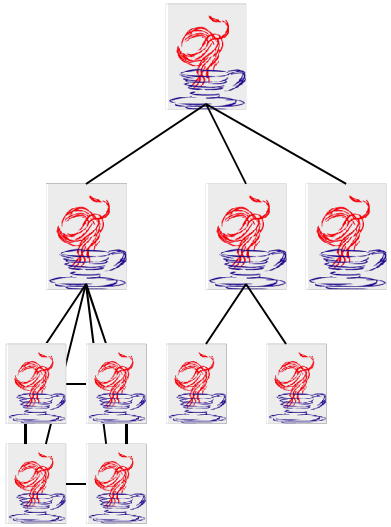


Grid computing with an extension of ProActive Groups

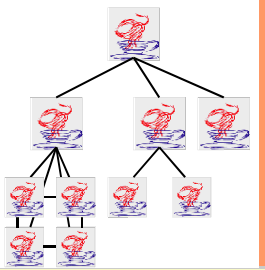
E. Zimeo, N. Ranaldo

RCOST- University of Sannio **– Benevento – Italy**

{zimeo, ranaldo}@unisannio.it

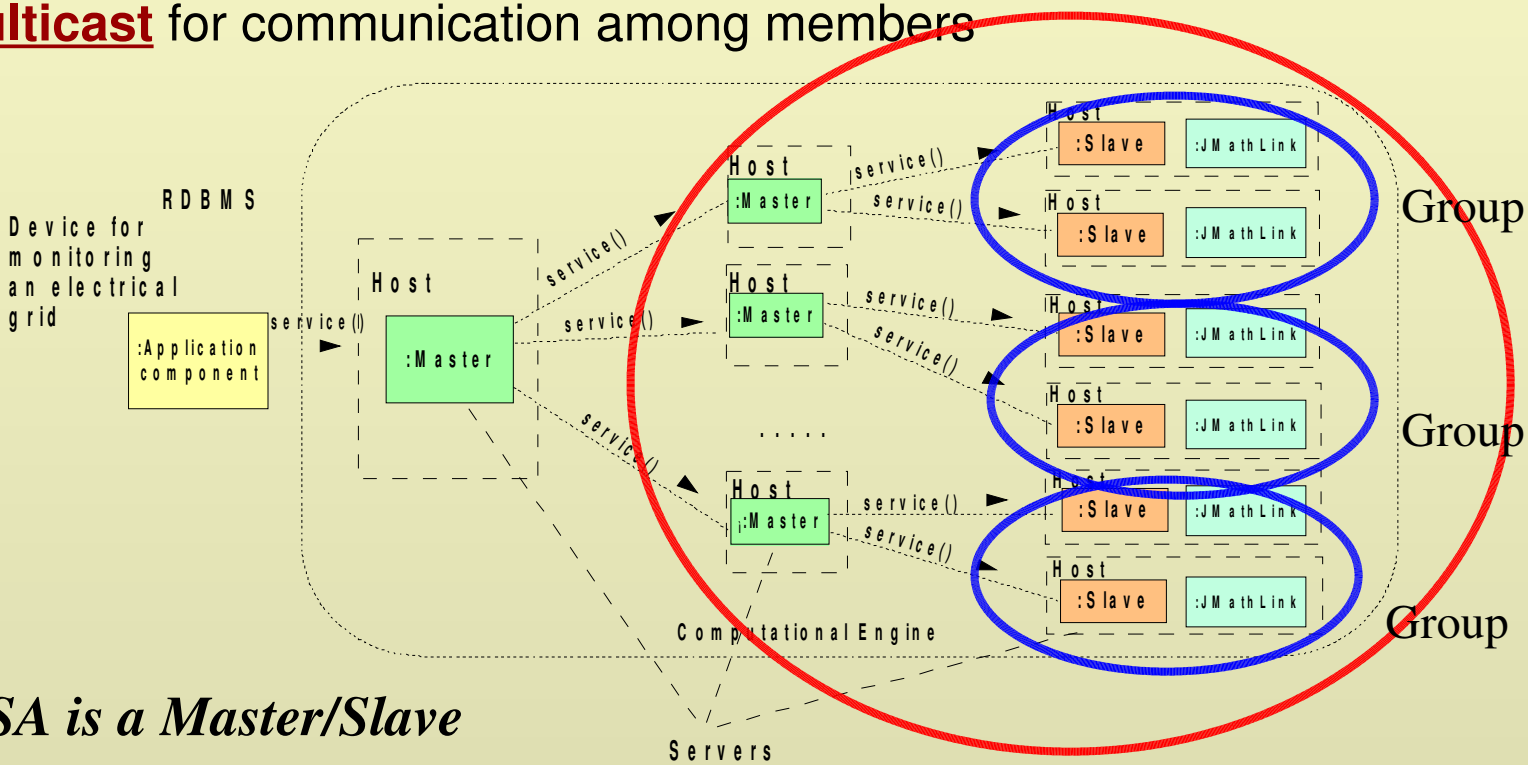
In collaboration with Francoise Baude, Laurent Baduel

