Interactive and Descriptor-based Deployment of Object-Oriented Grid Applications

Françoise Baude, Denis Caromel, Fabrice Huet, Lionel Mestre, Julien Vayssière

INRIA Sophia Antipolis, CNRS - I3S - Univ. Nice Sophia Antipolis,
BP 93, 06902 Sophia Antipolis Cedex - France
First.Last@inria.fr

Abstract

Increasing complexity of distributed applications and commodity of resources through grids are making the tasks of deploying those applications harder. There is a clear need for standard tools allowing versatile deployment and analysis of distributed applications.

We present here a solution for the deployment and monitoring of applications written using ProActive, an experimental Java-based library for concurrent, distributed and mobile computing. We describe the use of XML-based descriptor for the deployment part of a distributed application and the use of IC2D (Interactive Control and Debugging of Distribution), for the monitoring and steering of the running application.

Those ideas, concepts, and experiments are a contribution towards the construction of integrated environments for component-based grid programming.

1 Introduction

1.1 Presentation of the problem

If libraries for parallel and distributed application development exist (RMI in Java, jmpi [6] for MPI programming, etc.) there is no standard yet for the deployment of such applications. The deployment is commonly done manually through the use of remote shells for launching the various virtual machines or daemons on remote computers, clusters or grids. The commoditization of resources through grids and the increasing complexity of applications are making the task of deploying central and harder to perform.

Questions such as "are the distributed entities correctly created?", "do the communications among such entities execute correctly?", "where a given mobile entity is actually located?", etc. are usually left unanswered. Moreover, there is usually no mean to dynamically modify the execution environment once the application is started.

Clearly said, the management of the mapping of processes (such as JVMs, PVM or MPI daemons) onto hosts, the deployment of activities onto those processes have generally to be explicitly taken into account, in a static way, sometimes inside the application, sometimes through scripts. The application cannot be seamlessly deployed on different runtime environments.

To solve those critical problems, classical and somehow ideal solutions follow 3 steps:

1. abstract away from the hardware and software runtime configuration by introducing and manipulating in the program virtual processes where the activities of the application will be subsequently deployed,
2. provide external information regarding all real processes that must be launched and the way to do it (it can be through remote shells or job submission to clusters or grids), and define the mapping of virtual processes onto real processes,
3. provide a mean to visualize, complete or modify the deployment once the application has started.

1.2 Contribution

Taking into consideration this 3-steps approach, this paper presents an *integrated* solution targeted to distributed object-oriented applications written using object-oriented libraries and more specifically, using the *ProActive* library [5] (`www.inria.fr/oasis/ProActive`), a Java based solution for seamless parallel and distributed programming. The solution we have experimented is applicable to other object-oriented programming environments.

We solve the two first steps by introducing XML-based descriptors able to describe activities and their mapping onto processes. We solve the third step by having a monitoring application: *IC2D* . It is a graphical environment for

monitoring and steering distributed and *ProActive* applications.

The XML-based descriptor allows to describe: (1) virtual nodes, which are logical entities representing containers of activities, (2) Java virtual machines and the way to launch or find them, (3) the mapping of those virtual nodes onto the JVMs. In the application, activities are only mapped on virtual nodes allowing complete separation with the actual processes. As a consequence, a price to pay is the engineering and the inclusion in the source code of structured virtual nodes; the solution is not targeted at legacy code, but rather geared towards new applications and object-oriented design.

Once a *ProActive* application is running, *IC2D* enables the user to graphically visualize fundamental distributed aspects such as topology and communications, and allows the user to control and modify the execution (e.g. the mapping of activities onto real processes, i.e. JVMs, either upon creation or migration) adding dynamicity in the configuration and deployment.

2 Background on Distributed, Mobile, and Active Objects with *ProActive*

ProActive is a 100% Java library for concurrent, distributed and mobile computing implemented on top of RMI [16] as the transport layer. Besides RMI services, *ProActive* features transparent remote active objects, asynchronous two-way communications with transparent futures, high-level synchronisation mechanisms, and migration of active objects with pending calls. As *ProActive* is built on top of standard Java APIs[1], it does not require any modification to the standard Java execution environment, nor does it make use of a special compiler, preprocessor or modified virtual machine. The model of distribution and activity that we present in this section is part of a larger effort to improve simplicity and reuse in the programming of distributed and concurrent object systems [3, 4].

2.1 Base model

A distributed or concurrent application built using *ProActive* is composed of a number of medium-grained entities called *active objects*. Each active object has one distinguished element, the *root*, which is the only entry point to the active object. Each active object has its own thread of control and is granted the ability to decide in which order to serve the incoming method calls that are automatically stored in a queue of pending requests. Method calls (see figure 1) sent to active objects are always asynchronous with transparent *future objects* and synchronization is handled by a mechanism known as *wait-by-necessity* [3]. There is a short rendez-vous at the beginning of each asynchronous remote call, which blocks the caller until the call has reached the context of the callee (on Figure 1, step 1 blocks until step 2 has completed). The *ProActive* library provides a way to migrate any active object from any JVM to any other one through the `migrateTo(...)` primitive which can either be called from the object itself or from another active object through a public method call.

