Skeleton programming environments Using ProActive Calcium

Patrizio Dazzi
ISTI - CNR
Pisa Research Campus
mail: patrizio.dazzi@isti.cnr.it

Master Degree (Laurea Magistrale) in Computer Science and Networking Academic Year 2009-2010

Outline

- **ProActive framework Overview**
  - *Active Object concept*
  - *Future concept*
- **Calcium**
- **Using Calcium Skeletons**
- **Sample Applications**
- **Demo on my machine**
- **Installing ProActive**

ProActive Parallel Suite

- **Framework conceived by Denis Caromel**
  - *INRIA, France*
- **Based on Active Objects and Futures**
  - *Wait-By-Necessity semantics*
- **Java based framework**
- **Home page at: http://proactive.inria.fr/**

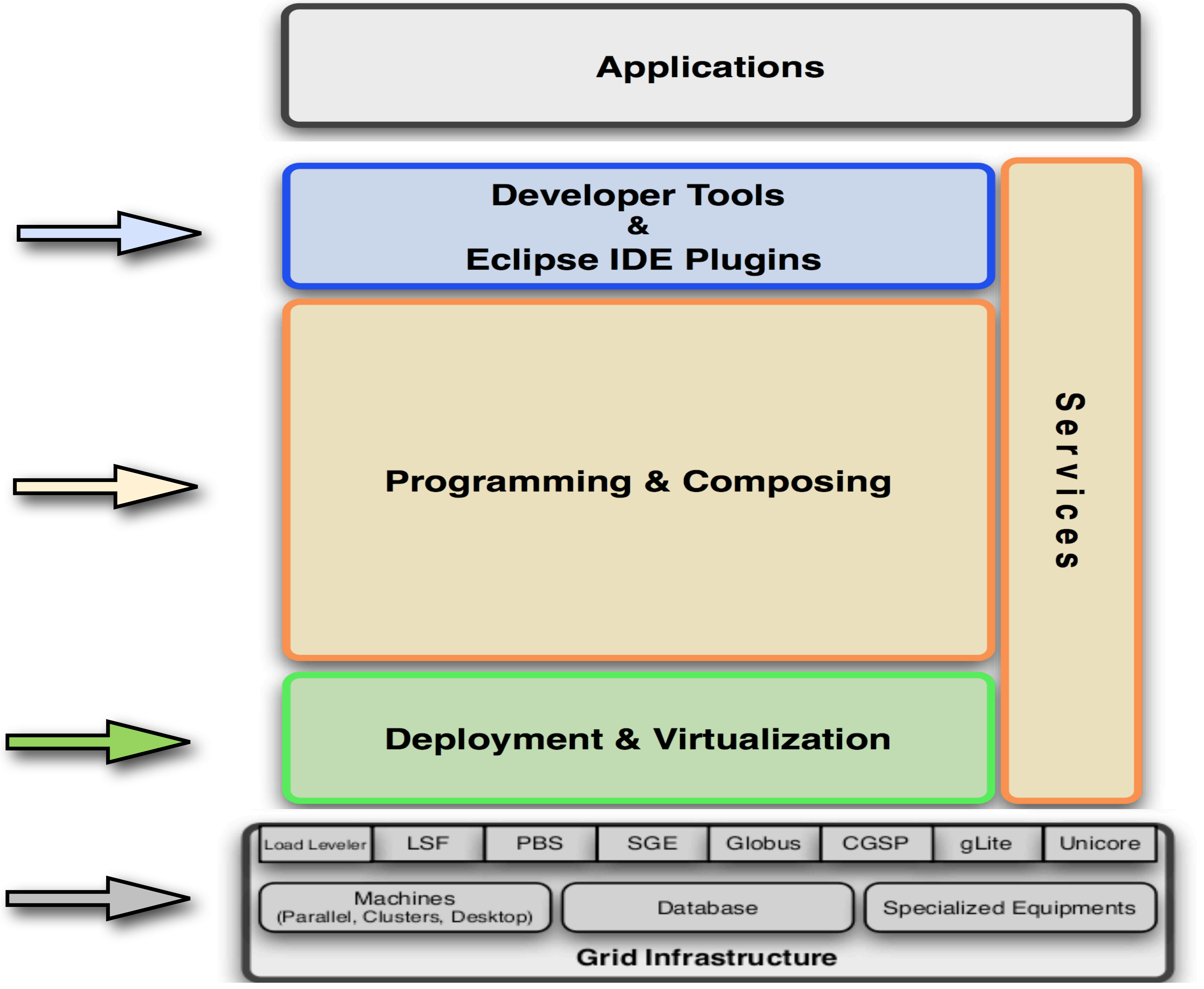
ProActive Layered Structure

Tools for easing the development, e.g. IC2D

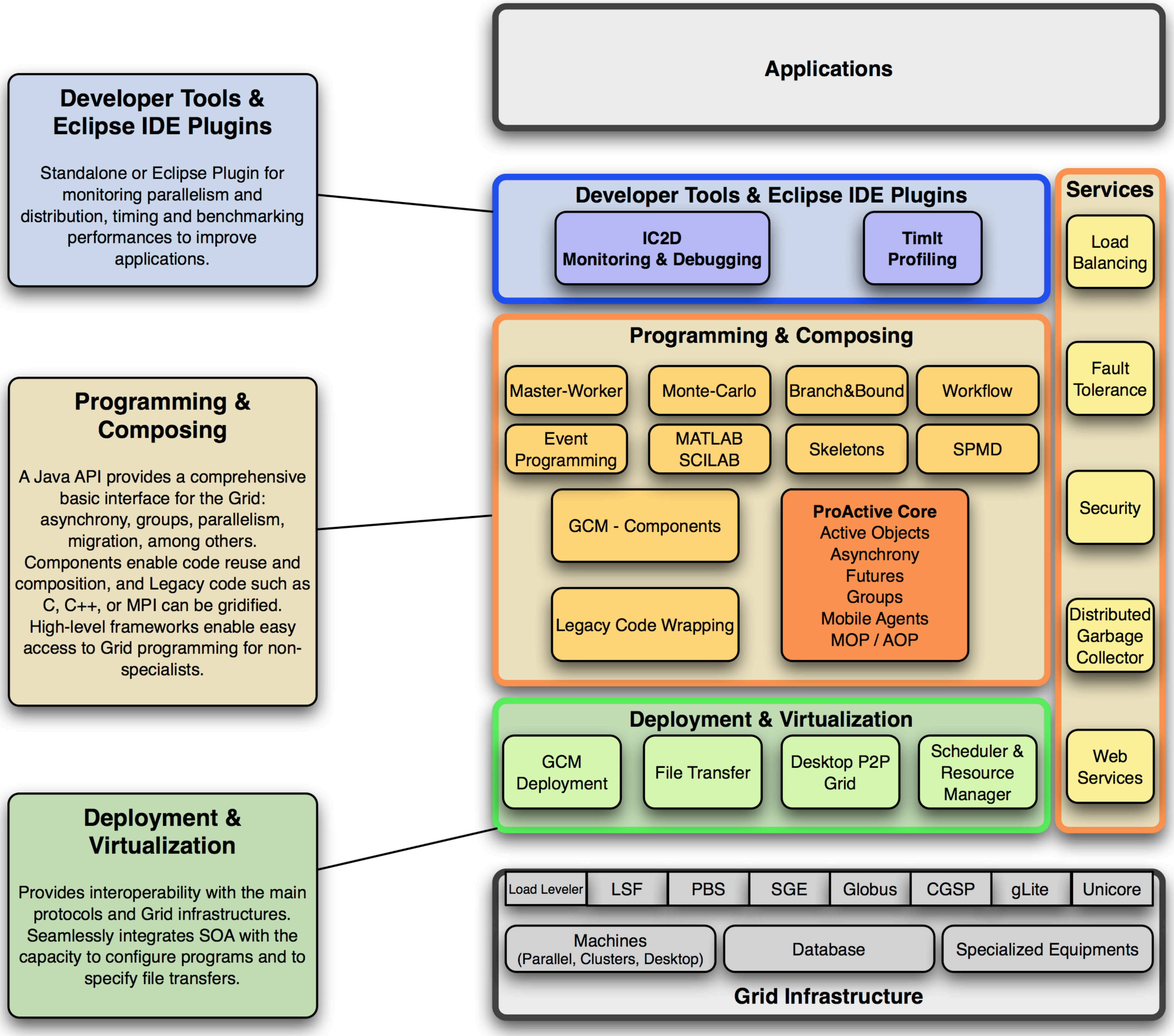
Active Objects, Components, Skeletons, ...

