Object-Oriented Middleware and Components for the GRID:

Java, Corba Techniques and Tools

Denis Caromel -- Christian Perez

Univ. Nice Sophia Antipolis — IRISA Rennes

INRIA, CNRS, IUF — INRIA

Tutorial Middleware 2003

Rio de Janeiro, June 16th 2003

Objectives of the Tutorial

- The main principles of component technology
- Object-oriented middleware for parallel and distributed programming on the Grid
- State the main principles of Grid components
- Provide comprehensive examples with Java and Corba

Table of Contents (1)

1. Principles and Definition of Software Components
   - 1.1 Basics ideas
   - 1.2 JavaBeans
   - 1.3 EJB
   - 1.4 .Net
   - 1.5 Corba 3 CCM
   - 1.6 Hierarchical components
   - 1.7 Summary and classification

Table of Contents (2)

2. Parallel Objects, Java, and Components
   - 2.1 Some academic research on GRID components
   - 2.2 Programming vs. Composing
   - 2.3 The Java ProActive middleware
   - 2.4 ProActive components
   - 2.5 Tools, and Demonstration

Table of Contents (3)

3. Parallel CORBA Objects and Components
- 3.1 Motivations
- 3.2 CORBA-based approaches
- 3.3 PaCO++: a Portable Parallel CORBA Object Implementation
- 3.4 PaCO++ in action
- 3.5 GridCCM: toward Parallel CORBA Components
- 3.6 Concluding remarks on Corba

4. Conclusion

1. Basic Ideas and Definition - What is it ?

A Component = a unit of Composition and Deployment

From Objects (Classes) to Components:

- Objects:
  - Programming in the small
  - Composition by programming (Inheritance, Instantiation/Aggregation)
- Components:
  - Building software in the large
  - Tools for assembling and configuring the execution

Component = a module (80s!) but subject to:

- Configuration (variation on Non Functional Properties)
- Instantiation, life Cycle management

To be deployed on various platforms (some portability)

Characteristics -- How ?

How it works --- Common characteristics

- A standardized way to describe a component:
  - a specification of what a component does:
    - Provide (Interfaces, Properties to be configured)
    - Require (services, etc.)
    - Accept as parameterization
- Usually dynamic discovery and use of components:
  - Auto-description (Explicit information: text or XML, reflection, etc.)
- Usually components come to life through several classes, objects
- Legacy code: OO code wrapper to build components from C, Fortran, etc.

My Definition of Software Components

A component in a given infrastructure is:

- a software module,
- with a standardized description of what it needs and provides,
- to be manipulated by tools for Composition and Deployment

1.2 A typical example: JavaBeans

Graphical components in Java

Quite simple :

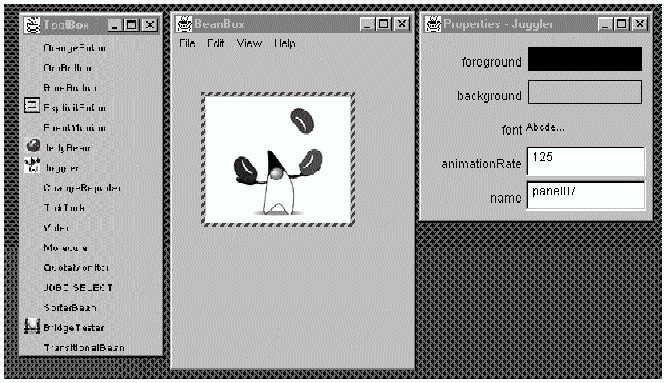
- a Java class (or several)
- a naming convention to identify **properties:**
  - method: public T getX ()
  - method: public void setX ()
  - an attribute: private T X = <default value>;
- a communication pattern: **Events, Source, Listeners**

and … a class is turned into a graphical component !

The Java introspection allows to discover dynamically the properties,
and to configure them, assemble JB interactively

JavaBeans (2)

The BeanBox



So for JavaBeans:

Software module =
Java Class
Standardized description =
getX, setX, X,
listeners
Tools:
Composition = BeanBox
Deployment = JVM

Nothing very new (cf. NeXTStep Interface Builder),
but life made a bit easier with byte code and introspection

Deploying and Executing Components

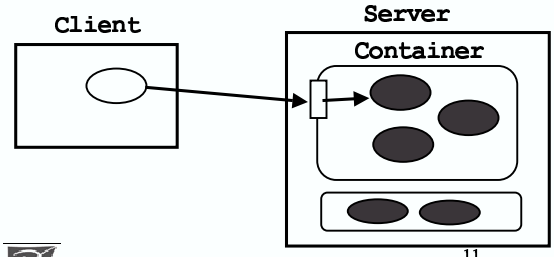
Components have to be configured on their Non-Functional Properties:

- Functional Properties, Calls (Def.):
  - Application level services a component provides (e.g. Balance, Saxpy)
- Non-Functional Properties, Calls (Def.):
  - The rest, mainly infrastructure services:
    - Transaction, Security, Persistence, Remote/Asynchronous Com., Migration, …
    - Start, Stop, Reconfiguration (location, bindings), etc.

so, Typical Infrastructure : Container for Isolation

Allows to manage and implement:

- the non-functional properties
- Life cycle of components



1.3 Example: Enterprise Java Beans

A 3 tiers architecture (Interface, Treatment, DB), in Java

- Objectives: ease development + deployment + execution
- Java portability

A few concepts and definitions:

- EJB Home object:
  - management of life cycle: creation, lookup, destruction, etc.
- EJB Remote object:
  - Client view and remote reference
- EJB object (or Bean):
  - the component itself (Session Stateless or Statefull, Entity Bean)
- Functional Properties = Business Methods

Summary: Enterprise Java Beans

So for EJB:

Software module =

- Java Classes and Interfaces (Home, Remote, Beans, …)

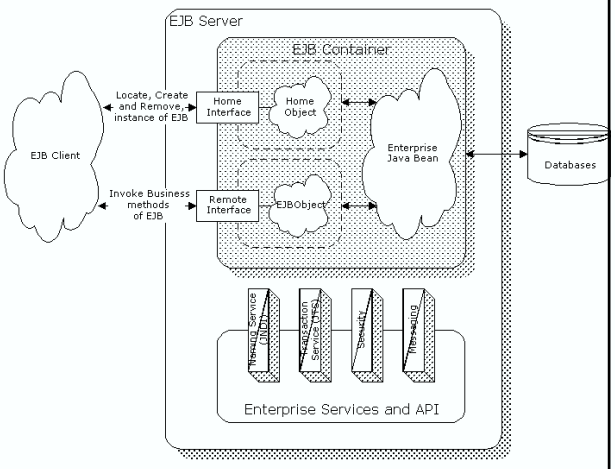
Only Provides (server), no Uses

Standardized description =

- a file with a standard format

Tools:

- Composition = ? EJBrew ?
- Deployment = JVM + RMI, JTS, + Generators + EJB Servers



From www.tripod.com , G. S. Raj article

1.4 Components in Windows .Net

.Net basics:

- A VM designed for several languages (C, C++, VB, + others)
  - CLR (Common Language Runtime)
  - CIL (Common Intermediate Language, MSIL) wider than ByteCode
    - Boxing/Unboxing (value type <--> object), etc.
- A new language: C#
- An interactive tool (Visual Studio) to manipulate the "components"

A key choice: Extraction of description from program code

- C# introduces language constructions for component information:
  - Properties
  - Attributes
  - XML tags in source code (in Attributes)

Components in Windows .Net (2)

Example of Attributes, and Properties in C#:

```
[HelpUrl ("http://someUrl/Docs/SomeClass")]
class SomeClass {
private string caption;
public string Caption {
   get { return caption; }
   set { caption = value;
         Repaint (); }
}
```

An attribute: HelpUrl

It is actually a user define class (derive from Syst.Att.)

