CMST 12(1) 69-77 (2006)

DOI:10.12921/cmst.2006.12.01.69-77

COMPUTATIONAL METHODS IN SCIENCE AND TECHNOLOGY **12(1)**, 69-77 (2006)

ProActive: an integrated platform for programming and running applications on Grids and P2P systems

Denis Caromel, Christian Delbé, Alexandre di Costanzo, Mario Leyton

INRIA Sophia-Antipolis, CNRS, I3S, UNSA, 2004, Route des Lucioles, BP 93
F-06902 Sophia-Antipolis Cedex, France
e-mail: First.Last@sophia.inria.fr

Abstract: We propose a grid programming approach using the ProActive middleware. The proposed strategy addresses several grid concerns, which we have classified into three categories. I. *Grid Infrastructure* which handles the resource acquisition and creation using deployment descriptors and Peer-to-Peer. II. *Grid Technical Services* which can provide non-functional transparent services like: fault tolerance, load balancing, and file transfer. III. *Grid Higher Level* programming with: group communication and hierarchical components. We have validated our approach with several grid programming experiences running applications on heterogeneous Grid resource using more than 1000 CPUs.

Key words: Grid, middleware, active object, descriptor, deployment, peer-to-peer, fault-tolerance, load balancing, file transfer

1. INTRODUCTION

Programming for the Grid raises new challenge for parallel and distributed programming. Mainly characterized by resource heterogeneousness, location dispersity, and high volatility (among others); the Grid requires the adoption of new programming paradigms that address these issues.

Based on our research, in this article we propose an approach for Grid programming using the ProActive middleware. ProActive was originally created as an implementation of Active Object programming model [11], and has developed into a multifeatured middleware for programming and deploying distributed applications on the Grid. ProActive is still evolving; and currently represents the manifestation of our studies.

Released under the LGPL license ProActive is a Java library for parallel, distributed, and concurrent computing, also featuring mobility and security in a uniform framework. With a reduced set of simple primitives, ProActive provides a comprehensive API allowing to simplify the programming of applications that are distributed on Local Area Networks (LAN), on clusters, or on Internet Grids.

The *Grid Infrastructure* based on resource creation and acquisition through deployment descriptors, provides a level of abstraction that allows removing from the application source code any reference of infrastructure (hardware, software, hosts, protocol, and hardware). Applying several *Grid Technical Services* non-functional and transparent aspects such as: fault tolerance, load balancing and file transfer can be transparently used to overcome the burden of programming distributed applications. ProActive also provides *Higher Level Programming* strategies that provide a further abstraction for Grid programming using paradigms like: typed group communication [11] and hierarchical components [7].

This document is organized as follows. In section 2 we give a general background on the active object programming model and ProActive, then in section 3 we discuss how the Grid Infrastructure can be built using Descriptors and Peer-to-Peer. Once we have established the Grid Infrastructure, in section 4 we discuss the different services required for Grid programming. We then proceed to section 5 where we show Higher Level Grid programming mechanisms. Then we show in section 6 some Grid programming experiences using our proposed approach, and finally in section 7 we conclude and show our current perspectives on Grid programming.

2. ACTIVE OBJECT PROGRAMMING WITH PROACTIVE

The ProActive middleware is a 100% Java library, which aims at achieving seamless programming for concurrent, parallel, distributed, and mobile computing. It does not require any modification of the standard Java execution environment, nor does it make use of a special compiler, pre-processor, or modified virtual machine.

The ProActive core is a uniform *active object* (AO) programming model. As shown in Fig. 1, AO are remotely accessible *via* method invocation. Each AO has its own thread of control and is granted the ability to decide in which order to serve the incoming method calls, automatically stored in a queue of pending requests. Method calls on AO are asynchronous with automatic synchronization. This is achieved using automatic *future objects* as a result of remote methods calls, and synchronization is handled by