Uniformed Deployment Infrastructure

Deployment achieved also via other tools



Detailed Structure



Developer Tools & Eclipse IDE Plugins
Standalone or Eclipse Plugin for monitoring parallelism and distribution, timing and benchmarking performances to improve applications.
Programming & Composing
A Java API provides a comprehensive basic interface for the Grid: asynchrony, groups, parallelism, migration, among others. Components enable code reuse and composition, and Legacy code such as C, C++, or MPI can be gridified. High-level frameworks enable easy access to Grid programming for non-specialists.
Deployment & Virtualization
Provides interoperability with the main protocols and Grid infrastructures. Seamlessly integrates SOA with the capacity to configure programs and to specify file transfers.
Applications
Developer Tools & Eclipse IDE Plugins
IC2D Monitoring & Debugging
TimIt Profiling
Programming & Composing
Master-Worker
Monte-Carlo
Branch&Bound
Workflow
Event Programming
MATLAB SCILAB
Skeletons
SPMD
GCM - Components
Legacy Code Wrapping
ProActive Core
Active Objects
Asynchrony
Futures
Groups
Mobile Agents
MOP / AOP
Deployment & Virtualization
GCM Deployment
File Transfer
Desktop P2P Grid
Scheduler & Resource Manager
Services
Load Balancing
Fault Tolerance
Security
Distributed Garbage Collector
Web Services
Load Leveler
LSF
PBS
SGE
Globus
CGSP
gLite
Unicore
Machines (Parallel, Clusters, Desktop)
Database
Specialized Equipments
Grid Infrastructure