[1] Java RMI [16], the Reflection API [15],...

2.2 Mapping active objects to JVMs: Nodes

Another extra service provided by *ProActive* (compared to RMI for instance) is the capability to *remotely create remotely accessible objects*. For that reason, there is a need to identify JVMs, and to add them few services. *Nodes* provide those extra capabilities : a *Node* is an object defined in *ProActive* whose aim is to gather several active objects in a logical entity. It provides an abstraction for the physical location of a set of active objects. At any time, a JVM hosts one or several nodes.The traditional way to name and handle nodes in a simple manner is to associate them with a symbolic name, that is a URL giving their location, for instance: `rmi://lo.inria.fr/Node1`.

As an active object is actually created on a *Node* we have instruction like `a = (A) ProActive.newActive("A", params, "rmi://lo.inria.fr/Node1")`. Note that an active object can also be bound dynamically to a node as the result of a migration.

As we identify earlier, the first step toward seamless deployment is to abstract away from hardware and software details. The next section will introduce the concept of *Virtual Node*: a solution not to reference *node* by their URL in the application.

3 Descriptor-based Mapping and Deployment

In order to solve the two first steps of the seamless deployment problem, we introduced XML-based descriptors as a way to define the logical entities the application needs to run, the computing resources available and the mapping of those logical entities onto those resources.

3.1 Principles

A first principle is to *fully* eliminate from the source code the following elements:

- machine names,
- creation protocols,
- registry and lookup protocols,

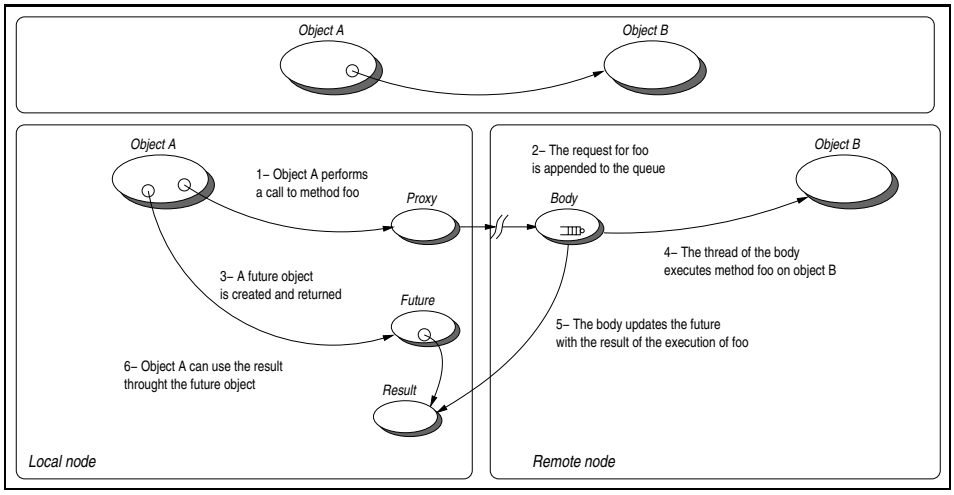

Figure 1. Execution of a remote method call

the goal being to deploy any application anywhere without changing the source code. For instance, we must be able to use various protocols, `rsh, ssh, Globus, LSF,` etc., for the creation of the JVMs needed by the application. In the same manner, the discovery of existing resources or the registration of the ones created by the application can be done with various protocols such as `RMIregistry, Jini, Globus, LDAP, UDDI,` etc. Therefore, we see that the creation, registration and discovery of resources has to be done externally to the application.

A second key principle is the capability to abstractly describe an application, or part of it, in term of its conceptual activities. The description should indicate the various parallel or distributed entities in the program or in the component. As we are in a (object-oriented) message passing model, to some extend, this description indicates the maximum number of address spaces. For instance, an application that is designed to use three interactive visualization nodes, a node to capture input from a physic experiment, and a simulation engine designed to run on a cluster of machines should somewhere clearly advertise this information.

Now, one should note that the abstract description of an application and the way to deploy it are not independent piece of information. In the example just above, if there is a simulation engine, it might register in a specific registry protocol, and if so, the other entities of the computation might have to use that lookup protocol to bind to the engine. Moreover, one part of the program can just lookup for the engine (assuming it is started independently), or explicitly create the engine itself.

To summarize, in order to abstract away the underlying execution plate-form, and to allow a *source-independent deployment*, a framework has to provide the following elements:

- an abstract description of the distributed entities of a parallel program or component,
- an external mapping of those entities to real *machines*, using actual *creation, registry*, and *lookup* protocols.

```
Component Name: C3D-Dispatcher-Renderer
Dependencies:
   Provides: class C3DDispatcher
   Needs: <none>
   Use: class C3DUser
VirtualNodes:
   Dispatcher RegisteredIn RMIregistry, Globus
   Renderer1
   Renderer2
   Renderer3
   Renderer4
Mapping:
   Dispatcher --> DispatcherJVM
   Renderer1 --> JVM1
   Renderer2 --> JVM1
   Renderer3 --> JVM2
   Renderer4 --> JVM2
JVMs:
   DispatcherJVM = Current // (the current JVM,
running the main),
   JVM1 = //lo.inria.fr/ Protocol rsh
   JVM2 = //ClusterSophia.inria.fr/ Protocol
          LSF <1> VIA galere1JVM
   galere1JVM = //galere1.inria.fr Protocol SSH
ARCHITECTURE:
   ...// How to launch a JVM on a given machine
      // using for instance the same kind of
      // information as shown on figure 6
```

Figure 2. Example of ProActive Descriptor file

3.2 Concepts and Definitions

Besides the principles above, we want to eliminate as much as possible the use of scripting languages, that can sometimes become even more complex than application code. Instead, we are seeking a solution with XML descriptors, XML editor tools, interactive ad-hoc environments to produce, compose, and activate descriptors.