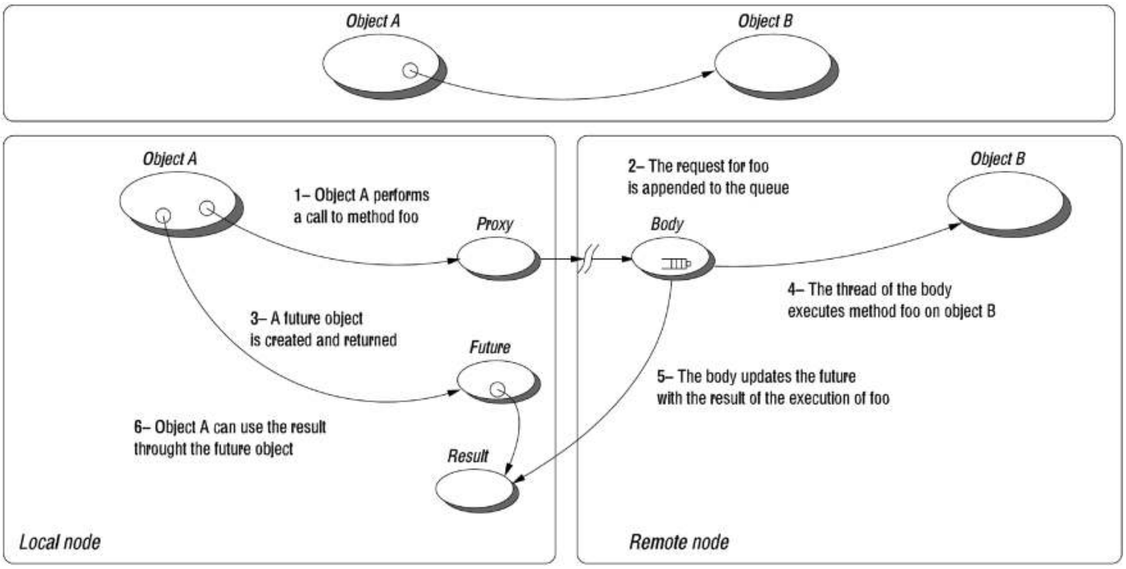

Fig. 1. Execution of a remote method call

a mechanism known as *wait-by-necessity* [11]. To move any AO from any Java Virtual Machine (JVM) to any other, a *migration* mechanism is provided. An AO with its pending requests (method calls), futures, and passive (mandatory non-shared) objects can migrate from JVM to JVM through the migrateTo(...) primitive. The migration can be initiated from outside the AO, but it is the responsibility of the AO to execute the migration, this is known as *weak migration*. Automatic and transparent forwarding of requests and replies provide location transparency, as remote references toward *active mobile objects* remain valid.

ProActive uses by default the RMI Java standard library as a portable communication layer, supporting the following communication protocols: RMI, HTTP, Jini, RMI/SSH, and Ibis [25].

Another Grid communication mechanism is *the typed group communication* model [2]. Group communication allows triggering method calls on a distributed group of *active objects* with compatible type, dynamically generating a group of results. It has been shown in [2] that this group communication mechanism, plus a few synchronization operations (WaitAll, WaitOne, *etc*.), provides quite similar patterns for collective operations such as those available in *e.g*. MPI, but in a language centric approach [3]. The typed group communication model is detailed in section 5.1.

ProActive also provides other higher-level abstractions for grid programming, implementing the hierarchical Fractal component model [7].

Graphical visualization and monitoring of any ongoing ProActive applications is possible through *IC2D* (Interactive Control and Debugging of Distribution) tool. In particular, IC2D enables to migrate executing tasks by a graphical drag-and-drop.

3. GRID INFRASTRUCTURE PROGRAMMING

3.1. Descriptor-based Deployment of Grid Applications

The deployment of distributed applications is commonly done manually through the use of remote shells for launching the various virtual machines or daemons on remote computers and clusters. Deployment on the grid increases the complexity of applications because of the heterogeneousness of resources, thus making the deploying task central and harder to perform.

ProActive succeeds at providing a generic approach for deployment. Using grid *descriptors*, infrastructure details can be removed from the user application in a uniform and abstract way [5]. References to hosts, protocols and other infrastructure details are removed from the application code, and specified in the descriptors using XML.

The grid application is thus contracted with the descriptor through a **VirtualNode**. VirtualNodes are abstractions for the location of resources, and correspond to the actual references in the application code. They have a unique identifier, and can be mapped on to one or several Java Virtual Machines (JVM). These JVMs can be created or acquired (on local or remote sites), through the mapping of processes. A process holds the protocol specific information. The result of mapping a VirtualNode on the resources corresponds to one or several ProActive Nodes.

Effectively, a user can change the mapping of the VirtualNode → JVM → Process to deploy on different sites, without modifying a single line of code in the application.

Figure 2 shows a simple descriptor example. The VirtualNode named *Example* is mapped on to a JVM called *JVMExample*, which in turn is mapped on to a process called *sshProcess*. The sshProcess will perform an ssh connection to the host *example.host* and insantiate a JVM using the defined *jvmProcess*.

The *XML Deployment Descriptors* currently provide interfaces with various protocols: rsh, ssh, LSF, PBS, SGE, Globus, Jini, RMIregistry, EGEE gLite, Unicore, Nordugrid, *etc*., which enable to effectively deploy grid applications.

```
<ProActiveDescriptor xmlns:xsi="http://www.w3.org/2001/XMLSchema
      – instance" xsi:noNamespaceSchemaLocation="DescriptorSchema.
      xsd">
 <componentDefinition>
  <virtualNodesDefinition>
   <virtualNode name="Example"/>
  </virtualNodesDefinition>
 <componentDefinition/>
 <deployment>
  <mapping>
   <map virtualNode="Example">
      <jvmSet>
         <vmName value="JvmExample"/>
      </jvmSet>
   </map>
  </mapping>
  <jvms>
   <jvm name="JvmExample">
    <creation> <processReference refid="sshProcess"/> </creation>
   </jvm>
  </jvms>
 </deployment>
 <infrastructure>
  <processes>
   <processDefinition id="sshProcess">
    <processReference refid="jvmProcess"/>
    <sshProcess class="org.objectweb.proactive.core.process.
         SSHProcess"
        hostname="example.host" username="smith"/>
   </processDefinition>
   <processDefinition id="jvmProcess">
    <jvmProcess class="org.objectweb.proactive.core.process.
         JVMNode Process"/>
   </processDefinition>
  </processes>
 </infrastructure>
</ProActiveDescriptor>
```

Fig. 2. XML Deployment Descriptor Example

In addition, descriptors provide support for other infrastructure services, such as P2P, *File Transfer* and others. In particular, the File Transfer support allows transferring of files, such as data, libraries, Java Virtual Machines, code, ProActive runtime, etc., to remote locations and retrieve of files from remote nodes. In section 4.3 we discuss *File Transfer* in further detail.

