IC2D: Interactive Control and Debugging of Distribution

Françoise Baude, Alexandre Bergel, Denis Caromel,
Fabrice Huet, Olivier Nano, and Julien Vayssière

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

Abstract. Within the trend of object-based distributed programming, we present a non-intrusive graphical environment for remote monitoring and steering, *IC2D*: Interactive Control and Debugging of Distribution. Applications developped using the 100% Java ProActive PDC (Parallel, Distributed and Concurrent) computing library are monitored for 'free' by *IC2D*. As those targetted applications can run on any distributed runtime support ranging from multiprocessor workstations, clusters, to grid-based infrastructures (through the Globus toolkit), *IC2D* turns out to be a grid-enabled programming environment.

Keywords: distributed computing, metacomputing, active object, migration, graphical visualisation, debugging, monitoring, steering, object-oriented.

1 Introduction

The results we present in this paper capitalise on research performed over the last few years on the *ProActive* PDC (Parallel Distributed and Concurrent) library [4]. *ProActive* is a library for concurrent, distributed and mobile computing in Java. As *ProActive* is a 100% Java application, applications built using it can run on any kind of machine (workstations, multiprocessors servers, clusters, etc) and under any operating system, provided that there exists an implementation of the Java virtual machine for the platform in question.

In this paper we describe *IC2D*, which is a graphical environment for monitoring and steering applications built using *ProActive*. It enables the programmer to dynamically visualise the inner workings of a *ProActive* application at runtime and also allows the user to interactively control the mapping of tasks onto machines, either upon creation or at migration time. The underlying motivation is to help users to deploy, monitor and control *ProActive* computations running on either kind of distributed platforms including grids.

Section 2 provides some background on *ProActive*. Then, in section 3, we present the main features that *IC2D* brings to *ProActive* applications. We then provide a comparison with related work in section 4.

S. Margenov, J. Wasniewski, and P. Yalamov (Eds.): ICLSSC 2001, LNCS 2179, pp. 193–200, 2001.

2 Distributed and Mobile Active Objects with *ProActive*

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 [2,3].

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 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 sent to active objects are always asynchronous with transparent *future objects* and synchronisation is handled by a mechanism known as *wait-by-necessity*.

The *ProActive* library provides a way to migrate any active object from any JVM to any other one. This is done through a `MigrateTo(...)` primitive which can either be called from the object itself or through a method call from another active object.

2.2 Mapping Active Objects

A *Node* is an object defined in *ProActive* whose aim is to host several active objects. It provides an abstraction for the physical location of a set of active objects. An active object can be bound to a node either at creation time or as the result of a migration. As active objects execute within Java Virtual Machines, there is actually a simple way to think about nodes: nodes can be seen as entry points to JVMs. If the programmer does not need or want to explicitly work with nodes, a default node is created on each JVM and active objects are automatically bound to it.

In order to name and handle nodes in a simple manner in the entire *ProActive* system, each node must be labelled with a name. This name is usually an URL that consists of the machine hostname and a string (e.g. `//sakuraii/Node1`). This URL is then registered with `rmiregistry`. Active objects, just like nodes, can also be named in order to be registered and subsequently located. An additional way to register and locate nodes or active objects is to use the Lookup Service of Jini [5]. New participants will then be able to dynamically discover nodes or active objects, and join an on-going *ProActive* computation. These various means of registering and locating are of uttermost interest for collaborative distributed applications for instance.

[1] Java RMI [15], the Reflection API [14],...

Execution in a Metacomputing Environment. In order to launch ProActive nodes on 'foreign' hosts, a metacomputing system must be brought in. We currently rely on the Globus system [7] and the Java CoGKit interface [16], in order to start JVMs and ProActive nodes. We also can make use of dynamic class loading, thanks to a RMI class file server in order to avoid to manually transfer class files before their use. Foreign nodes will be registered as usual in distributed instances of the `rmiregistry`. As a consequence, the only change to deploy *ProActive* applications is to modify command-line parameters in order to specify which 'globus' machines are used. Notice here that the deployment is done by hand.

