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)

Table of Contents (2)

Table of Contents (3)

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 !

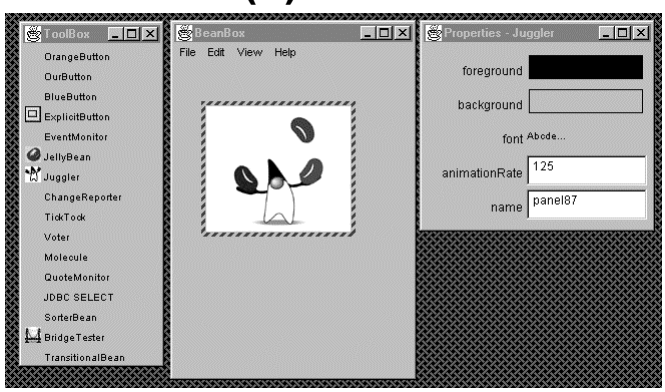
The Java introspection allows to discover dynamically the properties,
and to configure them, assemble JB interactively

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

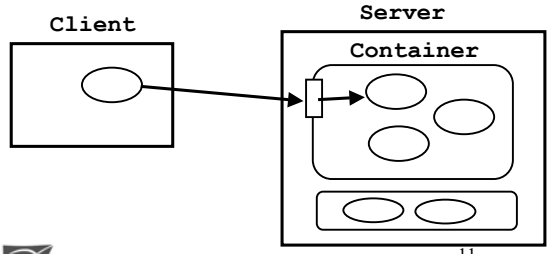
Components have to be configured on their Non-Functional Properties:

- 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, …)

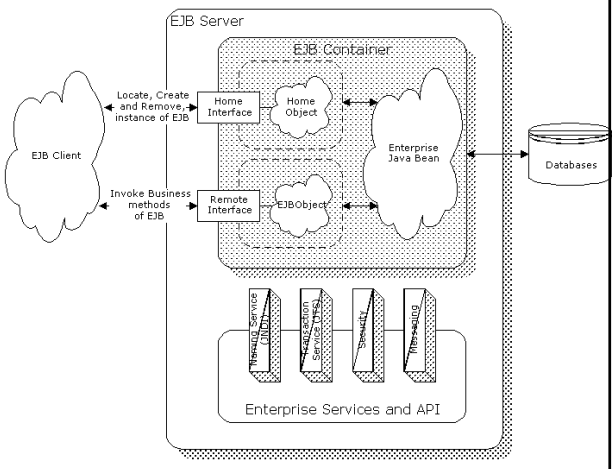
Only Provides (server), no Uses

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

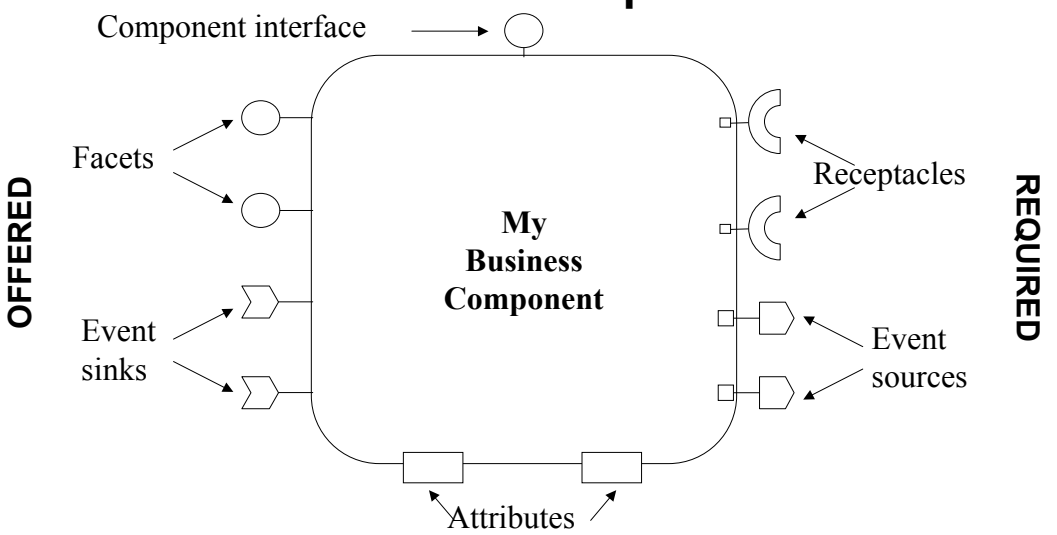
A Web Service: the instance of a component, … running...

1.5 Assembly of Components Corba 3 and CCM

CCM: Corba Components Model =

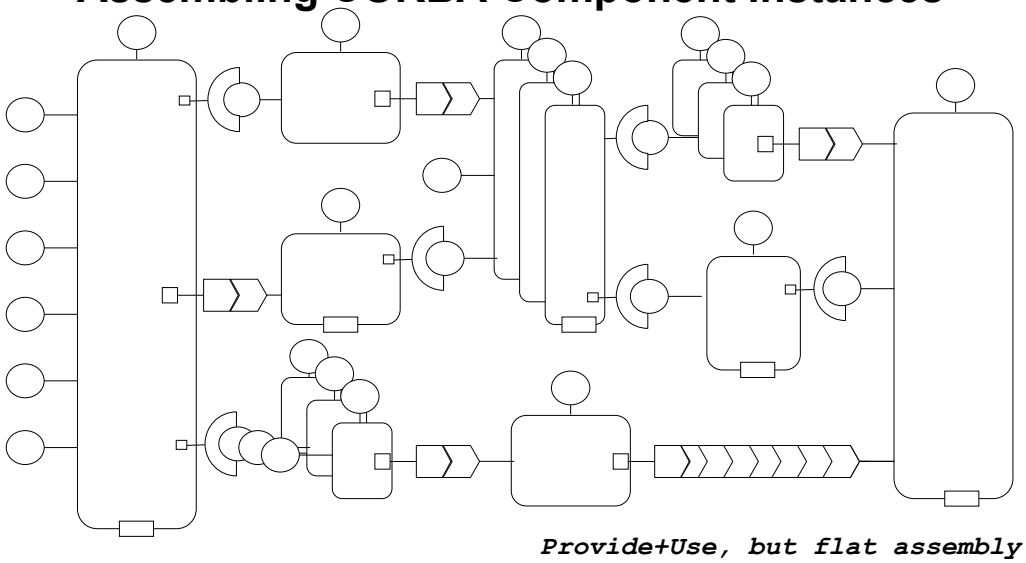
- EJB + a few things :
  - More types of Beans defined:
    - **Service**, Session, **Process**, Entity, ...
  - Not bound to Java (Corba IDL)
  - Provides but also Uses :
    - Specification of the component needs, dependencies
    - "Client Interfaces''
  - A deployment model (ongoing at OMG)

A CORBA Component



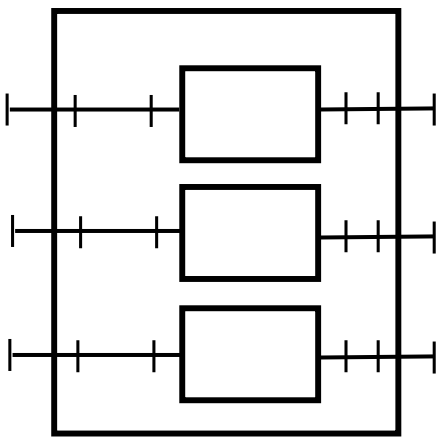
Courtesy of Philippe Merle, Lille, OpenCCM platform

Building CCM Applications = Assembling CORBA Component Instances



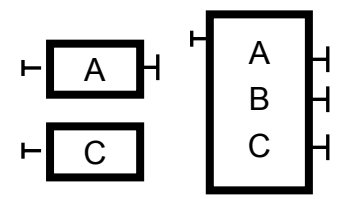
Provide+Use, but flat assembly

1.6 Hierarchical Component Fractal model

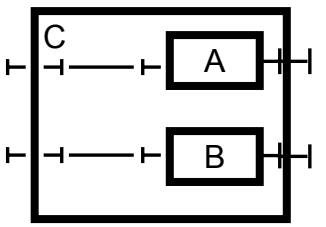


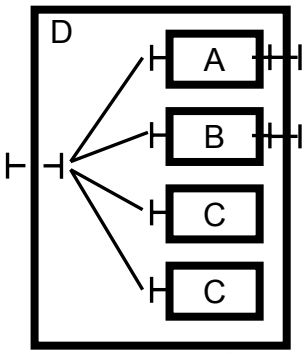
Defined by E. Bruneton, T. Coupaye, J.B. Stefani, INRIA & FT

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

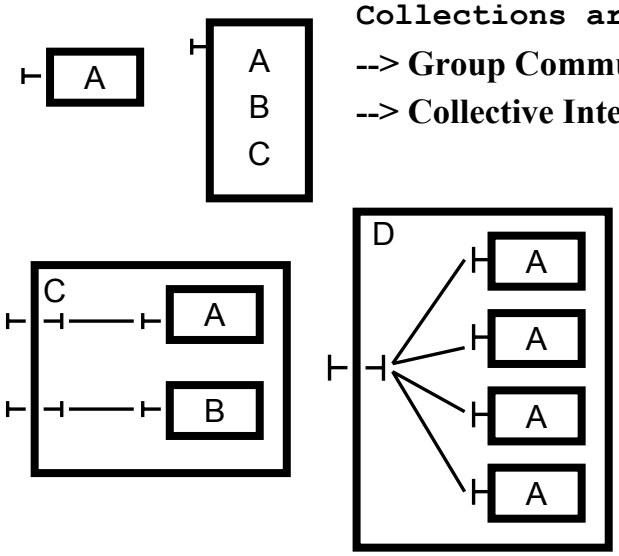
- Level 1: Instantiate - Deploy - Configure
  - Simple Pattern
  - Meta-information (file, XML, etc.) — JavaBeans, EJB
- Level 2: Assembly (flat)
  - Use and client interfaces — CCM
- Level 3: Hierarchic
  - Composite — Fractal
- Level 4: Reconfiguration
  - Binding, Inclusion, Location — On going work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

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

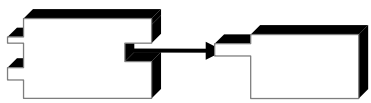
Java and C++ components

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?

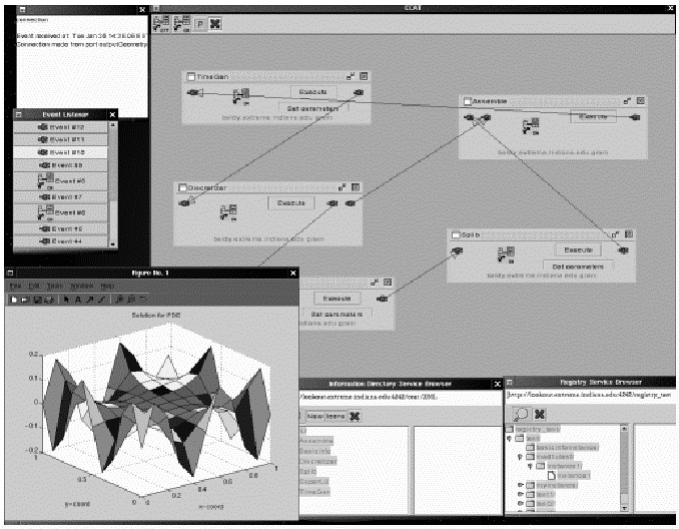
- Use Remote Procedure Call (RPC) Mechanism
  - XCAT uses **SOAP 1.1**
  - XCAT ports can serve as web services
- Events/Messages
  - Objects encoded as XML documents