To reach that goal, the system relies on a specific notion, **Virtual Nodes** (VNs):

- a VN is identified as a name (a simple string),
- a VN is used in a program source,
- a VN is defined and configured in a descriptor file (XML),
- a VN, after activation, is mapped to one or to a set of *actual ProActive Nodes*.

Of course, distributed entities (active objects), are created on Nodes, not on Virtual Nodes (see example below, Section 3.3). There is a strong need for both Nodes and *Virtual Nodes*. *Virtual Nodes* are a much richer abstraction, as they provide mechanisms such as *set* or *cyclic mapping*. Another key aspect is the capability to describe and trigger the mapping of a single VN that generates the allocation of several JVMs. This is critical if we want to get at once machines from a cluster of PCs managed through Globus or LSF, or even more critical in a Grid application, the co-allocation of machines from several clusters across several continents.

Moreover, a Virtual Node is a concept of a distributed program or component, while a Node is actually a deployment concept: it is an object that lives in a JVM, hosting active objects. There is of course a correspondence between Virtual Nodes and Nodes: the function created by the deployment, the mapping. This mapping can be specified in an XML descriptor. By definition, the following operations can be configured in such a descriptor:

- the mapping of VNs to Nodes and to JVMs,
- the way to create or to acquire JVMs,
- the way to register or to lookup VNs.

3.3 Examples

Figure 2 gives a simple example of a ProActive Descriptor file (rendered to eliminate XML tags for the sake of concision and readability). This descriptor declares 5 VNs (Dispatcher, and Renderer 1 to 4) in the `VirtualNodes` section. Further down, in the `Mapping` section, the 5 VNs are mapped on only three different JVMs (DispatcherJVM, JVM1, JVM2), resulting in the co-allocation in the same JVM of Renderer1 and Renderer2, and of Renderer3 and Renderer4. A dedicated section, `JVMs`, allows to explicitly state the creation of JVM, using various protocols. Note that `JVM2` cannot be created directly, due to security limitation, and the descriptor specifies that another JVM (`galere1JVM`)is used just for the sake of accessing the cluster. Finally, as the Dispatcher VN needs to be lookup from independently started clients, that VN name is registered in two systems (RMIregistry and Globus).

Within the source code, the programmer can manage the creation of active objects without relying on machine names and protocols. For instance, the piece of code given at Figure 3 will allow to create an active object onto the Virtual Node `Renderer1`. The JVMs associated in a descriptor file with a given VN are started (or aquired) only upon activation of a VN mapping (`pad.activateMapping("Renderer1")` in the code above).

```
ProActiveDescriptor pad = ProActive.
  getProActiveDescriptor("file:.ProActiveDescriptor");
VirtualNode vn = pad.activateMapping("Renderer1");
  // "Renderer1" is the virtual node name described
  // in the XML-descriptor. It triggers the
  // JVM on which "Renderer1" is mapped to.
Node node = vn.getNode();
C3DRenderingEngine re = (C3DRenderingEngine)
  fr.inria.proactive.ProActive.newActive(
  "fr.inria.proactive.examples.c3d.C3DRenderingEngine",
  param, node);
log("New engine " +  "Renderer1" + "created at Node " +
  node.name() + // also Renderer1 in that case
  " on JVM " + node.JVM() + " on Host " + node.host());
...
```

Figure 3. Example of ProActive source code for descriptor-based mapping

4 Graphical Visualisation and Control within *IC2D*

IC2D is the graphical environment for the monitoring of *ProActive* applications. It provides information about the support of the *ProActive* computation, and about the progress of the computation.

The communication flow that takes place between the various components is one of the most significant features to track in distributed applications. This is why *IC2D* tracks, on demand, all events that relate to remote method calls between active objects.