3.2 A self-organized and flexible Peer-to-Peer Infrastructure

Existing models and infrastructures for Peer-to-Peer (P2P) computing are rather limited: only targeting independent worker tasks, usually without communications between tasks. Therefore, we propose a *P2P infrastructure* of computational nodes for distributed communicant applications. The infrastructure provides large scale grids for computational intensive applications; such Grids provide a mix of clusters and desktop machines.

The main goal of the P2P infrastructure is to provide a new way to build and use Grids. The infrastructure allows applications to transparently and easily obtain computational resources from Grids composed of both clusters and desktop machines. The application deployment burden is eased by a seamless link between applications and the infrastructure. This link allows: applications to communicate, and to manage the resources volatility.

The proposed P2P infrastructure has three main characteristics. First, the infrastructure is not centralized and completely *self-organized*. Second, it is flexible, thanks to parameters for adapting the infrastructure to the location where it is deployed. Last, the infrastructure is portable since it is built on top of Java Virtual Machines, which run on cluster nodes and on desktop machines. Thus, the infrastructure contributes to ProActive, providing a new way for: deploying applications and acquiring already running JVMs (instead of starting new ones).

For us P2P is defined as "Pure Peer-to-Peer Network" [19]. This definition focus on sharing, decentralization, instability, and fault tolerance.

The proposed P2P infrastructure is an unstructured P2P network, such as Gnutella [15]. Therefore, the infrastructure resource query mechanism is similar to the Gnutella communication system, which is based on the Breadth-First Search algorithm (BFS). The system is message-based with application-level routing. Messages are forwarded to each acquaintance, and if the message has already been received (looped), then it is dropped. The number of hops that a message can take is limited with a *Time-To-Live* (TTL) parameter [1].

As previously mentioned, the main problem of P2P infrastructures is the peers high volatility, since peers are usually desktop machines and clusters nodes available for a short time. Therefore, the proposed infrastructure aims at maintaining the network alive, while available peers exist; this is called *self-organizing*. When it is impossible (or undesired) to have external entities, such as centralized servers, which maintain peer databases, all peers should be capable of staying in the infrastructure by their own means. A widely used strategy for achieving self-organization consists in maintaining, for each peer, a list of acquaintances.

At the beginning, when a fresh peer joins the network, it only knows acquaintances from a list of potential network members, such as with super-peer architectures. The initially known peers will not be permanently available, and therefore peers have to update their list of acquaintances to stay connected in the infrastructure.

Therefore, the proposed infrastructure uses a specific parameter called *Number of Acquaintances* (NOA): the minimum number of known acquaintances for each peer. Peers

[1] The TTL is one of the parameters configurable by the administrator, which has deployed the P2P infrastructure.

update their acquaintance list every *Time to Update* (TTU)[2] checking their own acquaintance list to remove unavailable peers, *i.e.* they send heartbeat messages to them. When the number in the list is less than NOA, a peer will try to discover new acquaintances. To discover new acquaintances, peers send exploring messages through the infrastructure. Note that each peer can have its own parameter values, and that they can be dynamically updated.

Using the proposed P2P infrastructure we have conducted several Grid programming experiences. Some of which are detailed in section 6.1.

4. GRID PROGRAMMING: TECHNICAL SERVICES

In this section we present *Technical Services*, which are non-functional aspects of applications. Those services can be added to the application functional code at the deployment time. Services configurations are based on application needs, potentially taking into account the underlying characteristics of the infrastructure.

4.1. Fault-Tolerance

As the use of desktop grids goes mainstream, the need for adapted fault-tolerance mechanisms increases. Indeed, the probability of failure is dramatically high for such systems: a large number of resources imply a high probability of failure of one of those resources. Moreover, public Internet resources are by nature unreliable.

Rollback-recovery [13] is one solution to achieve fault-tolerance: the state of the application is regularly saved and stored on a stable storage. If a failure occurs, a previously recorded state is used to recover the application. Two main approaches can be distinguished: the *checkpoint-based* [16] approach, relying on recording the state of the processes, and the *log-based* [1] approach, relying on logging and replaying inter-process messages.

