Efficient, Flexible, and Typed Group Communications in Java

Laurent Baduel, Françoise Baude, Denis Caromel

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

ABSTRACT

Group communication is a crucial feature for high-performance and Grid computing. While previous works and libraries proposed such a characteristic (e.g. MPI, or object-oriented frameworks), the use of groups imposed specific constraints on programmers – for instance the use of dedicated interfaces to trigger group communications.

We aim at a more flexible mechanism. More specifically, this paper proposes a scheme where, given a Java class, one can initiate group communications using the standard public methods of the class together with the classical dot notation; in that way, group communications remains typed. Furthermore, groups are automatically constructed to handle the result of collective operations, providing an elegant and effective way to program gather operations.
This flexibility also allows to handle results that are groups of remotely accessible objects, and to use a group as a means to dispatch different parameters to different group members (for instance in a cyclic manner). Futhermore, hierarchical groups can be easily and dynamically constructed; an important feature to achieve the use of several clusters in Grid computing.

Performance measures demonstrate the viability of the approach. The challenge is to provide easy to use, efficient, and dynamic group management for objects dynamically distributed on the Grid.

Categories and Subject Descriptors

D.3.2 [Java] : Library, Distributed Computing

General Terms

Languages, Experimentation, Design, Performance, Measurement

Keywords

Group Communications, Active Objects, Hierarchical and Dynamic Groups

1. INTRODUCTION

Programming high-performance applications requires the definition and the coordination of parallel activities. Hence, a library for parallel programming should provide not only point-to-point but collective communication primitives on groups of activities.

In the Java world, the RMI [12] mechanism is the standard point-to-point communication mechanism, and it is adequate mainly for client-server interactions, via synchronous remote method call. In a high-performance computing context, asynchronous and collective communications should be accessible to programmers, so the usage of RMI is not sufficient.

We have developed a 100% Java library, *ProActive* (`www.inria.fr/oasis/ProActive`), for parallel, distributed, concurrent computing with security and mobility. RMI is currently used as the transport layer. Besides remote method invocation services, *ProActive* features transparent remote active objects, asynchronous two-way communications with transparent futures, high-level synchronisation mechanisms, migration of active objects with pending calls and an automatic localisation mechanism to maintain connectivity for both "requests" and "replies".

This paper presents the design of a method invocation mechanism on groups of active objects and its implementation in the framework of the *ProActive* library. Alternate approaches for parallel and distributed computing in Java include in the use of more dedicated parallel programming frameworks, such as parallel and distributed collections [7] which hide the presence of parallel processes, or in implementing MPI-like libraries in an SPMD programming style [10]. Our group mechanism is more general, as it enables to build such alternate parallel programming models, while being able to provide group communication to distributed applications originally not aimed at being parallel, thus enabling code reuse. For instance, an active object is able to execute a remote method invocation on multiple active objects at once, without their active involvement. By comparison, in MPI, a collective operation is executed by having all processes in the group call the communication routine, with matching arguments.

The work presented in [9] is the closest to ours: the objectives and the approach are quite similar. It will be discussed in section 2.2. But, as further explained, we significantly advance its capabilities with more flexibility and dynamicity.

This paper is organized as follows: after a background and related work part, the principles and design of the typed group communication mechanism are presented. The implementation is sketched in section 4, and some performance measurements of the basic mechanism are provided and analysed. The section 4 ends up with the implementation of a real example.

2. BACKGROUND AND RELATED WORK

2.1 Distribution and Mobility 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, pre-processor or modified virtual machine. The model of distribution and activity of *ProActive* is part of a larger effort to improve simplicity and reuse in the programming of distributed and concurrent object systems [3, 4], including a precise semantics [1].

2.1.1 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.

2.1.2 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 a 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 instructions 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. In order to help in the deployment phase of *ProActive* components, the concept of virtual nodes as entities for mapping active objects has been introduced [2]. Those virtual nodes are described externally through XML-based descriptors which are then read by the runtime when needed.

2.2 Related work

The aim of group communication mechanism presented in [9] is to generalize all kind of communications (point-to-point or collective, synchronous or asynchronous, local or remote). As so many different communication modes are available, it requires some effort from the programmer in order to choose the desired communication mode. The main difference in the mechanism we present here is that the group communication mechanism is an additional and smoothly integrated mechanism, built around an already existing rich underlying framework for point-to-point communications. Thus, programmers can benefit at the same time from all kind of communication patterns in a flexible way and without additional work. For instance, here is a code example from [9]:

```
class SumImpl extends GroupMember implements Sum{...};
   // On one place, for instance, on the group
   // member whose rank is 0:
   // Creation of a group with name "Name"
   // N is the number of expected members
Group.create("Name", N);

   // On every member of the group:
   // (1) create a group member
SumImpl sum = new SumImpl();

   // (2) Enroll this object as a member of the
   //     group with name "Name"
   //     join blocks until N members have joined
Group.join("Name", sum);

   // On one place, for instance, on the group
   // member whose rank is 0:
   // (1) gain access to the group as a whole
   //     via a stub:
Sum stub = (Sum) sum.createGroupStub();

   // (2) choose the required communication mode
   //     for each method to call in the program:
Group.setInvoke(
            stub, "void add(double v)", Group.GROUP);
   // If method with signature above does not
   // exists, it raises an exception at run-time
   // and the communication mode is not changed
...

   // (3) triggers a method call towards each
   //     member of the group:
   // each member adds the value 42.0 to its own value
stub.add(42.0);
```