CCAT session with Java GUI: Composition

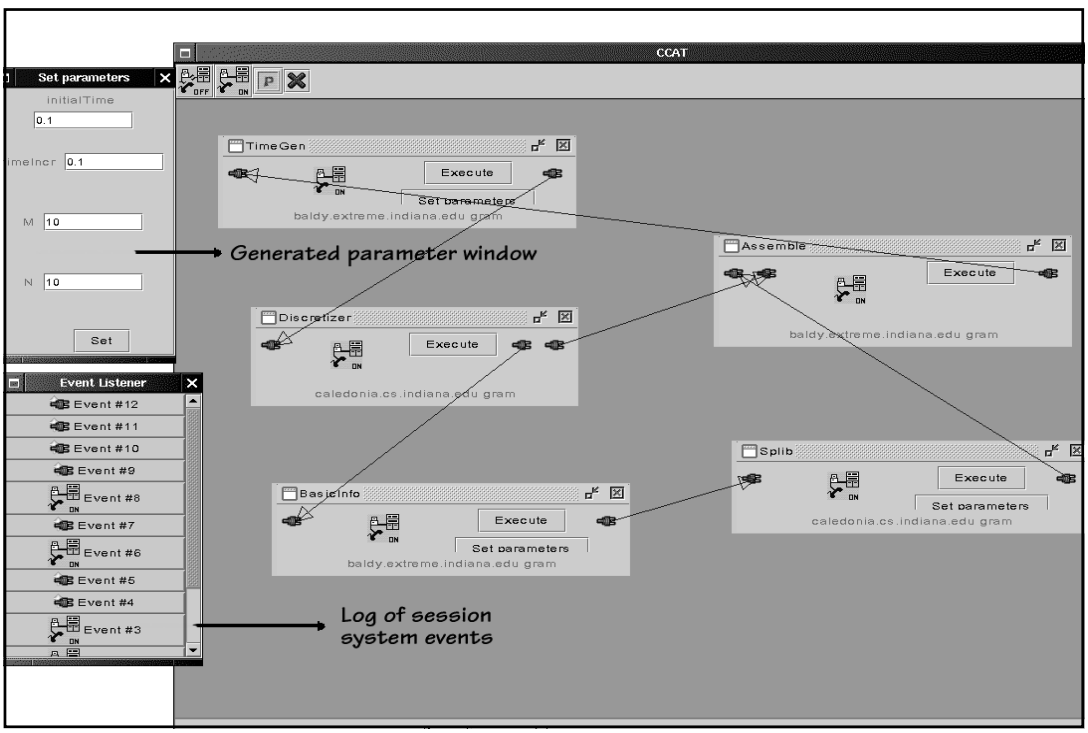
Composition Tool:

- Select
- Connect
- But also:
  Test and Execute

Level 2 Provide/Use model



Courtesy of D. Gannon et al.

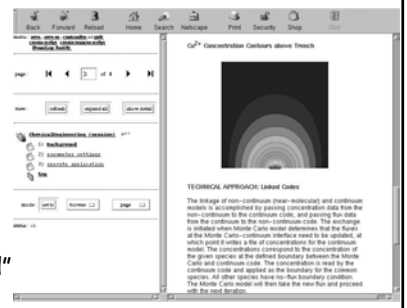


XCAT:

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

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

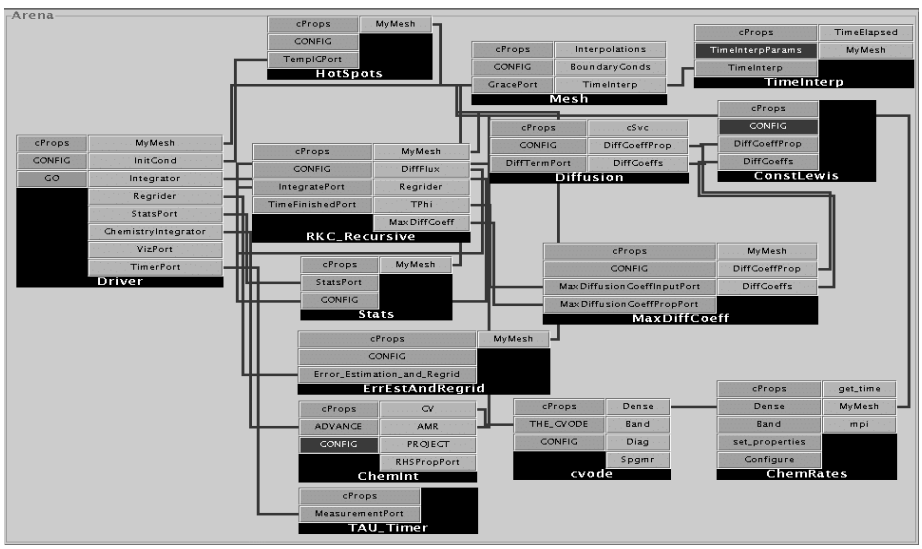
GUI and Scripted Interface

Interactive or Batch

Serial or Parallel

Components written in C++

Ccaffeine



CCA Characteristics
XCAT, CCAT, Ccaffeine, ...

So for CCA, etc.:

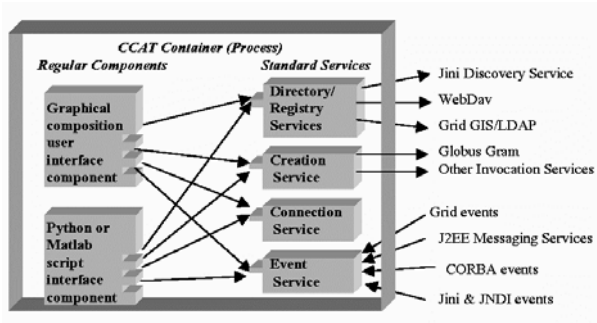
Software module =
- Any Code + wrappers

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:

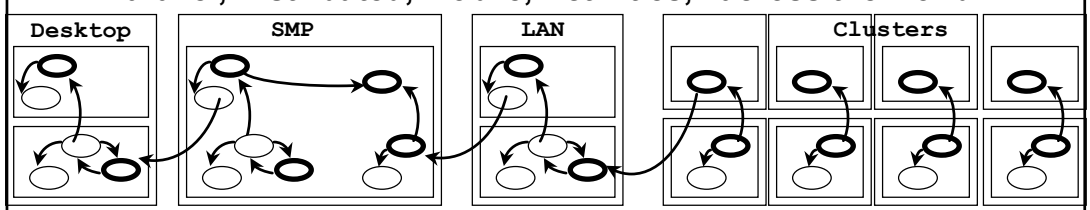
- IC2D GUI

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.**

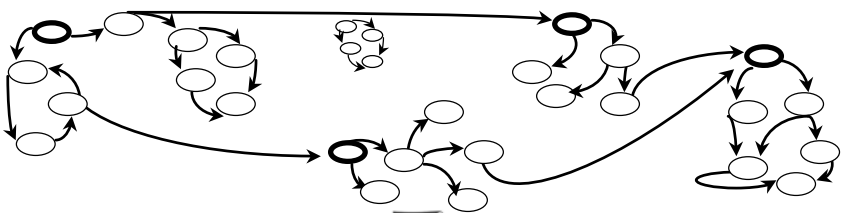
Main features:

- **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 { … }
```

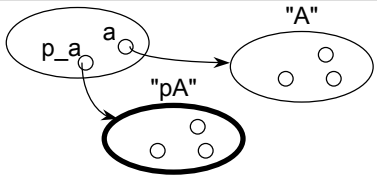
- **Object-based:**

```
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 (...);
}
```

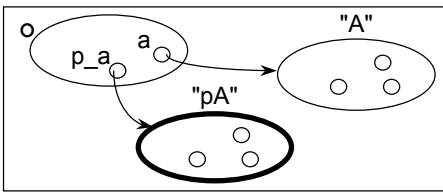
- Wait-by-necessity: inter-object synchronization
  - Systematic, implicit and transparent futures

    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)




```
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 synchronization, and the reuse of routines

Group Communications

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

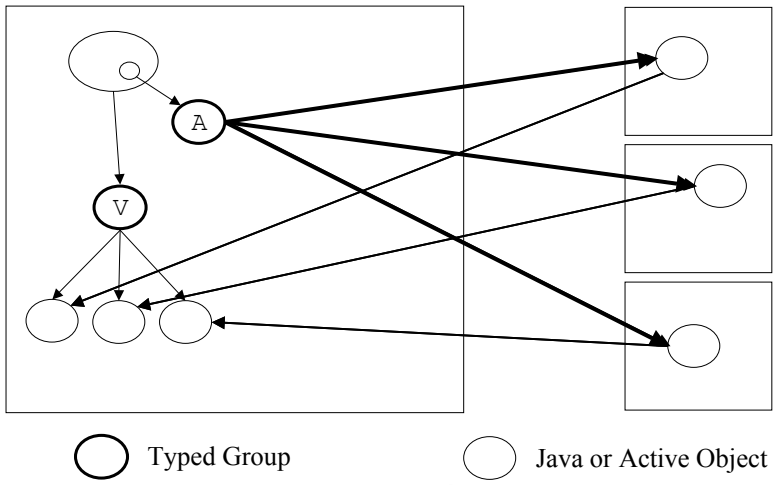
  ⇨ Typed and polymorphic Groups of active and remote objects
  Dynamic generation of group of results

- 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:

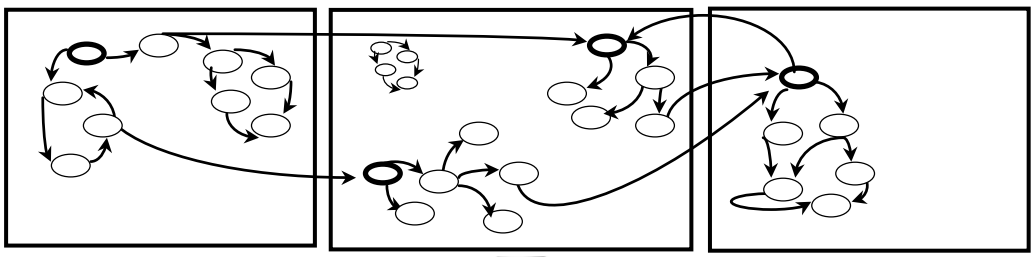
- all pending requests
- all its passive objects
- all its future objects

Automatic and transparent forwarding of:

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

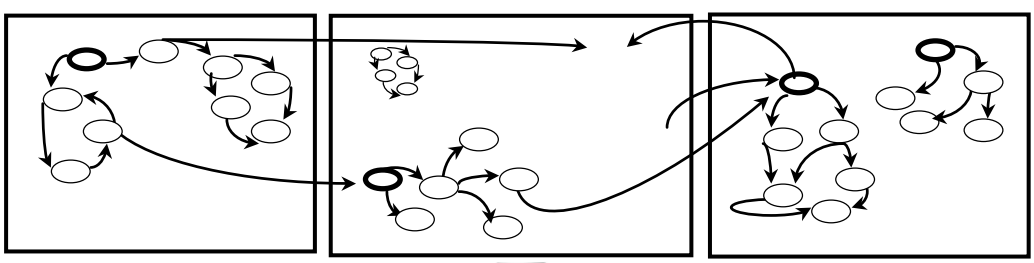
Characteristics and optimizations

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)



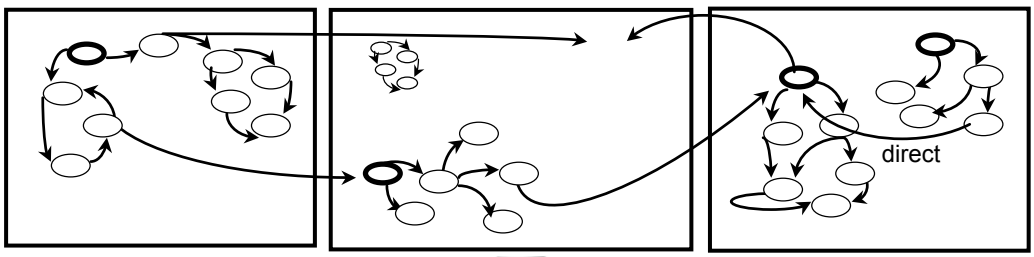
Characteristics and optimizations

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)