Attribute exists at RT.

A Property: Caption

JavaBeans in a language

Also: Indexed properties

Components for Web program. : WSDL (Web Services Description Lang.)

- WSDL (Def. of Web callable methods) + Directories +
- SOAP as wire format + Classes with Attributes and properties,

Components in Windows .Net

Components characteristics:

Software module =

- Classes and Interfaces in various languages, but focus on C#

Standardized description =

- Still the COM, DCOM interfaces
- Extraction of Attributes, Properties **from source code!**
- WSDL

Tools:

- Composition = Visual Studio, etc.
- Deployment = Windows, .NET CLR

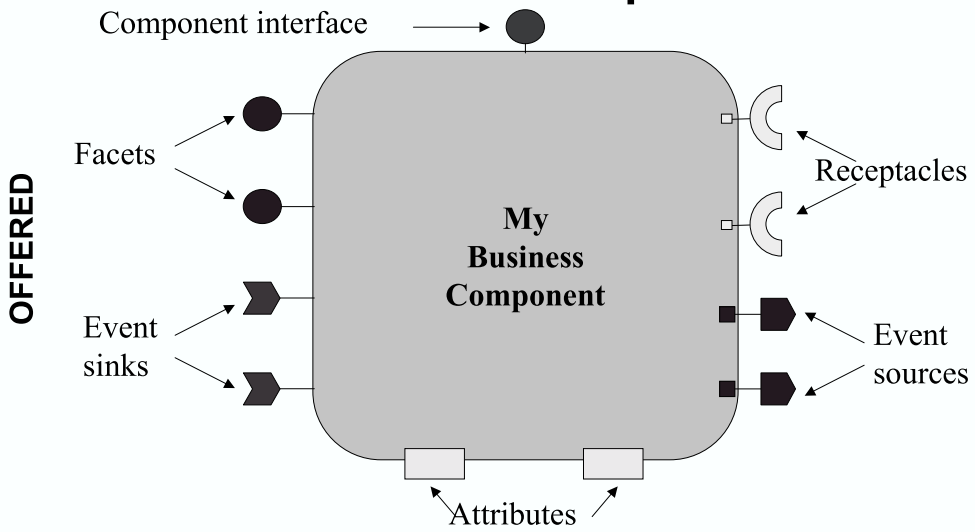
A Web Service: the instance of a component, … running...

1.5 Assembly of Components Corba 3 and CCM

CCM: Corba Components Model =

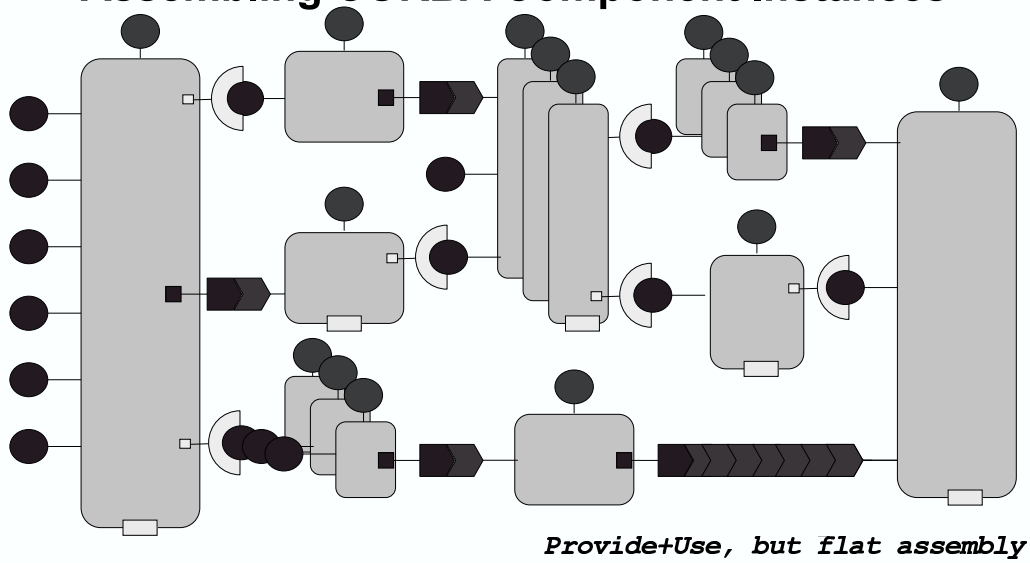
- EJB + a few things :
  - More types of Beans defined:
    - **Service**, Session, **Process**, Entity, ...
  - Not bound to Java (Corba IDL)
  - Provides but also Uses :
    - Specification of the component needs, dependencies
    - "Client Interfaces''
  - A deployment model (ongoing at OMG)

A CORBA Component



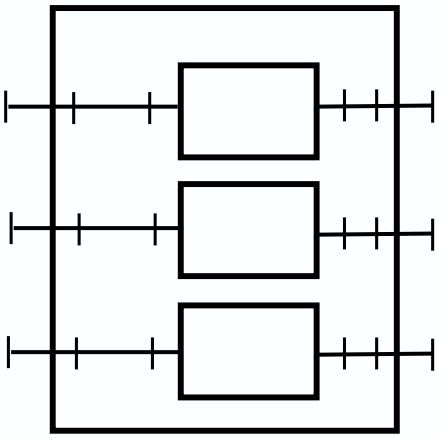
Courtesy of Philippe Merle, Lille, OpenCCM platform

Building CCM Applications = Assembling CORBA Component Instances



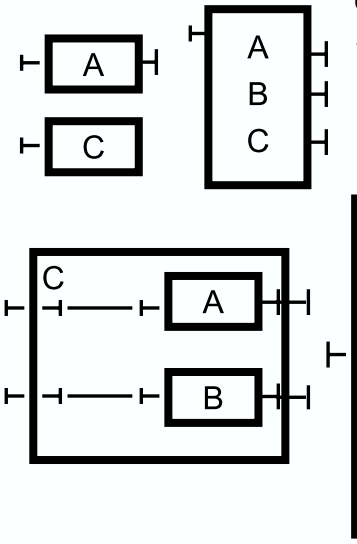
Provide+Use, but flat assembly

1.6 Hierarchical Component Fractal model



Defined by E. Bruneton, T. Coupaye, J.B. Stefani, INRIA & FT

Composite Components



Composition of components
to build new component:
Composite (vs. Primitif)
Cp. Inside Cp.

Composite components:
- Internal interfaces
- Internal Bindings
- Imbrication
- IN/OUT operations
  - Controllers
  - Life cycle Mng.

