Effective and Efficient Communication in Grid Computing with an Extension of ProActive Groups

Laurent Baduel Francoise Baude
OASIS –INRIA
University of Nice Sophia Antipolis – France
{lbaduel, fbaude}@sophia.inria.fr

Nadia Ranaldo Eugenio Zimeo
RCOST -Department of Engineering
University of Sannio – Benevento – Italy
{ranaldo, zimeo}@unisannio.it

Abstract

Grid applications typically deal with huge amount of data and often the same data have to be transferred and processed on many resources. Nevertheless, the majority of existing middleware platforms for Grid computing do not provide suitable programming and communication models to make easy software development and to improve communication performances when a large set of receivers is involved. Some middlewares for wide area network computing, such as ProActive, provide the group abstraction to transparently deal with a number of similar receivers. We propose an extension of such a mechanism in order to improve its features for Grid environments. In particular, ProActive native groups have been extended both at programming and communication levels in order to support both different internal behaviors and high performance communication based on IP multicast. A case study shows the effectiveness of the new mechanism and its efficiency compared with the original one.

1. Introduction

Many Grid applications (such as simulations applied to scientific and engineering fields, or data acquisition and analysis from distributed measurement instrumentations and sensors) deal with intensive computations and management of huge amount of data and often the same data have to be transferred and processed on multiple resources in order to improve the performance.

In recent years, many Grid middleware platforms and toolkits have been developed (Globus [1], Legionz [2], Unicore [3], Condor-G [4], HiMM [5], etc.). These middleware platforms, typically, adopt unicast communication mechanisms implemented atop unicast reliable protocols. However, Grid systems could strongly benefit in many applications of a one-to-many or many-to-many communication mechanisms [6] [7].

Providing a middleware for Grid computing with an effective and efficient implementation of the *group abstraction* at programming level could ease software development and reduce the communication overhead both in a small scale and in a large scale.

According to the Object Group design pattern [8], a group is a local surrogate for a group of objects distributed across networked machines to which can be assigned the execution of a task. The object group pattern specifies that when a method is invoked on a group, the runtime system sends the method invocation request to the group members, waits for one or more member-replies on the basis of a policy, and returns the result back to the client. Groups are usually dynamic, i.e. the set of group members can continuously change.

At programming level, groups can ease software development since they simplify the implementation of some high-level computing models, such as master-slave, master-worker, pipeline and work-stealing.

At communication level, groups can reduce the communication overhead for several reasons. First, the delivery of the same content to a collection of receivers can benefit of the group abstraction since specific optimizations can be applied even if the underlying transport layer is based on unicast communication [9]. For instance, the network transfer of objects requires serialization before sending them. Since serialization takes a significant processing time, sending the same object to the members of the group is easily improved if the same serialized copy of the object is used for a unicast transfer towards each member. Second, group communication can be implemented (only for some internal behaviors) through its mapping on a multicast transport layer. In this case, differently from real-time multimedia distributed systems, which tolerate unreliable data streaming to reduce latency, Grid systems often require reliable multicast protocols to deliver replicated application data without losses and errors.

These considerations have motivated an intense research activity which has led to many protocol definitions for implementing reliability in multicast communication. The paper in [10] proposes an interesting

solution integrated in a Java framework, JRMS (Java Reliable Multicast Service) [11], that provides several reliable multicast protocols.

In this paper, an extension of the groups provided by ProActive [12] is proposed. The ProActive API was modified in order to (1) support the definition of specific semantics for groups; (2) dynamically install defined semantics in running groups; (3) establish a mapping with a transport layer; (4) select IP multicasting when the group semantics require members to be clones and methods arguments to be delivered to all replicas. The paper presents also the integration of a transport layer based on IP multicast with ProActive and discusses, through the implementation of a case study, the benefits that the above extensions introduce in some computing models.

The rest of the paper is organized as follows. Section 2 describes the typed groups provided by ProActive and the current limitations as concerns the internal behavior and communication layer. Section 3 introduces some possible semantics for groups. Section 4 proposes a new ProActive API and the integration of a specific transport layer based on IP multicast. Section 5 discusses a case study, which is an application based on the master/slave computing model implemented through ProActive groups.