Fault-tolerance in ProActive is achieved by rollback-recovery; two different mechanisms are available. The first is a Communication-Induced Checkpointing protocol (CIC): each active object has to checkpoint at least every *TTC* (Time To Checkpoint) seconds. Those checkpoints are synchronized using the application messages to create *a consistent* global state of the application [12]. If a failure occurs, *every active object*, even the non faulty, must restart from its latest checkpoint. The second mechanism is a Pessimistic Message Logging protocol (PML): the difference with the CIC approach is that there is no need for global synchronization, because all the messages delivered to an active object are logged on a stable storage. Each checkpoint is independent: if a failure occurs, only the faulty process has to recover from its latest checkpoint.

Basically, we can compare those two approaches regarding two metrics: the failure-free overhead, *i.e.* the additional execution time induced by the Fault-Tolerance mechanism without failure, and the recovery time, *i.e.* the additional execution time induced by a failure during the execution. The failure-free overhead induced by the CIC protocol is usually low [6], as the synchronization between active objects relies only on the messages sent by the application. Of course, this overhead depends on the TTC value, set by the programmer; the TTC value depends mainly on the assessed frequency of failures. A small TTC value leads to very frequent global state creation and thus to a small rollback in the execution in case of failure. But a small TTC value leads also to a higher failure free overhead. The counterpart is that the recovery time could be high since all the application must restart after the failure of one or more active object.

As for CIC protocol, the TTC value impacts on the global failure-free overhead, but the overhead is more linked to the communication rate of the application. Regarding the CIC protocol, the PML protocol induces a higher overhead on failure-free execution. But the recovery time is lower as a single failure does not involve all the system: only the faulty has to recover.

Choosing one of those two approaches highly depends on the characteristics of the application and of the underlying hardware. We thus aim to provide a fault-tolerance mechanism that allows to choose the best approach *at deployment time*: the programmer can specify *in the deployment descriptor* if the application must be started with fault-tolerance, and can select the best mechanism and configuration regarding the hardware environment. In particular, there is no need to alter the original source code of an application to make it fault-tolerant: all the fault-tolerance concerns are handled transparently *at the middleware level.*

4.2. Load balancing of Active Objects

An important feature of Grid systems, is the ability to redistribute tasks among its processors. This requires a redistribution policy to gain in productivity by dispatching the tasks in such a way that the resources are used efficiently, *i.e.* minimizing the average idle time of the processors and improving applications performance.

Load balancing is the process of distributing parallel application tasks on a set of processors while improving the performance and reducing the application response time. The decisions of when, where and which tasks have to be transferred are critical; and therefore the load information has to be accurate and up-to-date [17]. When these decisions are taken at runtime, this process is called *dynamic load balancing*.

Dynamic load balancing requires the collection of information such as: underloaded and overloaded Grid nodes.

[2] NOA and TTU are also both configurable.

For communication-intensive applications[3], in [10] we experimentally showed that *Distributed* oriented policies (opposed to *Centralized*) have the best performance (using response time and bandwidth as metrics). And, that sharing underloaded nodes information, *Eager* policies (opposed to *Lazy*), is the best decision. Therefore, for this kind of applications, we concluded that the best strategy is the *Eager Distributed* policy: overloaded nodes trigger the balancing using previously collected information of underloaded nodes.

For *load balancing* in P2P networks, we presented [9] an active object load balancing algorithm based on well known algorithms [21] and adapted for an heterogeneous[4] P2P infrastructure. This algorithm is a dynamic, scalable, and fully distributed load balancer, which reacts to load perturbations on the processor and the system.

Using these experiments, we have set up, and continue to improve, a load balancing mechanism for programming on the Grid using ProActive.

4.3. File Transfer

File transfer in the Grid can be addressed from two different perspectives: *infrastructure* and *programming*.

From the *infrastructure* point of view, the Grid must provide tools for transferring / accessing files. Given the heterogeneous nature of the Grid, the file transfer tools will be miscellaneous: scp, rcp, gridftp, unicore [24], nordugrid-arc [18], *etc*. Providing a general abstraction for all the file transfer tools diversity is addressed in ProActive through the Descriptor Deployment model (see section 3.1). This allows transferring files at deployment time to the remote Grid nodes. For example, to provide the input for the application. Later on, at the end of the application, the gathering of remote files from the remote nodes can take place. For example, gathering the result of the application.

From the programming perspective, a uniform methodology is required. Different approaches have been used to address this issue, for example GAT transparent remote access [20]. From the active object Grid Programming model (see section 2), the ProActive solution [4] is to provide asynchronous file transfer with future objects through a programming API:

```
//Sends a file to Node n
static public void pushFile(Node n, File source, File
        destination);
//Gets a file from Node n
static public File pullFile(Node n, File source, File
        destination);
```

With the use of asynchronism and futures, a file transfer can take place in parallel with the user application. The wait-by-necessity mechanism will automatically synchronize the threads only if the file needs to be accessed while the transfer is still taking place.

5. HIGHER LEVEL GRID PROGRAMMING

5.1. Typed Group Communications

Group communication is an important feature for high-performance and Grid computing, for which MPI is generally the only available coordination model [3].