4.1 Hosts, VMs, Nodes, and Active Objects

In Figure 4, one can visualize the layout of hosts, JVMs, and ProActive nodes. In the Figure, each rectangle inside a grey box is a JVM, which means that there are exactly two VMs running on `pf19` and `galere8` and one on `pf9`. In

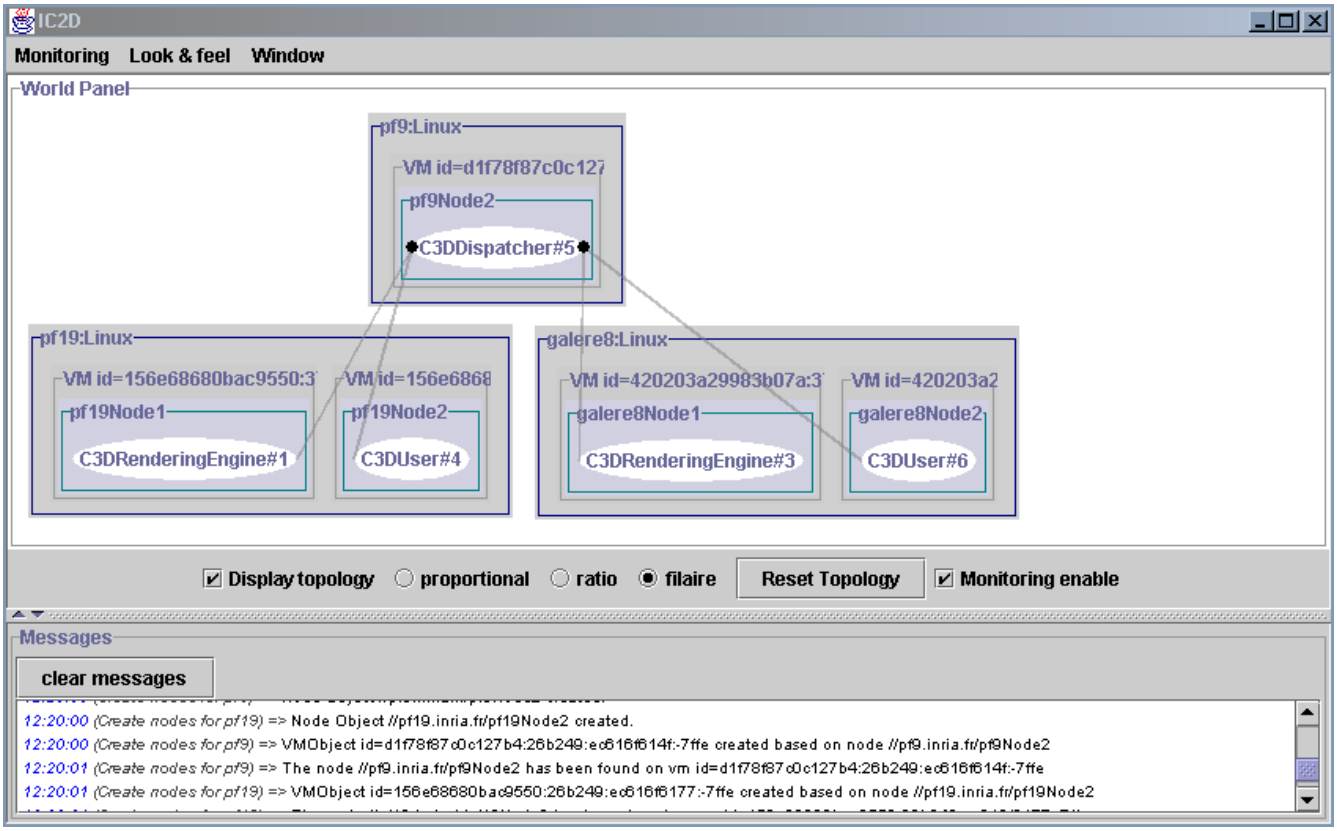

Figure 4. General view of what *IC2D* displays when an application is running

this example, a single node is running on each VM. Each active object is uniquely identified by *IC2D* using the type of the active object, and an index number managed globally for a world Panel (`C3DRenderingEngine #1` for instance).

4.2 Topology and message traffic

What is called *topology* is the set of references of active objects. Only the method calls that actually occur build up the drawn topology. The dot at each end of a grey line depicts the target of the remote call. As the amount of traffic generated by messages is a good indication of the way the application is structured into its various components, it is possible to display communication lines proportional to traffic towards an active object (see on Figure 9 the width of grey lines between communicating active objects).

4.3 Distributed events

The topology gives only quantitative information. Another view provides qualitative information and details all monitored events occurring in the distributed application (see Figure 5). The user can clicks on events, like for example the event `[C3DUser#4] set Pixels` in the active object numbered `5` (`C3DDispatcher`) (which corresponds to a request send), the pair of events pertaining to the same method call (for instance, request send and receive) are displayed using exactly the same color (in yellow). A given maximum (5 in fact) of events (and their matching ones) that happened before (resp. after) on object `5` are displayed using a graduation of blue (resp. red). Pairs of events are easily matched thanks to a sequence number attributed to each call towards active objects, and to the fact that communications between active objects are FIFO ordered (due to the rendez-vous, see section 2.1).

Some events that occurred locally but are indirectly related to method calls towards active objects are also shown: `ObjectWaitByNecessity` means that the reply of a call has not yet arrived and that the caller has incurred a blocking wait on it because it needed it in order to resume its computation; `ObjectWaitForRequest` means that the active object has incurred a wait because its queue of pending requests was empty. This gives a good feedback on the activity of the object and its workload.