2. ProActive Groups

The basic unit of activity and distribution used by ProActive to build concurrent applications is the *Active Object*. An active object is remotely created on a host involved in the computation. Methods calls sent to active objects are always asynchronous with transparent future objects and the synchronization is handled by a mechanism known as wait-by-necessity [13].

In addition to simple active objects, ProActive offers a group communication mechanism that allows for method invocations on sets of active objects, grouped together and referenced by a single collective name. A ProActive group is also called **typed group** since it is composed of objects belonging to classes inheriting from the same superclass or implementing the same interface. Typed group is the "clonation" of an active object on a set of nodes and a group communication is the "replication" of a remote method invocation on them. Each member can be an instance of a different class but all the members must have the same ancestor.

While many libraries and programming frameworks delivering group abstraction impose specific constraints on programmers, thanks to the use of a Meta-Object Protocol (MOP) [14], ProActive delivers a more transparent and flexible mechanism. ProActive MOP, through the reification of method invocation and constructor call, makes it possible to initiate group communication invoking a method of the group object. As a consequence a typed group takes exactly the same form as using only one active object. When a method call is invoked towards a group, the semantics of communications are implemented on an asynchronous underlying communication system which internally handles execution requests as sequences of events related to request transmissions, request dispatching, failure notifications, result collecting, etc. Such communication system asynchronously and efficiently propagates the call to all members of the group using multithreading. A method call on a group is asynchronous and provides a transparent future object to collect the results.

Currently, ProActive groups provide the programmer with some mechanisms for the management of input parameters, such as broadcasting and scattering. By adopting the broadcasting, the same parameter is sent to all the members. On the other hand, by adopting the scattering, a part of the overall parameter is transferred to the members. In this case, the parameter has to be explicitly passed as a group, which is built splitting the original parameter in several parts. The default behavior is the broadcasting, while in order to scatter a parameter the programmer has to invoke the static method `setScatterGroup` of the ProActiveGroup class to the input parameter group. So, the scatter policy is tied only to a specific input parameter instance.

Some synchronization policies can also be adopted to block the caller when a return parameter is used. The limitation of this approach is that synchronization policies can be associated to the returned group but not to the group instance which invokes the method. The result of a typed group communication is also a group, requiring so an explicit management of its group members when an aggregation policy has to be adopted. The result will be dynamically updated with the incoming partial results. Thanks to the wait-by-necessity synchronization mechanism, a result can be immediately used to execute a method call, even if all the results are not available.

In order to simplify distributed programming, more abstractions and high-level distributed models should be delivered by a group communication mechanism at programming level, in order to free the programmer from the implementation details of system aspects of programming such as object distribution, mapping and load balancing mechanisms. This leads also to a performance improvement, thanks to the possibility to automatically and transparently adapt the application to the system configuration.

We propose to extend the syntax of group creation and to change the syntax and semantics of group management. To this end, we introduce a dynamic internal behavior, called Group Behavior, for each ProActive group, so as to define the semantics adopted by the group for a method invocation. Through the definition of a behavior and its dynamic assignment to a group, this

one can change its internal behavior at run-time and new policies can be easily implemented and attached without interventions on the library or even on the application code. In fact, through the Java reflection, a newly created group behavior can be loaded during the program execution to install a different behavior in a running group. This way, a group can transparently adapt its behavior to the context in which it operates.

3. Group semantics and communication

In recent years, several group semantics have been defined. Each of them contributes to specify the behavior of a group. In particular, from the point of view of the method invocation the following semantics can be individuated:

- **Request mapping:** it handles the mapping of each request to the group members. Some examples are (1) ***One***, the request is assigned to only one group member, selected with a scheduling policy (for example random, round-robin, more sophisticated policies based on QoS) [15]; (2) ***Fixed***, the request is scheduled for a defined number of group members; (3***)*** ***All***, the request is propagated to all the group members.
- **Input parameters distribution:** it allows for splitting the input parameter of each group method before sending the request to the group members selected for the request mapping. Examples are: (1) ***Broadcast***, an input parameter of the method invocation is sent to all the scheduled group members; (2) ***Scatter***, a group that receives the invocation of a method could be able to split the value, received as parameter, in a number of chunks and to pass each one to the same method of each member.
- **Output parameters collection:** it handles the return value replied to the caller. Examples are (1) ***Gather***, the output parameter is obtained collecting the partial results of the group members; (2) ***Merging***, the output parameter is obtained by assembling the partial results of the group members.
- **Synchronization:** it specifies the condition that blocks the caller when a return parameter of a group method invocation is used. (1) ***All***, the totality of the scheduled group members execute the request, and all the results are to be collected and returned to the caller; (2) ***Majority***, the execution request is active until the majority of the scheduled group members have executed the request and replied the results; (3) ***One***, in this case, groups can be used to improve the reactivity related to the processing triggered by a method of the group by moving the invocation to all the scheduled members and collecting the result coming from the more reactive or nearer member; (4) ***Fixed***, a number of executions specified by the user are required.