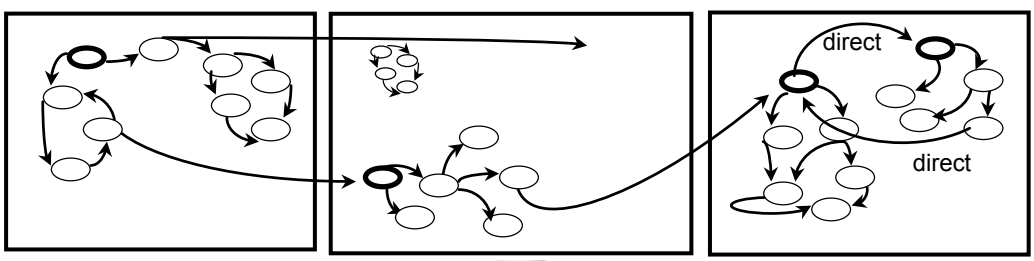
Characteristics and optimizations

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)



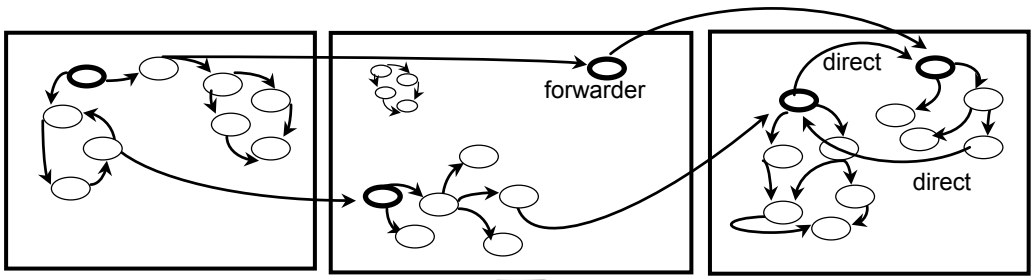
Characteristics and optimizations

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)



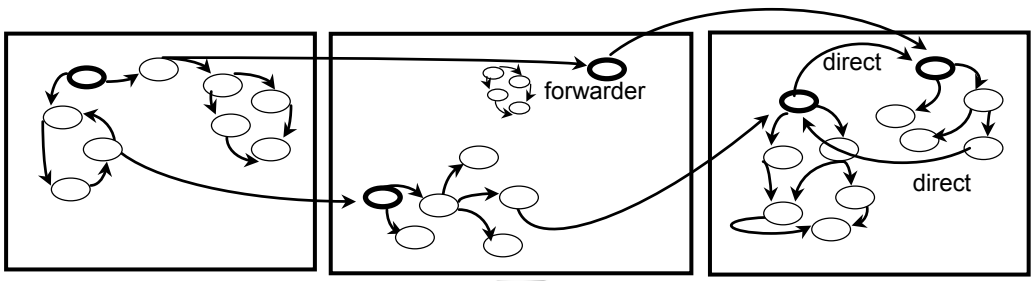
Characteristics and optimizations

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)



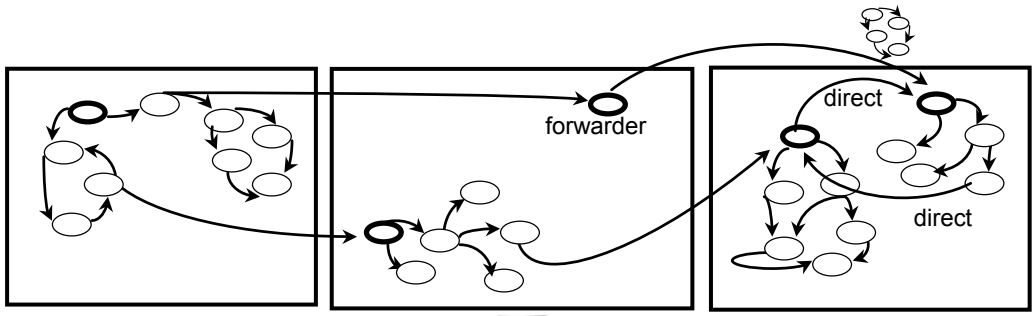
Characteristics and optimizations

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)



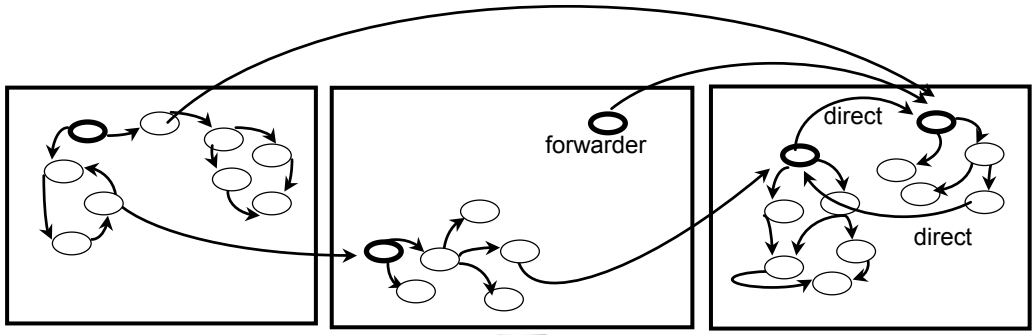
Characteristics and optimizations

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**

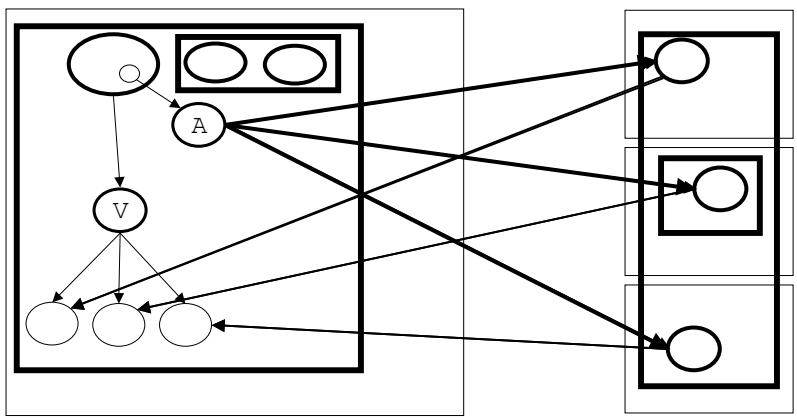
- **Abstract Deployment model**

- **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:

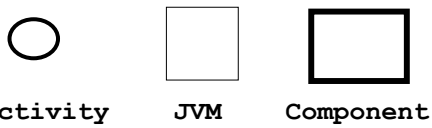
- Primitive component: define with Java code and a descriptor
- Composite component: composition of primitive components

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

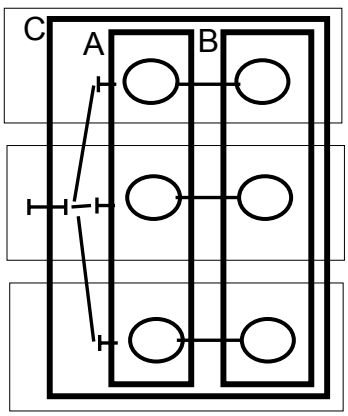


Cp. are rather orthogonal to activities and JVMs:

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

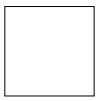
```
Descriptor pad = ProActive.getDescriptor ("file:.ProActiveDescriptor.xml");
VirtualNode vn = pad.activateMapping ("Dispatcher");  // Triggers the JVMs
Node node = vn.getNode();
...
C3D c3d = ProActive.newActive("C3D", param, node);
                log (  ... "created at:  " + node.name() + node.JVM()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



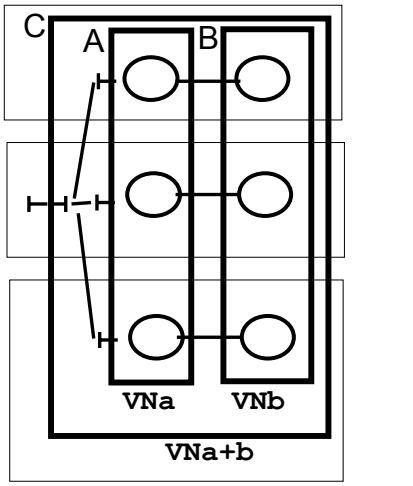
JVM

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

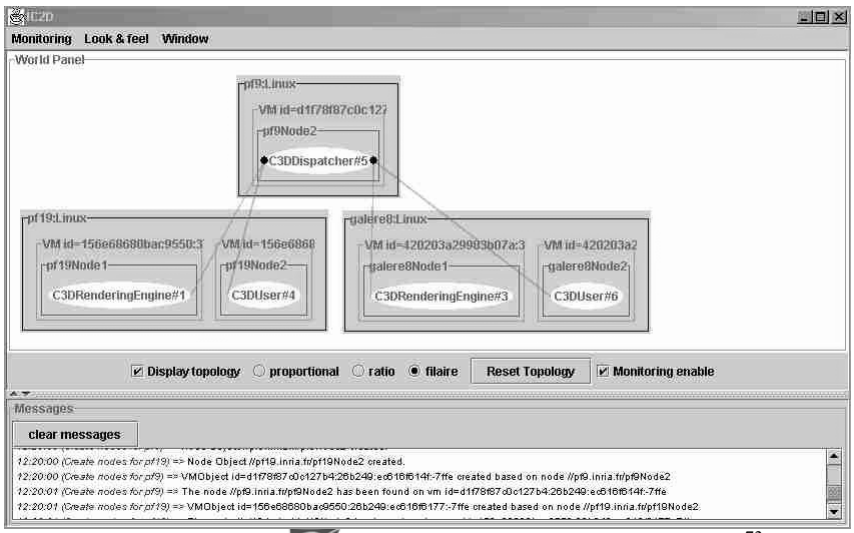
- **Component Tools: Composing, Deploying**

- **Screenshots or Demo**

IC2D: Interactive Control and Debugging of Distribution

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

- Logical Clock



IC2D: Basic features

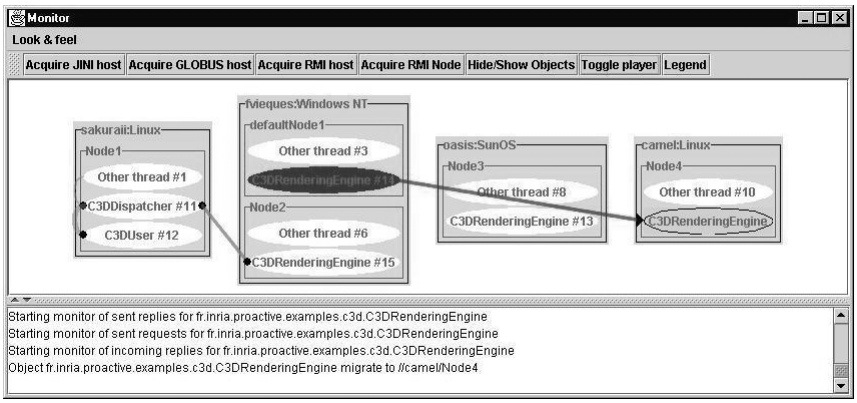
Graphical Visualisation:
- Hosts, Java Virtual Machines, Nodes, Active Objects
- Topology: reference and communications
- Status of active objects (executing, waiting, etc.)
- Migration of activities

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

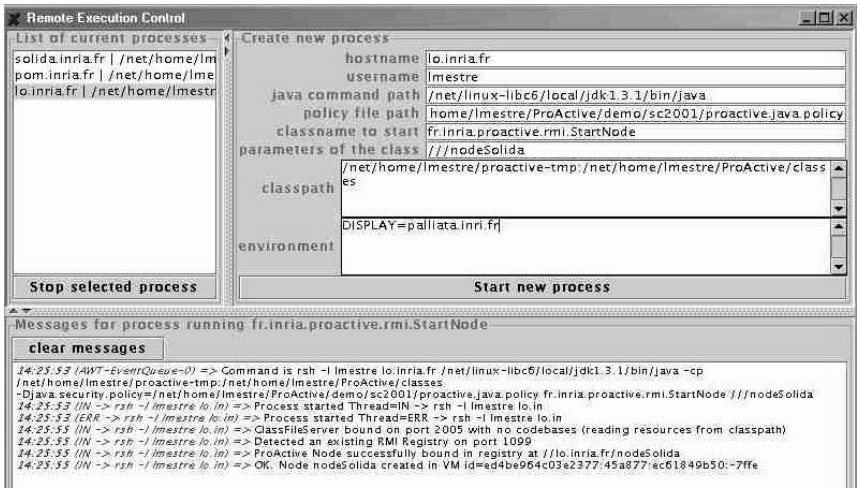
Events:

- 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

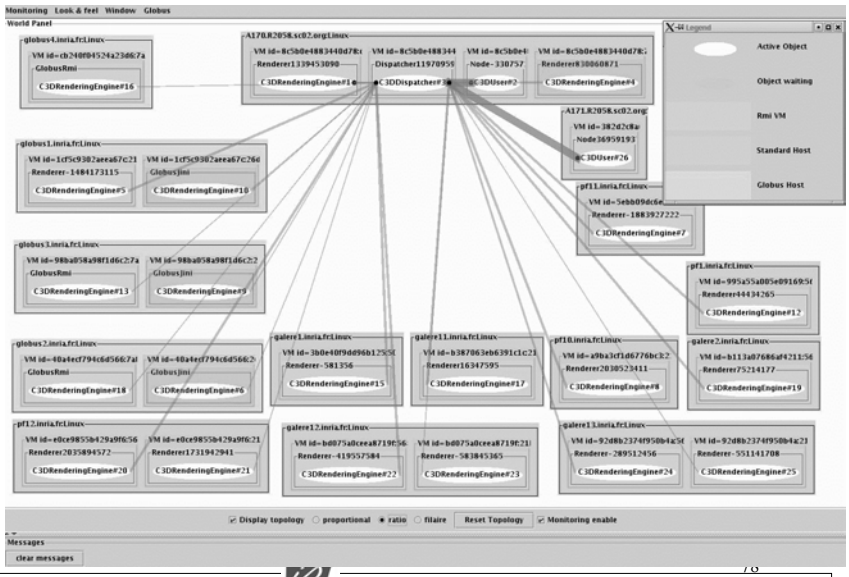
Creation, Acquisition of new JVMs, and Nodes

Protocols: rsh, ssh Globus, LSF



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

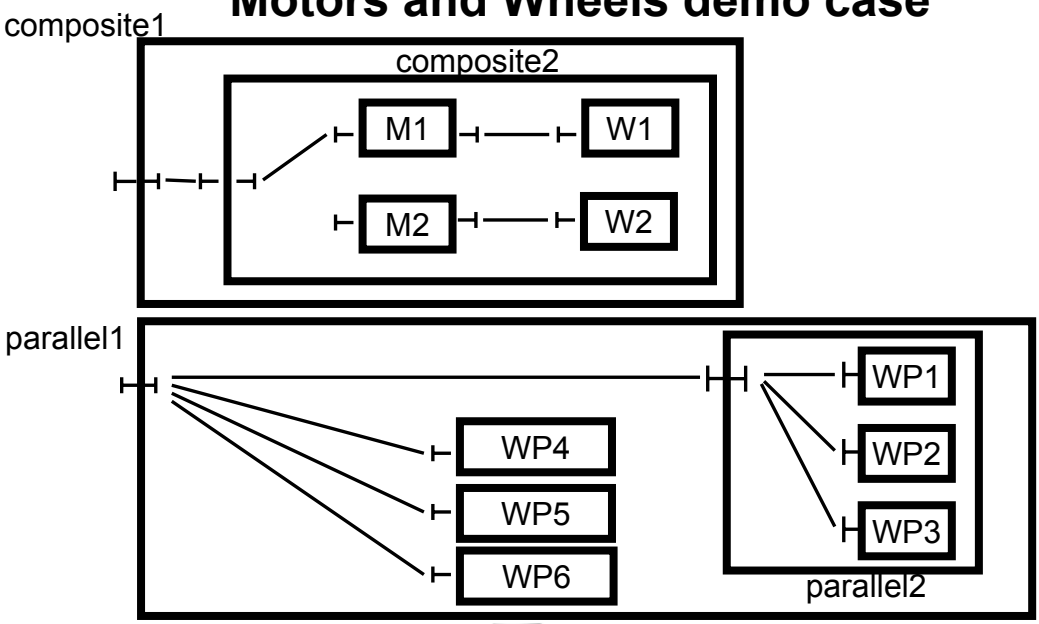
Width of links proportional to the number of com-munications



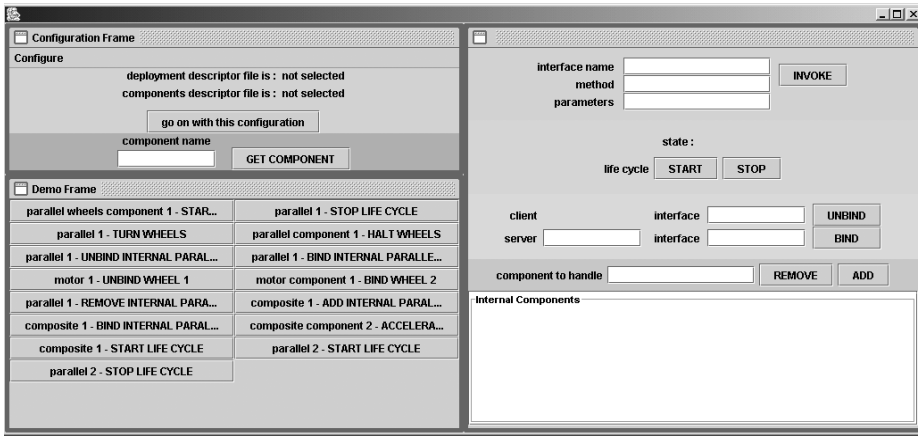
DEMO:
Components with the IC2D monitor

- 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

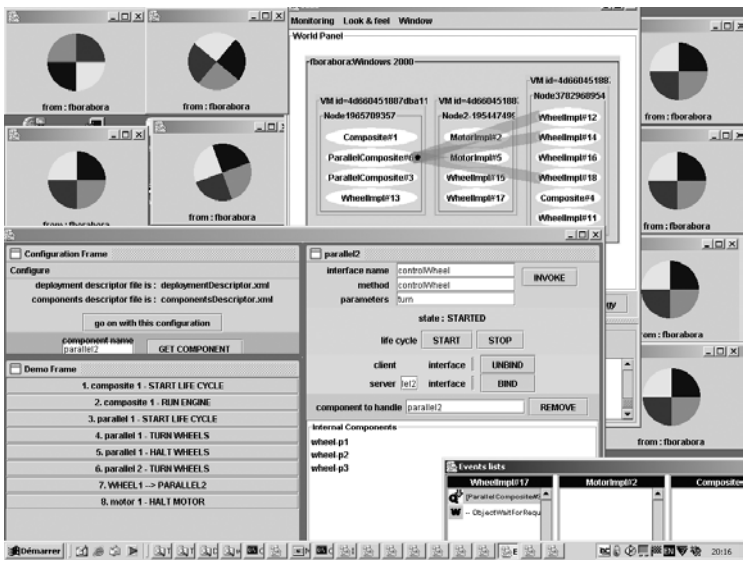


Component Manipulation



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)

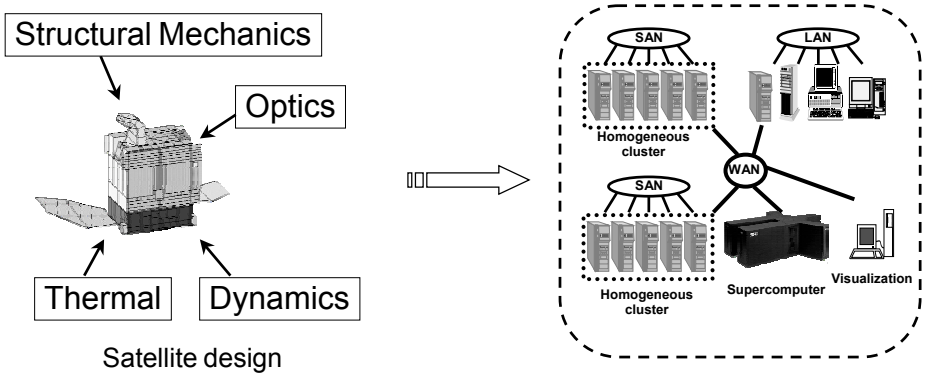
Next steps

- 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, …)

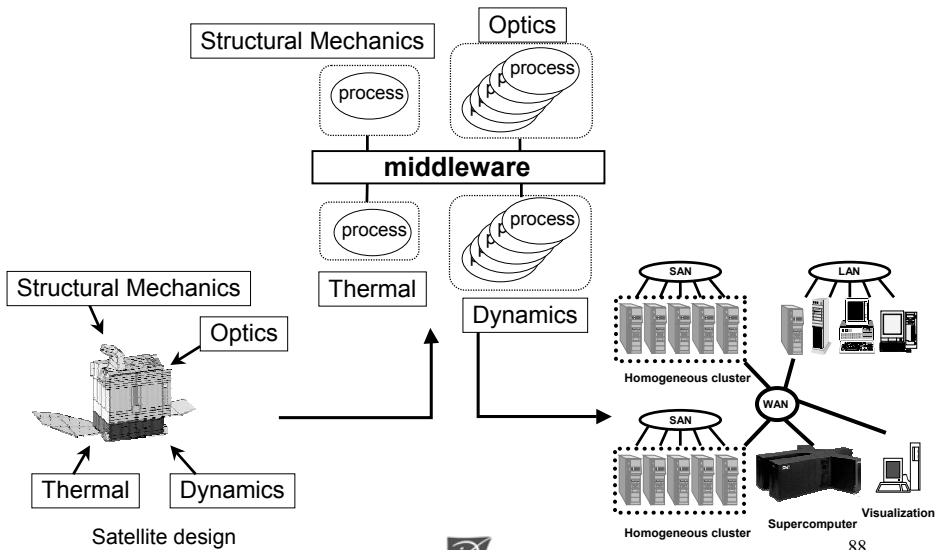
3. Parallel CORBA Objects and Components

Contents

Code Coupling on Grids



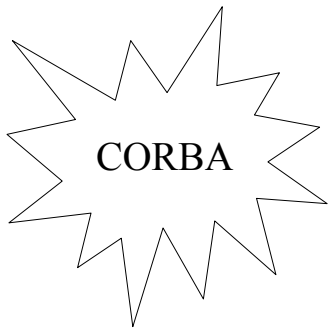
Coupling Through a Middleware



Features to support

Re-use legacy codes

- Support several languages
- Support parallel codes



Support of heterogeneous machines

Dynamical code interconnection

Transfer data & control

- Message passing (MP) vs remote method invocation (RMI)

Communication Models

Message Passing (MP)

- Explicit receive operation
