl. Reflect)>,
                                      <parameters passed to proxy>,
                                      < parameters of reified object constructor > );
```

- On *instance per instance* :

```
Vector v = (Vector) MOP.newInstance ( <name of class standard>,
                                      <class proxy name to use>,
                                      <parameters given to proxy>,
                                      <parameters of reified object constructor > );
```

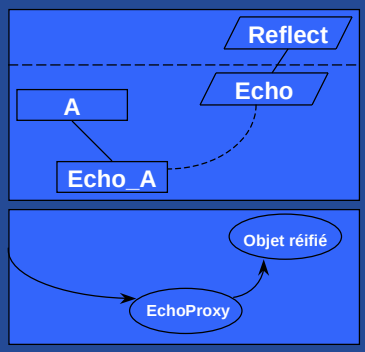
- On *object per object* :

```
Vector v = (Vector) MOP.turnReified  ( <objet standard>,
                                       <class proxy name to use>,
                                       <parameters given to proxy> );
```

Example : EchoProxy

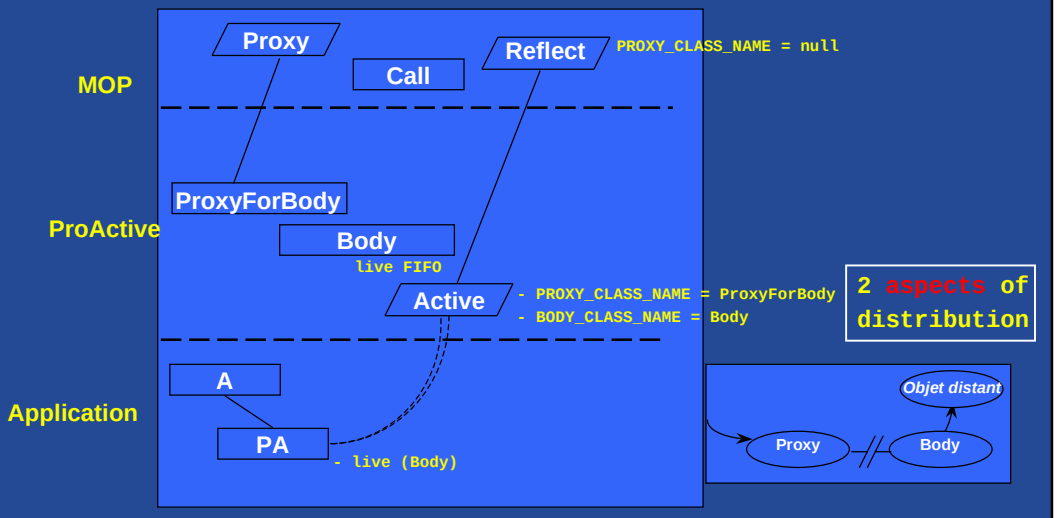
```
public class EchoProxy implements Proxy {
// Attributes
  Object myobject;
// Constructor
  public EchoProxy (Call c, Object[] p) {
     this.myobject = c.execute();
  }
// Method of the Proxy interface
  public Object reify (Call c) {
     System.out.println ("Echo->"+c.methodname+");
     return result = c.execute (myobject);
  }
}

public interface Echo_A extends Reflect
  PROXY_CLASS_NAME = "EchoProxy"; }
```



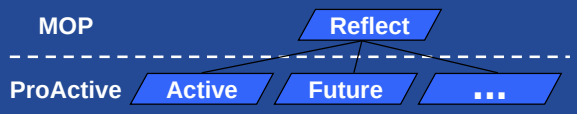