{fbaude, lbaduel}@sophia.inria.fr

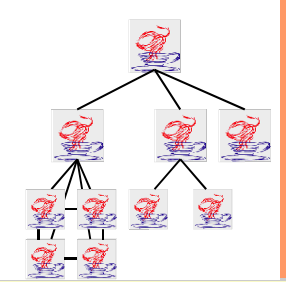


On going work – ProActive user group 2004

- Implementing the On line Power System Security Analysis (OPSSA) by using ProActive **Hierarchical Groups**
- Providing ProActive groups with **programmable behaviors** and **reliable multicast** for communication among members



OPSSA is a Master/Slave application



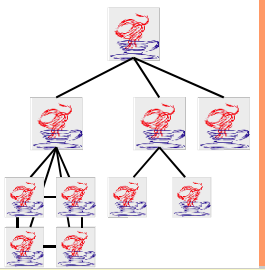
Outline

- Introduction: Groups and Grids

- Group semantics
- ProActive Groups extension

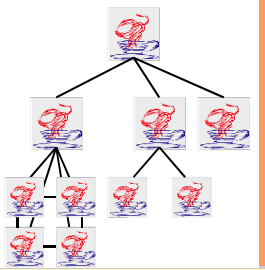
- Reliable multicast for Grid computing
- Performance analysis of TRAM
- ProActive over HiMM/TRAM