- User has to poll/wait

Remote Method Invocation (RMI)

- Method implicitly called
- Better in multithreaded environments / for code independence

Simulating MP on top of RMI or RMI on top of MP

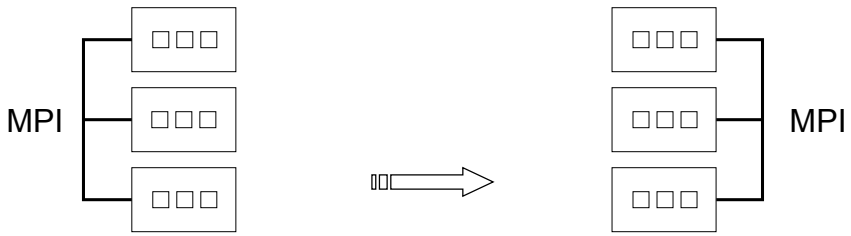
- It is known that each paradigm can be simulated on top of the other

RMI appears better as a foundation

Higher-level abstractions (like PAWS) provide their best-suited models

Communication between codes

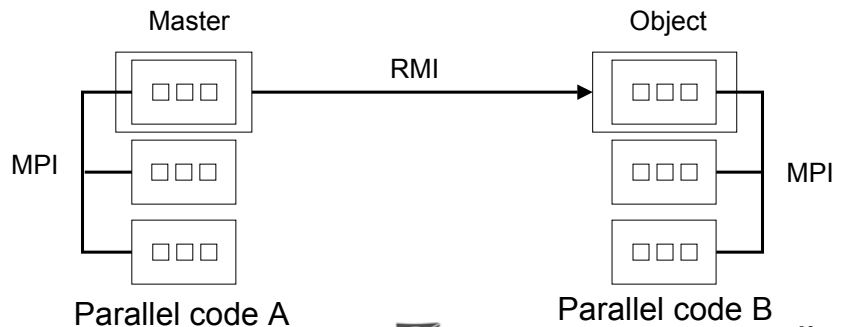
How to transfer distributed data between parallel codes ?



Communication between codes

Embedding **a** process into **an** object

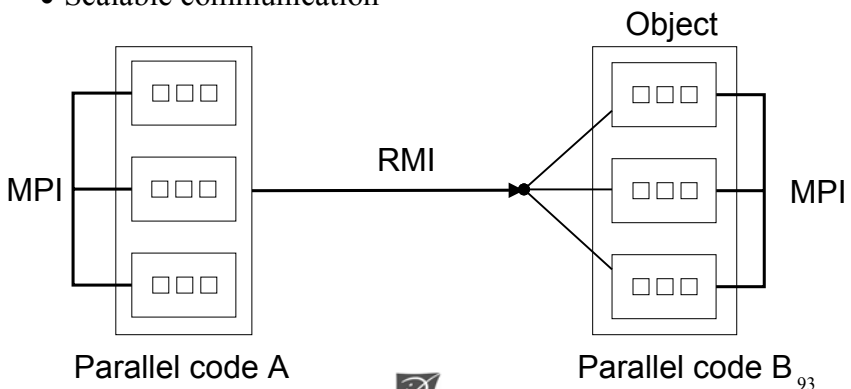
- Communication scalability issue
- Code modification



Communication between codes

Embedding ***all*** processes into an object

- Parallel information associated to an object
- Scalable communication



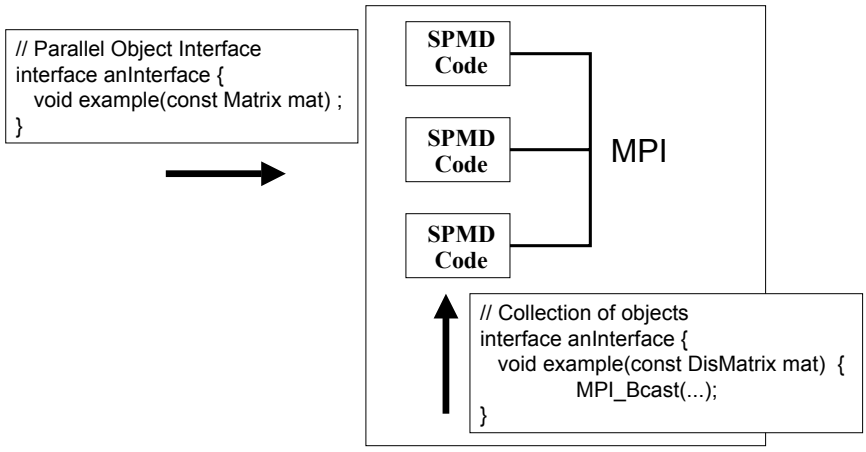
Definition of a parallel object

Definition : *A parallel object is an object whose execution model is parallel.*

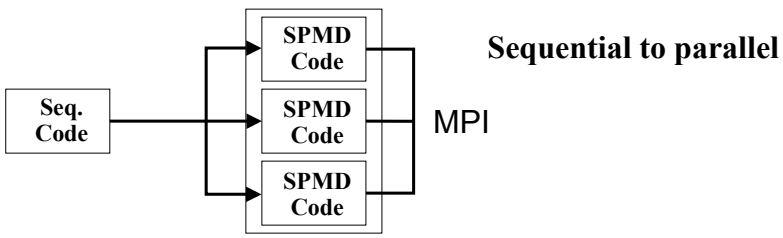
In practice,

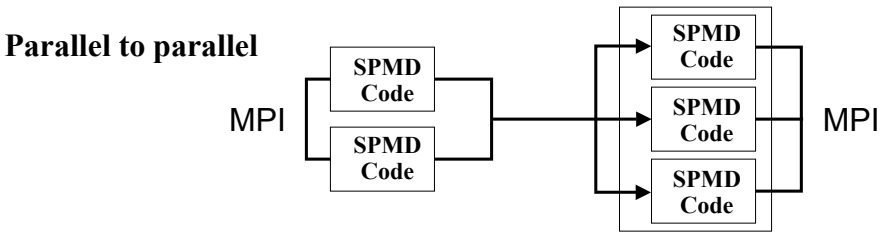
- A parallel object is often incarnated by a collection of object.
- The execution model is Single Program Multiple Data
  - *Invoking a method on a parallel object invokes the corresponding method on all the objects of the collection.*

An example of a parallel object



Examples of invocations of a parallel object





Contents

Performance of CORBA

« Towards High Performance CORBA and MPI Middleware for Grid Computing »

- Alexandre Denis, Christian Pérez and Thierry Priol
- Presented at GridComputing'01

Myrinet-2000 network

- CORBA performance : OmniORB / PadicoTM
  - Bandwidth : 240 MB/s of 250 MB/s
  - Latency : 20 µs
- MPI performance : MPICH
  - Bandwidth : 240 MB/s
  - Latency : 11 µs

Object oriented middleware systems

Parallel objects

- ParDIS: *K. Keahey and D. Gannon*
- PaCO: *C. René and T. Priol*
- Data Parallel CORBA: *OMG*
- PaCO++: C. *Pérez, T. Priol and A. Ribes*

Main differences

- Description of the parallelism
- Support for distributed data

ParDIS

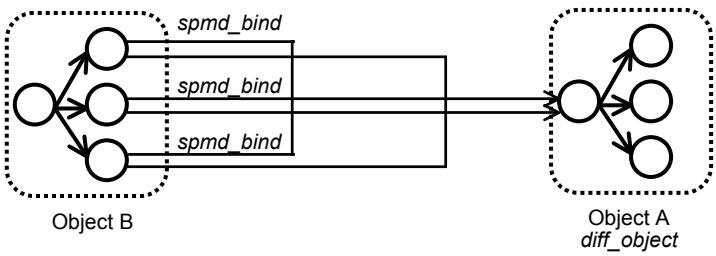
Developed by K. Keahey and D. Gannon (U. of Indiana)

Model

- Extension of CORBA objects to "SPMD objects"
- Concept of distributed sequences
- Mechanism of "future" to handle asynchronous requests

IDL specification