In our approach, thanks to reification and meta-object protocol techniques, it is never required, as in [9], to pass the signature of the remote method as a parameter of a group communication related instruction.

In the code above, the requirement to pass

```
      "void add(double v)"
```

[1]Java RMI [12], the Reflection API [11],...

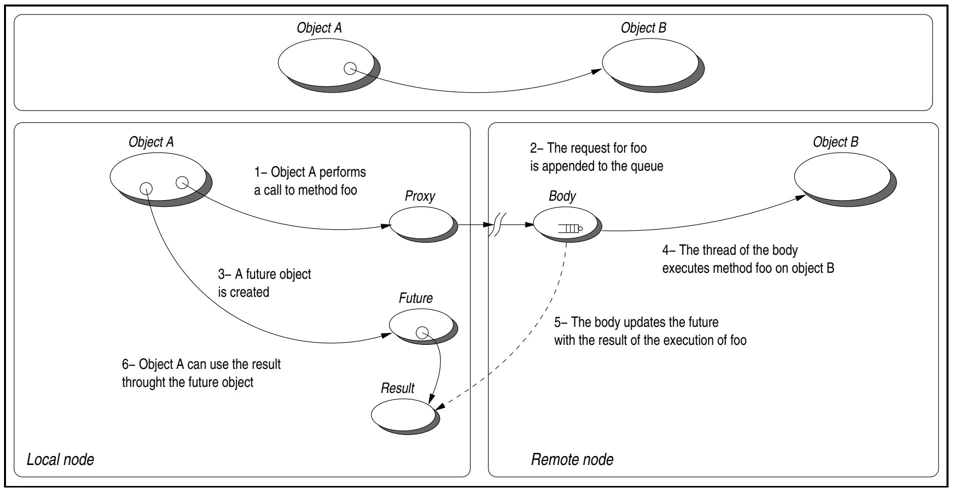

Figure 1: Execution of an asynchronous and remote method call

as a parameter of `Group.setInvoke()` is to set the invocation mechanism of this method `add` to be GROUP (by default, it would have been set to local method call mode; an alternative would have been to set it to be REMOTE, such as to get the standard RMI semantics). If for instance another method might be called on the group members, for instance, `"double get()"`, the programmer should set the corresponding communication mode as follows:
`Group.setInvoke(stub, "double get()" , Group.GROUP);`
The communication mode towards a group of objects must be set on a per method basis. On the contrary, in our group mechanism, as soon as an active object gets enrolled in a group, all the public methods of group members might be invoked via a group communication. Moreover, the methods can still be called on each member, via a standard point-to-point remote communication mode.

Like in [9], our mechanism provides *typed group communication*, *typed* in the sense that only methods defined on classes or interfaces implemented by members of the group can be called. This is enforced at compile-time.

3. TYPED GROUP COMMUNICATIONS

3.1 Principles

Our group communication mechanism is built upon the *ProActive* elementary mechanism for asynchronous remote method invocation with automatic future for collecting a reply. As this last mechanism is implemented using standard Java, such as RMI, the group mechanism is itself platform independant and 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, our mechanism is a generalization of the remote method call mechanism of *ProActive*, built upon RMI, but nothing prevents from using other transport layers in the future.

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: the class of an object that we want to make active, and thus remotely accessible, is reified at the meta level, at runtime. In a transparent way, method calls towards such an active object are executed through a stub which is type compatible with the original object. The stub's role is to enable to consider and manage the call as a first class entity and applies to it the required semantics: if it is a call towards one single remote active object, then the standard asynchronous remote method invocation of *ProActive* is applied; if the call is towards a group of objects, then the semantics of group communications is applied. The rest of the section will define this semantics.

3.2 Group creation

Groups are created using the static method
`ProActiveGroup.newActiveGroup(‘‘ClassName’’,...)`
The superclass common for all the group members has to be specified, thus giving the group a minimal type. Groups can be created empty and existing active objects can be added later as described below. Non-empty groups can be built at once using two additional parameters: a list of parameters required by the constructors of the members and a list of nodes where to map those members. In that case the group is created and new active objects are constructed using the list parameters and are immediately included in the group. The n^{th} active object is created with the n^{th} parameter on the n^{th} node. If the list of parameters is longer than the list of nodes (i.e. we want to create more active objects than the number of available nodes), active objects are created and mapped in a round-robin fashion on the available nodes.