From the point of view of communication inside a group, the following schemes can be adopted:

- **Unicast**, a point-to-point communication. In this case each member is contacted separately in order to receive different input data.
- **Multicast**, a point-to-multi-point communication. In this case the group is subdivided in two or more subgroups and, for each one, input data are delivered to all the members.
- **Broadcast**, all the members receive the same input.

Communication semantics have to be selected according to the behavior chosen for the group. For example the multicast semantic is adopted when a request execution is sent to a part of the group members and the input parameters are sent with the broadcast semantic, etc., whereas the unicast semantic is adopted for a request execution when an input parameter is scattered and each part has to be sent to a different group member.

For each one of the semantics reported above, a **reliable** or **unreliable** schema can be adopted, depending on the selected semantics of the group.

Some group semantics for the creation phase can also be individuated. Examples are the policy for the selection of host nodes on which allocate the group nodes, the management of each constructor parameter of the group members and also the semantic that determines the condition of success of a group creation. In this paper only the method invocation semantics are analyzed, whereas those related to the group creation phase are currently under study.

4. Extended ProActive Groups

To ensure flexibility and extensibility the configuration and customization of a behavior for a group is obtained through **GroupBehavior**. Such class specifies the behavior of a group in response to the method invocation request and is the composition of the four semantics defined above. Each semantic has a default implementation and can be modified at run-time.

A semantic is associated to an instance of one of the following interfaces:

- *RequestMappingSemantic*
- *InputDistributionSemantic*
- *SinchronizationSemantic*
- *OutputCollectionSemantic*

Each interface has some methods that have to be implemented to define a specific semantic. Such methods are invoked by a component of the framework, called **GroupBehaviorEnactor**.

RequestMappingSemantic The implementation of the method:
`Vector getMembers(MethodCall mc, Vector memberList)` specifies the group members at which the

request has to be sent. It receives an instance of the MethodCall class, which contains information on the current method invocation on the group (opportunely captured at run-time by the MOP), in particular on the method signature and the effective arguments. The other input parameter is a list of the current group members.

InputDistributionSemantic The implementation of the method **`Vector manageInputs(MethodCall mc, Vector memberList, Communicator comm)`** specifies how the input parameters have to be distributed to the group members for a method invocation request. It receives the `MethodCall` instance which represents the