The typed group communication mechanism [2] is built upon the ProActive elementary mechanism for asynchronous remote method invocation with automatic futures. The group mechanism must be thought of as a replication of more than one (say *N*) ProActive remote method invocations towards *N* active objects. Of course, the aim is to incorporate some optimizations into the group mechanism implementation, in such a way as to achieve better performances than a sequential achievement of *N* individual ProActive remote method calls. In this way, the mechanism is a generalization of the remote method call mechanism of ProActive.

The availability of such a group communication mechanism, simplifies the programming of applications with similar activities running in parallel. Indeed, from the programming point of view, using a group of active objects of the same type, subsequently called a typed group, takes exactly the same form as using only one active object of this type. This is possible due to the fact that the ProActive library is built upon reification techniques.

```
Object[][] constructorArray = {{...},{...},...};
Node[] nodes = {...,...,... };
A ag1 = (A) ProActiveGroup.newActiveGroup("A", constructorArray,
        nodes);
...
ag1.foo(...);
// A group communication //
A method call on a typed group
V vg = ag1.bar();
// To wait and capture the first returned member of vg
V v = (V) ProActiveGroup.waitAndGetOne(vg);
// To wait all the members of vg are arrived
ProActiveGroup.waitAll(vg);
```

Fig. 3. Typed group communications

Figure 3 shows an example using typed group communication.

5.2. Distributed Hierarchical Components

Components are attracting research for developing grid applications. In [7] we proposed a parallel and distributed component framework for building Grid applications, adapted to the hierarchical, distributed and heterogeneous nature of the Grid. We extended ProActive implementing

[3] Parallel applications which transfer a large amount of data among processors.

[4] Heterogeneous in processing capacity.

a hierarchical and dynamic component model named Fractal [8, 14]. This implementation aims at simplifying the composition, deployment, re-usability and efficiency of grid applications.

Using components, a complex Grid software can be composed of services (or sub-components). Each component has a well defined interface for accessing the service that it provides, and a well defined interface for requiring services[5]. Also, a component can be composed hierarchically of other sub components. The process of linking the component's interfaces is called binding.

From the outside, an application is viewed as a component providing a service. Once deployed and running on the grid, if the application or one of its sub components needs to be replaced or migrated (for example, because of load balancing), this can be achieved by replacing or migrating hierarchically the component.

Components are thus a very promising paradigm for Grid programming.

6. GRID EXPERIENCES

6.1. Experimentations on a large scale grid, mixing Clusters and Desktop Machines

In order to run experiments, the INRIA Sophia Desktop Grid has been deployed on the 250 desktop machines of the INRIA Sophia Antipolis laboratory; this grid is now a permanent grid managed by the P2P infrastructure (see section 3.2). All these desktop machines have heterogeneous operating systems (GNU/Linux and Windows) and CPU generations (Intel Pentium 2 to Pentium 4).

To avoid disturbing desktop users, a daemon was developed to start a JVM with a P2P Service at fixed times. The 250 desktop machines by default work during night (from 8:00 pm to 8:00 am) and during weekend (from Friday 8:00 pm to Monday 8:00 am), this group is called *INRIA-ALL*, and about 40 of those machines always work, this sub-group is called *INRIA-2424*.

In addition, of that desktop grid, we have access to a large scale national french wide infrastructure for grid research, *Grid'5000* (Grid5K). Grid5K project aims at building a highly reconfigurable, controllable and monitorable experimental grid platform gathering 9 sites geographically distributed in France, and currently featuring a total of 1700 CPUs.

The Figure 4 shows the grid used for our experimentations, this grid is a mix of INRIA Sophia Desktop Grid and Grid5K clusters. The left of the figure shows the INRIA Sophia Desktop Grid wherein INRIA-2424 peers are used as registries, all registries use themselves as registries; and at fixed moments the rest of INRIA-ALL machines join the P2P infrastructure by contacting those registries. In addition, the right part of the figure shows the Grid5K platform.

NQueens: Computation Record. With the INRIA Sophia Desktop Grid we managed, using the previously detailed P2P infrastructure, to be the first [22] to solve the NQueens counting problem for a 25 × 25 chessboard. The experimentation took six months at solving this problem instance.

The NQueens counting problem consists in placing n non attacking queens on a $n \times n$ chessboard (no two queens are on the same vertical, diagonal, or horizontal line). The problem's complexity comes from counting all the satisfying solution for a given n. The approach used to solve the NQueens problem, was to divide the global set of