Detailed Structure

Developer Tools & Eclipse IDE Plugins

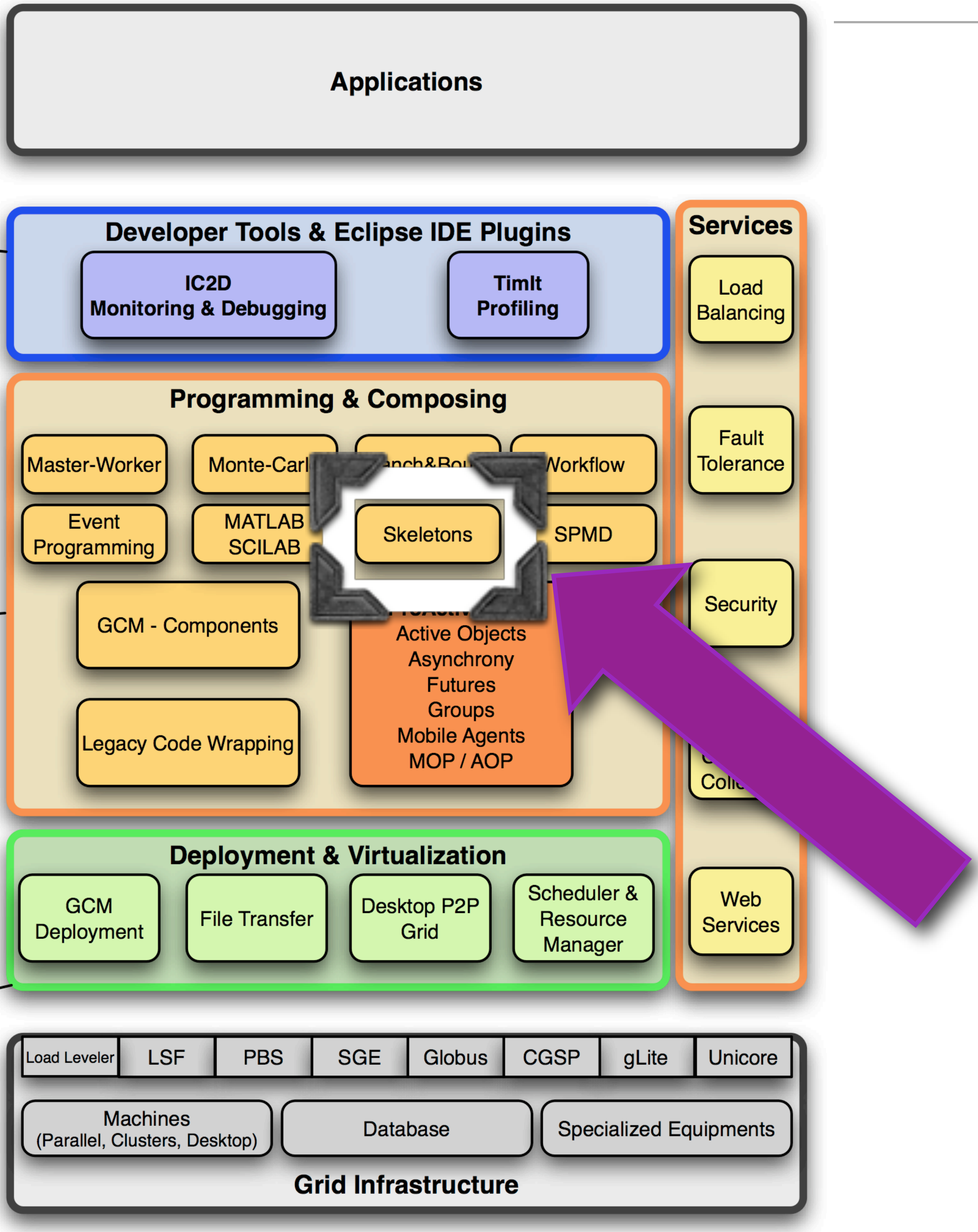
Standalone or Eclipse Plugin for monitoring parallelism and distribution, timing and benchmarking performances to improve applications.

Programming & Composing

A Java API provides a comprehensive basic interface for the Grid: asynchrony, groups, parallelism, migration, among others. Components enable code reuse and composition, and Legacy code such as C, C++, or MPI can be gridified. High-level frameworks enable easy access to Grid programming for non-specialists.