--> Reconfiguration

1.7 Conclusion on the basics: Component Orientedness

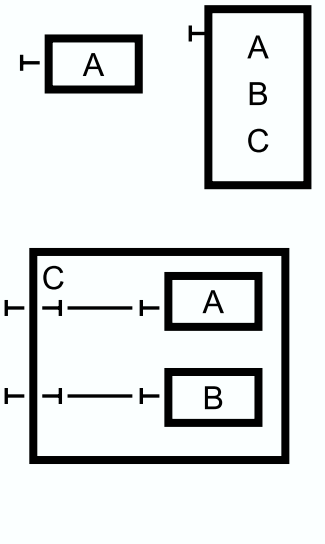
- Level 1: Instantiate - Deploy - Configure
  - Simple Pattern
  - Meta-information (file, XML, etc.) — JavaBeans, EJB
- Level 2: Assembly (flat)
  - Use and client interfaces — CCM
- Level 3: Hierarchic
  - Composite — Fractal
- Level 4: Reconfiguration
  - Binding, Inclusion, Location — On going work …

Interactions / Communications:

Functional Calls: service, event, stream

Non-Functional: instantiate, deploy, start/stop, inner/outer, re-bind

Towards GRID Components



Collections are essential:

--> Group Communications

--> Collective Interfaces

Parallel component vs. Distributed: A given component instance can be distributed over machines

Reconfigurations: bindings, in/out

Towards GRID Components

Parallel and Distributed:

--> Group Communications

Plus specificity :

- High performance
- Communication :
  - Important Bandwidth
  - Very High Latency
- Deployment complexity: **--> Abstractions**
  - Various remote execution tools (rsh, ssh, Globus, Web Services)
  - Various registry and lookup protocols (LDAP, RMI, WS, etc.)
  - Large variations in nodes being used (1 to 5000, … 200 000)
- Debugging, Monitoring, and Reconfiguring
  - Across the world ??

High-Performance a specificity ? Not sure: an EJB component handling 1 000 000 of requests already needs High-Performance!

Networks grow faster than Procs

Techniques for hiding it

2.
Java Parallel Objects and Components

2.1
Some Academic Research on GRID Components

SciRun from Utah

- scalable parallel applications and visualization

Webflow from Syracuse

- graphical composition palette

CCA:

- CCAT and XCAT from Indiana University
- Ccaffeine from Sandia Nat. Lab. in Livermore

A quick look at CCA

The U.S. Dept of Energy DOE2000 project

- The Common Component Architecture: CCA
  - Lawrence Livermore National Lab
  - Sandia Labs
  - Argonne National Labs
  - Los Alamos National Labs
  - Universities: Utah, NCSA, Indiana

Initiative to define minimal specification for scientific components

Targeting Parallel and distributed

Draws ideas from CCM and other models

Provide/Use ports, Calls/Events/Streams, Scientific IDL

CCAT and XCAT: Common Component Architecture

D. Gannon et al.

CCAT: Common Component Architecture **Toolkit**

- D. Gannon's team implementation of CCA
- Based on :
  - HPC++
  - Globus, SSH
  - Java for GUI, JPython, Matlab interface

A focus on : Composition

Novel MxN work at MPI-I/O level

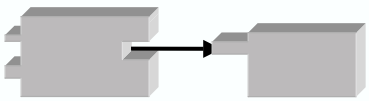
Java and C++ components

CCAT Components

Main principles:

- CCA framework
- Core Services are components
  - Flexibility, Higher-level services from core
- GUI for composition is a component
  - connected to Provides-Port of core service components
- Standard services:
  - Directory, Registry, Creation, Connection, Events
- XML description of components

Component Communication



How do components communicate?

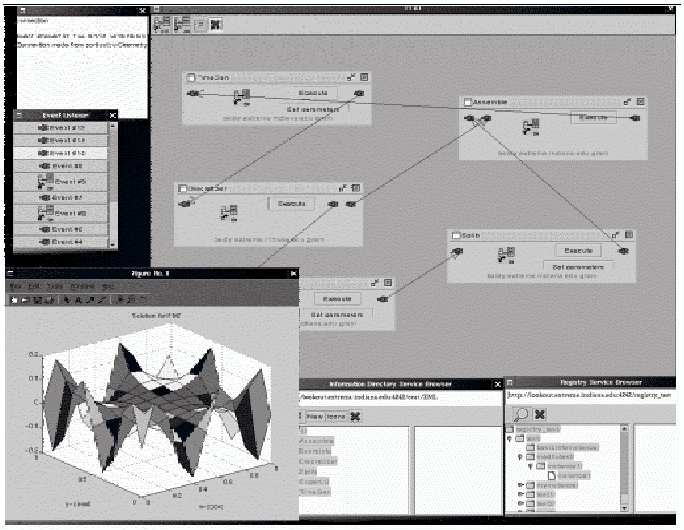
- Use Remote Procedure Call (RPC) Mechanism
  - XCAT uses **SOAP 1.1**
  - XCAT ports can serve as web services
- Events/Messages
  - Objects encoded as XML documents

CCAT session with Java GUI: Composition

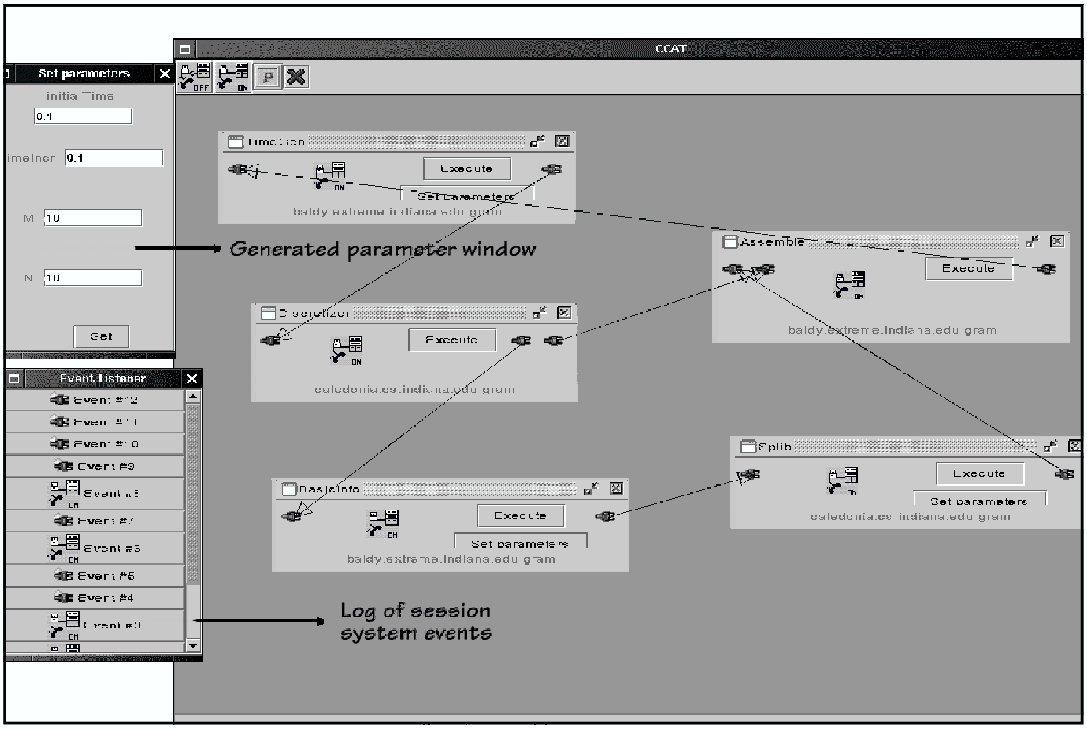
Composition Tool:

- Select
- Connect
- But also:
  Test and Execute

Level 2 Provide/Use model



Courtesy of D. Gannon et al.