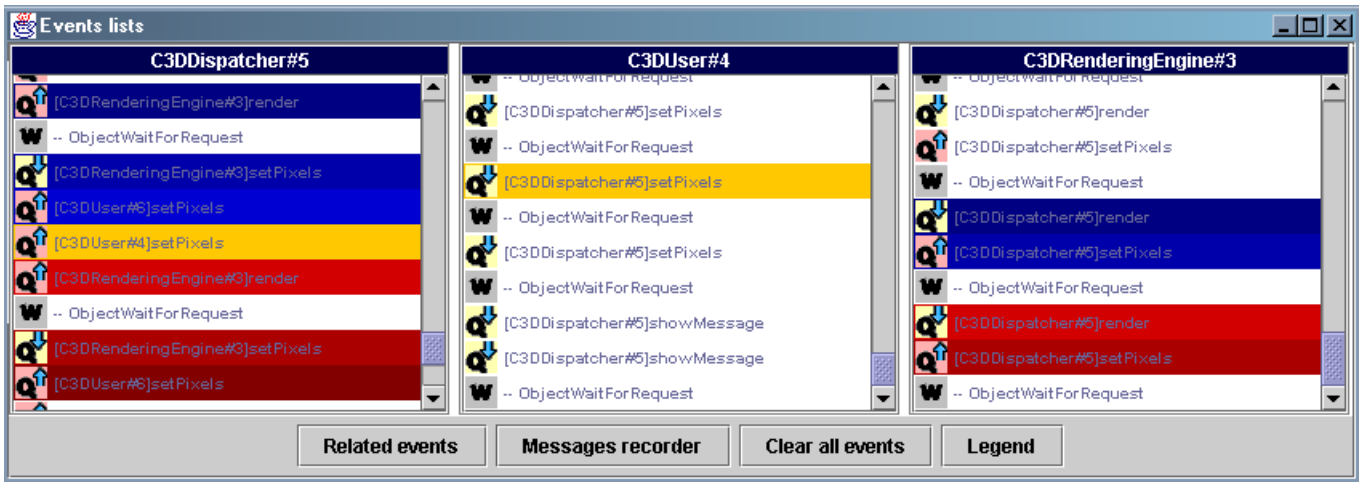

Figure 5. Communication related events that *IC2D* displays when an application is running

4.4 Control

As the availability of computing resources varies over time, especially in grid-based computing environments where many users share hosts, there are strong needs for easy-to-use deployment and control tools, even if they are basic. Without any change to the existing application *IC2D* addresses some of those needs by providing :

- A way to interactively **create** ProActive nodes on local, remote or Globus-enabled hosts, which implies to first launch corresponding JVMs (see Figure 6).
- Correspondingly, a way to **discover** or **locate** ProActive nodes that are already running on local or remote hosts and that could serve as entry points within JVMs for active objects.
  IC2D can be launched even if ProActive applications are already running. There is an easy way to monitor those ongoing applications by asking *IC2D* to *acquire* a specific or all ProActive nodes that already execute on a given host. In the case of RMI, the registry running on a host will be asked to return references to ProActive nodes, that subsequently will be visualized in the `World Panel` (see Figure 4). *IC2D* can also use other facilities such as Jini or Globus to acquire nodes.
- A way to interactively **drag-and-drop any running active object** to move it to any ProActive node displayed by *IC2D*.
  The effect of the drag-and-drop event is to dynamically insert inside the target active object requests queue, a `migrateTo()` request with the target node location as parameter. Additional load metrics would be needed to use such an interactive feature for load-balancing purposes. Yet, it has proven to be very useful for deployment purposes, for instance enabling the user to carry with him, from desktops to desktops, an active object acting as a graphical interface assuming he has previously launched nodes on the desktops he will be sitting in front of.

A complete listing of IC2D features is given by Figure 7.

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:
- Step by step execution
- Drag and Drop migration of executing tasks
- Creation of additional JVMs and nodes

Figure 7. The basic features of IC2D

4.5 Design and Implementation

The *IC2D* system is an external part of the core *ProActive* library. It is built according to the usual pattern for event notification. *IC2D* is composed of a central monitor and a spy on each *ProActive* observed JVM (there is for the moment no security considerations in order to limit acquisition