```
A a =(A)newInstance ("Echo_A",null,null);
A a =(A)newInstance ("A","EchoP.",null,null);
A a =(A)turnReified ( a, "EchoP.", null);
```


ProActive : implementation principle

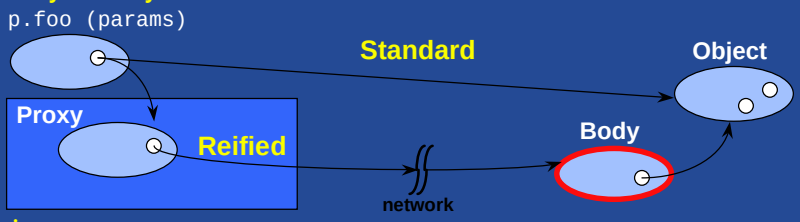


Proxy and Body

Based on interface **`Active`** and class **`ProActive`**



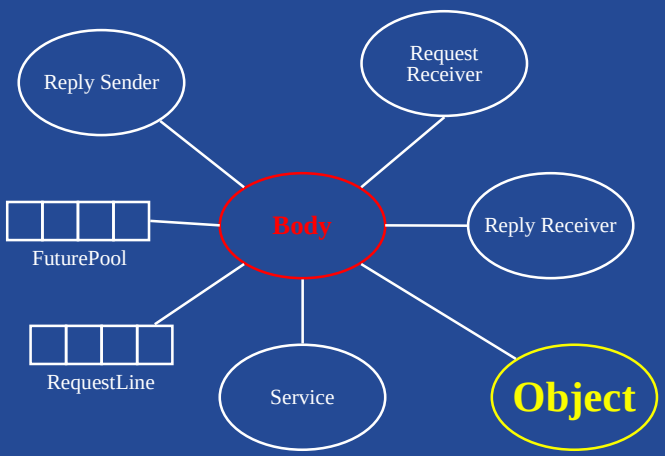
A proxy / body model



Originalities:

- Extensions through inheritance of the **`Reflect`** interface
- 3 ways to turn a standard object into a reified one
- Reuse of existing classes, polymorphism between standard and reified objects

1.4 Meta-Objects for Distribution
An Active Object

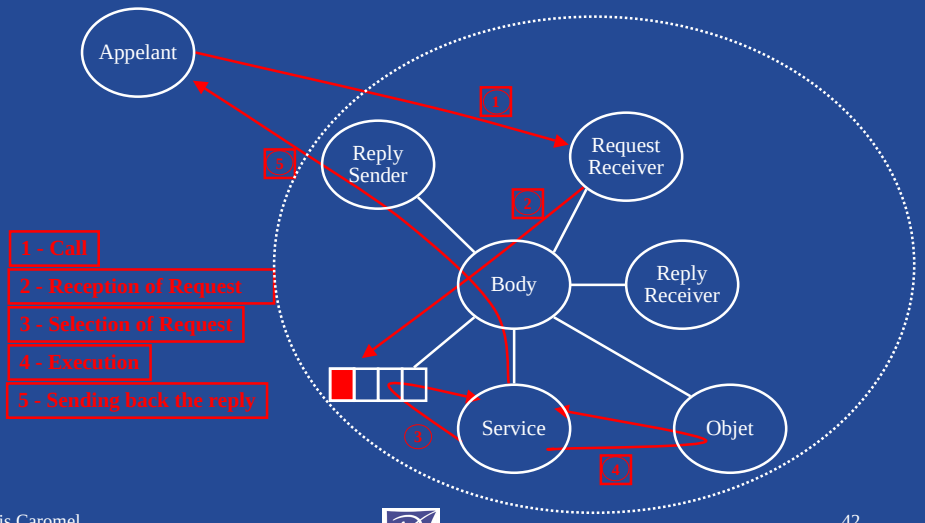


Composition d'un objet actif

Multiples Objets

- ***RequestSender:*** *Send requests (proxy + body)*
- ***RequestReceiver:*** Receve the requests
- ***ReplySender:*** Send back the result to the caller
- ***ReplyReceiver:*** Receive the future updates
- ***Service:*** Chose (select) and executes the requests
- ***RequestLine:*** Pending Requests
- ***FuturePool:*** Pending Futures

Request to an Active Object



Listener

- Pattern Event Listener
- Events are (if needed) generated for each important step
- If asked for, sent to the listeners
- These Listeners can be added/suppressed dynamically

Event Types (1)

3 main categories

Active Object:
- Creation
- Migration (activation, Inactivation : Cycle de vie)

Communications:

Requests:
- RequestSent
- RequestReceived

Reply:
- ReplySent
- ReplyReceived

Service (activity of an AO):
- WaitForRequest
- WaitByNecessity

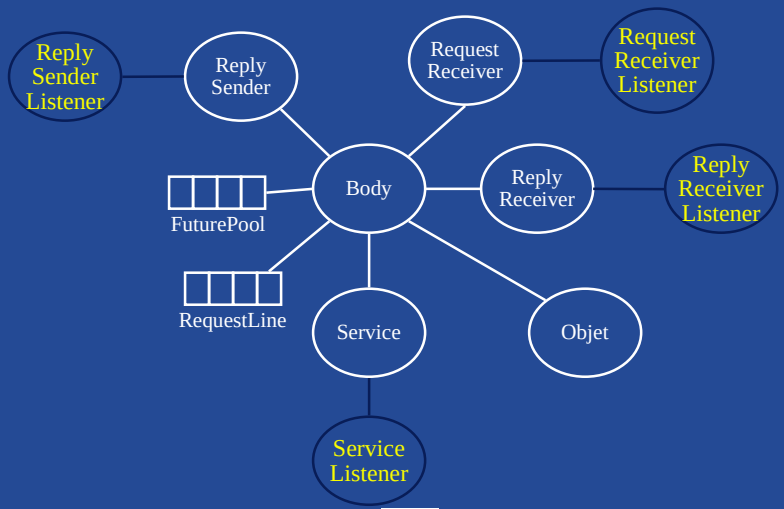
Listener - Modifier

Idem Listener + modification of the AO execution:

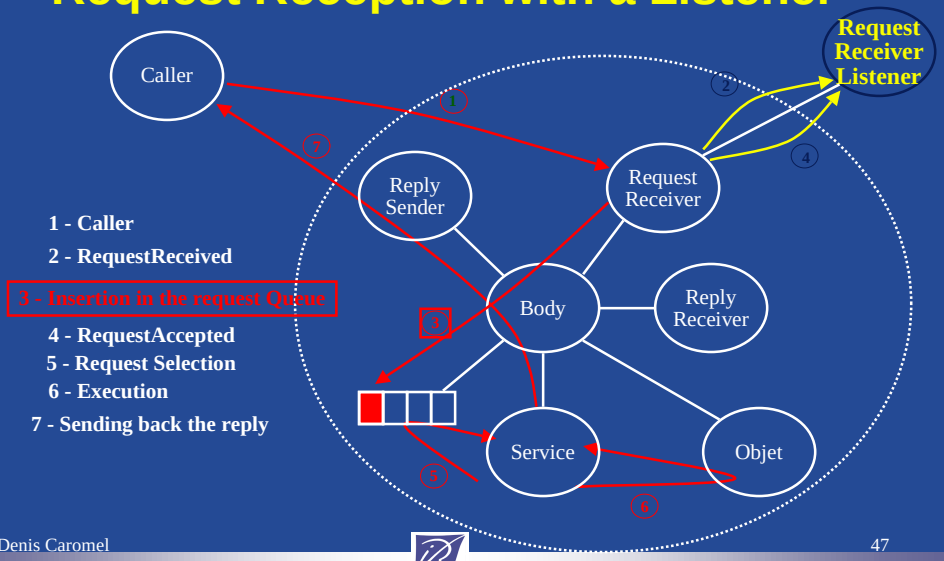
- At creation: change the VM of creation
- At migration: changer the destination VM
- Step-by-step on communications
- etc.

Application: debugging, monitoring, interactif Control of execution

Localization of listeners



Request Reception with a Listener



1 - Caller
2 - RequestReceived
3 - Insertion in the request Queue
4 - RequestAccepted
5 - Request Selection
6 - Execution
7 - Sending back the reply

1.5 : Abstract Deployment Model
Objectives

Problem:

- Difficulties and lack of flexibility in deployment
- Avoid scripting for: configuration, getting nodes, connecting, etc.

A key principle:

- Abstract Away from source code:
  - Machines
  - Creation Protocols
  - Lookup and Registry Protocols

Context:

- Grid
- Distributed Objects, Java
- Not legacy-code driven, but adaptable to it

Descriptors: based on Virtual Nodes

Virtual Node (VN):

- Identified as a string name
- Used in program source
- Configured (mapped) in an XML descriptor file --> Nodes

Operations specified in descriptors:

- Mapping of VN to JVMs (leads to Node in a JVM on Host)
- Register or Lookup VNs
- Create or Acquire JVMs

Program Source Descriptor (RunTime)

|----------| |----------|

Activities (AO) --> VN VN --> JVMs --> Hosts

Runtime structured entities: 1 VN --> n Nodes in n JVMs

Descriptors: Mapping Virtual Nodes

Example of an XML file descriptor:

```
Component Dependencies:
  Provides: …  Uses: ...
VirtualNodes:
  Dispatcher  <RegisterIn RMIregistry, Globus, Grid Service, … >
  RendererSet
Mapping:
  Dispatcher --> DispatcherJVM
  RendererSet --> JVMset
JVMs:
  DispatcherJVM = Current  // (the current JVM)
  JVMset=//ClusterSophia.inria.fr/ <Protocol GlobusGram … 10 >
...
```

Mapping Virtual Nodes: example (1)

Definitions and mapping

```
<virtualNodesDefinition>
    <virtualNode name="Dispatcher"/>
</virtualNodesDefinition>

<map virtualNode="Dispatcher">
    <jvmSet>
        <vmName value="Jvm1"/>
    </jvmSet>
</map>

<jvm name="Jvm1">
    <acquisition method="rmi"/>
    <creation>
        <processReference refid="linuxJVM"/>
    </creation>
</jvm>
```

Definition of Virtual Nodes

Mapping of Virtual Nodes

Mapping Virtual Nodes: example (2)

Definitions and mapping

```
<virtualNodesDefinition>
    <virtualNode name="Jem3DNode"/>
</virtualNodesDefinition>

<map virtualNode=" Jem3DNode">
    <jvmSet>
        <vmName value=”clusterJvm"/>
    </jvmSet>
</map>

<jvm name="clusterJvm">
    <acquisition method="rmi"/>
    <creation>
        <processReference refid=”clusterProcess"/>
    </creation>
</jvm>
```

Definition of Virtual Nodes

Mapping of Virtual Nodes

Mapping Virtual Nodes: example (3)

Infrastructure informations

```
<processDefinition id="linuxJVM">
    <jvmProcess class="org.objectweb.proactive.core.process.JVMNodeProcess"/>
</processDefinition>
```

JVM on the current Host

```
<processDefinition id="rshProcess">
    <rshProcess class="org.objectweb.proactive.core.process.rsh.RSHJVMProcess"
                hostname="sea.inria.fr">
        <processReference refid="linuxJVM"/>
    </rshProcess>
</processDefinition>
```

JVM started using RSH

Mapping Virtual Nodes: example (4)

Infrastructure informations

```
<processDefinition id=”singleJVM">
    <jvmProcess class="org.objectweb.proactive.core.process.JVMNodeProcess"/>
</processDefinition>

<processDefinition id=" clusterProcess ">
  <bsubProcess class="org.objectweb.proactive.core.process.lsf.LSFBSubProcess"
               hostname=”cluster.inria.fr">
    <processReference refid=”singleJVM"/>
    <bsubOption>
        <processor>12</processor>
    </bsubOption>
  </bsubProcess>
</processDefinition>
```

Definition of bsub process

Virtual Nodes in Programs (1)

Load the descriptor file

```
Descriptor pad = ProActive.getDescriptor ("file://ProActiveDescriptor.xml");
```

Activate the mapping

```
VirtualNode vn = pad.activateMapping ("Dispatcher");
    // Triggers the JVMs
```

Use nodes

```
Node node = vn.getNode();
...
C3D c3d = ProActive.newActive("C3D", param, node);
```

Descriptors: Virtual Nodes in Programs (2)

```
Descriptor pad = ProActive.getDescriptor ("file:.ProActiveDescriptor.xml");
VirtualNode vn = pad.activateMapping ("Dispatcher");  // Triggers the JVMs
Node node = vn.getNode();
...
C3D c3d = ProActive.newActive("C3D", param, node);
            log (  ... "created at:  " + node.name() + node.JVM() + node.host() );
```

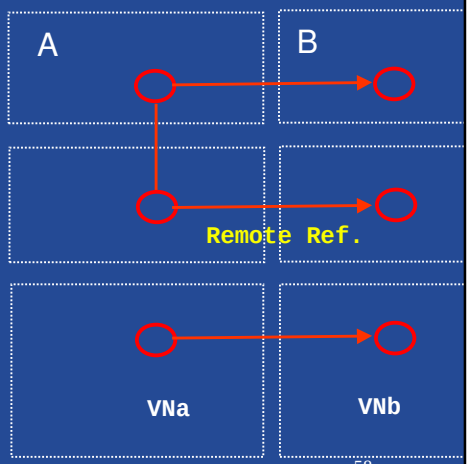
Descriptors: Virtual Nodes in Programs (3)