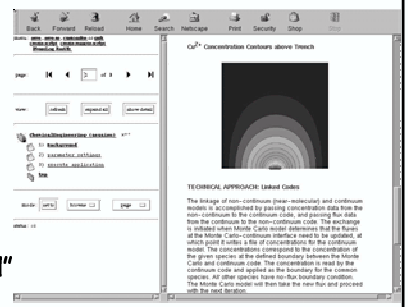
XCAT:

D. Gannon et al. current project

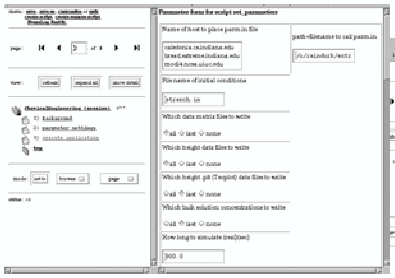
- A Java-based web server (Tomcat) on the client:
  - Java Servlets
- A Browser on the client as well
  - Script Editing
- Scripts in JPython

- Web Services: SOAP communications
  - XML wire format

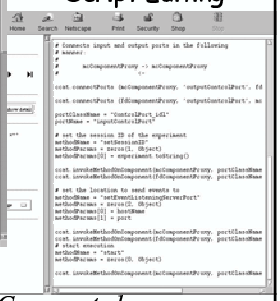
Notebook Browsing Controls



Notebook Scripts can be "parameterized" by web forms



Script Editing



Courtesy of D. Gannon et al. 33

XCAT Services Architecture

Default services for all components

XCAT services

- Directory
  - locate components based on port types and other attributes
- Registry
  - locate running instances of components
- Creation
  - create running instance of a component
- Connection
  - connect ports of two running instances
- Events
  - publish/subscribe framework for messages

Ccaffeine

Mainly from HPC Research Div. At Sandia Nat. Lab. in Livermore

SPMD

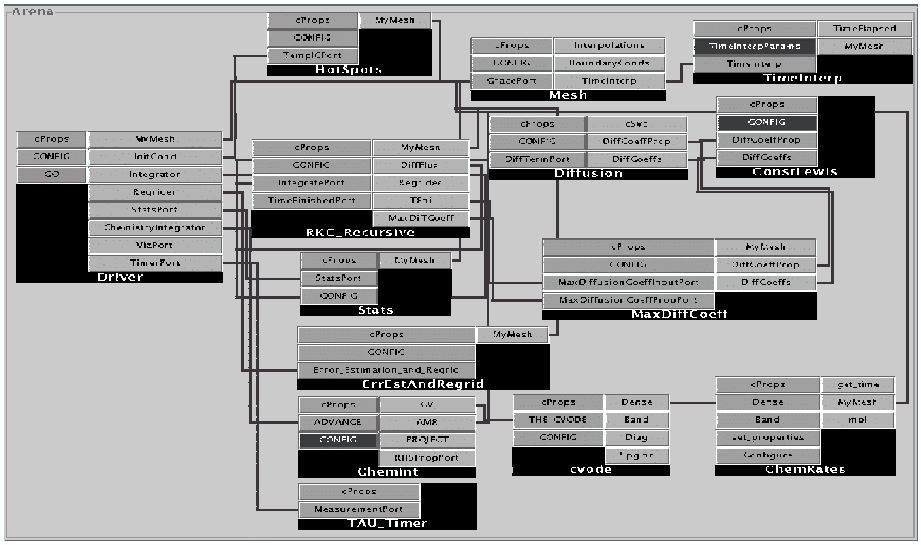
GUI and Scripted Interface

Interactive or Batch

Serial or Parallel

Components written in C++

Ccaffeine



CCA Characteristics
XCAT, CCAT, Ccaffeine, ...

So for CCA, etc.:

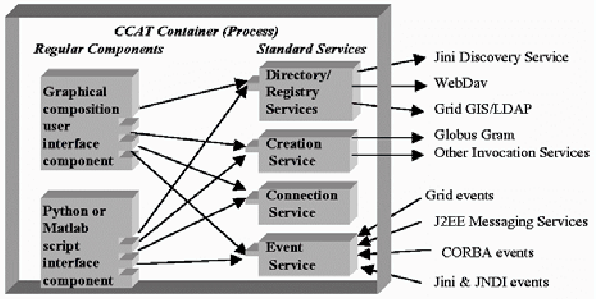
Software module =
- Any Code + wrappers

Standardized description =
- XML
- Interfaces (Provide+Use)

Tools:
- Composition = GUI (Java)
- Deployment = some

Level 2 Provide/Use model

From D. Gannon et al._article



CCAT Container: User and Service Components
Service Components are mainly wrappers for external services (factory,registry,...)

But: CCA not specific to any underlying distributed object model

2.2 Programming vs. Composing

A model of computation is still needed

Programming vs. Composing

The underlying model of parallel and distributed computing being used is FUNDAMENTAL.

How to build components that actually compose:

- semantics, correctness,
- efficiency, predictability of performance, ...

without a clearly defined programming model ?

For 50 years, Computer Science have been looking for abstractions that compose: functions, modules, classes, objects, …
The semantics of a composite is solely and well defined from the semantics of inner components. *The quest is not over !*

Techniques for Components Interactions

Interactions / Communications:

Functional Calls: service, event, stream

Non-Functional: instantiate, deploy, start/stop, inner/outer, re-bind

Alternative:

- A unique infrastructure and model, e.g.: ---> 2.2 ProActive
  - RMI for functional and parallel calls
  - RMI for component management
- 2 different infrastructures: ---> 3.2 GridCCM
  - MPI, openMP, … for functional and parallel calls
  - Corba, WebServices (SOAP), … for non-functional

2.3 The ProActive middleware

A programming model for the Grid:

- Asynchronous and typed communications
- Data-driven synchronization: Wait-By-Necessity
- Group communications
- Migration

ProActive Components:

- Parallel and distributed
- Abstractions of deployment: Virtual Nodes
- Composition: composite components
- Interactive deployment and monitoring

Demonstration:

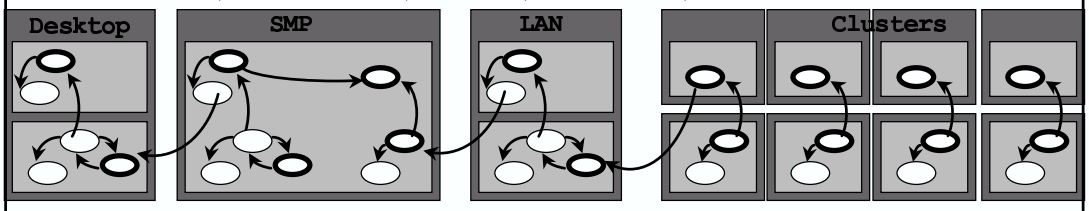
- IC2D GUI

ProActive: Basic Features and Model

Overall Goal

- Library for Parallelism, Distribution, Mobility, CSCW , GRID
- 100 % Java

Parallel, Distributed, Mobile, Activities, across the world !



Goals:

- Change in distribution: smooth + incremental transitions
- Interactive Configuration, Deployment
- Strong Semantics, performance, safety and security issues
- Distributed Components: Structured, Hierarchical

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.**

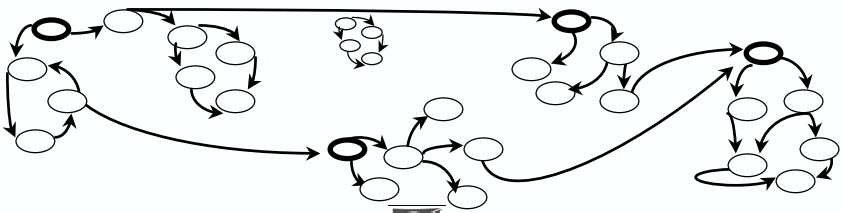
Main features:

- **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 : model

- Active objects : coarse-grained structuring entities (subsystems)
- Each active object: - possibly owns many passive objects
  - semantically 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)



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 { … }
```

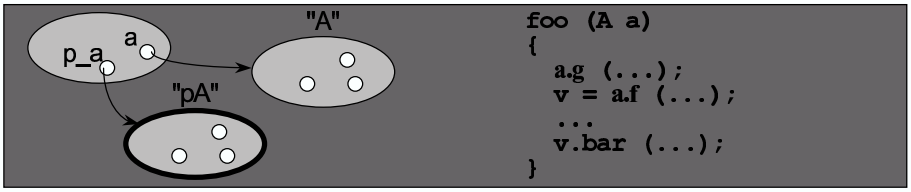
- **Object-based:**

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

ProActive : flexibility

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 (...);
}
```