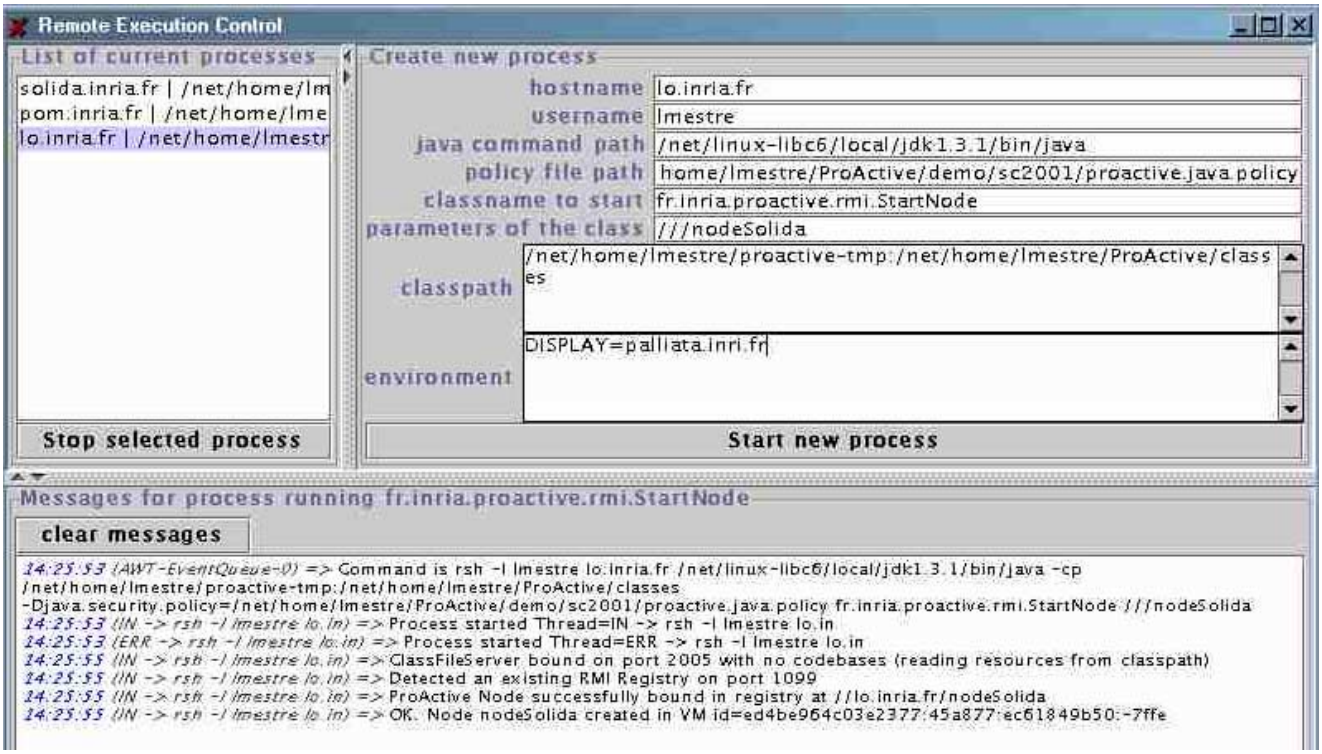

Figure 6. Interactive creation of a new JVM and associated node

of nodes that run on JVMs to authorized users). A spy is an active object that receives events occurring on nodes or on any active object attached to those nodes. Events are eventually sent back to the *IC2D* application owner of the spy and displayed to the end-user by the monitor. The graphical representation of those events could be chosen dynamically from textual, hierarchical form for instance. At the present time, only the view shown on Figure 4 is available.

IC2D does not require to instrument the application at all, as long as it is built using the *ProActive* library. As the library is built on top of a Meta-Object Protocol (MOP), all information that pertains to monitoring or steering the distributed features are implemented at the meta-level: active object creations or migrations, remote method calls are reified in the meta-level, where it is possible for each of those operations to trigger a corresponding event (i.e. the notification of this event to the spy acting as a listener).

For the sake of scalability and readability, the listening of each of these events can be turned on and off on a per-active object, per-node or per-host basis, or for all the currently acquired nodes. Of course, the corresponding events are not generated by the MOP, nor sent to spies if the listening has been turned off. Figure 8 lists events that are notified *on demand* to spies on every node, then displayed by the monitor.

Activities:
- Creation of active objects
- Migration of active objects

Communication:
- Sending a Request
- Receiving a Request
- Sending a Reply
- Receiving a Reply

Waits:
- Waiting for a reply (wait-by-necessity)
- Waiting for a request

Figure 8. The basic events monitored by IC2D

5 Comparison with related work

5.1 Monitoring and Steering

On-line monitoring, visualising and debugging distributed applications is a very broad area and we only briefly mention here some of the most relevant works, like for example xpvm for assisting in debugging PVM applications [10] or ParaGraph, a performance visualisation tool