Let us take a standard Java class:

```
class A {
  public A() {}
  public void foo (...) {...}
  public B bar (...) {...}
  ...
}
```

Here are examples of some group creation operations:

```
   // Pre-construction of some parameters:
   //   For constructors:
Object[][] params = {{...} , {...} , ... };
   //   Nodes to identify JVMs to map objects
Node[] nodes  = { ... , ..., ... };

   // Solution 1:
   // create an empty group of type "A"
A ag1 = (A) ProActiveGroup.newActiveGroup("A");

   // Solution 2:
   // a group of type "A" and its members are
   // created at once,
   // with parameters specified  in params,
   // and on the nodes specified in nodes
A ag2 = (A) ProActiveGroup.newActiveGroup(
                                   "A", params, nodes);

   // Solution 3:
   // a group of type "A"  and its members are
   // created at once,
   // with parameters specified in params,
   // and on the nodes directly specified
A ag3 = (A) ProActiveGroup.newActiveGroup(
                       "A", params[],
                       {rmi://globus1.inria.fr/Node1,
                        rmi://globus2.inria.fr/Node2});
```

Elements can be included into a typed group only if their class equals or extends the class specified at the group creation: the classes of all the members of a group have a common ancestor. Note that we do allow and handle *polymorphic* groups. For example, an object of class B (B extending A) can be included to a group of type A. However based on Java typing, only the methods defined in the class A can be invoked on the group.

The main limitation of the group construction is that the specified class of the group has to be *reifiable*, according to the constraints imposed by the Meta-Object Protocol of *ProActive*: the type has to be neither a primitive type (`int`, `double`, `boolean`,...), nor a final class, in which cases, the MOP would not be able to create a typed group object. However, those constraints are easy to explain, to identify, and to check.

3.3 Group representation and manipulation

The typed group representation we have presented in the preceding subsection corresponds to the functional view of groups of objects. In order to provide a dynamic management of groups, a second and complementary representation of a group has been designed. In order to manage a group, this second representation must be used instead. This second representation follows a more standard pattern for grouping objects: the interface `Group` extends the Java `Collection` interface which provides management methods like `add`, `remove`, `size`, ... Those group management methods feature a simple and classical semantics (add in group, remove the nth element, ...) which provides a ranking order property of elements of a group.

The management methods for a group are not available on the *typed group representation*, but instead, on the *group representation*. It is a design choice among two possibilities: one that would have consisted in using static methods of the `ProActiveGroup` class in order to manage groups, and as such, yielding to just one representation of a group. The other consists in associating to a group two complementary representations, one for functional use only, the other for management purposes only. At the implementation level, we are careful to have a strong coherence between both representations of the same group, which implies that modifications executed through one representation are immediately reported on the other one. In order to switch from one representation to the other, two methods have been defined (see figure 2): the static method of the `ProActiveGroup` class, named `getGroup`, returns the Group form associated to the given group object; the method `getGroupByType` defined in the `Group` interface does the opposite.

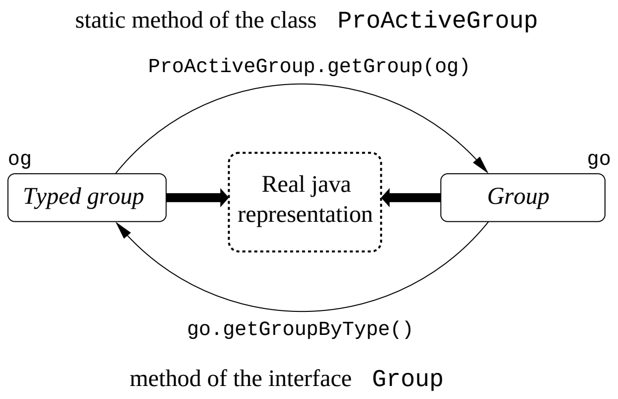
static method of the class ProActiveGroup
ProActiveGroup.getGroup(og)
og
go
Typed group
Real java representation
Group
go.getGroupByType()
method of the interface Group

Figure 2: Typed group and Group representations

Below is an example of when and how to use each representation of a group:

```
   // definition of one standard Java object
   // and two active objects
A a1 = new A();
A a2 = (A) ProActive.newActive("A", paramsA[], node);
B b  = (B) ProActive.newActive("B", paramsB[], node);
   // Note that B extends A

   // For management purposes, get the representation
   // as a group given a typed group, created with
   // code on the left column:
Group gA = ProActiveGroup.getGroup(ag1);

   // Now, add objects to the group:
   // Note that active and non-active objects
   // may be mixed in groups
gA.add(a1);
gA.add(a2);
gA.add(b);

   // The addition of members to a group immediately
   // immediately reflects on the typed group form,
   // so a method can be invoked on the typed group
   // and will reach all its current members
ag1.foo(); // the caller of ag1.foo() may not belong to ag1

   // A new reference to the typed group
   // can also be built as follows
A ag1new = (A) gA.getGroupByType();
```

Notice that groups do not necessarily contain only active objects, but may contain standard Java objects as members.