- Case Study: master/slave computing
- Performance evaluation
- On going work



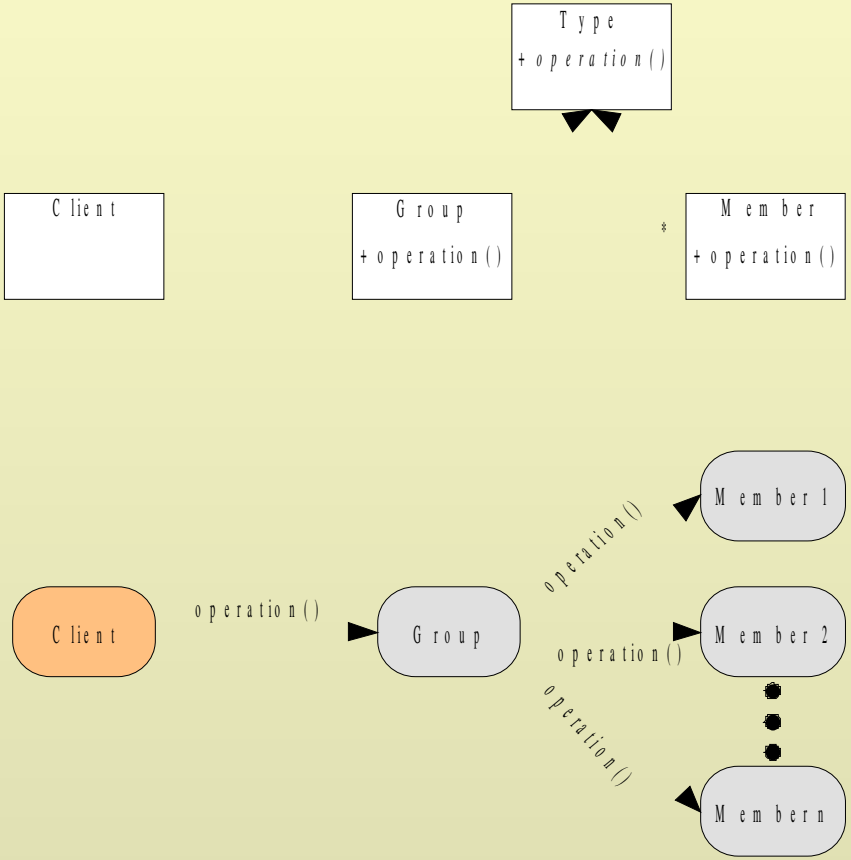
Introduction

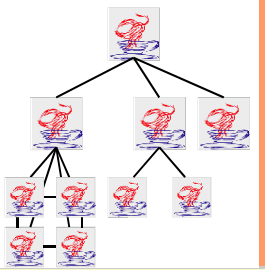
- Most of Grid applications and middleware platforms demand for large sets of data to be delivered to a wide collection of resources
  - to perform program and data remote submissions
  - to implement information and naming services
  - to manage data replication
- In this work we are interested only to manage **data replication**; program submission and naming services will be considered in future works
- For object-oriented programming, a high-level abstraction is necessary to hide the complexity of the network when data replication is to be performed:
  - **Groups**



What is a group ?

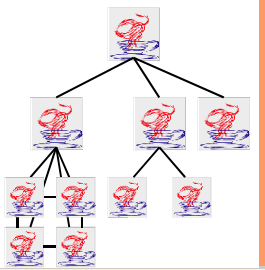
- A group is a local surrogate for a group of objects (members)
  - The members can be distributed across networked machines
  - When an operation is invoked on the group, it is forwarded to the members according to a policy and the results are returned back to the client
  - Groups are dynamic: the set of members can change