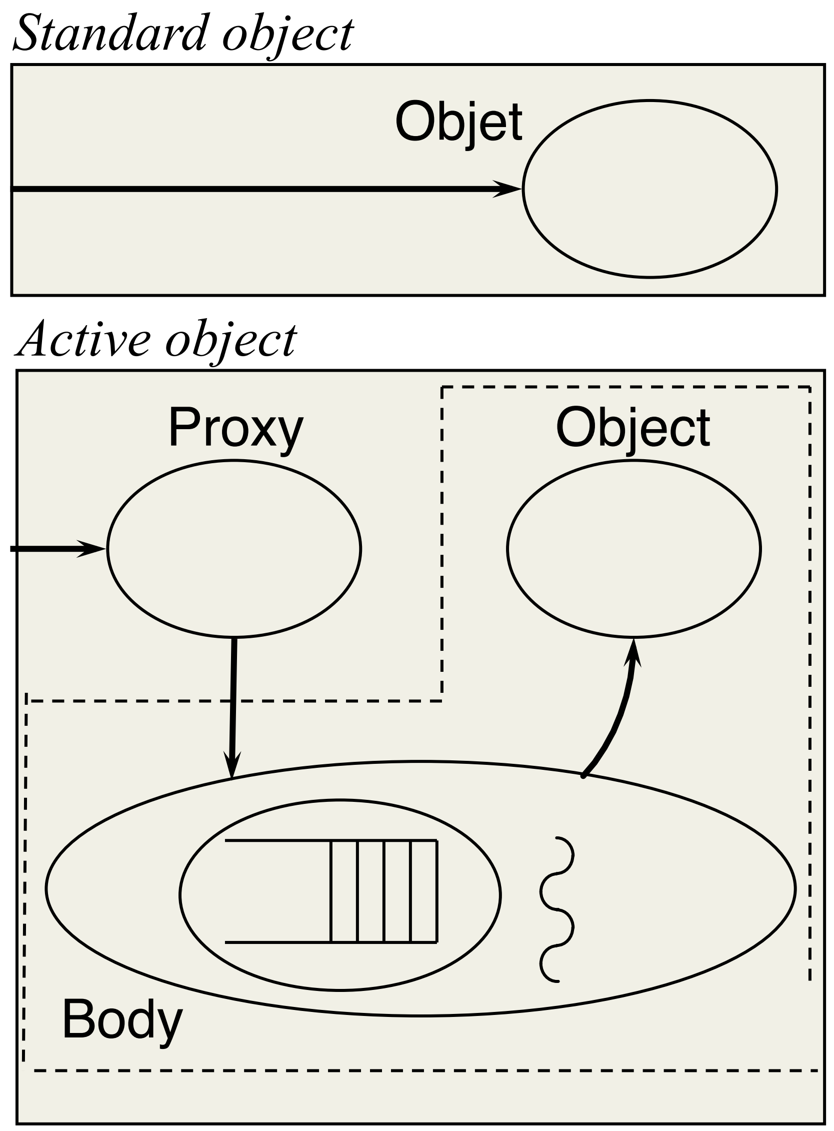
Deployment & Virtualization

Provides interoperability with the main protocols and Grid infrastructures. Seamlessly integrates SOA with the capacity to configure programs and to specify file transfers.



Active Objects

- **An active object is composed of several objects**
    - *The object being activated*
    - *A set of standard Java objects*
    - *A single thread*
    - *The queue of pending requests*



Futures

- **Result Placeholder**

- **Needed to achieve Asynchrony with standard Java**
  - *without callbacks!!!*

Suppose each call requires 3 seconds and you have 2 processors or 2 distributed machines

- **E.g.**
  Future<Integer> fi1 = methodCall(A);
  Future<Integer> fi2 = methodCall(B);

Calcium (1)

- **Conceived and Developed by Mario Leyton**
- **Uses ProActive**
  - *Active Objects as Skeleton Stages*
  - *for Application Deployment*
- **Includes Task and Data Parallel Skeletons**

Calcium (2)

- **Available Skeletons:**
  - *Task Parallel:*
    - farm
    - pipeline
    - divide and conquer
  - *Data Parallel*
    - map
    - fork

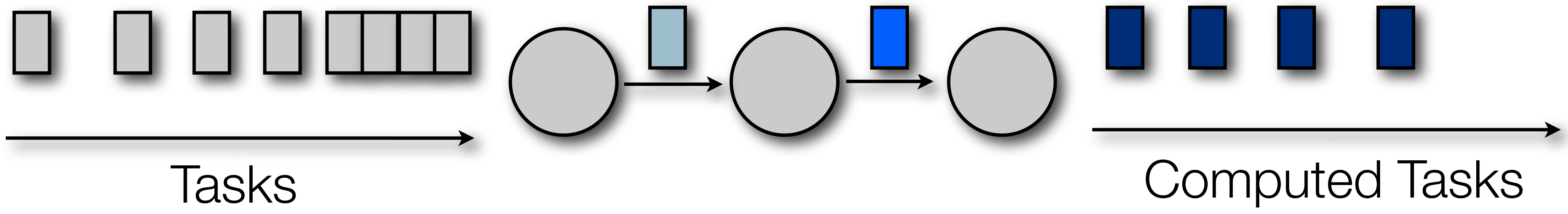
Today we will see

“Special” Classes

- **Environment**
  - *different execution environments available.*
  - *3 supported environments:*
    - MultiThreadedEnvironment
    - ProActiveEnvironment
- **Input and Output Files from the framework**
  - *Stream*
  - *CalFuture*