```
typedef dsequence <double,1024,
                   (BLOCK,BLOCK)> MyArray;

interface diff_object {
     void diffusion(     in  long     timestep,
                         inout MyArray darray);
};
```



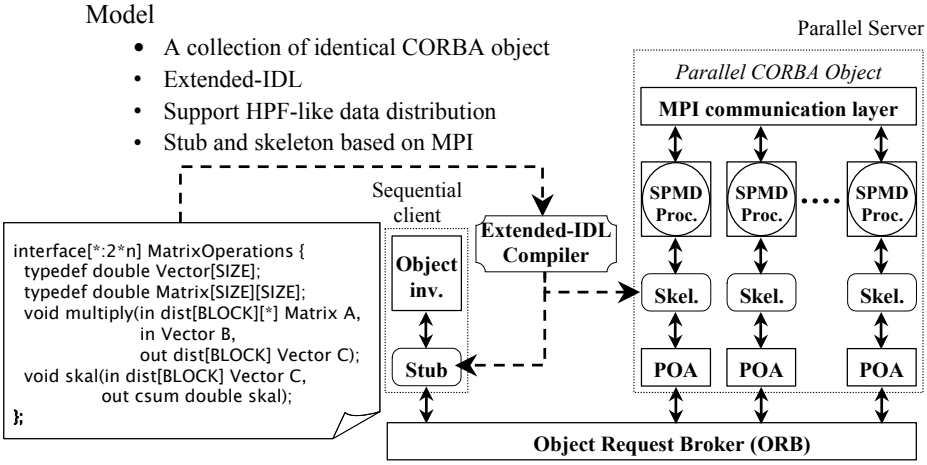
PaCO

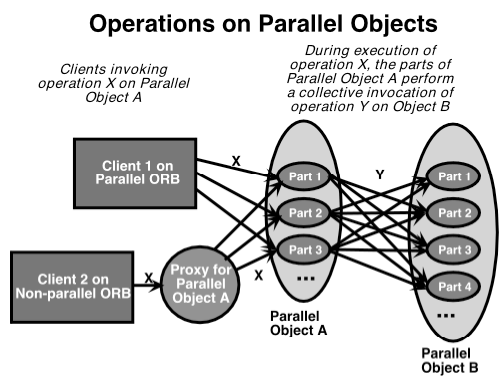
Developed by T. Priol and C. René (PARIS Research team)

- Available at http://www.irisa.fr/paris/

Model

- A collection of identical CORBA object
- Extended-IDL
- Support HPF-like data distribution
- Stub and skeleton based on MPI





Data Parallel CORBA

OMG initiative to extend CORBA

Standardization procedure completed

- Technology adopted : orbos/2001-11-09

Model

- Optional ORB feature
- Runtime-based solution

- No IDL modification
- New Data-Parallel POA
- Explicit parallelism
- No data redistribution
- Interoperability (Proxy)

Contents

PaCO++ objectives

Extends CORBA

- No modification of CORBA specifications
- Parallelism is a non-functional property of an object implementation
- Implementation on top of existing CORBA implementations

Parallel object

- Collection of sequential object
- SPMD execution model
- Support of parallel operations with distributed arguments
- Support for redistribution libraries as plug-in
- Interoperability with standard CORBA objects

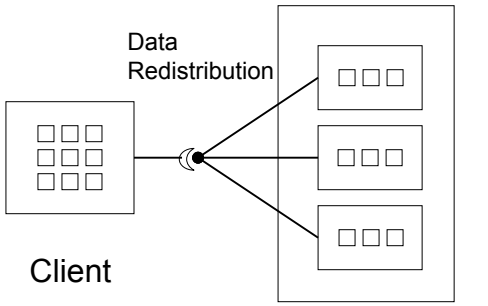
A simple example

```
// IDL
interface MyInterface  {
    void example(in Matrix mat);
  };
```

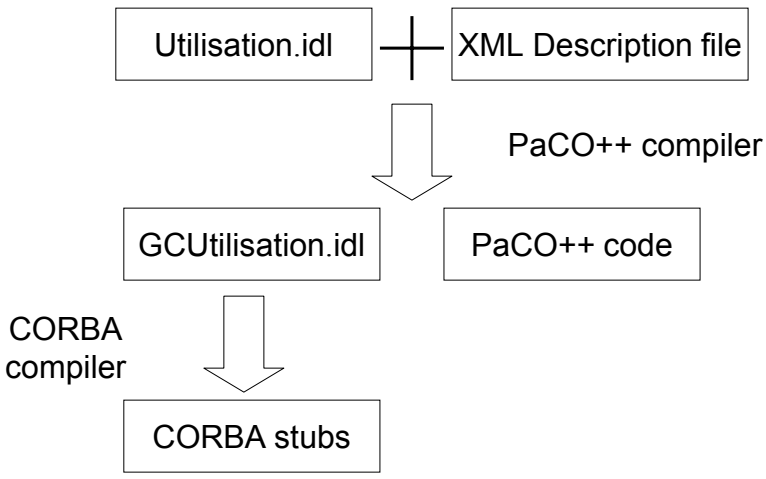
```
// XML
Name: MyInterface.example
Type: Parallel
Argument1: *, bloc
ReturnType: noReduction
```

```
// Code SPMD on the client
o->example(m);
```

```
// Code SPMD on the server
class MyInterface_impl : public MyInterface_serv {
        void example(const Matrice_serv mat) {
                .... MPI_Bcast(...) ... ;
        }
}
```



Code generation



Example

Parallel client

- A bloc-distributed vector

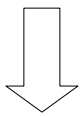
Parallel server

- A method expecting a block-distributed argument

Example : IDL

```
interface example
 {
   typedef sequence<long> lseq;
   void send(in lseq vect);
 };
```

```
XML Description

// declare 1st argument of send
// to be of type parallel
...
```



```
interface example : PaCO_Proxy {...};   // for the clients
interface example_serv : PaCO_Node  // for the implementer of the service
{
  void send( in PaCOData_vect vect);
};
```

Example: sequential client code

```
// Retrieving a standard CORBA reference
CORBA::Object obj = ...