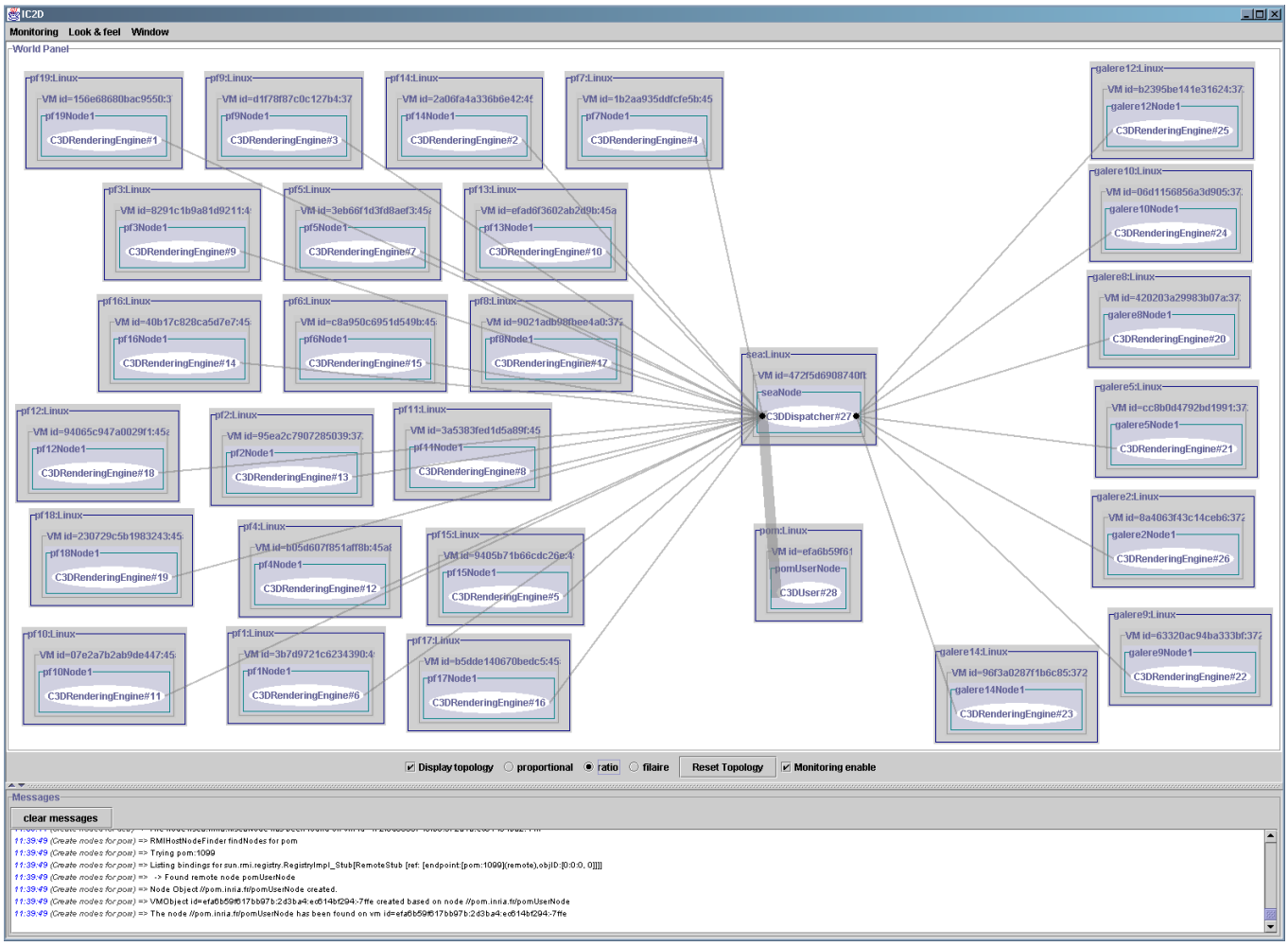

Figure 9. Visualisation of a *ProActive* application, executing on 2 LSF allocated clusters of PCs interconnected with 1Gbits/sec links.

for Paragon applications [13]. The aforementioned environments target preferably parallel computers or clusters. As *IC2D* plus *ProActive* can indifferently run on any support, we could execute it in on various kind of machines, including ones not on the same administrative domain but accessible through Globus. and also on clusters of PCs (see Figure 9).

Interactive program steering pertains to the *runtime manipulation* of an application program and its execution environment. Usually, application developers themselves, not library developers, create 'steerable' applications by identifying components of the application to export to the end-user, for instance with the Progress [18] (Program and Resource Steering System) toolkit, or in MOSS [7] introducing a mirroring level. Using *IC2D*, *ProActive* applications are steerable, but in the limited sense of distribution-related data and operations. But as a consequence, there is no need for *ProActive* application developers to instrument their application, as all objects are already mirrored in a sense, through reification, at the meta level.

5.2 Grid-enabled programming environments

One major problem in the development of grids is to define adequate programming models and environments [12]. We will only discuss a few distributed object-oriented or component-based programming environments, as they provide a better encapsulation and abstraction than any of the lowest-level programming systems, such as for example grid-enabled implementations of the Message Passing Interface or RPC systems [14]. Nevertheless, note that for those programming tools, there is also a need to have solutions to ease of set up, configuration and installation [1].

Note also that *ProActive* is not supposed to be a new object-oriented middleware targeted as using the Internet as "the computer", like Globe [17], Legion [11], etc. Instead it leverages a well-adopted platform, i.e. Java and some of its standard libraries.

Moba/G [19] is a grid-based Java thread migration system and as such shares some features with *ProActive*. But it lacks, for instance, some of the features *IC2D* provides: visualisation of the topology and objects, drag-and-drop migration, etc.

In CCA, a software component framework for building Grid applications [2], components define external interfaces, as provides ports or uses ports and a mechanism to interconnect both kinds of ports. In the XML-oriented implementation of CCA (XCAT [9]), attached XML documents to components contain information about where a component is installed, how an instance can be created, etc. Scripts written for instance in Jython (Python in Java) or embedded themselves into "Application Manager" components [9] interpret such XML documents in order to create instances, connect to other components, etc. *ProActive* descriptors provide the same kind of functionality targeted to Java components. *ProActive* with its descriptors tries to avoid as much as possible the use of scripting configuration files as they can become even more complex than application code itself. To further help in that purpose, *IC2D* acts as both a front-end and a monitor.