Pipeline in Calcium (1)

- Applications organized in Stages
- Each Stage performs a specific computation



Pipeline in Calcium (2)

- **Class Pipeline**

```
public class Pipe<P extends java.io.Serializable,
                  R extends java.io.Serializable> implements Skeleton<P, R>
```

- **Several Constructors: 2, 3 and 4 stages**

Sample Pipeline Usage (1)

First Stage

```
package Pipeline;
import org.objectweb.proactive.extensions.calcium.muscle.Execute;
import org.objectweb.proactive.extensions.calcium.system.SkeletonSystem;

public class Incr implements Execute<Integer, Integer> {
   public Integer execute(Integer arg0, SkeletonSystem arg1) throws Exception {

        return new Integer(arg0+1);
   }
}
```

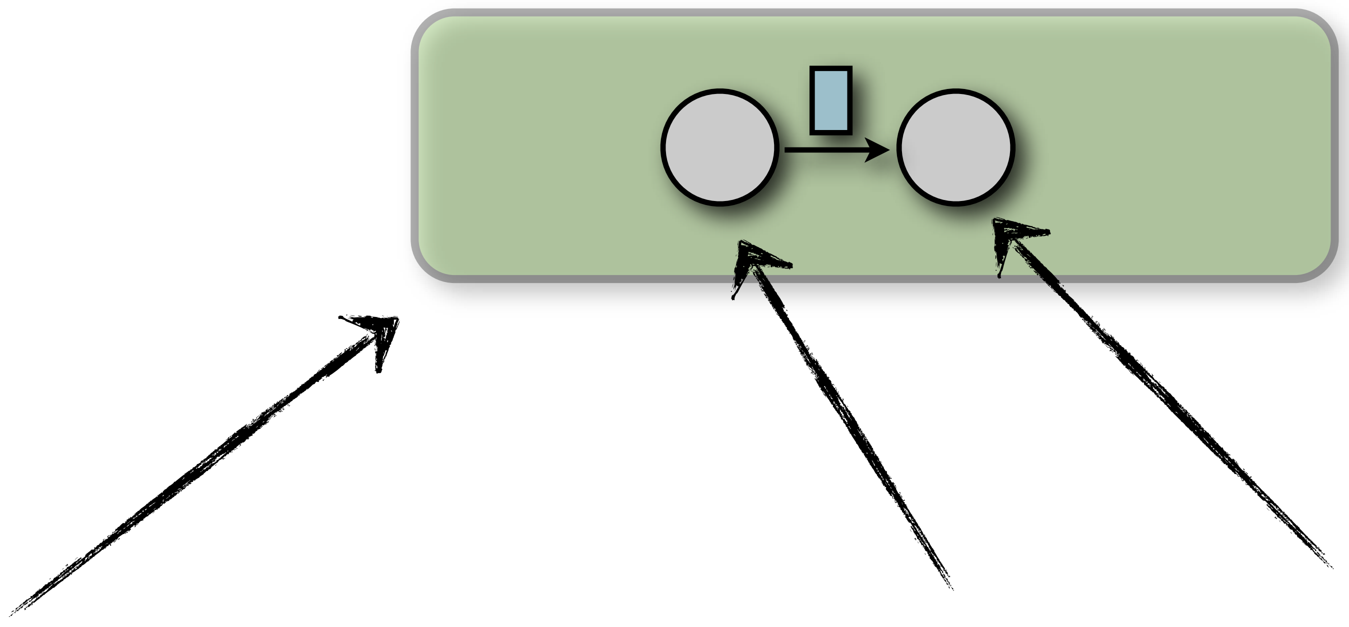
Second Stage

```
package Pipeline;
import org.objectweb.proactive.extensions.calcium.muscle.Execute;
import org.objectweb.proactive.extensions.calcium.system.SkeletonSystem;

public class StringMaker implements Execute<Integer, String> {
   public String execute(Integer arg0, SkeletonSystem arg1) throws Exception {
        return "This is a string with "+arg0+" in the middle";
   }
}
```