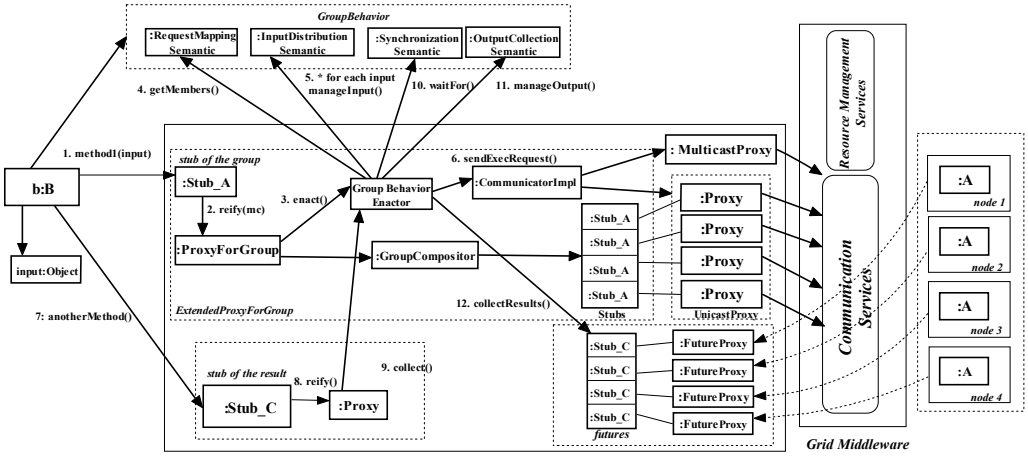

Figure 1. Extended Proactive Group organization

current method invocation request, the list of the group members chosen for the request execution by the `RequestMappingSemantic`. The last input parameter represents a component responsible for the implementation of the logical communication semantic to use for data transmission inside a group. Such class has the method **`setLogicalCommunication(String commSchema, Parameters qos)`** which permits a user to configure a logical communication semantic for a method execution request. The method uses a string representing a communication schema supported by the middleware and some parameters of QoS which have to be satisfied. **Parameters** is a class that contains instances of **Parameter**, which is a couple of attribute-value. Currently, we consider only a parameter, which represents the reliability level defined by the attribute "**reliability"** and can assume the values "**reliable"** and "**unreliable"**. The communication schemas currently supported by the middleware are: "unicast", "multicast" and "broadcast".

From the programming point of view the possibility to specify the logical communication semantic inside a group is delivered without any awareness on the leveraged transport layers supported by the physical networks. For example, although unicast group communication could be implemented by employing a unicast transport protocol such as TCP or UDP, multicast and broadcast group communication could be implemented both by using a unicast transport protocol and a multicast one, depending on the availability of the underlying transport layers.

Finally the return parameter is a vector which contains the result of the distribution semantic applied to each effective argument, obtained by the MethodCall instance, corresponding to the input parameter identified by its index in the parameter list.

SynchronizationSemantic Through the implementation of the method:
`void waitFor(MethodCall mc, Vector futures)`
it is possible to specify the synchronization policy when a result of a group method invocation is used for another method call. Such method is invoked on a vector of future objects, each of which is associated to the asynchronous call on a group member scheduled for the execution.

This method can be easily implemented leveraging the static methods of ProActive related to the synchronization on a future object or a vector of future objects.

OutputCollectionSemantic determines how to reply to the caller the final return parameter of a group method invocation through the following method:
`Object manageOutput(MethodCall mc, Vector futures)`
It receives an instance of `MethodCall` and a vector of future objects, containing the stubs of the results of the group members scheduled for the request execution.

The extension of the `ProActive` group requires only few modification to the syntax of the current version. In particular the client application creates an instance of a group specifying the `GroupBehavior` to apply. As a

consequence, the static methods of the ProActiveGroup class have been modified to include this parameter. See the example reported below that shows the creation of an empty group of class A. The code shows also how it is possible to build an object of `GroupBehavior` class specifying the instances of the semantic objects to adopt, and as it is possible to change at run-time one or more of them.

```
public class A{
  public A(){}
  public C method1(Object input) {...}
  public C method2(Object input) {...} }

public class AllScheduler implements
RequestMappingSemantic{
 public Vector getMembers(Vector memberList, MethodCall
mc) {    return memberList; } }

public class ScatterSemantic
                 implements InputDistributionSemantic{
public Vector manageInput(MethodCall mc,
               Vector memberList, Communicator comm) {
  Vector inputs = new Vector();
  for (int i=0; i<mc.getNumberOfParameter(); i++){
   Vector parts=new Vector();
   Object par = mc.getParameter(i);
   if (par.getClass().isArray()) { //default scatter
    Object[] o = (Object[])par;
    Class c = par.getClass().getComponentType();
    Object part = null ;
    int size = memberList.size();
    int elemNum = o.length/size;
    for (int k=0 ; k< size ; k++) {
       part = Array.newInstance(c,elemNum);
       for (int j=0;j< Array.getLength(part); j++)
         Array.set(part, j, o[(k*elemNum)+j]);
       parts.add(k,part);
    }
   } else if . . . .
   inputs.add(i, parts); }
 Parameters pars= new Parameters();
 Parameter p =new Parameter("reliability", "reliable");
 pars.addParameter(p);
 comm.setLogicalCommunication("unicast", pars);
 return inputs;   } }

public class MyInputSemantic extends ScatterSemantic {
Vector manageInputs(MethodCall mc, Vector memberList
Communicator comm)
{ Vector inputs = new Vector();
  if (mc.getName().equals("method1")) {
   //method1 with scatter semantic
    inputs = super.manageInputs(mc, memberList, comm);
  } else if (mc.getName().equals("method2")){
   . . .// method2: broadcast sem. and multicast comm.
  }
  return inputs; } }

public class OutputAssembler implements
OutputCollectionSemantic {
  // return par. is replied assembling partial results
  Object manageOutput(MethodCall mc, Vector futures){
   . . . } }

public class AllSynchronizator implements
SyncronizationSemantic {
  // synchr. on all parzial results
  public void waitFor(MethodCall mc, Vector futures){
    ProActive.waitForAll(futures);
  } }

public class Main{
 public static void main(String[] s){
  Node [] nodes = …;
  RequestMappingSemantic r = new AllScheduler();
  InputDistributionSemantic in = new ScatterSemantic();
  SynchronizationSemantic s = new AllSynchronizator();
  OutputCollectionSemantic out = new OutputAssembler();
  GroupBehavior beh = new GroupBehavior(r,in,s,out);
  A a = (A) ProActiveGroup.newGroup("A", null, nodes,
beh);
  . . . // creation of the parameter input
  C c = a.method1(input);
  in  = new MyInputSemantic();
  beh.setInputDistributionSemantic(in);
  Group g = ProActiveGroup.getGroup(a);
  g.setBehavior(beh);
  c = a.method2(input);
  c.anotherMethod(); } }
```