3 Visualisation and Control within the *IC2D* Environment

Figure 1 provides a quick summary of the features *IC2D* adds to *ProActive* applications.

3.1 Visualisation

Figure 2 gives an overview of the two sorts of information that *IC2D* provides to the user: information about the support of the *ProActive* computation, and information about the progress of the computation.

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

Textual Visualisation:
- Ordered list of messages exchanged by active objects
- Status of active objects: waiting for a request or for a reply
- Causal dependencies between messages
- Related events (corresponding send and receive, etc.)

Control and Monitoring:
- Interactive control of mapping of active objects upon creation
- Interactive control of destination of active objects upon migration
- Step by step execution
- Drag and Drop migration of executing active objects

Fig. 1. A summary of the basic features of IC2D

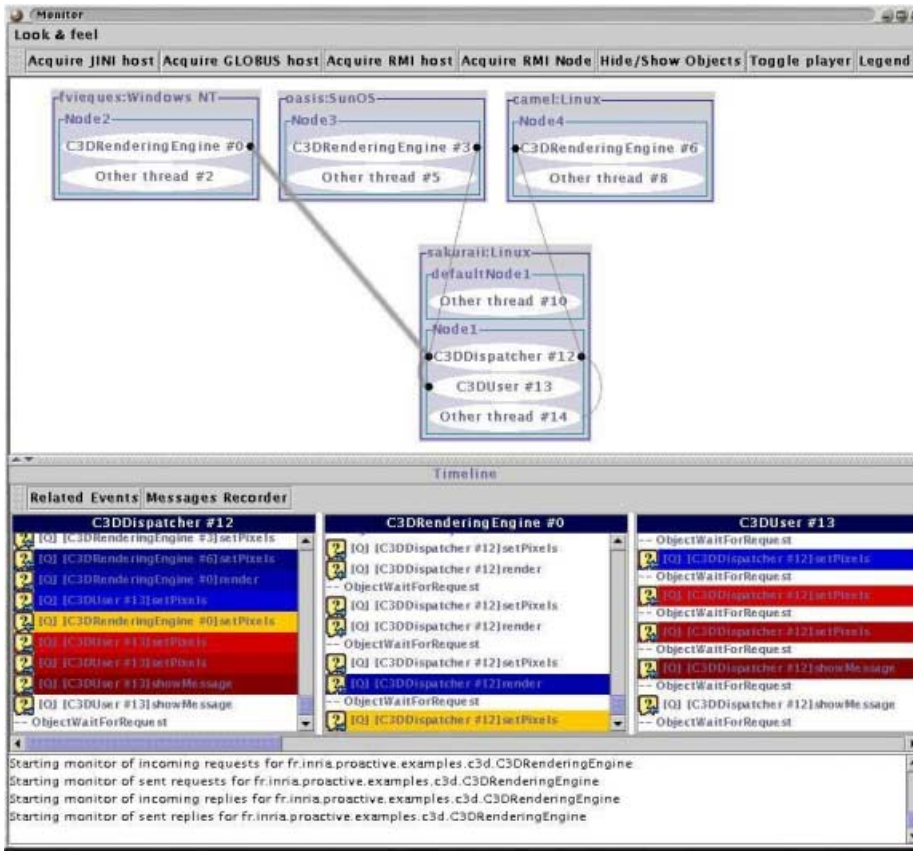

Fig. 2. General view of what *IC2D* displays when an application is running

In the top part of figure 2, one can visualise the imbrication of hosts, VMs, and ProActive nodes[2]. The *topology* shows the set of *used* references (i.e. communications) between active objects. The dot at each end of a grey line depicts the endpoint of the remote call. As the message traffic is a good indication of the way the application is structured into its various components, the bigger the message traffic towards an active object, the bigger the width of this line.