```
Descriptor pad = ProActive.getDescriptor ("file:.ProActiveDescriptor.xml");
VirtualNode vn = pad.activateMapping ("Dispatcher");  // Triggers the JVMs
Node node = vn.getNode();
...
C3D c3d = ProActive.newActive("C3D", param, node);
                log ( ... "created at:  " + node.name() + node.JVM() + node.host() );

// Cyclic mapping: set of nodes
VirtualNode vn = pad.activateMapping ("RendererSet");
while  ( … vn.getNbNodes … ) {
    Node node = vn.getNode();
    Renderer  re = ProActive.newActive(”Renderer", param, node);
```


Composing Virtual Nodes

VNa , VNb
--> mapped onto different JVM sets:



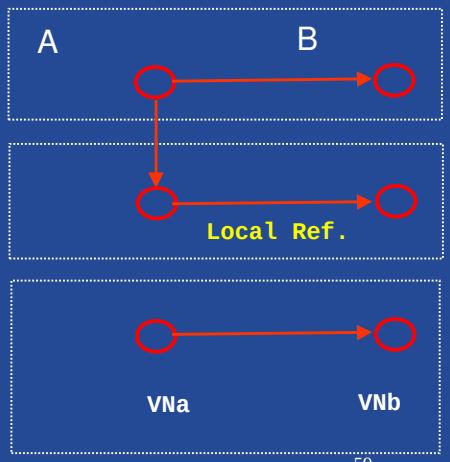
Composing Virtual Nodes

Activity JVM

VNa , VNb

--> mapped onto the same JVM sets:

Co-allocation



1.6 IC2D

Interactive Control & Debug for Distribution

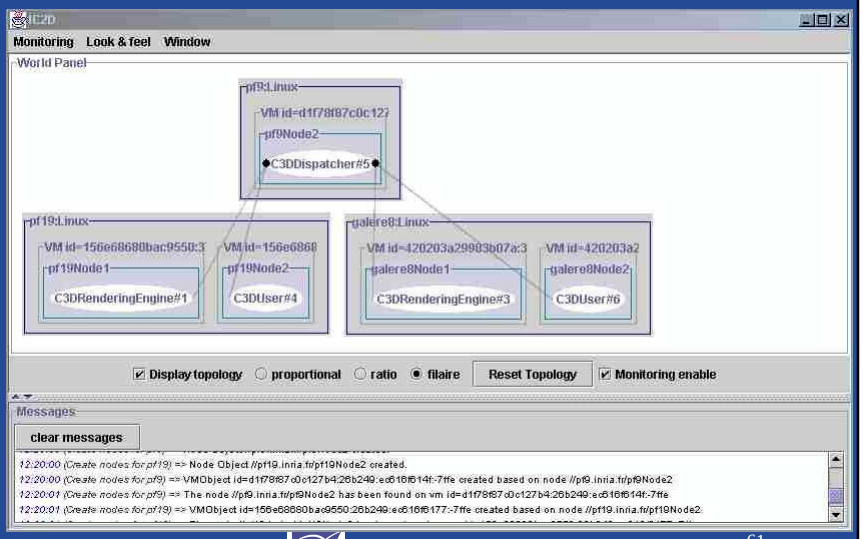
Features:

- Graphical visualization
- Textual visualization
- Monitoring and Control

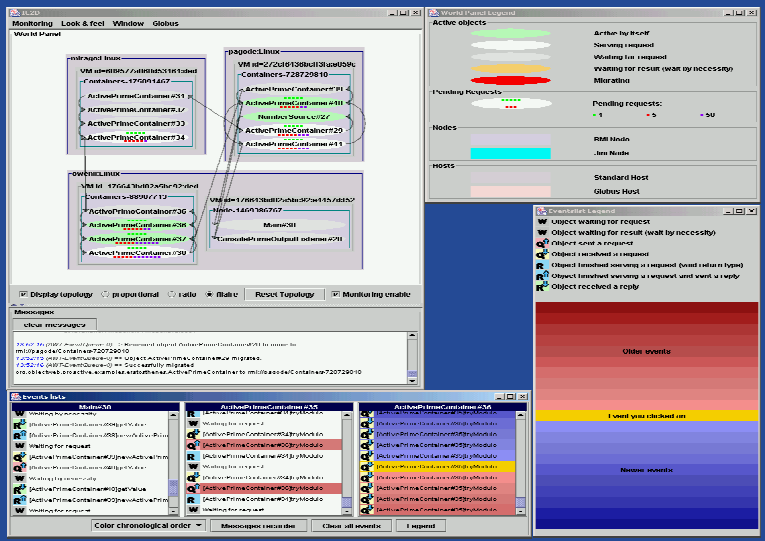
IC2D: Interactive Control and Debugging of Distribution

Main Features:
- Hosts, JVM,
- Nodes
- Active Objects
- Topology
- Migration

- Logical Clock