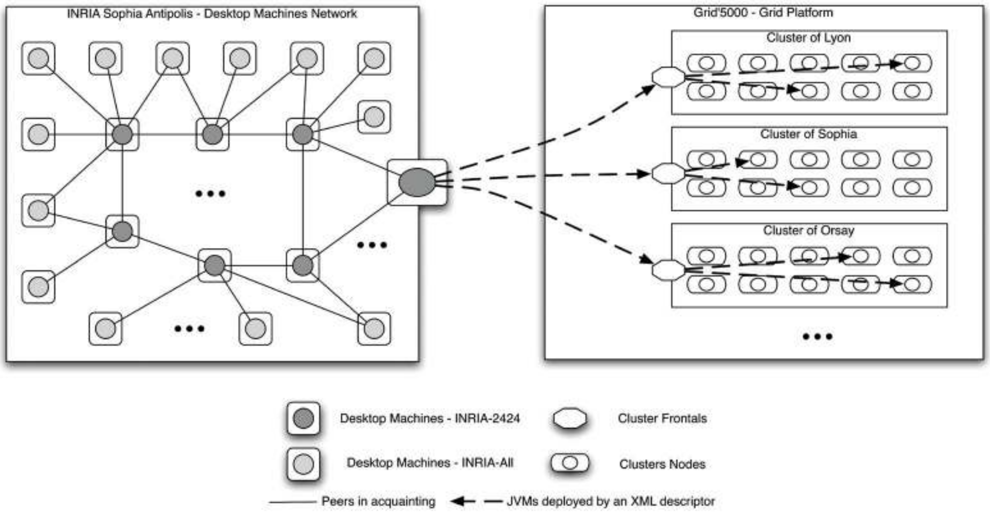
INRIA Sophia Antipolis - Desktop Machines Network
...
Grid'5000 - Grid Platform
Cluster of Lyon
Cluster of Sophia
Cluster of Orsay
...
Desktop Machines - INRIA-2424
Cluster Frontals
Desktop Machines - INRIA-All
Clusters Nodes
Peers in acquainting
JVMs deployed by an XML descriptor

Fig. 4. Environment of experimentations: Grid of desktop machines and of clusters

[5] As a matter of fact, a component has more interfaces like non functional interfaces, which are not denoted here for simplicity.

permutations into a set of independent tasks. A master-slave model was applied to distribute these tasks to the workers, which were dynamically deployed on the INRIA Sophia Desktop Grid.

The results of the NQueens experimentation are shown in Table 1.

Table 1. NQueens experimentation summary

n of NQueens	*n* = 25
Total of Solution Found	2, 207, 893, 435, 808, 352
Total of Tasks	12, 125, 199
Total of Computation Time	4 444 h (≈185 days)
Average Time of One Task Computation	2 min. and 18 sec.
One CPU Cumulated Time	464 344 h (≈53 years)
Total of Desktop Machines	250 (up to 220 working together)

The total number of solution was confirmed by Pr. Yuh-Pyng Shieh from the National Taiwan University. Using a different algorithm he found the same number of solutions [22] to place 25 queens on a 25 × 25 chessboard.

NQueens: Large Scale Grid. To experiment on a large scale grid, we took the same NQueens application, and run it on a grid. This grid is a mix of machines from INRIA Desktop Grid (INRIA-2424 and INRIA-All), and from clusters of Grid5K. Using these resources, we managed to deploy on 1007 CPUs. We chose the NQueen problem instance: *n* = 22.

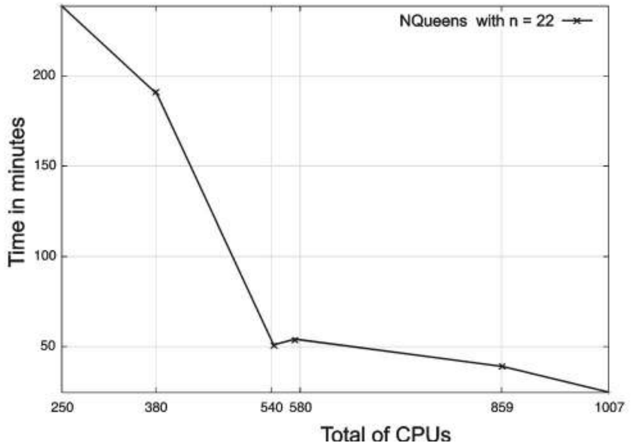

Fig. 5. NQueens with *n* = 22 benchmark results

Figure 5 shows computation time for the problem instance deployed on different number of CPUs. The speedup of the application depends on the ratio of desktop and cluster machines used. Note that, during the experimentation the availability of desktop machines varied affecting the ratio and therefore the speedup.

Flow-Shop: Communicant Application. The Flow-Shop problem aims to find a schedule of a set of jobs on a set of machines for minimizing the total execution time. We try to evaluate our P2P infrastructure using a Flow-Shop which requires communication.

The algorithm used to solve the Flow-Shop problem is not optimal, but provides some good characteristics for testing the P2P infrastructure. Firstly, we divide the solution tree of a given problem instance in a number of tasks. For finding the best solutions, we give the tasks to the workers. Our approach is based on a master-slave model. Workers share the best current solution, when a better solutions is found, the worker that finds it broadcasts it to all workers. Using this information workers can branch and bound the search tree.

Table 2 shows results from Flow-Shop computations with an instance of 17 jobs/17 machines. An analysis of the Table 2 shows that computation time decreases when the number of used CPUs increases. Note, there are two peaks, which can be explained because desktop machines are not constantly available. Machines leave and join the infrastructure, since they are desktop resources.