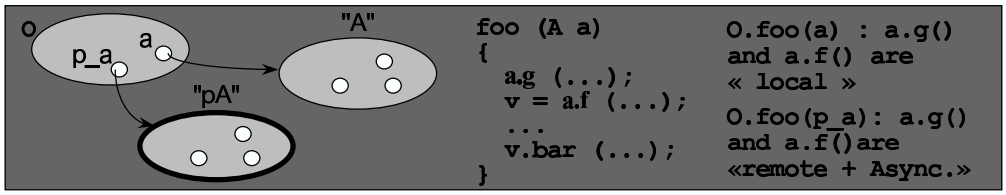
- Wait-by-necessity: inter-object synchronization
  - Systematic, implicit and transparent futures

    Ease the programming of synchronizations, and the reuse of routines

ProActive : flexibility

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)



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

    Ease the programming of synchronization, and the reuse of routines

Group Communications

- Manipulate groups of Active Objects, in a simple and typed manner:

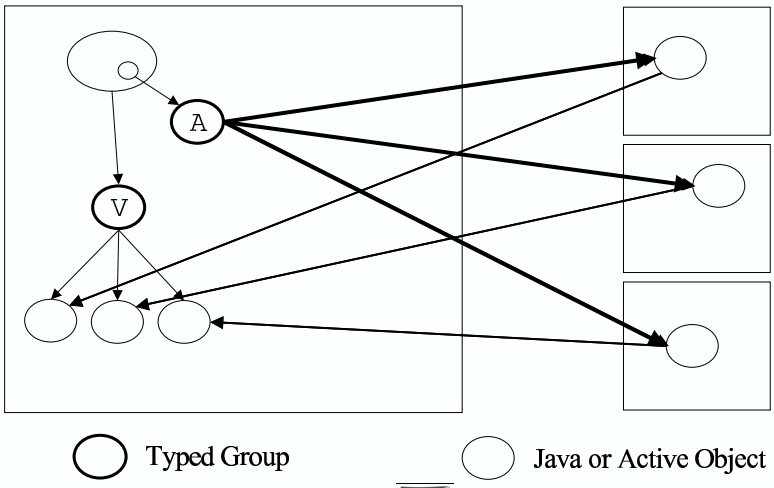
  ➡ Typed and polymorphic Groups of active and remote objects
  Dynamic generation of group of results