// Obtaining a correctly type reference
example_ptr ex_obj = example::_narrow(obj) ;

// Initializing some data

long vect[40];
...

// Real call to the (parallel) object
ex_obj->send(vect);
```

Example: the parallel client code (1)

```
// Retrieving parallel view from a standard CORBA reference
example ex_obj = ... ;
PaCO_example * server = PaCO_example::get_PaCO_example(ex_obj);

// Retrieving the parallel context of the parallel operation
PaCO_operation_client * send_ctx = server->getContext("send");

// Describing client-side data distribution: a bloc distribution
PaCO::Paco1DBlockData_t data;
data.gd.len = 10 * numprocs;       // let assume 10 elements per node
data.gd.unit_size = sizeof(long);
data.ld.rank = myid;
data.ld.start = 10 * myid;
data.ld.len   = 10;

// Description of the client topology
PaCO::PacoGridTopology_t topo;
topo.dim.length(1);          // 1D
topo.dim[0] = numprocs; // number of procs in 1st dim
```

Example: the parallel client code (2)

```
// Initialization of internal library : see server code
...

// Declaration of distribution types associated with operation send
send_ctx->init(myid, numprocs); // Id of local node
send_ctx->initArg(data, topo, 0); // 1st argument distribution type declaration

// Initializing some local data

long local_vect[10];
...

// Real call
server->send(local_vect); // Distribution assumed to be
```

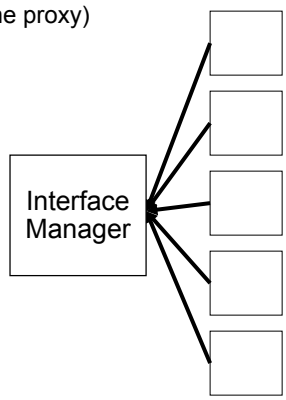

Server setting

1: Set up interface manager (contains the code of the proxy)

2: Set up of all objects belonging to the collection
   On each node:
   i: Initialization of the parallel context of each parallel operation
   ii: Initialization of redistribution library

3: Registering the nodes to the interface manager

4: The server is ready to receive requests



Example : the implementation code

```
class example_serv_impl : public example_serv
{
public:

  void send(const PaCOData_seq& seq) {
     // SPMD execution model

     // Parallel object specific interface access
     int myid      = InterfaceParallel::getMyRank();
     int nbprocs = InterfaceParallel::getTotalNode();

     // Accessing the distributed data is distribution dependant
     // This example is logical view for a block distribution
     for(int i=0; i<seq.ld.len; i++)    // local number of element
       cout << seq.data[i] << endl; // access to the ith element
     }
     MPI_Barrier(...); // Parallel operations can be used :
                              // There are not dependant on PaCO++
};
```

Example : the server code

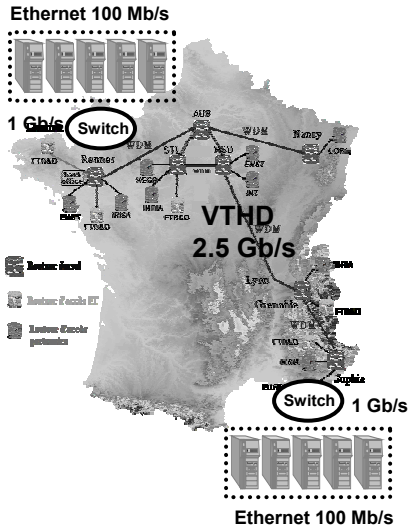
```
// Servant creation
example_serv_impl * servant = new example_serv_impl(orb,ior);

// Retrieving the parallel context of the parallel operation
PaCO_operation_server * send_ctx = servant->getContext("envoyer");

// Select the communication library
// Note: the code to manage communication library is not shown
MPI_Comm group = MPI_COMM_WORLD;
send_ctx->setLibCom("mpi",&group);

// Select the distribution library for the distributed arguments
send_ctx->setTypeArg(0, "Block");

...

// Actually declared the object as member of a collection
servant->deploy();
```

PaCO++ performance

Client and server programs
- PaCO++
- OmniORB (AT&T)
- MPI

WAN Network
- VTHD (2.5 Gb/s)
- 11 nodes to 11 nodes
- 826 Mb/s (103 MB/s)
  ⇒ at the 1 Gbit/s switch limit
- Pt-2-Pt at 75 Mb/s (9.4 MB/s)

SAN Network
- Myrinet 2000 (2 Gb/s pt-2-pt)
- 8 nodes to 8 nodes
- 12 Gbit/s (1.5 GB/s)
- Pt-2-Pt at 1.5 Gbit/s (187 MB/s)



PaCO++ Status

Portable parallel extension to CORBA
- Independent of the ORB
- Successfully test with Mico 2.3.x and OmniORB 3 et 4

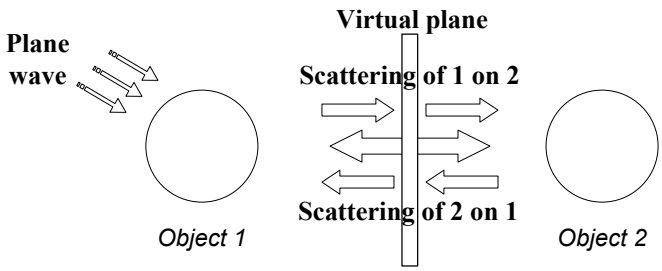
Still under development
- IDL compiler ready
- Beta C++ version of the PaCO++ layer
- 1st public version expected summer 2003

Web site
- http://www.irisa.fr/paris/PaCO++

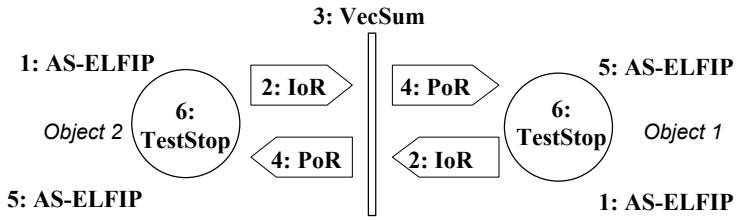
Contents

ACI GRID RMI Application from EADS

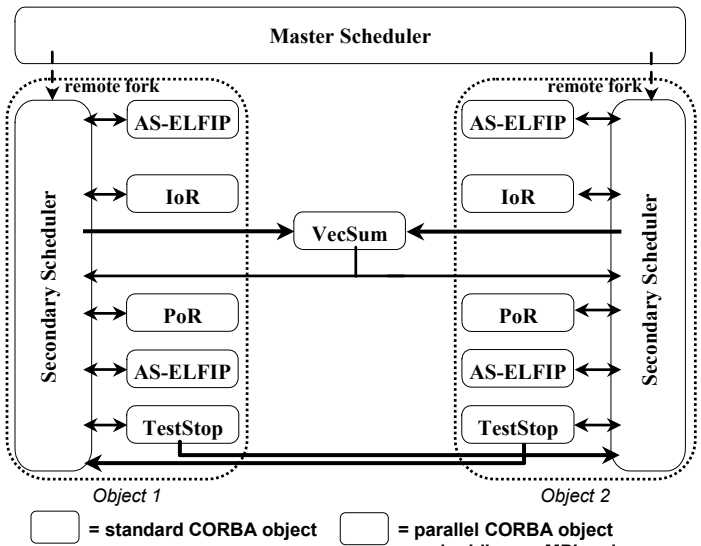
http://www.irisa.fr/Grid-RMI



Logical MPI code scheduling



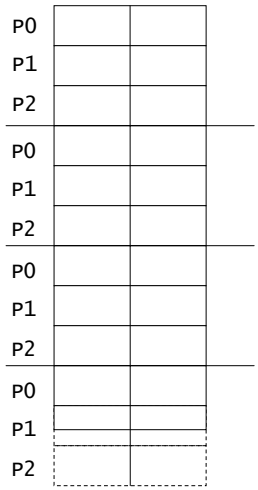
Parallel Scheduling



Data to be transferred

Data structure

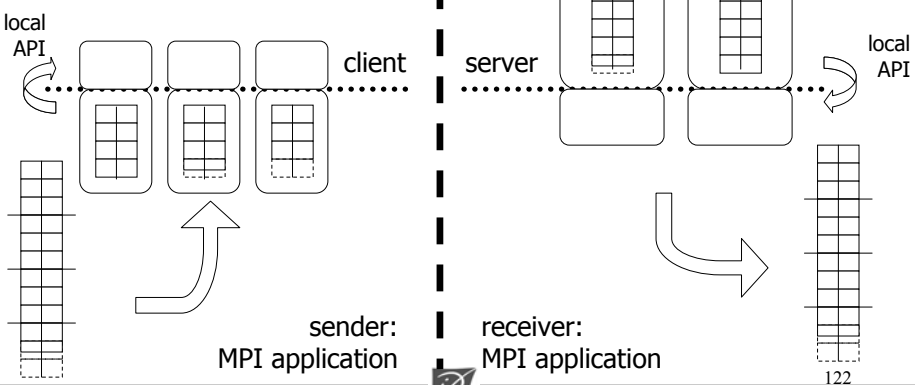
- Two-column matrix
  - Data : `double complex`
- Block-cyclic distribution
- Matrix size: up to many GigaBytes!



Data transfer

```
CODE SPMD CLIENT
...
tobj->send(id, data);
...
```

```
CODE SPMD SERVER
void OBJ::send(id, data) {
   store data locally
}
```



Remarks

Application is independent of deployment consideration

- Two secondary schedulers on the same cluster
- Two secondary schedulers on different clusters

Well-defined programming model

- Object-oriented model
- Separation of distributed and parallel issues

CORBA and MPI co-existence

- Co-existence and network transparency can be achieved
- PadicoTM : An Open Integration Framework for Communication Middleware and Runtimes
  http://www.irisa.fr/paris/Padicotm

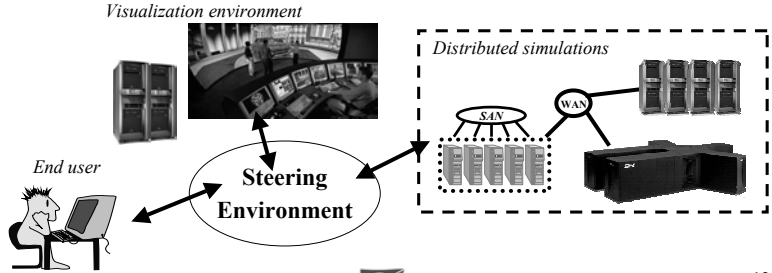
ACI EPSN

http://www.labri.fr/Recherche/PARADIS/epsn/

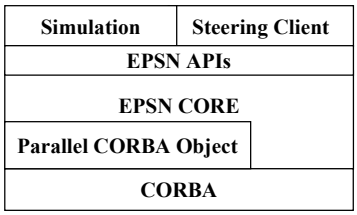
A computational steering environment for numerical distributed application

EPSN capitalizes CORBA advantages

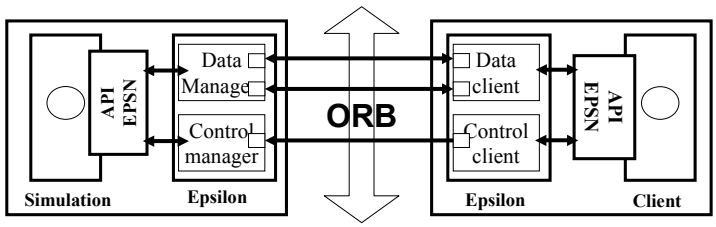
- Portability, interoperability, network transparency
- Based on PaCO++ for parallel applications



Epsilon: a prototype of EPSN



Software layer



Contents

From CORBA 2 . . .

A distributed object-oriented model

- Heterogeneity: OMG Interface Definition Language (OMG IDL)
- Portability: Standardized language mappings
- Interoperability: GIOP / IIOP
- Various invocation models: SII, DII, and AMI
- Middleware: ORB, POA, etc.
  minimum, real-time, and fault-tolerance profiles

No standard packaging and deployment facilities !!!

Explicit programming of non functional properties !!!

- lifecycle, (de)activation, naming, trading, notification, persistence, transactions, security, real-time, fault-tolerance, ...

No vision of software architecture

From OMG document ccm/2002-06-01

. . . to the CORBA Component Model

A distributed component-oriented model

- An architecture for defining components and their interactions
  - From client-side (GUI) to server-side (business) components
- A packaging technology for deploying binary multi-lingual executables
- A container framework for injecting lifecycle, (de)activation, security, transactions, persistence, and events
- Interoperability with Enterprise Java Beans (EJB)

The Industry's First Multi-Language Component Standard

- Multi-languages, multi-OSs, multi-ORBs, multi-vendors, etc.
- Versus the Java-centric EJB component model
- Versus the MS-centric .NET component model

From OMG document ccm/2002-06-01

GridCCM objectives

Re-define the parallel object concept in terms of parallel components

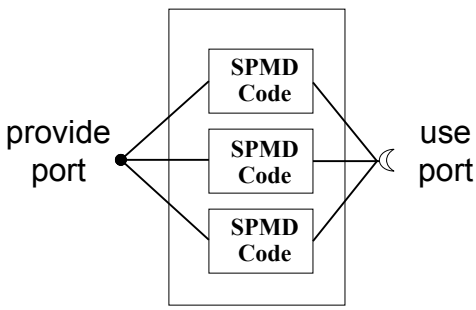
- Benefit from the PaCO++ experience
- Benefit from the CORBA component model

Component model

- Definition of non-functional properties
- CCM: packaging and deployment model!

A GridCCM Parallel component

Definition : *A parallel component is a collection of identical sequential component that executes some of the operations attached to its ports in parallel.*



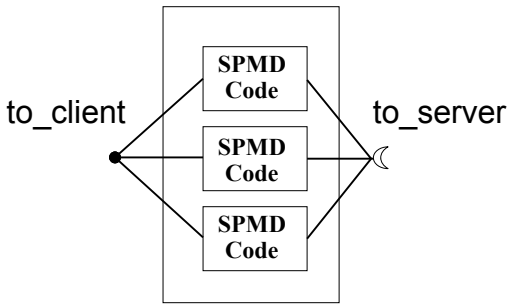
GridCCM component description