Groups and Grids

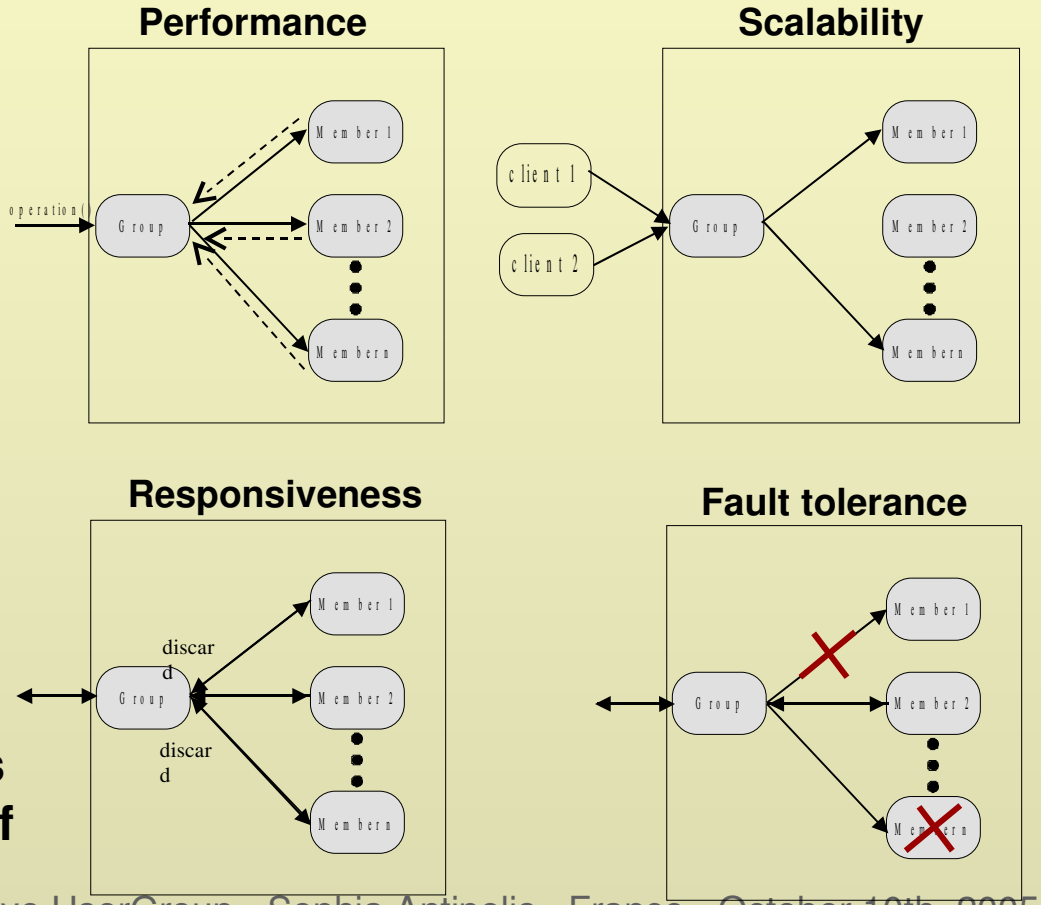
- Why group communication in Grid computing ?
  - Some grid applications require the same data to be delivered to a multitude of receivers
  - Such receivers could be treated as a unique coherent entity at programming level
  - The underlying middleware should map the interaction with a conceptual entity to a dynamic group of members
  - The mapping should be performed for several reasons:
    - Performance
    - Scalability
    - Responsiveness
    - Availability and Fault tolerance

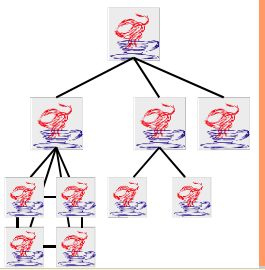


How can a group help ?

- At programming level
  - A group object looks like a single object
- At system level