Group creation is performed through the method **`newGroup`** which specifies the group class, the constructor parameters, the nodes and the group behavior. In the current implementation, the broadcast semantic, and the multicast communication schema are adopted for the constructor parameters. The reference created by the `newGroup` method is an instance of the class A, and more precisely, an instance of Stub_A, that is a subclass of A automatically and dynamically built by the MOP.

Thanks to the reification, the semantic related to the management of method invocations on groups can be intercepted and customized at run-time in order to logically show a specific behavior. In particular, when a method is invoked on a group instance, the MOP mechanism is enacted to start the reification of the method call (see fig. 1): (1) an object of MethodCall class is built and passed by the group stub to the group proxy to execute the method `reify(MethodCall)`; (2) the group proxy invokes the method `enact` of the Group Behavior Enactor; (3) the Group Behavior Enactor perform the execution request adopting the semantics specified in the GroupBehavior object received as parameter.

When a result is used for the invocation of a method the following steps are executed: (1) the proxy of the result invokes the method `collect` of the Group Behavior Enactor; (2) the Group Behavior Enactor executes the synchronization semantic and builds the final result adopting the semantics specified by the GroupBehavior object received as parameter.

From the point of view of the communication inside a group, some improvements can be made. The idea is to perform the data transmission leveraging the potentialities of the network connections effectively available at the moment. For example some network information can be used in order to adopt, when it is possible, as an alternative to the commonly used unicast transport communication based on TCP/IP, a transport layer based on multicast protocols.

ProActive is particularly suitable to implement such a mechanism, thanks to its high modularity and customization mechanisms related to the mapping of logical application data communication to the real services available at transport level for data transmission on physical networks.

Our solution is based on the definition of a new ProActive component, the **Communicator**, which has the main task to manage the data transmission inside a group for each method execution request. Such component is the only component to be aware of the communication services delivered by the physical networks and so to be able to map the logical communication semantic onto an available transport layer, that is the most suitable one.

For unicast communication, the Communicator can access to the Proxy, one for each member scheduled for the request execution, which is able to handle the transmission on the network of a request adopting one among the available unicast transport layers. For multicast communication, the Communicator can access to the MulticastProxy, a completely new component, able to handle the transmission of a request adopting a multicast mechanism.

The default middleware in ProActive adopted for group communications, that is based on RMI, limits the possibility to improve the performances of an application written using the group communication mechanism. In fact RMI is currently implemented on TCP which requires a group method invocation to be implemented as the sequential invocation of a remote method call on each active object. For this aspect, we propose an implementation of ProActive groups atop a transport layer based on IP multicast.