Table 2. Flow-Shop experimentation results
with an instance of 17 jobs and 17 machines

Max. CPUs	Computation Time	Cumulated Time
80	125.55 min.	9 603 minutes (≈160 h)
201	61.31 min.	10 676 minutes(≈178 h)
220	86.19 min.	14 396 minutes (≈240 h)
313	56.03 min.	13 257 minutes (≈220 h)
321	83.73 min.	14 628 minutes (≈243 h)
346	59.14 min.	15 036 minutes (≈250 h)

Thanks to the P2P infrastructure we successfully managed to deploy a communicant application on different sites which provided about 350 CPUs. Those CPUs were composed of heterogeneous architectures, using desktop machines and clusters.

6.2. 2nd Grid Plugtests

During the 10th-14th of October 2005 the 2nd Grid Plugtests [23] was held. Organized by ETSI and INRIA, the objectives were: to test Grid interoperability, and to learn, through the user experience and open discussion, about the future features needed for Grid middlewares.

Two Grid challenges took place with 8 participating teams. A grid was setup using the ProActive middleware, which inter-operated with several other middlewares and protocols. This grid was deployed on 13 different countries, in more than 40 sites, gathering **2700 processors** with a computing power of approximately **450 GFlops**[6]. Given the heterogeneousness of the sites, each one had to be configured and fine tuned. This involved figuring out the Architecture (*x86, ia64, x86-64, PPC, AIX, SGIIrix,* and *Sparc*) and Operating System (*Linux, MacOS, AIX, SGIIrix,* and

[6] Measured with the SciMark 2.0 benchmark.

Solaris), installing an adequate Java Virtual Machine (*Sun, IBM, Apple*, and *AIX*), figuring out the network/firewall configuration (*Firewalls*, and *NAT*), Job Scheduler (*GLite, Globus, LSF, NorduGrid ARC, OAR, PBS, PRUN, SGE, SSH,* and *Unicore*). The deployment and interoperability between all resources/sites was achieved using ProActive.

The Grid deployment was thus made very simple and transparent for the contestants, who had all the architecture details hidden by the ProActive layer. The contestants had to implement their own solutions for the challenge problems. All teams used the ProActive grid programming library.

The criterion for deciding the winners were based on: a) Greatest number of solutions found. b) Biggest number of processors used. c) Fastest algorithm.

Each team was allocated one hour of exclusive access to the Grid for computing. The first challenge was the NQueens counting problem. The Brazilian team from LSC/UFSM got ahead of the other participants. They managed to compute 2 202 billions of solutions, deployed on 1106 nodes, and solved the NQueens instance n = 21 in 13 minutes.

The second challenge was the Flow-Shop scheduling problem. The first place was awarded to the Polish Team PUTaT3AM. They computed all exact cases for FlowShop challenge for 20 jobs and 20 machines. Only this team was able to do this in less than one hour, and using 370 CPUs.

The Grid Plugtests gave us the opportunity to develop new and interesting features, while testing the middleware at a new level of complexity. The results of the NQueens and Flow-Shop challenges showed that programming in the heterogeneous Grid can be achieved using ProActive.

7. CONCLUSIONS

Using the proposed approach, which targets Grid programming at three different levels: *Grid Infrastructure*, *Grid Technical Services*, and *Grid Higher Level Programming*, we have shown it is possible to program applications for the Grid. Our experiences have shown that our approach can be used successfully on a large scale, highly heterogeneous and geographically dispersed grid.

As the grid continues to evolve, so will our proposed programming approach. We are currently concerned on *Distributed Non Functional Exception Handling* as a Grid Technical Service, where remote exceptions can be handled in distributed environments. We are also concerned with other Grid Higher Level strategies like *Skeletons*. A Skeleton can be defined as useful patterns of parallel computation and interaction that can be packaged up as "framework/second order/template" (*i.e.* parametrized by other pieces of code) constructs. The idea is to take advantage of Skeletons in order to ease the programming of Grid applications.

References

[1] L. Alvisi and K. Marzullo, *Message logging: Pessimistic, optimistic, causal, and optima*l. Software Engineering, **24(2)**, 149-159 (1998).

[2] L. Baduel, F. Baude, D. Caromel, *Efficient, Flexible, and Typed Group Communications in Java*. In Joint ACM Java Grande – ISCOPE 2002 Conference, pp. 28-36, Seattle, 2002. ACM Press. ISBN 1-58113-559-8.

[3] L. Baduel, F. Baude, D. Caromel, *Object-Oriented SPMD*. In Proceedings of Cluster Computing and Grid, Cardiff, United Kingdom, May 2005.

[4] F. Baude, D. Caromel, M. Leyton, R. Quilici, *Integrating deployment and file transfer tools for the grid*. In Preliminary Proceedings 1st Coregrid Integration Workshop (IW'05), Pisa Italy, pp. 457-466 (2005).

[5] F. Baude, D. Caromel, L. Mestre, F. Huet, J. Vayssière, *Interactive and descriptor-based deployment of object-oriented grid applications*. In Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing, pp. 93-102, Edinburgh, Scotland, July 2002. IEEE Computer Society.