Sample Pipeline Usage (2)

```
package Pipeline;
<required imports>

public class CalciumFirst {

   public static void main(String[] args) throws Exception {

        Skeleton<Integer, String> root = new Pipe<Integer, String>(new Incr(), new StringMaker());
        MultiThreadedEnvironment enviroment =
               (MultiThreadedEnvironment)MultiThreadedEnvironment.factory(2);
        Calcium calcium = new Calcium(enviroment);
        Stream<Integer, String> stream = calcium.newStream(root);

        Vector<CalFuture<String>> futures = new Vector<CalFuture<String>>();
        futures.add(stream.input(new Integer(2)));
        futures.add(stream.input(new Integer(3)));

        calcium.boot(); //begin the evaluation

        for(CalFuture<String> future:futures){
            String res = future.get();
            System.out.println(res);
        }

        calcium.shutdown(); //release the resources
        System.exit(0);
   }
}
```

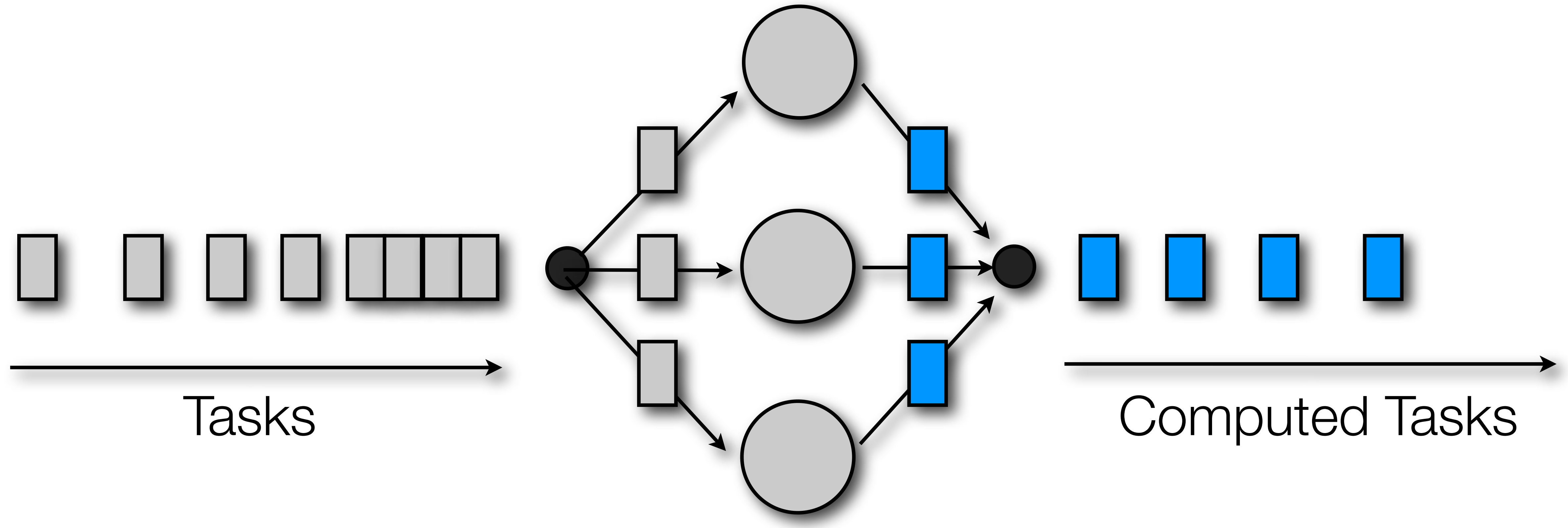
Sample Pipeline Usage (3)



```
Skeleton<Integer, String> root = new Pipe<Integer, String>(new Incr(), new StringMaker());
```

Farm in Calcium (1)

- Elaborations performed by multiple “workers”
- Each worker computes the same application code