In the bottom part of figure 2, one can visualise any portion of the message flow on graphically selected active objects (here, `C3DUser #13, C3DDispatcher #12, C3DRenderingEngine#0`) and more precisely for a given event (here, the request reception `[C3DDispatcher #12]`), all its causally-related events in the whole *ProActive* computation. Some events that occurred locally but are indirectly related to method calls towards active objects are also shown:

[2] In the figure, each rectangle inside a grey box is a JVM, which means that there is exactly one VM running on each host, except on `sakurai` where 2 are running

`ObjectWaitByNecessity` (a reply is awaited), `ObjectWaitForRequest` (the queue of requests is empty) This gives a good feedback of the activity of the object and its workload.

3.2 Monitoring

As the availability of computing resources varies over time, especially in grid-based computing environments where many users share hosts, there is a strong need for easy-to-use deployment and control tools. We now detail the most significant features IC2D provides as a solution to various monitoring needs. Notice that all needs are satisfied without any change, nor recompilation, of the existing application.

At creation or at migration time, a way to interactively ***associate*** *the new or mobile active object to any already-running ProActive node:* An active object creation or migration arising on a given node is instrumented: the event corresponding to this action is notified and then triggers a dialog box with the *IC2D* user, see figure 3.

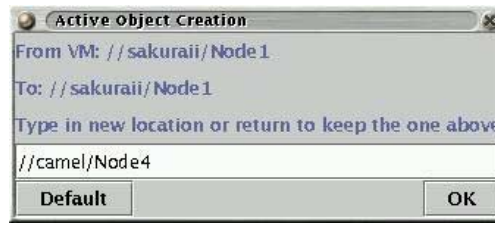

Fig. 3. Interactive mapping of a new *ProActive* active object

A way to interactively ***move*** *an ongoing active object to an other ProActive node:* As illustrated by figure 3, even if it is already possible to dynamically modify the location of a new or a migrating active object *IC2D* adds the feature to **drag-and-drop any running active object** such as to move it on any ProActive node displayed by *IC2D*. The only constraint for an active object to be the target of such user-driven migration is to implement a specific ProActive interface called *movable*. The effect of the drag-and-drop event is to dynamically put in front of the target active object requests queue, a `MigrateTo()` method call with the target node location as parameter.

By using the *ProActive* library only without *IC2D*, the programmer has the ability to hand-code the solution to all the above requirements, but this will lead to the intermixing of the application code with the code for monitoring and debugging.

3.3 Design

The *IC2D* system is an external part of *ProActive* applications, and moreover it is not mandatory to run it as a permanent part. It is built according to the

usual pattern for event notification. This external part, composed of a central and unique monitor and a spy on each node, plays the role of an observer: events are delivered to the spy, processed and eventually displayed to the end-user by the monitor. The spy is implemented as an active object.

In order to control or steer the application, some kind of events, such as active object creations or migrations should not only be notified but should in addition trigger a given action, that is an interactive modification of some parameters pertaining to the operation the event notifies. In this case, the spy does not only act as a listener, but as a *listener-modifier*.

4 Related Work

4.1 Monitoring

On-line monitoring, visualising and debugging distributed applications is a very broad area. Two widely known examples are XPVM for assisting in debugging PVM applications [8] or ParaGraph, a performance visualisation tool for Paragon applications [11]. *IC2D* compares well with them.

4.2 Steering

Interactive program steering pertains to the *runtime manipulation* of an application program and its execution environment. Usually, application developers themselves create 'steerable' applications by identifying components of the application to export to the end-user. For example, through the Progress [9] toolkit, the programmer must first define and register steering objects (for instance for some complex data of its program) and the operations on them. He then must instrument its application in order to call those operations, synchronise with their execution, etc. As *IC2D* is dedicated to monitor and steer distributed features only, it does not require the instrumentation of the application. Instead, only the *ProActive* library methods that manage meta-objects dedicated to distribution need to be instrumented.

4.3 Grid-Enabled Programming Environments

The development of grid computing environments, problem solving environments (PSEs) and computing portals is a very active and challenging area [10]. We will only discuss a few object-oriented 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 [12]. Nevertheless, *ProActive* has a similar aim, but with a strong emphasise on code reuse, flexibility, extensibility. At the other end of the spectrum (i.e. at a level closer to applications) we can mention problem solving environments, like for example Cactus [1], but which are quite difficult to compare with *IC2D* as they are dedicated to specific application domains.