The only restriction is that they be type compatible with the class of the group. We will see in 3.4 the implication of such heterogenous groups on the management of communications towards group elements.

```
public interface Group extends Collection {
...
void add (Object o)
        //Add an element into the group.
void addMerge (Object ogroup)
        //Merge a group into the group.
Object getByType ()
        //Return an object representing the group under the typed form.
Class getType ()
        // Return the (upper) class of member.
int indexOf ()
        //Return the index in the group of the first occurence of the
        //specified element. (−1 if the list does not contain this element).
iterator iterator ()
        //Return an Iterator on the members in the group
void remove (int index)
        //Remove the object at the specified index.
int size ()
        //Return the number of members
... }
```

Figure 3: The Group Interface

3.4 Group communications

A method invocation on a group has a similar syntax to a standard method invocation:

```
Object[][] constructorArray = {{...},{...},...};
Node[] nodes = {...,...,... };
A ag1 = (A) ProActiveGroup.newActiveGroup(
                    "A", constructorArray, nodes);
...
ag1.foo(...); // A group communication
```

Of course, such a call has a different semantics which is as follows: the call is asynchronously propagated to all members of the group using multithreading. Like in the *ProActive* basic model, a method call on a group is non-blocking and provides a transparent future object to collect the results. A method call on a group yields a method call on each of the group members. If a member is a *ProActive* active object, the method call will be a *ProActive* call and if the member is a standard Java object, the method call will be a standard Java method call (within the same JVM).

The parameters of the invoked method are broadcasted to all the members of the group. As described in 3.6, another semantics is available in order to scatter the parameters to the group members instead of broadcasting them.

3.5 Group as result of group communications

The particularity of our group communication mechanism is that the *result* of a typed group communication *is also a group*. The result group is transparently built at invocation time, with a future for each elementary reply. It will be dynamically updated with the incoming results, thus gathering results. Nevertheless, the result group can be immediately used to execute another method call[2], even if all the results are not available. In that case the *wait-by-necessity* mechanism implemented by *ProActive* is used: if all replies are awaited, then, the future enables to block the caller, but as soon as one reply arrives in the result group, then the method call on this result is executed. In the code below, a new `f1()` method call is automatically triggered as soon as a reply from the call `vg = ag1.bar()` comes back in the group `vg`:

```
   // A method call on a group, returning a result
V vg = ag1.bar();
   // vg is a typed group of "V": operation
   // below is also a collective operation
   // triggered on results
vg.f1();
```

The instruction `vg.f1()` completes when `f1()` has been called on all members.

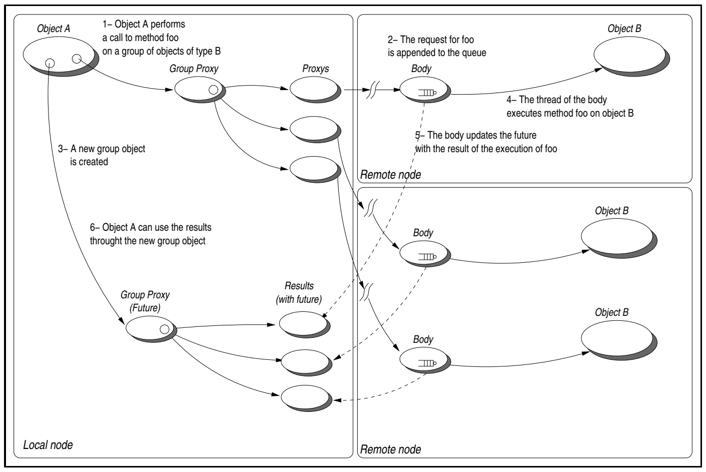
Object A
1– Object A performs a call to method foo on a group of objects of type B
Group Proxy
Proxys
2– The request for foo is appended to the queue
Body
Object B
4– The thread of the body executes method foo on object B
5– The body updates the future with the result of the execution of foo
3– A new group object is created
Remote node
6– Object A can use the results throught the new group object
Body
Object B
Group Proxy (Future)
Results (with future)
Body
Object B
Local node
Remote node

Figure 4: Execution of a remote method call on a group

The ranking order of elements in a group is a property that is kept through a method invocation: the nth member of a result group (i.e., of `vg`) corresponds to the result of the method executed by the nth member in the calling group (i.e., of `ag1`). We will see later in section 3.7, that another property is maintained between the group onto which the call is performed and the group of corresponding results: hierarchy.