Farm in Calcium (2)

- **class Farm**

```
public class Farm<P extends java.io.Serializable,
                 R extends java.io.Serializable> implements Skeleton<P, R>
```

- **Only one Constructor**

```
public Farm(Execute<P, R> muscle)
```

Sample Farm usage (1)

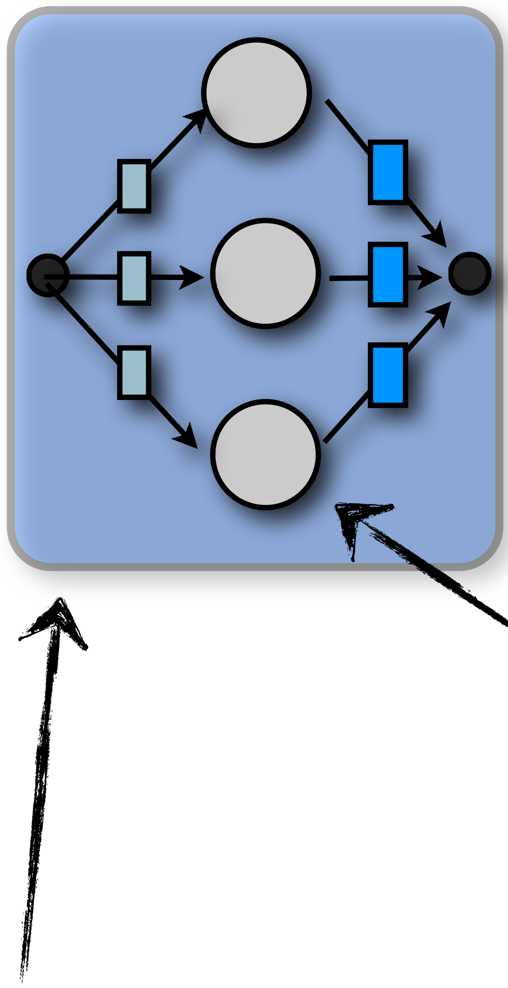
Worker

```
package Farm;

import org.objectweb.proactive.extensions.calcium.muscle.Execute;
import org.objectweb.proactive.extensions.calcium.system.SkeletonSystem;

public class Worker implements Execute<Integer, Integer> {

    public Integer execute(Integer arg0, SkeletonSystem arg1) throws Exception {

        return new Integer(((int)Math.pow(2, arg0)));
    }

}
```

Sample Farm usage (2)

```
package Pipeline;
<required imports>

public class CalciumFirst {

  public static void main(String[] args) throws Exception {

        Skeleton<Integer, Integer> root = new Farm<Integer, Integer>(new Worker());
        MultiThreadedEnvironment enviroment =
               (MultiThreadedEnvironment)MultiThreadedEnvironment.factory(2);
        Calcium calcium = new Calcium(enviroment);
        Stream<Integer, Integer> stream = calcium.newStream(root);

        Vector<CalFuture<Integer>> futures = new Vector<CalFuture<Integer>>();
        futures.add(stream.input(new Integer(2)));
        futures.add(stream.input(new Integer(3)));

        calcium.boot(); //begin the evaluation

        for(CalFuture<Integer> future:futures){
            String res = future.get();
            System.out.println(res);
        }

        calcium.shutdown(); //release the resources
        System.exit(0);
  }
}
```

Sample Farm Usage (2)



```
Skeleton<Integer, Integer> root = new Farm<Integer, Integer>(new Worker());
```

Compiling a Calcium Application

- Sun Java Compiler, version >= 5
- Add to the Java CLASSPATH the jar file in dist/lib
- Use javac <source code>

Running a Calcium Application

- **On a single machine**
  - *Use as MultiThreadEnvironment as Calcium Environment*
  - *run java <compiled class>*
- **On a set of distributed machines (we will see an more detailed how-to in the next lesson)**
  - *Define a deployment descriptor*
  - *Configure the ssh accesses*

Demo

- I'll show now how these things actually work on my machine ....

Questions ?