- 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();
```

Construction of a Result Group

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



ProActive : Mobility of active objects

Migration is initiated by the active object itself through a primitive: migrateTo

Can be initiated from outside through any public method

The active object migrates with:

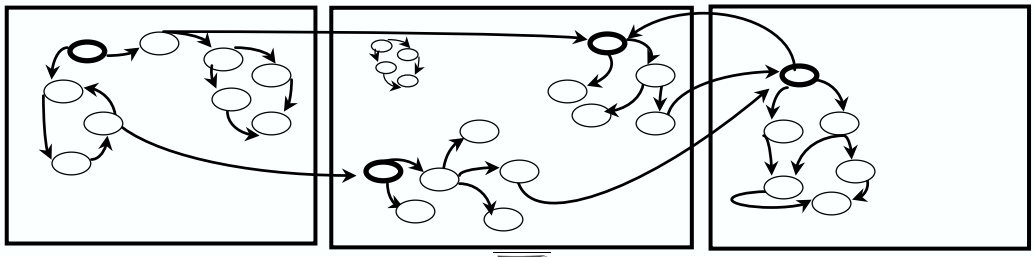
- all pending requests
- all its passive objects
- all its future objects

Automatic and transparent forwarding of:

- requests (remote references remain valid)
- replies (its previous queries will be fulfilled)

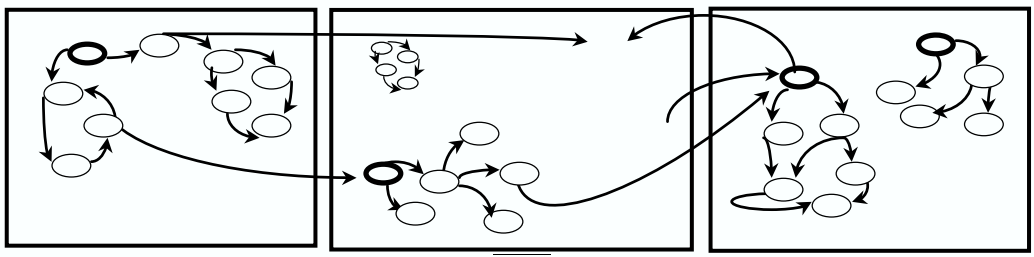
Characteristics and optimizations

Same semantics guaranteed (RDV, FIFO order point to point, asynchronous)
Safe migration (no agent in the air!)
Local references if possible when arriving within a VM
Tensionning (removal of forwarder)



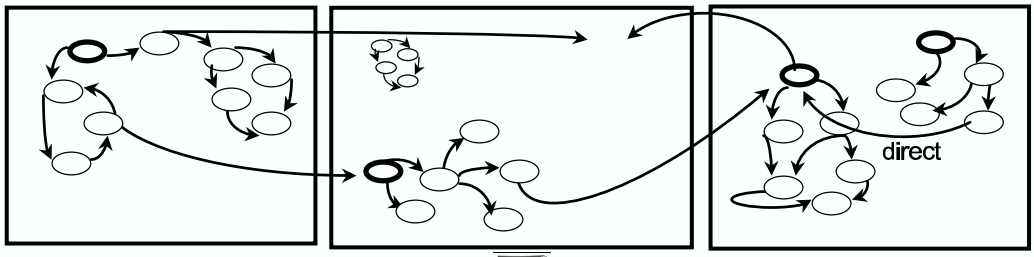
Characteristics and optimizations

Same semantics guaranteed (RDV, FIFO order point to point, asynchronous)
Safe migration (no agent in the air!)
Local references if possible when arriving within a VM
Tensionning (removal of forwarder)



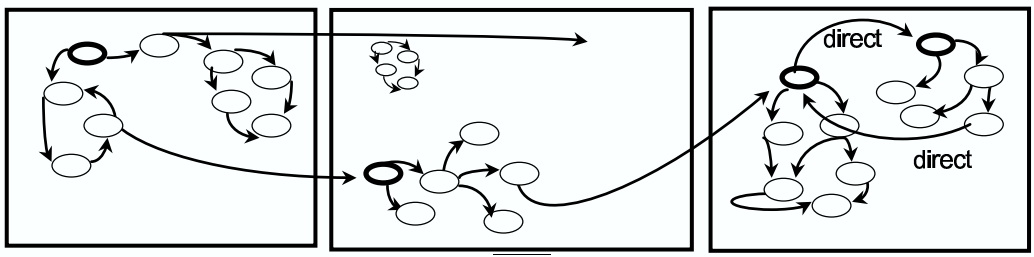
Characteristics and optimizations

Same semantics guaranteed (RDV, FIFO order point to point, asynchronous)
Safe migration (no agent in the air!)
Local references if possible when arriving within a VM
Tensionning (removal of forwarder)



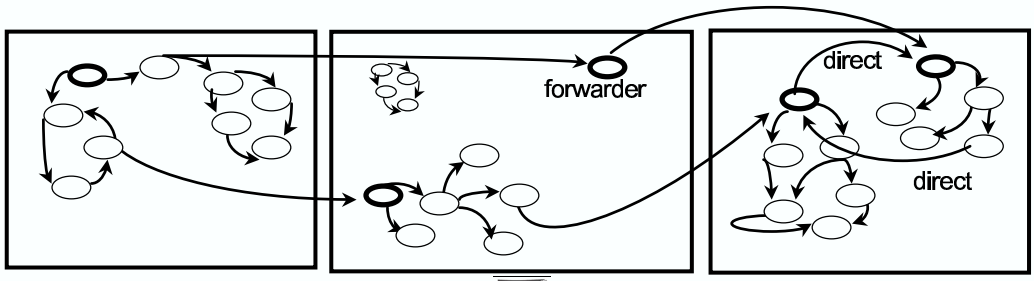
Characteristics and optimizations

Same semantics guaranteed (RDV, FIFO order point to point, asynchronous)
Safe migration (no agent in the air!)
Local references if possible when arriving within a VM
Tensionning (removal of forwarder)



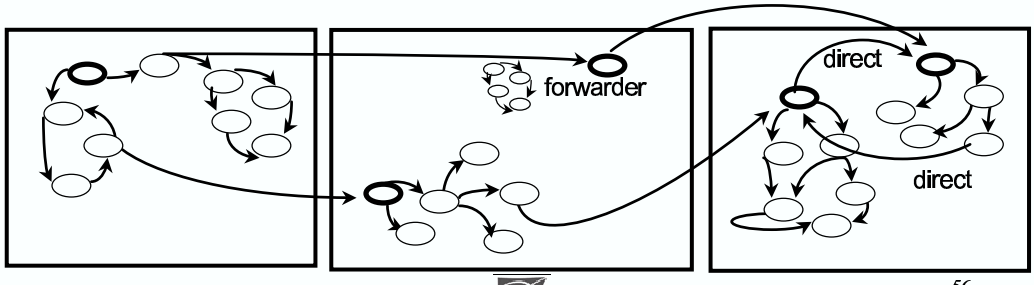
Characteristics and optimizations

Same semantics guaranteed (RDV, FIFO order point to point, asynchronous)
Safe migration (no agent in the air!)
Local references if possible when arriving within a VM
Tensionning (removal of forwarder)



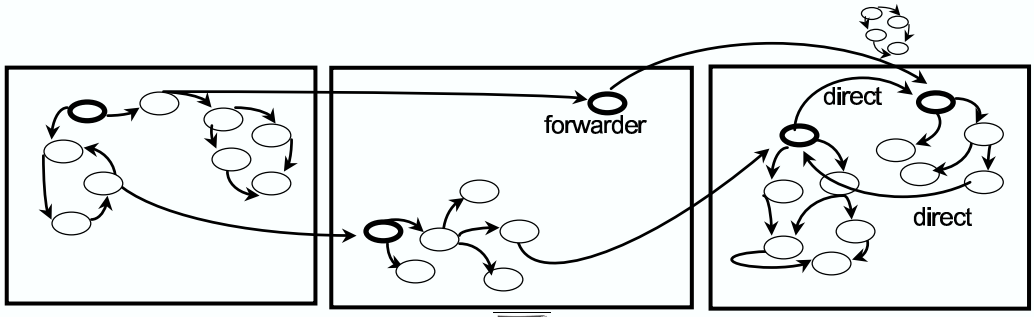
Characteristics and optimizations

Same semantics guaranteed (RDV, FIFO order point to point, asynchronous)
Safe migration (no agent in the air!)
Local references if possible when arriving within a VM
Tensionning (removal of forwarder)



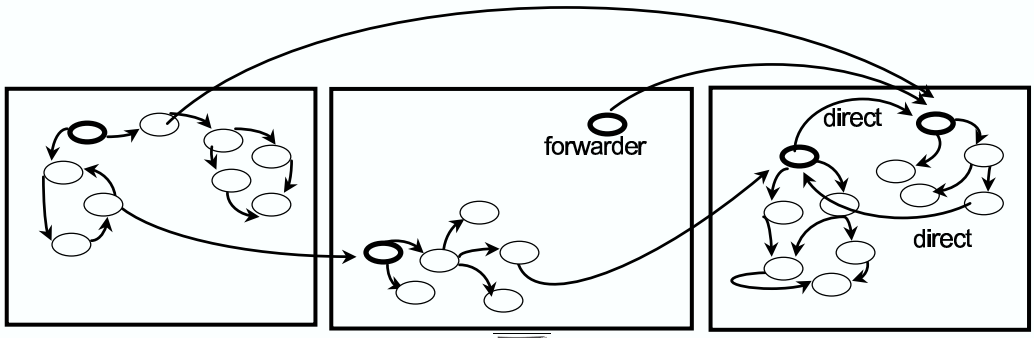
Characteristics and optimizations

Same semantics guaranteed (RDV, FIFO order point to point, asynchronous)
Safe migration (no agent in the air!)
Local references if possible when arriving within a VM
Tensionning (removal of forwarder)



Characteristics and optimizations

Same semantics guaranteed (RDV, FIFO order point to point, asynchronous)
Safe migration (no agent in the air!)
Local references if possible when arriving within a VM
Tensionning (removal of forwarder)

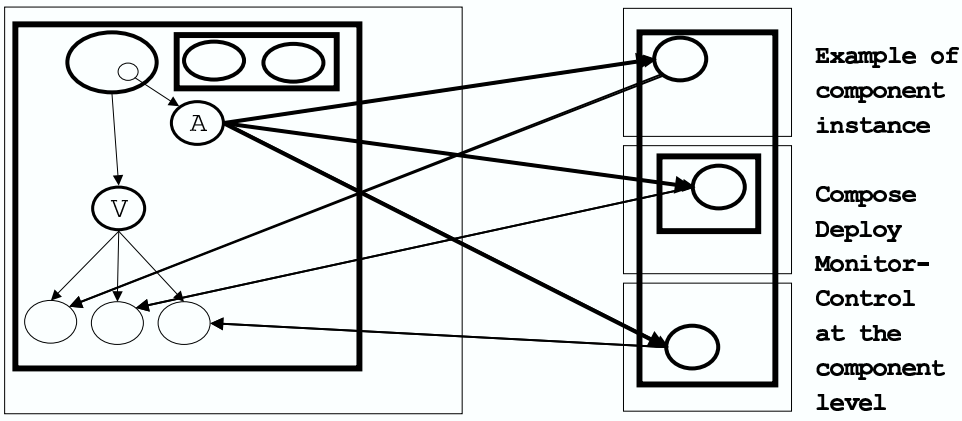


2.4 ProActive Components

- **Principles for Distributed Components**
- **Abstract Deployment model**
- **Composing Virtual Nodes**
- **Descriptors: Primitive and Composite**

Towards Distributed Components

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



Example of component instance

Compose
Deploy
Monitor-Control at the component level

Typed Group

Java or Active Object

ProActive Component Definition

A component is:

- Formed from one (or several) Active Object
- Executing on one (or several) JVM
- Provides a set of server ports (Java Interfaces)
- Uses a set of client ports (Java Attributes)
- Point-to-point or Group communication between components

Hierarchical:

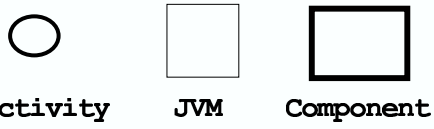
- Primitive component: define with Java code and a descriptor
- Composite component: composition of primitive components

Descriptor:

- XML definition of primitive and composite
- Virtual node captures the deployment capacities and needs

Virtual Node is a very important abstraction for GRID components

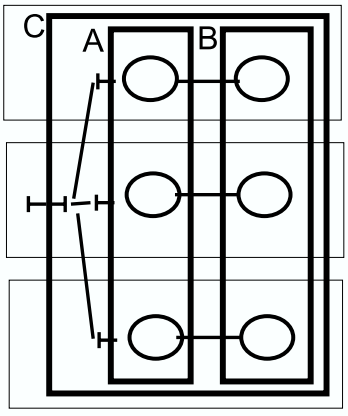
Components vs. Activity and JVMs



Cp. are rather orthogonal to activities and JVMs:
contain activities, span across several JVMs

Here, co-allocation of two components,
within a composite one,
with a collective port using group com.

Components are a way to globally manipulate distributed, and running activities



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:

- 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 >
...
```