As explained in 3.2, groups whose type is based on final classes or primitive types cannot be built. So, the construction of a dynamic group as a result of a group method call is also limited. Consequently, only methods whose return type is either void or is a 'reifiable type', in the sense of the Meta Object Protocol of *ProActive* (see above), may be called on a group of objects; otherwise, they will raise an exception at run-time, because the transparent construction of a group of futures of non-reifiable types fails.

To take advantage with the asynchronous remote method call model of *ProActive*, some new synchronisation mechanisms have been added. Static methods defined in the `ProActiveGroup` class enable to execute various forms of

[2]This call will be either a standard call or a *ProActive* remote call, depending of the real type of results

synchronisation. For instance: `waitOne`, `waitN`, `waitAll`, `waitTheNth`, `waitAndGet`, ... For instance:

```
   // 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);
```

3.6 Dispatching parameters using groups

Regarding the parameters of a method call towards a group of objects, the default behaviour is to broadcast them to all members. But sometimes, only a specific portion of the parameters, usually dependent of the rank of the member in the group, may be really useful for the method execution, and so, parts of the parameter transmissions are useless. In other words, in some cases, there is a need to transmit different parameters to the various members.

A common way to achieve the scattering of a global parameter is to use the rank of each member of the group, in order to select the appropriate part that it should get in order to execute the method. There is a natural traduction of this idea inside our group communication mechanism:

the use of a *group of objects* in order to represent
a parameter of a group method call that must
be scattered to its members.

A *one to one* correspondence between the n^{th} member of the parameters group and the n^{th} member of the group is obtained by the ranking property already mentioned in 3.5.

Like any other object, a group of parameters of type P can be passed instead of a single parameter of type P specified for a given method call. The default behaviour regarding parameters passing for method call on a group, is to pass a deep copy of the group of type P to all members [3]. Thus, in order to scatter this group of elements of type P instead, the programmer must apply the static method `setScatterGroup` of the `ProActiveGroup` class to the group. In order to switch back to the default behaviour, the static method `unsetScatterGroup` is available.

```
   // Broadcast the group gb to all the members
   // of the group ag1:
ag1.foo(gb);

   // Change the distribution mode of the
   // parameter group:
ProActiveGroup.setScatterGroup(gb);

   // Scatter the members of gb onto the
   // members of ag1:
ag1.foo(gb);
```

Notice that, should the parameter group be bigger than the target group, the excess members of the parameter group will be ignored. Conversely, should the target group be larger than the size of the parameter group, then the members of the parameter group will be reused (i.e. sent more than once) in a round-robin (cyclic) fashion.

Note that this parameter dispatching mechanism is in many ways a very flexible one. It provides:

- automatic sending of a group to all members of a group (default),
- the possibility to scatter groups in a cyclic manner (`setScatterGroup`),
- the possibility to mix non-group, group, cyclic-scatter group as arguments in a given call.

All of this is achieved without any modification to the method signature.

3.7 Hierarchical groups

In order to be able to build large applications, the concept of *hierarchical group* is available:

a group of objects that is built as *a group of groups*.

This mechanism helps in the structuration of the application and makes it more scalable. A hierarchical group is easily built by just adding group references to a group:

```
   // Two groups
A ag1 = (A) ProActiveGroup.newActiveGroup("A",...);
A ag2 = (A) ProActiveGroup.newActiveGroup("A",...);

   // Get the group representation
Group gA = ProActiveGroup.getGroup(ag1);
   // Then, add the group ag2 into ag1:
gA.add(ag2);
```

As seen previously, a group of results reflects a group of method calls (i.e. the n^{th} member of the result group corresponds to the result of the method executed by the n^{th} member in the group). A similar correspondance exists for hierarchical groups: the n^{th} member of the result group will be in fact a future to a group of results that corresponds to the group method call executed by the n^{th} member in the calling group.

Note that one can merge two groups, rather than add them in a hierarchical way. This is provided through the member `addMerge` of the `Group` interface (see Figure 3). For instance, the instruction

```
   gA.addMerge(ag3);
```

merges a group (by first flattening it) into an existing one.

4. IMPLEMENTATION, EXAMPLE AND BENCHMARKS

4.1 Principles and basic performances

A stub is generated by the MOP of *ProActive* in order to locally represent a typed group (this is the same stub as generated for the representation of an active object). The stub is dynamically built, extending the class of the objects that can be included as members, thus yielding a *typed* group. Thanks to inheritance and polymorphism, the stub object acts as representative of the group and is polymorphically compatible with the typed group (and also with a single active object of this type).

The stub is connected to a *proxy for group*, programmed in the *ProActive* implementation. The proxy for group stores one reference to each group member. The role of the proxy

[3]If the members of the group of type P are in fact active objects of type P, then only copies of the stubs are done. Indeed, the group collecting such members does not effectively contain a copy of those active objects, but only references to them.

for group is to transmit the method call to each of the members. This is done with multithreading, so as to introduce asynchronism and communication overlap for the multiple method calls. Figure 4 gives the details of the implementation, except that the stub for a B in figure 1 or for a group of B in figure 4 are not represented (the field of the object "A" pointing to an object of type "B", does not actually store a reference to a "B", but a reference to a stub for type "B").