IC2D: Interactive Control and Debugging of Distribution



IC2D: Basic features

Graphical Visualisation:

- Hosts, Java Virtual Machines, Nodes, Active Objects
- Topology: reference and communications
- Status of active objects (executing, waiting, etc.)
- Migration of activities

Textual Visualisation:

- Ordered list of messages
- Status: waiting for a request or for a data
- Causal dependencies between messages
- Related events (corresponding send and receive, etc.)

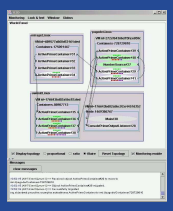
Control and Monitoring:

- Drag and Drop migration of executing tasks
- Creation of additional JVMs and nodes

IC2D: Basic features

Graphical Visualization:

- Hosts, Java Virtual Machines, Nodes, Active Objects
- Topology: reference and communications
- Status of active objects (executing, waiting, etc.)
- Migration of activities



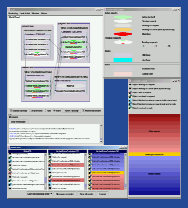
IC2D: Basic features

Graphical Visualization:

- Hosts, Java Virtual Machines, Nodes, Active Objects
- Topology: reference and communications
- Status of active objects (executing, waiting, etc.)
- Migration of activities

Textual Visualization:

- Ordered list of messages, Logical Clock
- Status: waiting for a request or for a data
- Causal dependencies between messages
- Related events (corresponding send and receive, etc.)



IC2D: Basic features

Graphical Visualization:

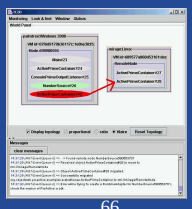
- Hosts, Java Virtual Machines, Nodes, Active Objects
- Topology: reference and communications
- Status of active objects (executing, waiting, etc.)
- Migration of activities

Textual Visualization:

- Ordered list of messages, Logical Clock
- Status: waiting for a request or for a data
- Causal dependencies between messages
- Related events (corresponding send and receive, etc.)

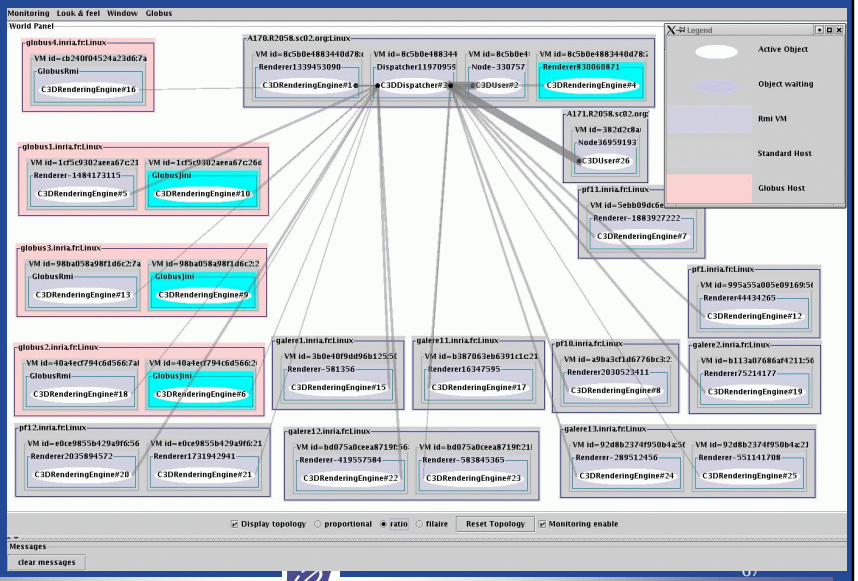
Control and Monitoring:

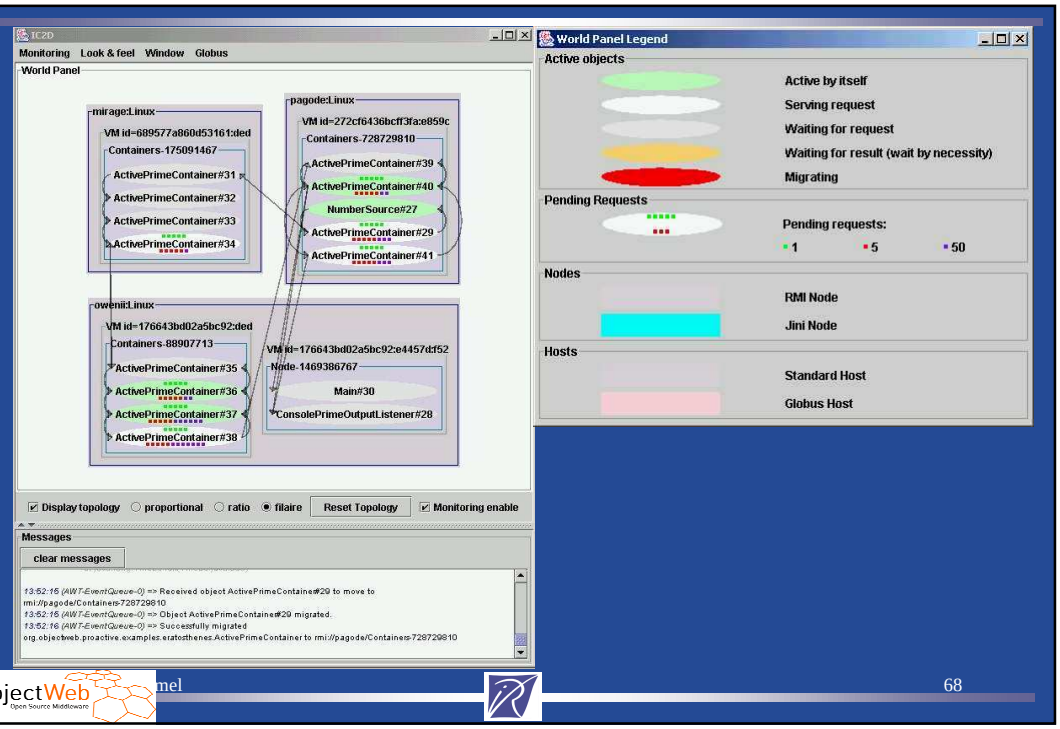
- Drag and Drop migration of executing tasks
- Creation of additional JVMs and nodes

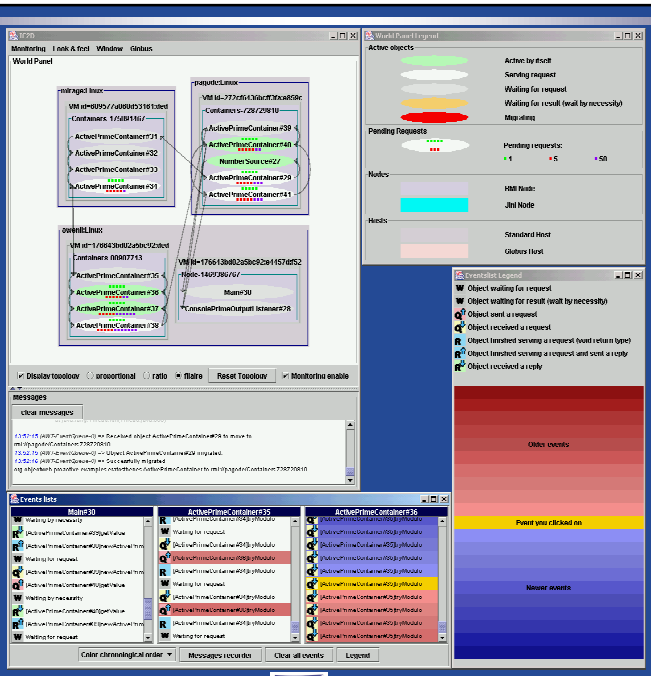


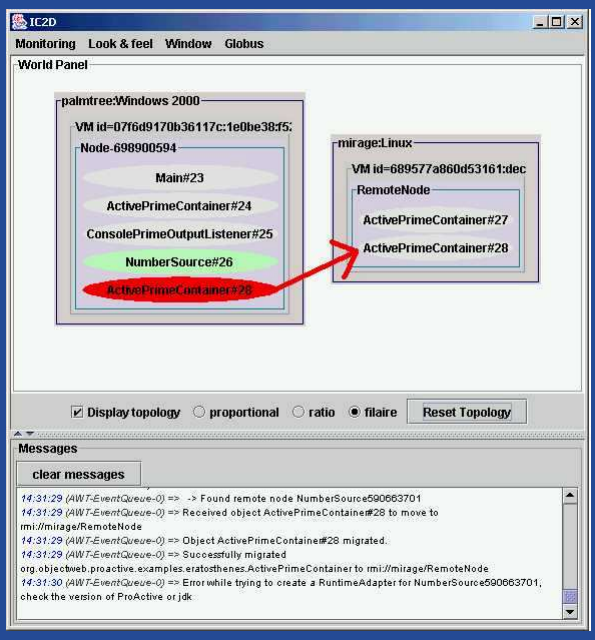
Monitoring of RMI, Globus, Jini, LSF cluster Nice -- Baltimore at SC'02

Width of links proportional to the number of com-munications








IC2D
Monitoring
Look & feel
Window
Globus
World Panel
palmtree:Windows 2000