Descriptors: Virtual Nodes in Programs

```
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() );
```

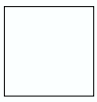
Descriptors: Virtual Nodes in Programs

```
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

Activity



JVM

Component

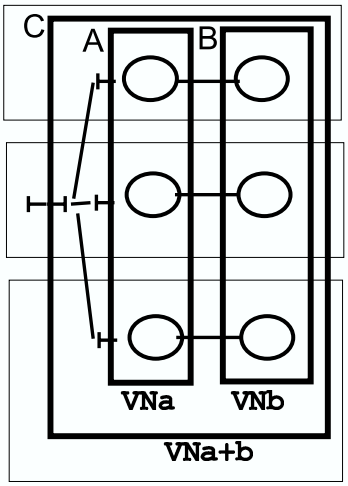
When composing A and B to form C

VNa , VNb --> 2 VNs : Distributed mapping

VNa , VNb --> VNa+b : Co-allocation

Composition can control distribution of composite

Co-allocation in a composite



Component Descriptors

- Defining Provide and Use ports (Server, Client)
- Defining Composite

- Using the Fractal component model, and
  ADL: Architecture Description Language
  [ObjectWeb, **Bruneton-Coupaye-Stefani**]

- XML descriptors
- Integration with Virtual Nodes

Descriptor Example: Primitive Component

```
<primitive-component
  implementation="test.component.car.MotorImpl" name="motor_1"
      virtualNode="Node2">
 <requires> <interface-type cardinality="single" contingency="mandatory"
  name="controlWheel"      signature="test.component.car.Wheel" />
</requires>
 <provides>  <interface-type name="controlMotor"
  signature="test.component.car.Motor" />  </provides>
</primitive-component>
```

Descriptor Example: Composite Component