But, as a proxy for group contains copies of proxies for each of its elements, then, it might be the case that several proxies for the same group replicated on different virtual machines (i.e., on *ProActive* nodes) be incoherent. Indeed, should a member dynamically join a group, then, the local proxy for this group would be updated, but other copies of that might exist elsewhere would not be automatically updated. Nevertheless, it is possible to extend the basic group communication mechanism, in such a way as to maintain coherent representations of groups. A consequence of this design choice is that it is not necessary to have a centralized server that stores group representations and that would be asked to propagate method calls to members of a given group. In this way, our mechanism is very scalable.

We now present some performance measurements for group creation and communication. The experiments are run onto a 100 Mb/s Ethernet local area network connecting Pentium III PCs under Linux with Sun JVMs version 1.3. Each experiment is run 1,000 times, and the curves on figures 5 and 6 plots average durations.

Figure 5 shows the durations required to create remote active objects, for which no constructor parameter is required. The number of active objects, members of the group varies between 1 and 200. For measurement purposes, after a group creation call, the caller is blocked until all members have been created and have joined the group. The creation duration measures this elapsed time (shown as *...with guarantee of creation...*). Those remote objects are created in a cyclic manner on 10, 20 or 30 computers on the same local area network. When the number of computers increases, then, the number of group members to create on each decreases, so the total creation duration also proportionally decreases. As one thread is dedicated to the creation of each member of the group, then the group creation at the caller side almost immediately returns (see curve *with groups, with asynchronous creation (10 hosts)*): the object which initiated the group creation thereby is able to resume its job while effective group member creations are executed on remote *ProActive* nodes.

Figure 6 presents the durations for a group method call, depending on the number of members in the groups compared with standard *ProActive* method call towards active objects. The experiment *with guarantee of delivery* consists of one group method call, followed by a barrier synchronization of the object that has initiated the group method call. Recall that there is a short rendez-vous at the beginning of each asynchronous remote call (see section 2.1) in *ProActive*. Then, this experiment measures the total duration of the concurrent and remote executions of rendez-vous triggered by the group method call. As we use a thread for each call in a group communication, then, the execution of a method call on a group of N remote active objects is more efficient than triggering successively N *ProActive* remote method calls, one for each remote active object (see curve *without group*).

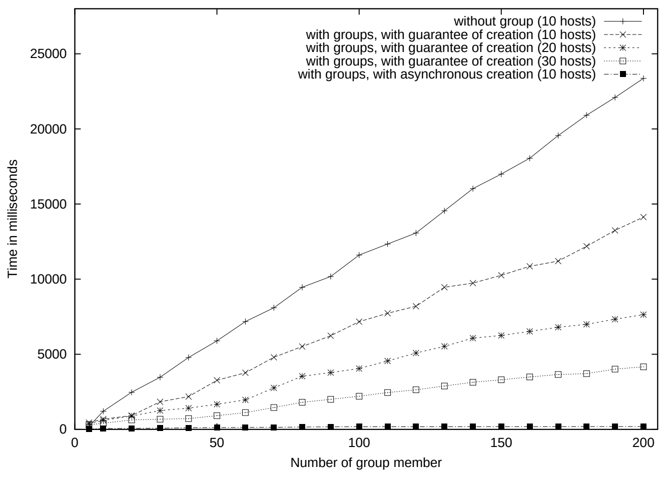

Figure 5: Performances of various group creation benchmarks

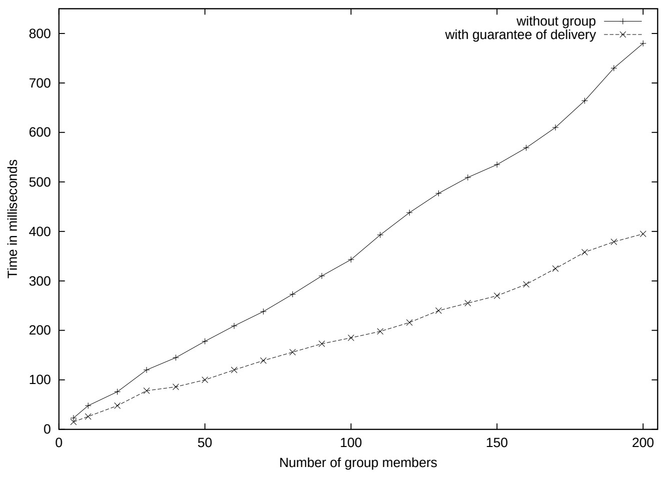

Figure 6: Performances of group communication. Group members are mapped onto 10 remote hosts

4.2 A standard example

To validate the design and implementation of group communication, we have programmed a basic numerical application pertaining to a parallel dense matrix multiplication. We have choosen the algorithm based on the *Broadcast-Broadcast Approach* described in [8]. This algorithm pertains to our work as it extensively uses collective communications. As our group communication features some asynchronism, we foresee performances improvements compared to the same algorithm implemented without using the group mechanism but only point-to-point *ProActive* method calls.