Integrating reliable multicast inside a middleware for Grid computing is still an open issue. Some solutions aim at easily porting existing applications to multi-destination environments by enriching TCP with multicast capabilities [6]. To efficiently exploit a multicast protocol, the Grid computing middleware should be able to manage the sub-parts of the Grid infrastructure in which the multicast communication is supported at data-link or network layers. This is the case of a cluster in which the resources are connected through a common LAN which supports broadcast communication at data-link layer, or a set of workstations directly connected to an IP multicast-enabled router.

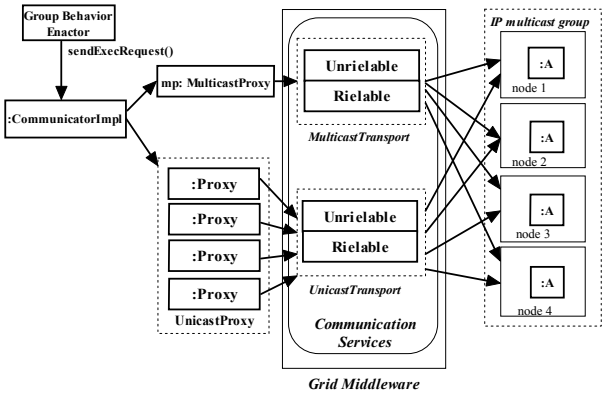

Figure 2. Logical communication mapping on real transport layers

Unreliable multicast typically provide scalability up to tens of thousands of nodes, but its semantics are generally too weak for application developers to depend upon. Messages are subject to long and unpredictable transmission delays, message loss, and out of order delivery. Processes may crash and network links may fail; such failures are hard to detect when the communication delays are unpredictable and messages can be lost. To avoid this, we can use a reliable multicast service to integrate into the transport layer of ProActive to ensure that a message from a correct process reaches all the correct participants.

5. A case study: the implementation of the master/slave model

The master-slave pattern [16] for distributed programming was implemented to test our proposal. Two implementations of this programming model are shown and compared, the first one adopts the native group mechanism and the second one adopts the extended group mechanism. A slave object is implemented by means of a group member, each of which is opportunely distributed onto remote machines.

The four semantics described above have to be specialized in order to define the specific behavior of a group of objects which has to implicitly implement such programming model.

RequestMappingSemantic. The request execution has to be sent to all the slaves, in order to leverage all the computational power of the distributed resources on which them are instanced, so the already defined **AllScheduler** class can be used to set such semantic.

InputDistributionSemantic. For the master/slave pattern it has to be defined how the initial workload is divided among the slaves. The decomposition mechanism on input parameters is delivered by the already defined class **ScatterSemantic**.

SynchronizationSemantic. The result of a master/slave computation is obtained combining all the results of the slaves, so the synchronization policy has to be a wait for all the results, already implemented by the class **AllSynchronizator.**

OutputCollectionSemantic**.** The reconstruction of the final result from the partial results of the slaves is defined by the default output collection semantic defined by the already defined class **OutputAssembler**.

In the following, the class MSGroupBehavior is shown. Such class represents the group behavior for the

master/slave pattern, built adopting the semantics defined above.

```
public class MSGroupBehavior extends GroupBehavior {
 public MSGroupBehavior() {
  RequestMappingSemantic r = new AllScheduler();
  InputDistributionSemantic in = new ScatterSemantic();
  SynchronizationSemantic s = new AllSynchronizator();
  OutputCollectionSemantic out = new OutputAssembler();
  super(r,in,s,out);  } }
```

The canonical matrix multiplication is used as case study. Class **`Matrix`** is used to represent the abstract data-type *matrix*, and delivers the methods necessary to perform the row-for-column multiplication. In particular Matrix delivers a constructor **`Matrix (float[][] m)`** where the parameter m is a two-dimensional array of float, and the method **`Matrix multiply (float[][] a)`**, that performs the multiplication algorithm where the current instance represents the right matrix and the matrix passed as parameter the left matrix. Such matrix has to be split in equivalent sub-parts, using a row-based decomposition, each of that has to be sent to a different group member that represents a slave object. On the other hand, the constructor parameter will be the overall right matrix, which so will be the same for each of them.