Moba/G [17] is a grid-based Java thread migration system and as such shares many features with *ProActive*. But it lacks some of the features *IC2D* provides: visualisation of the topology and objects, drag-and-drop migration, etc. GECCO (Grid Enabled Console COmponet) [16] is a high-level graphical tool for specifying and monitoring the execution of sets of tasks with dependencies between them. The main difference lies in the fact that *IC2D* non-intrusively and globally monitors and debugs activities at a finer level than the task/job granularity, i.e. at the level of a collective and connected set of distributed communicating objects.

5 Conclusion

We have presented *IC2D*, a graphical environment which enables a programmer to interactively control and debug the distribution aspect of *ProActive* applications, which can themselves be of various kinds: collaborative and/or high-performance distributed computations on clusters and/or grids, mobile object based system and network management platforms, etc. The important point is that no change at all to *ProActive* applications is required.

We are currently working on leveraging *IC2D* as a portal and using it for new or already existing applications. For instance, *IC2D* has been recently used in our team in order to monitor existing Enterprise Java Beans applications: the only modification to the code is to turn a bean into an active object; then through *IC2D* running as an applet, the bean can easily be distributed on servers, be moved by the user with a drag-and-drop action, etc.

References

1. G. Allen, W. Benger, T. Goodale, H-C. Hege, G. Lanfermann, A. Merzky, T. Radke, and E. Seidel. The Cactus Code: A problem solving environment for the grid, in *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC9)*, 2000.
2. D. Caromel. Towards a method of object-oriented concurrent programming, *Communications of the ACM*, 36(9), 90–102, 1993.
3. D. Caromel, F. Belloncle, and Y. Roudier. The C++// Language, in *Parallel Programming using C++*, MIT Press, 257–296, 1996. ISBN 0-262-73118-5.
4. D. Caromel, W. Klauser, and J. Vayssiere. Towards seamless computing and metacomputing in Java, in *Concurrency Practice and Experience*, 10(11–13), 1043–1061, 1998.
5. W. Keith Edwards. *Core JINI*, Prentice Hall, 1999.
6. 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)*.
7. I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit, *Int. Journal of Supercomputer Applications*, 11(2), 115–128, 1997.
8. 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.

9. W. Gu, G. Eisenhaur, E. Kraemer, K. Schwan, J. Stasko, and J. Vetter. Falcon: On-line monitoring and steering of parallel programs, in *Concurrency: Practice and Experience.*, 1998.
10. 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.
11. B. Ries, R. Anderson, W. Auld, D. Breazeal, K. Callaghan, E. Richards, and W. Smith. The Paragon performance monitoring environment, in *Proc. Supercomputing '93*, IEEE Computer Society, 850–859, 1993.
12. S. Sekiguchi, M. Sato, H. Nakada, S. Matsuoka, and U. Nagashima. Ninf: Network-based information library for globally high performance computing, *Parallel Object-Oriented Methods and Applications (POOMA)*, 39–48, 1996. http://ninf.etl.go.jp.
13. B. Sridharan, B. Dasarathy, and A. Mathur. On building non-intrusive performance instrumentation blocks for CORBA-based distributed systems, in *4th IEEE International Computer Performance and Dependability Symposium*, 2000.
14. Sun Microsystems. Java core reflection, 1998. http://java.sun.com/products/jdk/1.2/docs/guide/reflection/index.html.
15. Sun Microsystems. Java remote method invocation specification, October 1998. ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf.
16. G. von Laszewski, I. Foster, J. Gawor, W. Smith, and S. Tuecke. Cog kits: A bridge between commodity distributed computing and high-performance grids, in *ACM Java Grande Conference*, http://www.extreme.indiana.edu/java00., San Francisco, California, June 2000.
17. G. von Laszewski, K. Shudo, and Y. Muraoka. Grid-based asynchronous migration of execution context in Java virtual machines, in R. Wismüller, A. Bode, Th. Ludwig, (eds.), *Euro-Par 2000 - Parallel Processing*, *LNCS*, Springer-Verlag, 1900.