Like most of the algorithms for parallel dense matrix multiplication, the *Broadcast-Broadcast Approach* algorithm performs a multiplication of the form C = αAB + βC on a two dimensional logical process grid with P rows and Q columns. In this demonstration we consider only the case where P=Q.

Once the distribution is done, sub-matrices of the two matrices to multiply are located on each computer which takes part in the computation. The *Broadcast-Broadcast Approach* algorithm consists in four steps:

1. Broadcast the sub-matrices of A along the rows.
2. Broadcast the sub-matrices of B along the columns.
3. Update partial C sub-matrices with A and B sub-matrices multiplication in each process.
4. Repeat Step 1 through Step 3 P times.

At the end of those steps, the sub-matrices of C contain the result of the A*B multiplication.

It is obvious that each process of the logical grid will be represented by one active object, whose class represents a sub-matrix.

The active objects of each row (resp. column) of the logical grid build up one group. Broadcast communication of sub-matrices along one row (resp. one column) will be achieved thanks to the group method call mechanism. Here is an implementation of the algorithm :

```
// The method multiply is a basic centralized matrix
// multiplication; it updates the right sub-matrix
// of C.

// row[i] and column[i] return the i-th row and i-th
// column of the logical grid, in a typed group form.

// The distributed matrix multiply method
// implementation :
for (int i=0 ; i<P ; i++)
   A.row[i].multiply (B.column[i]);
```

The mechanism of group communication provides a simpler implementation. With just two lines of code, we replace about twenty lines of pseudo-instructions seen in [8].

Figure 7 shows the time spent in order to compute the matrix multiplication depending of size of one side of square matrix. Two implementations of the algorithm are tested. One uses the *ProActive* library without group communication mechanism, the other uses the group communication mechanism. Experimentations were done using either one (see curve *centralized*) or nine Intel Pentium3 933MHz on the same local area network as above (see curves *distributed* and *distributed using groups*).

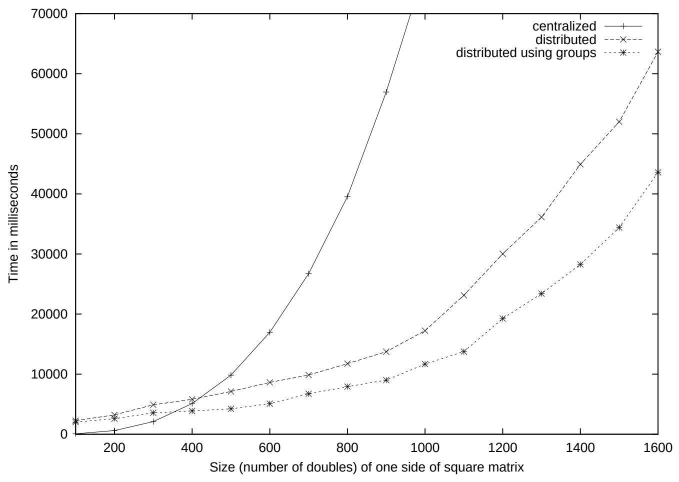

Figure 7: *Broadcast-Broadcast* performances

Again, this experiment proves that the implementation using groups is more efficient than the one without. We identify two reasons for this. The first point is that multi-threading in the multiple *ProActive* individual method calls covers the serialization phases of RMI, which is used at the transport layer. A second point is that some meta-level operations, like the method call reification, are done just once within the group communication implementation.

5. CONCLUSION AND PERSPECTIVES

Group communication is a crucial feature for high-performance and Grid computing, for which MPI is generally the only available coordination model. We try to show now how collective communications offered by our mechanism can be compared to that offered by MPI.

For the most important ones, such as broadcast, scatter, gather, our mechanism provides equivalent group communication patterns:

- *broadcast* in a group is simply achieved through a group method call with the information to broadcast as parameter;
- *scatter* is achieved in the same way, apart from passing the scatter group as parameter (see 3.6) to the group method call;
- *gather* is achieved within a group of results of a group method call (see 3.5).

The *reduce* collective operation may be executed by the group method caller, just after all results have been gathered (or, even as soon as they arrive). Reduction is carried on sequentially within one active object, the caller. Thus, we do not exploit the opportunity of parallel reduction. But, in a coarse-grained view of parallelism as used here, parallel reduction is not as important as for fine-grained parallelism.