5.1. Native vs Extended Groups: code writing

The implementation adopting the native Proactive group mechanism is the following:

```
public class Main1 {
 public static void main (String args[]) {
  Matrix mDxGroup, mSxGroup result;
  Node[] nodes = null; // nodes list for slaves
  float[][] a, b;
  // def. of the left mat. b and right mat. a
  int totalRows = b.length;
  Object[] po = new Object[1]= {a};
  mDxGroup = (Matrix)
       ProActiveGroup.newGroup("Matrix", po, nodes);
  Object[] parts = createSubMatrices(b, nodes.length);
  Object[][] pars = new Object[nodeList.length][];
    for (int i=0 ; i < nodeList.length ; i++) {
      po = new Object[1] {parts[i]};
      pars[i] = po; }
    mSxGroup =
(Matrix)ProActiveGroup.newGroup("Matrix", pars, nodes);
    ProActiveGroup.setScatter(mSxGroup);
    Matrix gResult = mDxGroup.multiply(mSxGroup);
    Matrix result = reconstruction(gResult, totalRows);
  } }

public Object[] createSubMatrices(float[][] m, int n){
    Object[] parts = new Object[n];
    int widthSubMatrix = m.length / n;
    for (int i=0 ; i < n ; i++) {
      float[][] d = new float[widthSubMatrix][];
      for (int j=0 ; j < d.length ; j++)
      d[j] = m[(i*widthSubMatrix)+j];
      parts[I]=d;
    }
    return parts; }

public Matrix reconstruction(Matrix group, int rows) {
  int index = 0;
  Matrix partial = null;
  int size =  ProActiveGroup.size(group);
  float[][] d = new float[rows][];
  for (int i=0 ; i < size ; i++) {
    partial = ((Matrix)(ProActiveGroup.get(group,i)));
    int  widthTmp = partial.getWidth();
    for (int j=0 ; j < widthTmp ; j++) {
     d[index] = partial.getRow(j); index++;
    } }
  return new Matrix(d); }
```

As it is possible to note, it is necessary to define a ProActive group which will used to perform the distributed multiplication. The tasks related to the master/slave pattern implementation, are explicitly provided by the programmer. Such tasks are essentially those performed by the master object, that are the configuration of the input parameter for each group member and the collection of the results and their assembling in order to deliver the final result matrix to the user. In particular, in order to perform the multiplication algorithm on a part of the left matrix, this has to be explicitly split, and the obtained sub-parts have to be used to build a group of Matrix objects to use with the scatter semantic. The result parameter is a group of Matrix objects, so an assembling algorithm has to be written in order to extract each group member and to merge the partial matrices in the final one. Such implementation of the master/slave pattern requires so to the programmer many tasks, and the group mechanism doesn't permit to effectively simplify the distributed programming.

The implementation adopting the extended Proactive group mechanism is the following:

```
public class Main2 {
  public static void main (String args[]) {
    Matrix mDxGroup, result;
    Node[] nodesList = null;// nodes list for slaves
    float[][] a, b;// … def.left mat. B, right mat. a
    GroupBehavior msbeh = new MSGroupBehavior();
    Object[] po = new Object[1] {a};
    mDx = (Matrix) ProActiveGroup.newGroup("Matrix",
      po, nodesList, msbeh);
    result = mDx.multiply(b);
   . . . // use of the result matrix  } }
```

It is possible to note as the distributed programming is really simplified adopting the extended group mechanism and the group behavior for the master/slave pattern, thanks to the fact that it maintains completely transparent to the programmer the details of its implementation.

5.2 Native vs Extended Groups: performance evaluation

A performance evaluation of the proposed approach was conducted by comparing the performances obtained with two different implementations of the case-study. An implementation was based on the original groups and unicast communication, the other one was based on extended ProActive groups and multicast communication. For the first case, the default ProActive implementation based on Java RMI was adopted, while for the second one

a prototypical version of the ProActive group mechanism was implemented and a reliable multicast protocol (TRAM), included in JRMS 1.1, was used.
The testbed was a cluster of eight nodes, each one equipped with Intel Pentium II 350 Mhz, 128 MB of RAM and 10/100 Mbps network card, and a machine equipped with Intel Pentium IV 2.4 Ghz, 256 MB of RAM and 10/100 Mbps network card.