VM id=07f6d9170b36117c:1e0be38:f5:
Node-698900594
Main#23
ActivePrimeContainer#24
ConsolePrimeOutputListener#25
NumberSource#26
ActivePrimeContainer#28
mirage:Linux
VM id=689577a860d53161:dec
RemoteNode
ActivePrimeContainer#27
ActivePrimeContainer#28
Display topology
proportional
ratio
filaire
Reset Topology
Messages
clear messages
14:31:29 (AWT-EventQueue-0) => -> Found remote node NumberSource590663701
14:31:29 (AWT-EventQueue-0) => Received object ActivePrimeContainer#28 to move to rmi://mirage/RemoteNode
14:31:29 (AWT-EventQueue-0) => Object ActivePrimeContainer#28 migrated.
14:31:29 (AWT-EventQueue-0) => Successfully migrated org.objectweb.proactive.examples.eratosthenes.ActivePrimeContainer to rmi://mirage/RemoteNode
14:31:30 (AWT-EventQueue-0) => Error while trying to create a RuntimeAdapter for NumberSource590663701, check the version of ProActive or jdk

IC2D: Related Events

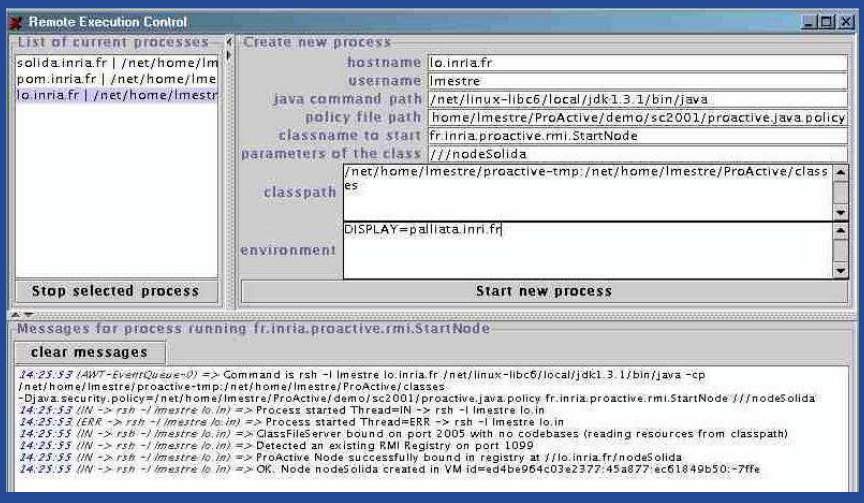
Events:

- Textual and ordered list of events for each Active Object
- Logical clock: related events, ==> Gives a Partial Order

IC2D: Dynamic change of Deployment New JVMs

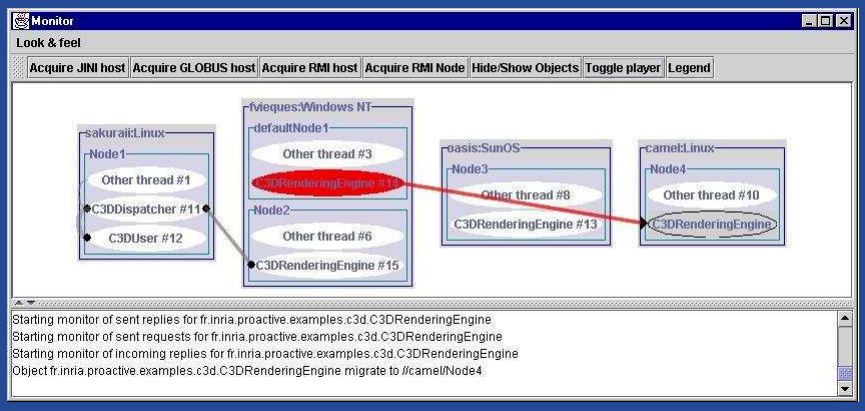