We do not provide the most sophisticated but rarely used collective operations of MPI, such as all-gather, all-to-all, all-reduce, reduce-scatter, scan. Even libraries such as CCJ for instance [10], aiming at providing communications in Java inspired by MPI (as opposed to direct bindings to native MPI libaries, like mpiJava [5] or Java-centric MPI [6]) do not provide all those variants. But, as those operations can be simply implemented by using those from the set "broadcast, gather, scatter, reduce", the motivation to effectively define them in the library seems to be primarily for performance optimization purposes.

Notice that the group communication does not directly support barrier synchronization (i.e. a global rendez-vous) as in MPI. Again, it is not so important for coarse-grained parallelism. Nevertheless, it can be programmed in standard *ProActive*. But, we have a group coordination feature that allows a similar effect in which we can guarantee that all group members have executed the method call and have returned their result (using the `WaitAll` primitive presented in 3.5).

A point to emphasize is that members of the group, except the one acting as the root (in the sense of MPI), do not need to explicitly participate in the group communication: the method called on the group is executed on each active object member of the group, by an automatic and FIFO (by default) service of pending requests, inherent to *ProActive*. The effective execution of a collective communication is thus really more asynchronous, according to an

MIMD model, than within an SPMD one as featured by MPI. This loose synchronization may have positive effects on performances, especially on heterogeneous and loosely coupled environments such as Grids. For instance, a root object can even launch a group method call, and, assuming this call returns a group of active objects, the root can immediately trigger the next group method call. Due to the rendez-vous of the basic *ProActive* method invocation model, there is a guarantee that both method calls will be received in the same order on every member of the group.

As our *ProActive* group communication model is more asynchronous than the one found in SPMD models such as MPI, we allow more than one group method call execution on the same group, triggered by different objects, to interleave. Indeed, as method call results may not be immediately used (due to the future semantics), then our asynchronous global communication model is less subject to deadlocks.

The current implementation optimizes only the network latency by overlapping point-to-point communications. We have noticed that the implementation of broadcast, scatter and gather operations could take advantage of a structuration, as a binomial tree for instance, of the communications among members of a group. As the meta-level of *ProActive* is structured in such a way as to dynamically enable to adapt the way requests and replies are sent or received, we plan to use this opportunity to adapt proxies for groups such as to structure collective communications. Another alternative we are considering, especially valuable for networks of workstations, would be the use of a multicast transport layer. Notice that the implementation of group communication on hierarchical groups naturally takes advantage of the underlying hierarchical structure. If a member is itself a group, for instance gathering a group of active objects mapped on a remote cluster, then, the proxy representing this member will be sent only one method call that it will then recursively propagate to its peers.
As a group method call is currently implemented by an RMI call towards each member of the group, we are currently working on another optimization that should prove to be effective in the Java context: achieving a unique RMI serialization for all identical objects in a group communication.

In summary, this paper has presented a group mechanism which we believe can be very effective for component-based parallel programming on Grids, with the following features:

- groups of objects need only to be defined at instanciation time, without any change elsewhere in the program;
- factories to build groups are provided;
- code for group management (“non-functional code”) is, by virtue of one group representation, kept separate from “functional” code, which has only to consider the typed group representation.

Acknowledgments:
This work was supported by the ACI Grid-RMI project, funded by the French government.

6. REFERENCES

[1] I. Attali, D. Caromel, and R. Guider. A step towards automatic distribution of java programs. In *FMOODS 2000, Stanford University, September 6-8, 2000*, pages 141–161. Kluwer Academic.

[2] F. Baude, D. Caromel, F. Huet, L. Mestre, and J. Vayssière. Interactive and Descriptor-based Deployment of Object-Oriented Grid Applications. In *11th IEEE International Symposium on High Performance Distributed Computing*, 2002. To appear.

[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] B. Carpenter, G. Fox, S. Ko, and S.Lim. mpiJava 1.2: API Specification. `http://www.npac.syr.edu/projects/pcrc/mpiJava/mpiJava.html`, October 1999.

[6] B. Carpenter, V. Getov, G. Judd, and A. Skjellum. MPJ: MPI-like message passing for Java. *Concurrency, Practice and Experience*, 12(11):1019–1038, 2000.

[7] V. Felea and B. Toursel. Methodology for Java Distributed and Parallel Programming Using Distributed Collections. In *Int. Workshop on Java for Parallel and Distributed Computing (IPDPS 2002).*

[8] J. Li, A. Skjellum, and R. D. Falgout. A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies. *Concurrency: Practice and Experience*, 9(5):345–389, 1997.

[9] J. Maassen, T. Kielmann, and H. Bal. Generalizing Java RMI to support efficient group Communication. In *ACM Java Grande Conference*, 2000.

[10] A. Nelisse, T. Kielmann, H. Bal, and J. Maassen. Object-based Collective Communication in Java. In *Joint ACM Java Grande - ISCOPE 2001 Conference.*

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

[12] Sun Microsystems. Java Remote Method Invocation Specification, Oct. 1998. `ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf`.