```
interface AnInterface
 {
   void example(in Matrix mat);
 };
```

```
component CoPa1
 {
   provides AnInterface to_client;
   uses Interfaces2 to_server;
 };
```

```
Component: CoPa1
Port: to_client
Name: AnInterface.example
Type: Parallel
Argument1: *, bloc
ReturnArgument: noReduction
```

XML



A parallel component of type CoPa1

Early Performance study

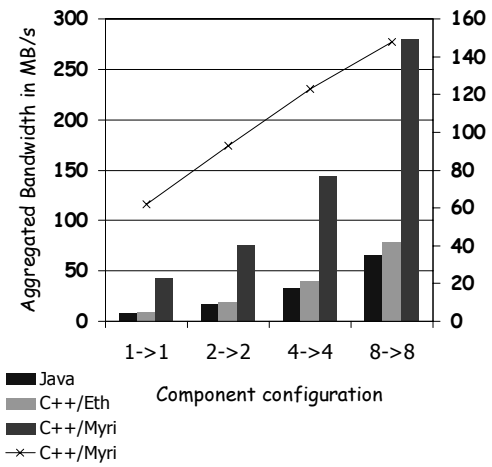
Platform

- 16 PIII 1 Ghz
- Linux 2.2
- Fast-Ethernet network
- Myrinet-2000 network

CCM-based for GridCCM

- JAVA: OpenCCM
- C++: MicoCCM

C++/Myri based on MicoCCM/PadicoTM



GridCCM status

Portable parallel extension to CORBA

- ORB-independent... but it needs a CCM implementation
- The accurate interface with redistribution library is under development

GridCCM will be based on PaCO++

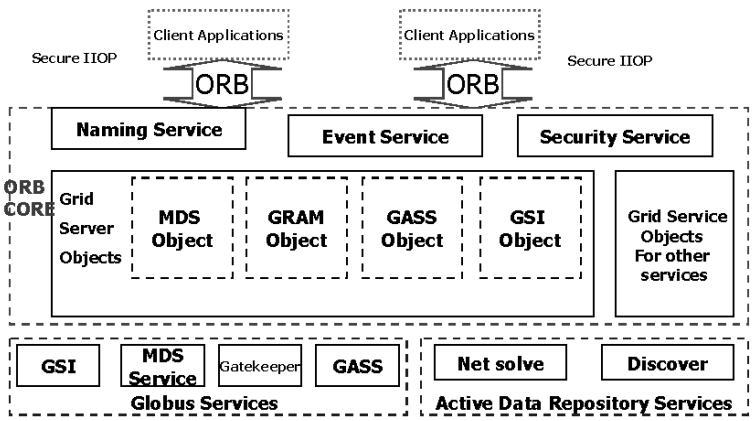
- Component model solves many PaCO++ implementation issues
- 1st public version expected by the end of year

Deployment

- Integrate CCM deployment model with Grid environment
  - CORBACoG

CORBACoG

http://www.caip.rutgers.edu/TASSL/Projects/CorbaCoG/



From CORBACoG web site

Contents

Concluding remarks on CORBA

CORBA is an industrial standard for distributed programming
- Legacy-based grid applications (multi-languages)
- Heterogeneous environment (OS independent)

CORBA 3 brings the software component technology
- Complete implementations in progress

Parallel CORBA
- Several parallel CORBA extension have shown the feasibility
  - Object-oriented and component-oriented models
- High performance can be achieved
- Integration with Grid environment under investigation

4. Conclusion

Sum Up (1)

Software component technology was developed to overcome object limitations

Software component key benefits:

- Reduce develop time by increasing code re-use
- Program by assembly rather than developing
- A unit of deployment

Still an evolving concept

- Hierarchical component model, Reconfiguration, ...

Sum Up (2)

Parallel Object/Component concepts offer a solid programming model basis for building complex applications

- Benefit of well-known technology
- Support parallelism for high performance

Most concepts are independent of the implementation technology

- Java
- CORBA
- ...

Where to manage heterogeneity ?

Language oriented view

- Hide heterogeneity by a virtual machine
- Manage parallelism and distribution inside a unified framework
- Java, ProActive, ..., and web services

Interface oriented view

- Manage heterogeneity at the interface level
- Interoperability
- Support of legacy codes
- CORBA, PaCO++, ..., and web services

Conclusion -- Perspectives

Not all models are equivalent: Component Orientedness

Level 1: Configuration 2: Assembly 3: Hierarchic 4:Reconfiguration

Specificity for GRID Components:

- Parallel (HPC), Distributed, Collective Op., Deployment, … Reconfiguration

Can programming models be independent of (Grid) Components ?

- Do not target the same objectives
- But can components … compose, … reconfigure without a clear model ?

Reconfiguration is the next big issue:

- Life cycle management, but with direct communications as much as possible
- For the sake of reliability and fault tolerance ---> GRID
  - Error, Exception handling across components
  - Checkpointing: independent, coordinated, memory channel, ...

Other pending issues:

- Peer-to-peer (even more volatile … reconfiguration is a must), Security, ...

Adaptive GRID

The need for adaptive middleware is now acknowledged,
with dynamic strategies at various points in containers, proxies, etc.

Can we afford adaptive GRID ?

with dynamic strategies at various points
(communications, checkpointing, reconfiguration, …)
for various conditions (LAN, WAN, network, P2P, ...)

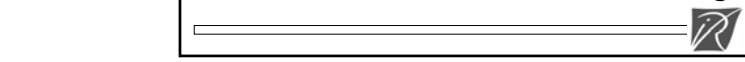
HPC vs. HPC
High Performance Components vs. High Productivity Components

Extra Material

DEMO: Applis with the IC2D monitor

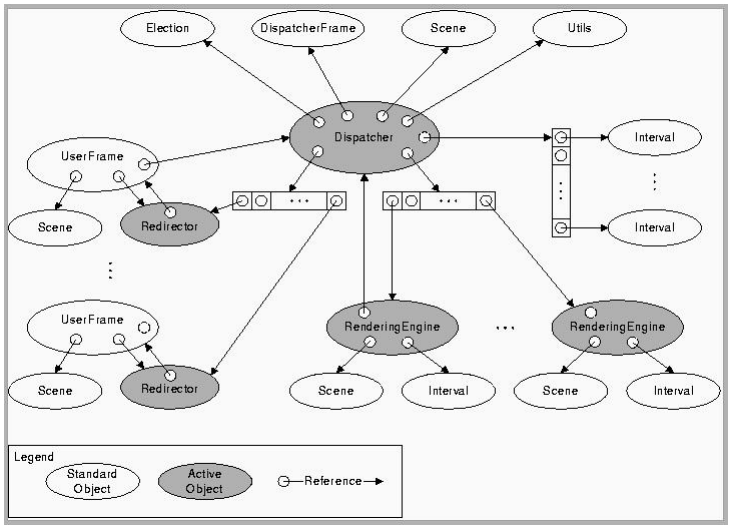
- 1. C3D : Collaborative 3D renderer in //
  a standard ProActive application
- 2. Penguin
  a mobile agent application

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

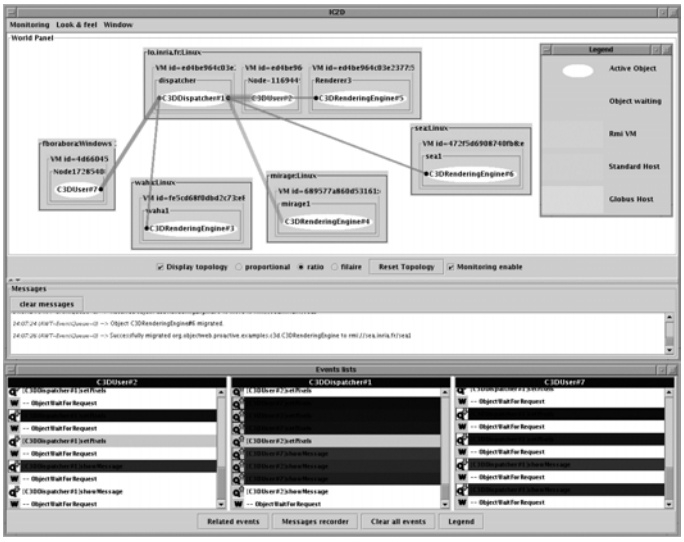


C3D: distributed-//-collaborative

Object Diagram for C3D



Monitoring: graphical and textual com.



ProActive

- A library:
  --> 100% Java, no change to the JVM,
  --> no change to user code
- Parallelism, Distribution, Synchronization, and mobility
- Typed Groups, subject to important optimizations,
- Interactive deployment and control: **IC2D**
  --> Towards Components for the Grid
- Formal properties:
  - A calculus: ASP: Asynchronous Sequential Processes
    --> Result on confluence
  - Markov Chains: Performance Evaluation
    -->Towards adaptive strategies (LAN, WAN, ...)

ProActive : API for Mobile Agents

- Mobile agents (active objects) that communicate
- Basic primitive: migrateTo
  - public static void migrateTo (String u)
    // string to specify the node (VM)
  - public static void migrateTo (Object o)
    // joinning another active object
  - public static void migrateTo (Node n)
    // ProActive node (VM)
  - public static void migrateTo (JiniNode n)
    // ProActive node (VM)

ProActive : API for Mobile Agents

- Mobile agents that communicate
- Primitive to automatically execute action upon migration
  - public static void onArrival (String r)

    // Automatically executes the routine r upon arrival

    // in a new VM after migration
  - public static void onDeparture (String r)

    // Automatically executes the routine r upon migration

    // to a new VM, guaranted safe arrival
  - public static void beforeDeparture (String r)

    // Automatically executes the routine r before trying a migration

    // to a new VM

ProActive : API for Mobile Agents
Itinerary abstraction

Itinerary : VMs to visit

- specification of an itinerary as a list of (site, method)
- automatic migration from one to another
- dynamic itinerary management (start, pause, resume, stop, modification, …)

API:

- myItinerary.add (“machine1’’, “routineX”); ...
- itinerarySetCurrent, itineraryTravel, itineraryStop, itineraryResume, …

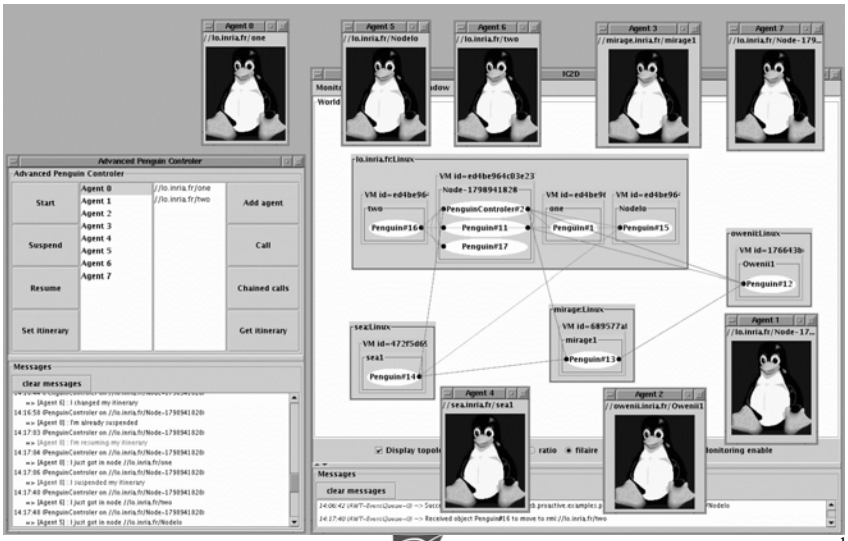
Still communicating, serving requests:

- itineraryMigrationFirst ();

  // Do all migration first, then services, Default behavior
- itineraryRequestFirst ();

  // Serving the pending requests upon arrival before migrating again

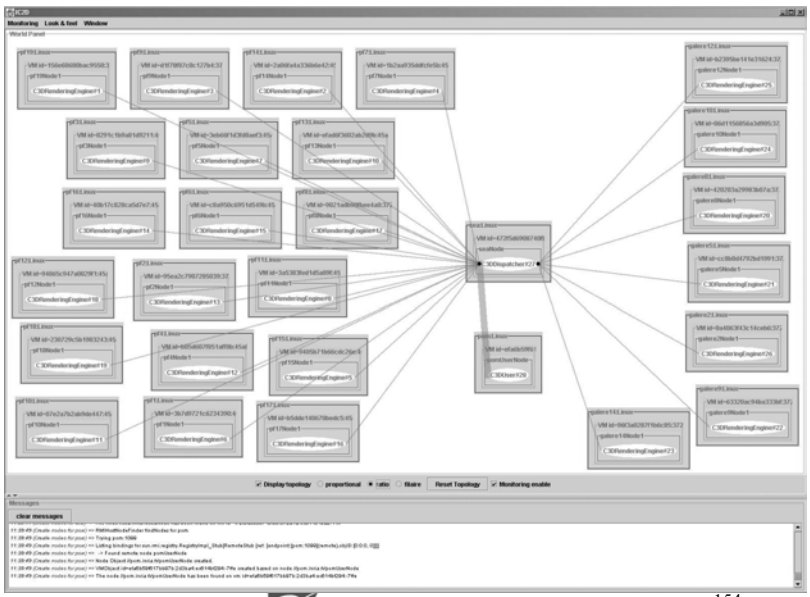
Mobile Application executing on 7 JVMs



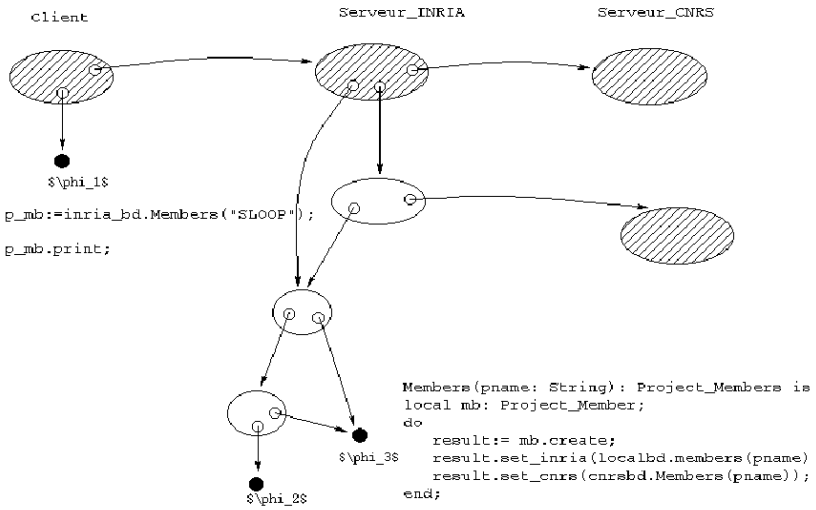
IC2D: Cluster Visualization

Visualization of 2 clusters (1Gbits links)

Featuring the current communications (proportional)



Automatic Continuations



Transparent Future transmissions (Request,Reply)

Formal Models and Properties inside

- **The ASP calculus:**
  Asynchronous Sequential Processes

- **Performance Evaluation of Mobile Agents:**
  Markov Chains

The ASP calculus: Asynchronous Sequential Processes

An Imperative and Parallel Object Calculus

Together with Ludovic Henrio, and Bernard Serpette

Objectives:

- Formally study the ProActive model
- Investigate various strategies for asynchronous calls
- Prove some equivalence between Sequential and Parallel programs
- Demonstrate the deterministic nature of sub-sets of the model

Parallel

Transition

System

$$\frac{(a_\alpha,\sigma_\alpha) \rightarrow_S (a'_\alpha,\sigma'_\alpha)}{\alpha[a_\alpha;\sigma_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha]\|Q \rightarrow_\| \alpha[a'_\alpha;\sigma'_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha]\|Q} \text{ (LOCAL)}$$

$$\frac{\begin{array}{c}\gamma \text{ nouvelle activité} \quad \iota' \notin Dom(\sigma_\alpha) \\ \sigma'_\alpha = \{\iota' \rightarrow OA(\gamma)\} :: \sigma_\alpha \quad copy_{\|}(\iota,\sigma_\alpha) = \sigma_\gamma\end{array}}{\begin{array}{l}\alpha[\mathcal{R}[Active(\iota)];\sigma_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha]\|Q \rightarrow_\| \\ \alpha[\mathcal{R}[\iota'];\sigma'_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha]\|\gamma[0;\sigma_\gamma;\iota;0;0;0]\|Q\end{array}} \text{ (NEWACT)}$$

$$\frac{\begin{array}{c}\sigma_\alpha(\iota) = OA(\beta) \quad \iota'' \notin Dom(\sigma_\beta) \\ \sigma'_\beta = Append(\sigma_\beta,copy_{\|}(\iota',\sigma_\alpha)\{\iota' \leftarrow \iota''\},\iota'') \\ f \text{ nouveau futur} \quad R'_\beta = R_\beta :: [l_j;\iota'';f^{\alpha\rightarrow\beta}] \\ \iota_f \notin Dom(\sigma_\alpha) \quad \sigma'_\alpha = \{\iota_f \rightarrow fut(f^{\gamma\rightarrow\beta})\} :: \sigma_\alpha\end{array}}{\begin{array}{l}\alpha[\mathcal{R}[\iota.l_j(\iota')];\sigma_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha]\|\beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta]\|Q \\ \rightarrow_\| \alpha[\mathcal{R}[\iota_f];\sigma'_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha]\|\beta[a_\beta;\sigma'_\beta;\iota_\beta;F_\beta;R'_\beta;f_\beta]\|Q\end{array}} \text{ (SENDREQUEST)}$$

$$\frac{R_\alpha = [l_j;\iota_r;f] :: R'_\alpha}{\alpha[0;\sigma_\alpha;\iota_\alpha;F_\alpha;R_\alpha;0]\|P \rightarrow_\| \alpha[\iota_\alpha.l_j(\iota_r);\sigma_\alpha;\iota_\alpha;F_\alpha;R'_\alpha;f]\|P} \text{ (NEWSERVICE)}$$

$$\frac{\begin{array}{c}\iota' \notin Dom(\sigma_\alpha) \quad F'_\alpha = \{f_\alpha \rightarrow \iota'\} :: F_\alpha \\ \sigma'_\alpha = append(\sigma_\alpha,copy_{\|}(\iota,\sigma_\alpha)\{\iota \leftarrow \iota'\},\iota')\end{array}}{\alpha[\iota;\sigma_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha]\|P \rightarrow_\| \alpha[0;\sigma'_\alpha;\iota_\alpha;F'_\alpha;R_\alpha;0]\|P} \text{ (ENDSERVICE)}$$

$$\frac{\begin{array}{c}\sigma_\alpha(\iota) = fut(f^{\gamma\rightarrow\beta}) \quad F_\beta(f^{\gamma\rightarrow\beta}) = \iota_f \\ \sigma'_\alpha = Append(\sigma_\alpha,copy_{\|}(\iota_f,\sigma_\beta)\{\iota_f \leftarrow \iota\},\iota)\end{array}}{\begin{array}{l}\alpha[a_\alpha;\sigma_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha]\|\beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta]\|P \rightarrow_\| \\ \alpha[a_\alpha;\sigma'_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha]\|\beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta]\|P\end{array}} \text{ (SENDREPLY)}$$

Performance Evaluation of Mobile Agent

Together with Fabrice Huet and Mistral Team

Objectives:

- Formally study the performance of Mobile Agent localization mechanism: Markov Chains
- Investigate various strategies (forwarder, server, etc.)
- Define adaptive strategies

Transitions for the Server localization

Analyse Markovienne du serveur centralisé

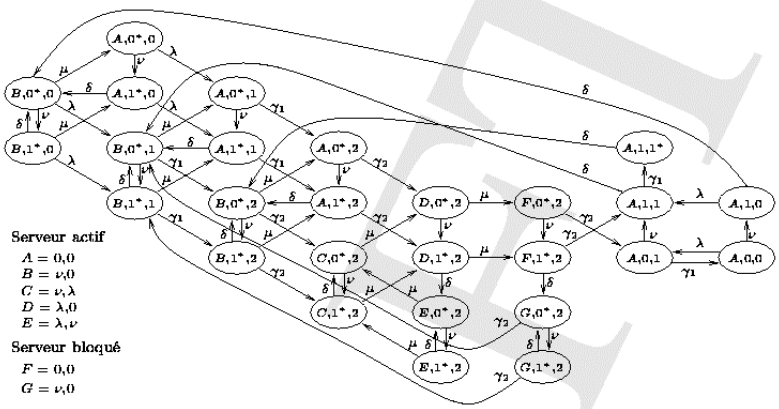


FIG. 4.6 – *État du système et taux de transitions dans le mécanisme du serveur.*

OASIS Team:
Active Objects, Semantics, Internet and Security

Sophia Antipolis (Nice)
www.inria.fr/oasis

- 6 Researchers, 8 Ph.D., 2 to 4 Eng.
- Distributed Objects, Grid, Middleware,
- Formal foundations
- An Open Source (ObjectWeb consortium) experimental platform: ProActive

PARIS Research Team
IRISA/INRIA (Rennes)
http://www.irisa.fr/paris

PARIS*: Programming parallel and distributed Systems for Large Scale Numerical Simulation

Head of the project: Thierry Priol

Members: 7 Researchers, 7 PhD, 3 Engineers

Make distributed and parallel systems easier to use by

- Designing operating systems for clusters of PC and workstations
- Designing runtimes for parallel languages (HPF, OpenMP, Java) to make the programming of clusters as simpler as possible
- Designing scalable middleware to hide distributed resources for both computational and storage Grids

*Common project with CNRS, ENS-Cachan, INRIA, INSA, University of Rennes 1