Creation, Acquisition of new JVMs, and Nodes

Protocols: rsh, ssh Globus, LSF



IC2D: Dynamic change of Deployment Drag-n-Drop Migration

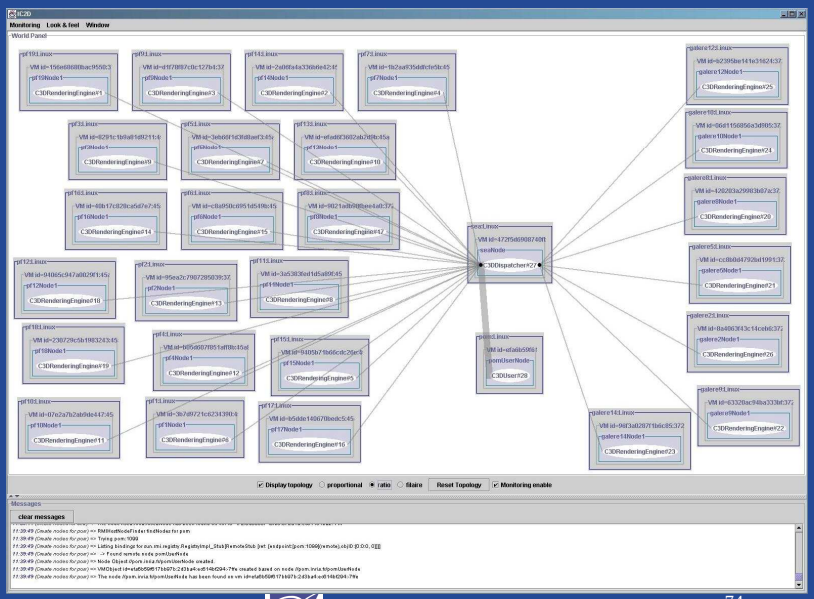
Drag-n-Drop tasks around the world



IC2D: Cluster Visualization

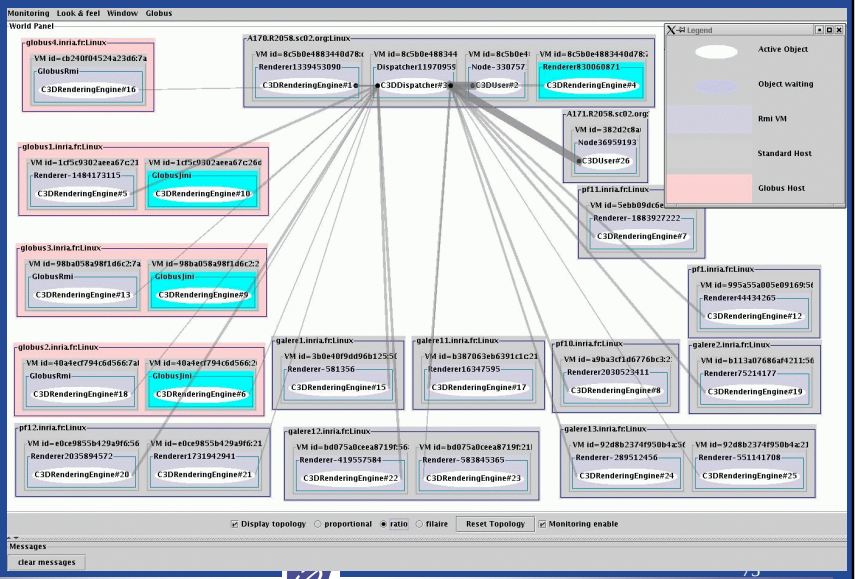
Visualization of 2 clusters (1Gbits links)

Featuring the current communications (proportional)

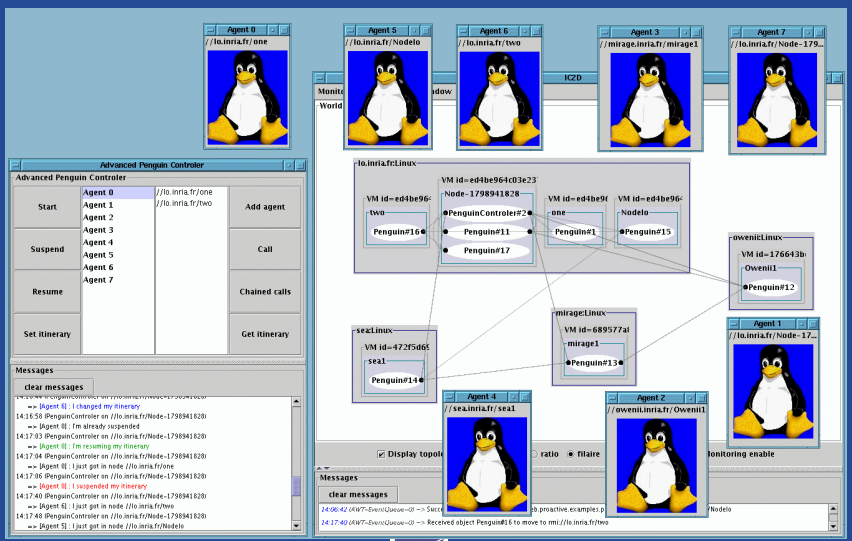


Monitoring of RMI, Globus, Jini, LSF cluster Nice -- Baltimore at SC'02

Width of links proportional to the number of com-munications



Mobile Application executing on 7 JVMs

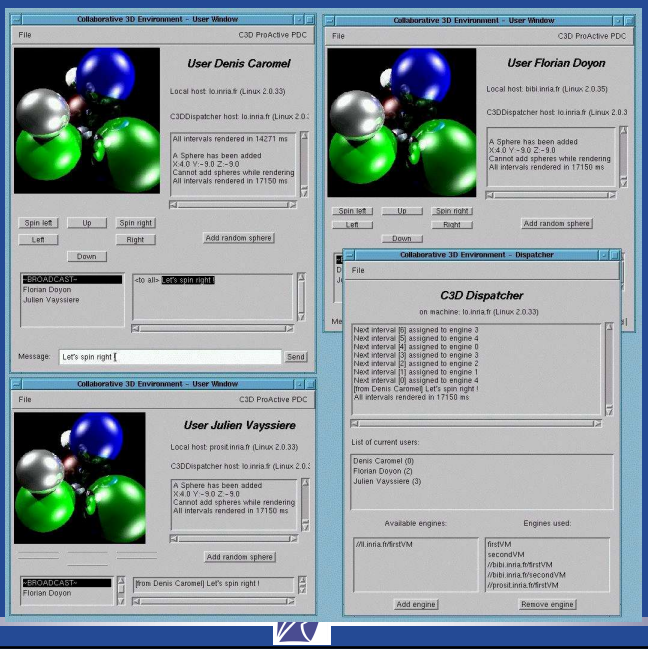


1.7 DEMO: Applis with the IC2D monitor

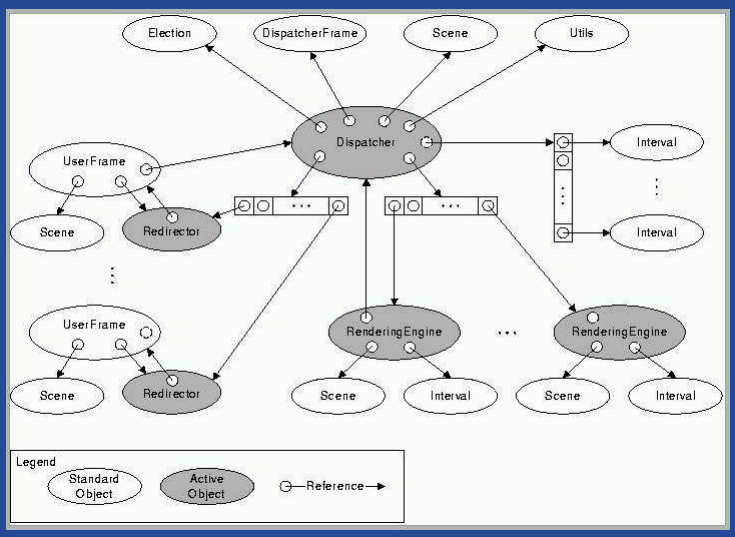
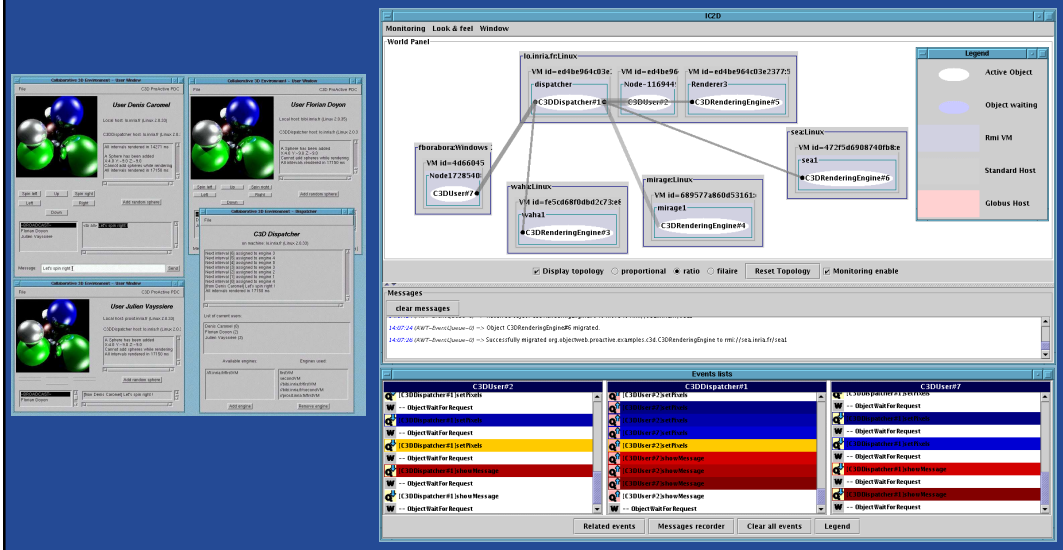
- 1. C3D : Collaborative 3D renderer in //
  a standard ProActive application
- 2. Penguin
  a mobile agent application

IC2D: Interactive Control & Debug for Distribution
work with any ProActive application
Features:
Graphical and Textual visualization
Monitoring and Control

C3D: distributed-//-collaborative



Object Diagram for C3D



Monitoring: graphical and textual com.