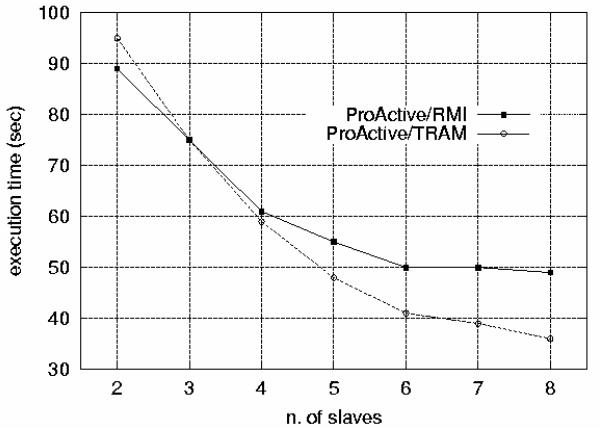

Fig. 3 Performance evaluation

Fig. 3 shows the execution times of the application matrix multiply, considering a fixed matrix dimension to 1000x1000 float number, and a varying number of slaves. As it is possible to note, the implementation based on reliable multicast exhibits better performances, mainly due to the reduced traffic on the network. The adopted implementation in fact employs multicast communication for the creation group members, each of which receives the right matrix with the broadcast semantic, so strongly reducing the network utilization compared to repeated unicast communication adopted by the original ProActive groups.

6. Conclusions

The paper presented and discussed an extension of ProActive groups in order to improve their features for Grid environments. The extension regards both the programming interface and communication. This way, programmers can define specific semantics for groups and dynamically install them in each group instance. These semantics specify the internal behaviour of a group and create a binding with a desired communication layer and protocol.
A case study shows both the flexibility of the proposed approach and the improvement of performance when IP multicast can be adopted for communication.
In the future, configurable semantics will be associated also to the creation phase of each group and resource management services provided by low-level Grid middlewares will be used to define dynamic mappings between group members and computational resources.

References

[1] The Globus Project. http://www.globus.org/.

[2] A. Natrajan, A. Nguyen-Tuong, M.A. Humphrey et al, "The Legion Grid Portal", *Concurrency and Computation: Practice and Experience*, 2002, 14(13-15), pp. 1365-1394.

[3] Unicore. http://unicore.sourceforge.net/.

[4] J. Frey, T. Tannenbaum, I. Foster, M. Livny, S. Tuecke, "Condor-G: A Computation Management Agent for Multi-Institutional Grids", *in proc. of the Tenth IEEE Symposium on HPDC*, California, August, 2001.

[5] M. Di Santo, N. Ranaldo, E. Zimeo, "A Broker Architecture for Object-Oriented Master/Slave computing in a Hierarchical Grid System", *Parallel Computing*, Germany, Sept 2003.

[6] K. Jeacle, J. Crowcroft, "Reliable High-speed Grid Data Delivery Using IP Multicast", *in proc. of e-Science All Hands Meeting*, Hottingham (UK), September, 2003.

[7] M. Miamour, C. Pham, "An Active Reliable Multicast Framework fro the Grids", *in proc. of the ICCS 2002*, April 2002, Amsterdam, The Nederlands, pp. 588-597.

[8] S. Maffeis, "The Object Group Design Pattern", *in proc. of the Second USENIX Conference on Object-Oriented Technologies*, Toronto, Canada, 1996.

[9] Laurent Baduel, Francoise Baude, Denis Caromel, "Efficient, Flexible, and Typed Group Communication in Java". *JGI'02*, November 3-5, USA, 2002.

[10] D. M. Chiu, S. Hurst, M. Kadansky, J. Wesley, "TRAM: A Tree-based Reliable Multicast Protocol", *Sun Microsystems Laboratories*, *SML TR-98-66*, July, 1998.

[11] S. Hanna, M. Kadansky, P. Rosenzweig, "Java Reliable Multicast Service Overview", *Sun Microsystems Laboratories, SMLI TR-98-68*, September 1998.

[12] D. Caromel, W. Klauser, J. Vayssiere, "Towards Seamless Computing and Metacomputing in Java", *Concurrency: Pract &Exp.*, 1998, 10(11-13), pp.1043-1061.

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

[14] G. Kiczales, J. des Rivires, and D. G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

[15] Andrew A. Chien and William J. Dally, Concurrent Aggregates (CA) , *in proc. of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, 1990.

[16] F. Bushmann et al., *Pattern-Oriented Software Architecture: A System of Patterns*. J. Wiley and Sons, 1996.