```
<composite-component name="composite2" virtualNode="Node2">
 <provides>  <interface-type name="controlComposite2"
             signature="test.component.car.Motor" />
 </provides>
<composite-component name="composite1" virtualNode="Node2">
      <provides>
            <interface-type name="controlComposite1"
                  signature="test.component.car.Motor" />
      </provides>
  <primitive-component …..
```

Not to be written nor read by humans !!

TOOLS

2.5 Tools for Distributed Objects and Components

- GUI: IC2D: Interactive Control and Debugging of Distribution
graphical visualization and control

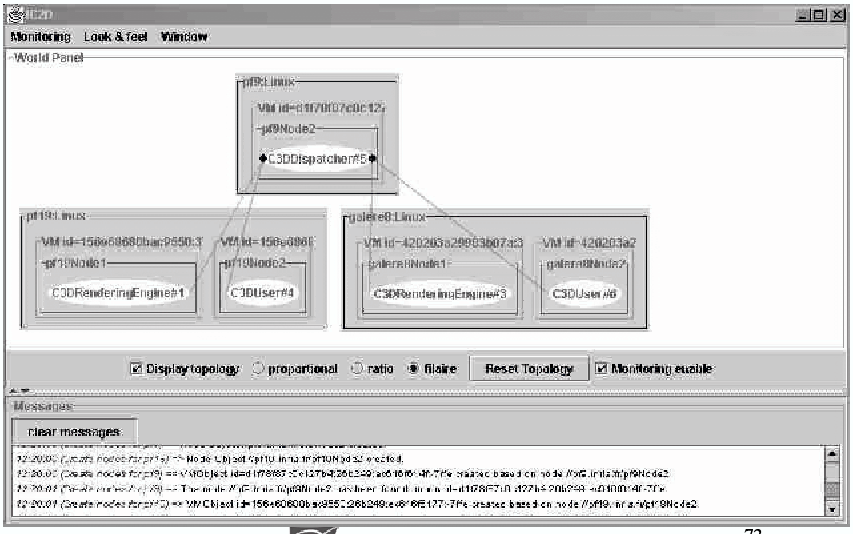
- Component Tools: Composing, Deploying

- Screenshots or Demo

IC2D: Interactive Control and Debugging of Distribution

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

- Logical Clock



IC2D: Basic features

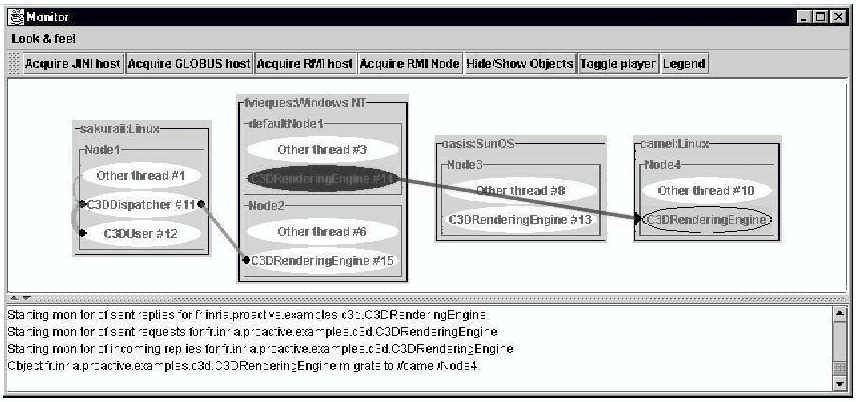
Graphical Visualisation:
- Hosts, Java Virtual Machines, Nodes, Active Objects
- Topology: reference and communications
- Status of active objects (executing, waiting, etc.)
- Migration of activities

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: Dynamic change of Deployment Drag-n-Drop Migration

Drag-n-Drop tasks around the world



IC2D: Related Events

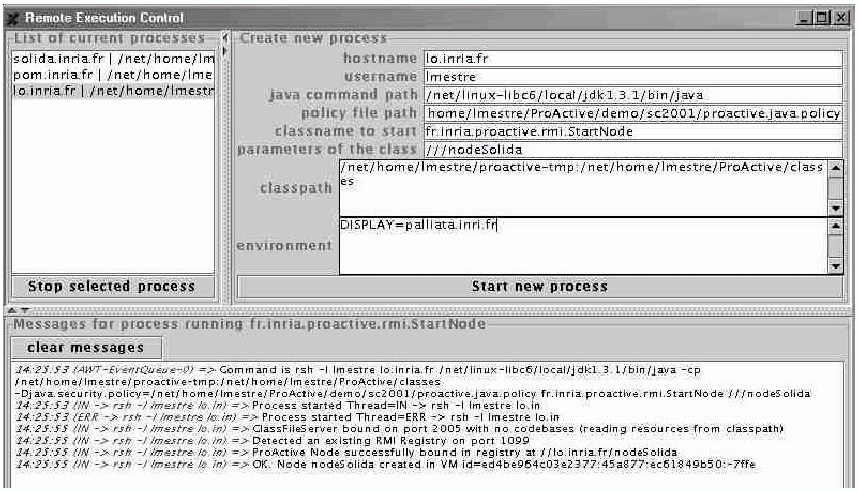
Events:

- 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

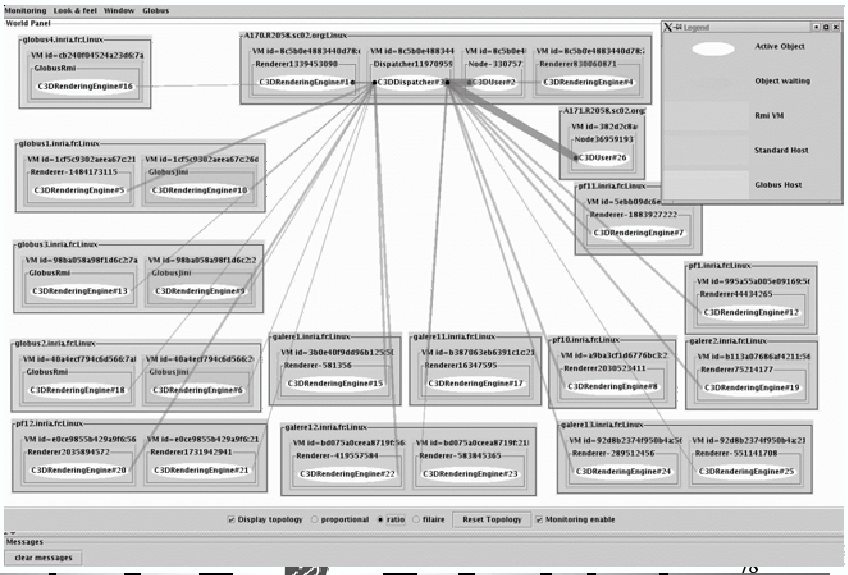
Creation, Acquisition of new JVMs, and Nodes

Protocols: rsh, ssh Globus, LSF



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

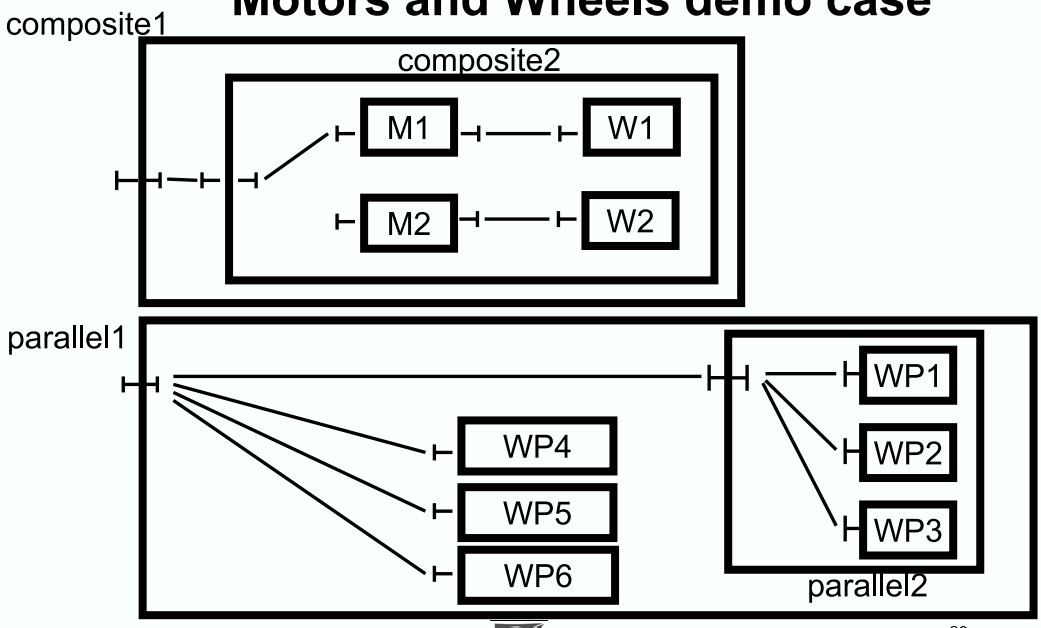
Width of links proportional to the number of com-munications



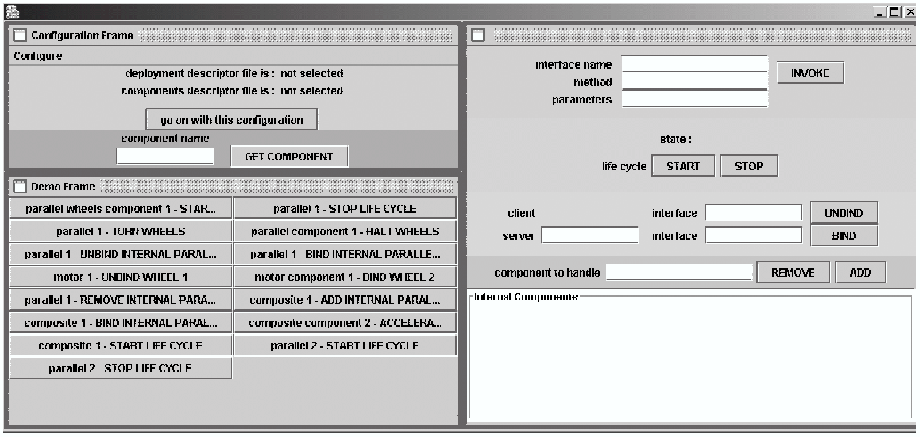
DEMO:
Components with the IC2D monitor

- A simple Motors and Wheels demo case
- Parallel:
  - several instances of components with collective interface
- Composite:
  - 3 levels of imbrication
- Level 3 component orientedness:
  - life cycle management, rebinding, in and out

Motors and Wheels demo case



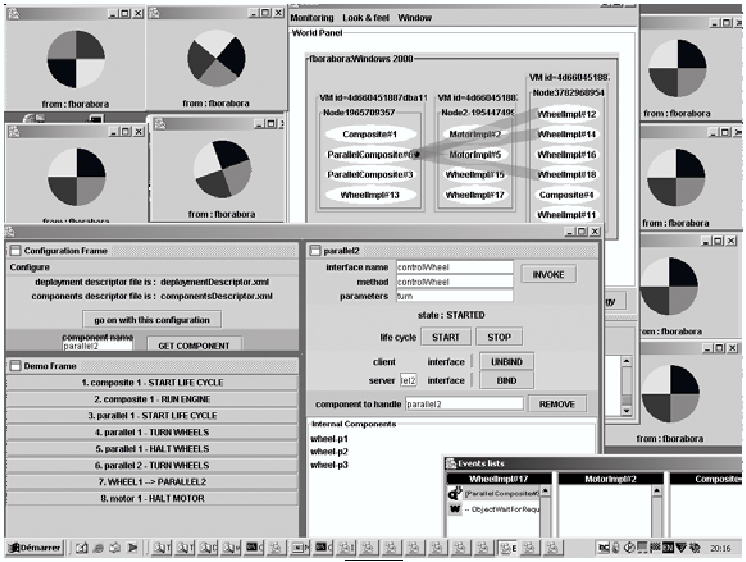
Component Manipulation



Selecting: component and deployment descriptors
DEPLOY

Managing: life cycle, rebinding, in and out

Component Interface with IC2D



ProActive Components: Characteristics

Components characteristics:

Software module =
- Java Classes and Interfaces + threads: forming Active Objects

Standardized description =
- In source: Virtual Nodes, newActive API
- .ProActiveDescriptor : an XML file per component

Tools:
- Composition = working on: IC2D--Compose
- Deployment = Java VM, and IC2D (Deploy + Monitor)

Next steps

- Interactively compose components with the component view
- Maintain component view at execution
- Formal Semantics of mixing:
  - Functional, with
  - Non Functional calls (start/stop, rebind, in/out, …)