[6] F. Baude, D. Caromel, Ch. Delb, L. Henrio, *A hybrid message logging-cic protocol for constrained checkpointability*. In Proceedings of EuroPar 2005, 3648 in LNCS, pp. 644-653, Lisbon, Portugal, August- September 2005. Springer.

[7] F. Baude, D. Caromel, M. Morel, *From distributed objects to hierarchical grid components*. In International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily, Italy, 3-7 November, pp. 1226-1242, Springer Verlag, 2003, Lecture Notes in Computer Science, LNCS.

[8] E. Bruneton, T. Coupaye, J. Stefani, *Recursive and dynamic software composition with sharing*, 2002.

[9] J. Bustos, D. Caromel, A. Di Costanzo, M. Leyton, J. Piquer, *Balancing active objects on a peer to peer infrastructure.* In Proceedings of XXV International Conference of SCCC, Valdivia, Chile. IEEE CS Press, November 2005.

[10] J. Bustos, D. Caromel, M. Leyton, J. Piquer, *Load information sharing policies in communication-intensive parallel applications.* In Proc. of Sixth IEEE International Symposium and School on Advance Distributed Systems (ISSADS 2006), Guadalajara, Mexico, Springer LNCS Series, 2006. To appear.

[11] D. Caromel, *Toward a method of object-oriented concurrent programming*. Communications of the ACM, **36(9)**, 90-102, 1993.

[12] K. M. Chandy and L. Lamport, *Distributed snapshots: Determining global states of distributed systems*. In ACM Transactions on Computer Systems, pp. 63-75 (1985).

[13] M. Elnozahy, L. Alvisi, Y.M. Wang, D.B. Johnson. *A survey of rollbackrecovery protocols in message passing systems.* Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, oct 1996.

[14] Fractal. http://fractal.objectweb.org.

[15] Gnutella. http://www.gnutella.com.

[16] D. Manivannan and M. Singhal, *Quasi-synchronous checkpointing: Models, characterization, and classification*. In IEEE Transactions on Parallel and Distributed Systems, **10**, 703-713 (1999).

[17] M. Mitzenmacher, *How useful is old information?* IEEE Transactions on Parallel and Distributed Systems, **11(1)**, 6-34 (2000).

[18] NorduGrid. http://www.nordugrid.org.

[19] R. Schollmeier, *A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications*. In Peer-to-Peer Computing, pp. 101-102 (2001).

[20] E. Seidel, G. Allen, A. Merzky, J. Nabrzyski, *Gridlab: A grid application toolkit and testbed.* Future Generation Computer Systems, **18**, 1143-1153 (2002).

[21] N. G. Shivaratri, P. Krueger, M. Singhal, *Load distributing for locally distributed systems*. IEEE Computer, **25(12)**, 33-44 (1992).

[22] Neil J. Sloane. Sloane a000170. http://www.research.att.com/projects/OEIS?Anum=A000170

[23] OASIS Team and ETSI. 2nd grid plugtests report. Technical report, INRIA, 2005. http://www-sop.inria.fr/oasis/plugtest2005/2ndGridPlugtestsReport.pdf.

[24] Unicore. http://www.unicore.org.

[25] R. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. F. H. Hofman, C. J. H. Jacobs, Th. Kielmann, H. E. Bal, *Ibis: a flexible and efficient java-based grid programming environment*. Concurrency – Practice and Experience, **17(7-8)**, 1079-1107 (2005).

Denis Caromel is full professor at University of Nice-Sophia Antipolis and CNRS-INRIA. He is also member of the Institut Universitaire de France (IUF), a multi-disciplinary national academia that select a few professors based on the excellence of their research records. His research interests include parallel, concurrent, and distributed object-oriented programming. He was an invited visiting scientist at various universities and research institutions (including Digital System Research Center in Palo Alto, NASA Langley Research Center in Hampton, Virginia, and IBM Tom Watson). He has published more than 70 scientific papers in referred international journals and conferences, and edited 5 volumes of Lecture Notes. In 2005 he published a monograph, A Theory of Distributed Objects.

Christian Delbe is graduated at the University of Nice Sophia Antipolis in 2003 in Networking and Distributed Systems and is currently PhD student at the INRIA-CNRS-University of Nice Sophia Antipolis. His research interests involve parallel, concurrent, and distributed object-oriented programming. His PhD deals with fault-tolerance for distributed systems, particularity in the context of Grid Computing.

Alexandre di Costanzo graduated at the University of Nice Sophia Antipolis in 2004 in Networking and distributed Systems and is currently PhD student at the INRIA-CNRS-Universty of Nice Sophia Antipolis. His main research interests are in parallel, distributed, and grid computing. In particular, in the field of Peer to Peer Computing: Model and Infrastructure for Communicating Applications. He was a visiting scientist for three months at the Computer Science Department of Indiana University.

Mario Leyton received his Computer Science Engineer degree from University of Chile in 2004. Currently he is a PhD student at University of Nice-Sophia Antipolis and INRIA. His main research interest are in parallel/distributed computing. In particular, the field of structured parallel programming models.