6 Conclusion

Through the use of deployment descriptors and the monitoring application *IC2D* we showed how to address the difficult problem of seamless deployment and control of distributed applications of various kinds: high-performance collaborative distributed computations on clusters or grids, mobile object systems, network management platforms, etc. The main points are:

- There is no more reference to machine names, creation protocols, registry and lookup protocols within the source code. Instead the application uses *virtual nodes* that are logical entities on which the active components of the application can be deployed.
- A descriptor maps the logical entities defined in the application onto computers allowing total flexibility of deployment and support of many different creation, lookup and registration protocols.
- Without any change to the application it is possible to monitor and control it using *IC2D*.

Deployment descriptors and *IC2D* are far from independent; for instance they both need and use the capability to launch JVMs (see Figures 2 and 6) with various protocols (ssh, Globus, LSF, etc.), to discover computation from many lookup protocols (Jini, Globus, UDDI), etc. Used in combination, they allow to write a distributed application focusing only on the logical computing entities it needs, deploy it on various environment (machine, cluster, grid), control and visualize the deployment graphically, and modify the deployment while the application is running.

If *IC2D* is already capable of reading a descriptor file to monitor a given running distributed application, we foresee many other improvements in the integration of the two. *IC2D* can be used to interactively deploy the components of a distributed application and automatically generate the descriptor, or at least an outline of it, based on the current deployment.

Ultimately, we would like to define reusable distributed components that can be hierarchically composed to build-up an application. Each component would be associated with its own descriptor trying here to converge towards a complete description as WSDL does for Web Services. There is here a central point of convergence as the current trend is to reuse the work done for Web Services for the Grid as in the emerging Open Grid Services Architecture [8]. The *ProActive* components could be seen as services accessible through a web-services based grid portal.

References

[1] M. Baker and G. Smith. Establishing a Reliable Jini Infrastructure for Parallel Applications. *Parallel Processing Letters*, 2001.

[2] R. Bramley, K. Chin, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yochuri. A Component-Based Services Architecture for Building Distributed Applications. In *9th IEEE International Symposium on High Performance Distributed Computing Conference*, 2000.

[3] D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[4] D. Caromel, F. Belloncle, and Y. Roudier. The C++// Language. In *Parallel Programming using C++*, pages 257–296. MIT Press, 1996. ISBN 0-262-73118-5.

[5] D. Caromel, W. Klauser, and J. Vayssiere. Towards seamless computing and metacomputing in java. *Concurrency Practice and Experience*, 10(11–13):1043–1061, November 1998.

[6] K. Dincer. Ubiquitous message passing interface implementation in java: Jmpi. In *Proc. 13th Int. Parallel*

Processing Symp. and 10th Symp. on Parallel and Distributed Processing. IEEE.

[7] G. Eisenhauer and K. Schwan. An Object-Based Infrastructure for Program Monitoring and Steering. In *2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*.

[8] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid. `http://www.globus.org/research/papers/ogsa.pdf`.

[9] D. Gannon, R. Bramley, G. Fox, S. Smallen, A. Rossi, R. Ananthakrishnan, F. Bertrand, K. Chiu, M. Farrellee, M. Govindaraju, S. Krishnan, L. Ramakrishnan, Y. Simmhan, A. Slominski, Y. Ma, C. Olariu, and N. Rey-Cenvaz. Programming the Grid: Distributed Software Components, P2P and Grid Web Services for Scientific Applications. In Grid 2001. Submitted to the Journal of Cluster Computing, 2002.

[10] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press.

[11] A. Grimshaw and W. Wulf et al. The Legion Vision of a World-wide Virtual Computer. *Communications of the ACM*, 40(1), 1997.

[12] C. Lee, S. Matsuoka, D. Talia, A. Sussman, N. Karonis, G. Allen, and M. Thomas. A grid programming primer. Draft 2.4 of the Programming Models Working Group presented at the Global Grid Forum 1, March 2001.

[13] B. Ries, R. Anderson, W. Auld, D. Breazeal, K. Callaghan, E. Richards, and W. Smith. The Paragon performance monitoring environment. In *Proc. Supercomputing '93*, pages 850–859. IEEE Computer Society, 1993.

[14] S. Sekiguchi, M. Sato, H. Nakada, S. Matsuoka, and U. Nagashima. Ninf: Network-based information library for globally high performance computing. In *Parallel Object-Oriented Methods and Applications (POOMA)*, pages 39–48, 1996. `http://ninf.etl.go.jp`.

[15] Sun Microsystems. Java core reflection, 1998. `http://java.sun.com/products/jdk/1.2/docs/guide/reflection/`.

[16] Sun Microsystems. Java remote method invocation specification, October 1998. `ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf`.

[17] M. van Steen, P. Homburg, and A.S. Tanenbaum. Globe: A Wide-Area Distributed System. *IEEE Concurrency*, 7(1):70–78, 1999.

[18] J.S. Vetter and K. Schwan. Progress: a toolkit for interactive program steering. In *International Conference on Parallel Processing*, 1995.

[19] G. von Laszewski, K. Shudo, and Y. Muraoka. Grid-based asynchronous migration of execution context in java virtual machines. In R. Wismller A. Bode, T. Ludwig, editor, *Euro-Par 2000 - Parallel Processing*, number 1900 in LNCS. Springer-Verlag.