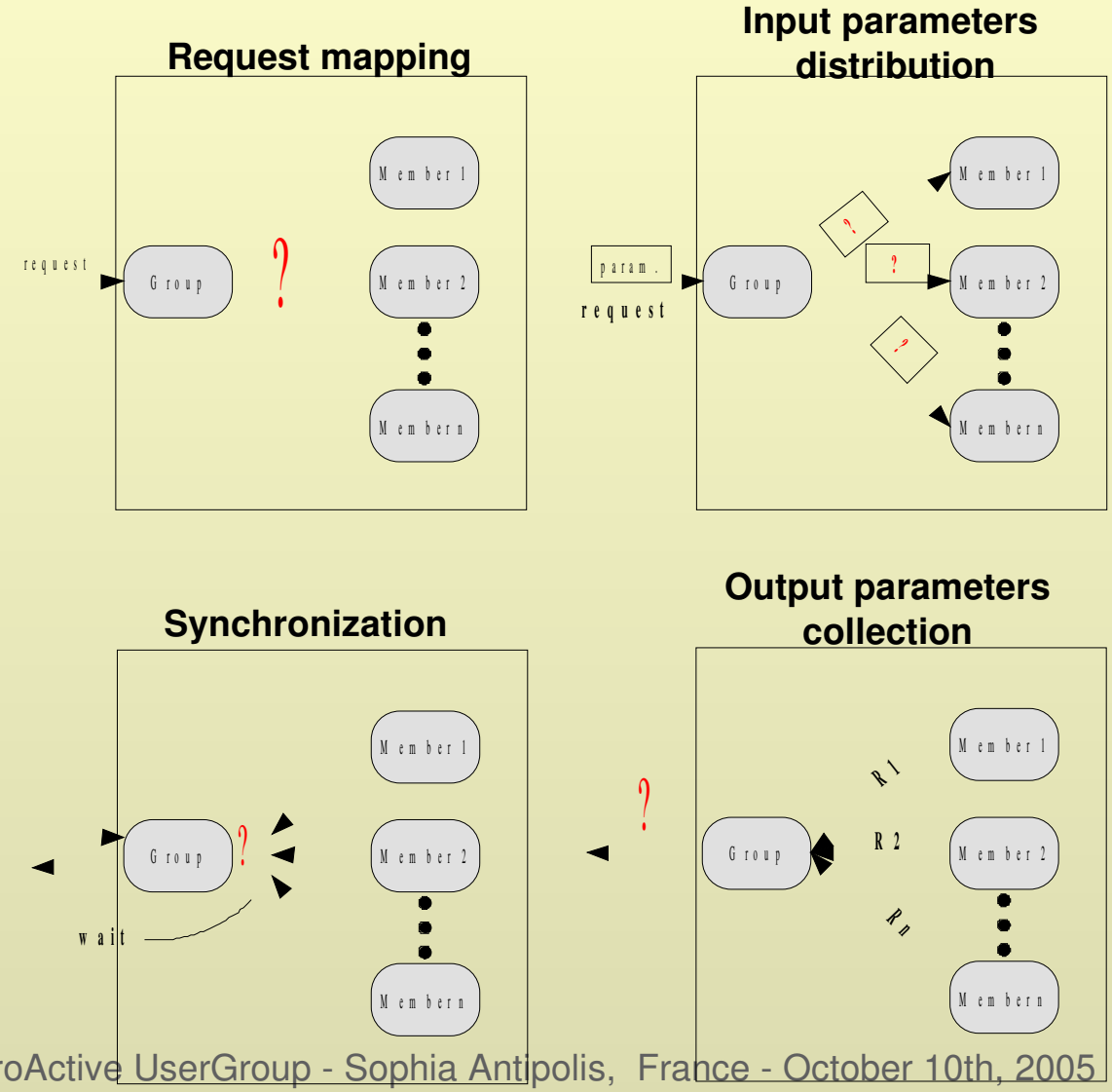
(performance) distributes a task in subtasks for different hw resources
(scalability) dispatches requests to different resources as the number of clients grows
(responsiveness) sends a request to many receivers for selecting the most reactive one
(fault tolerance) Sends a request to many receivers for surviving to crashes of network or nodes

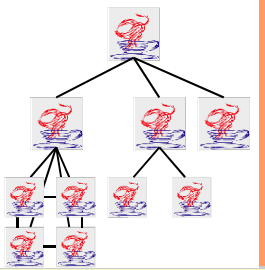




Group Semantics (1)

- Request mapping
- Input parameters distribution
- Synchronization
- Output parameters collection

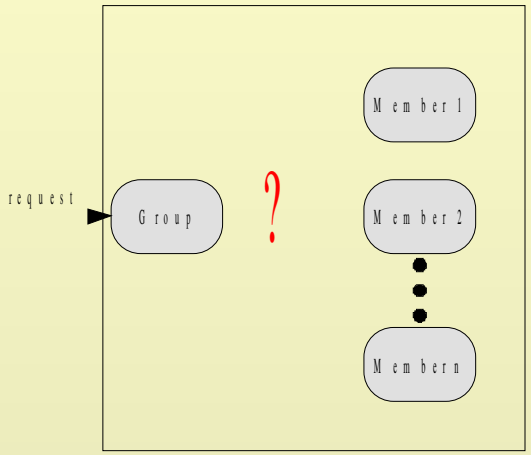




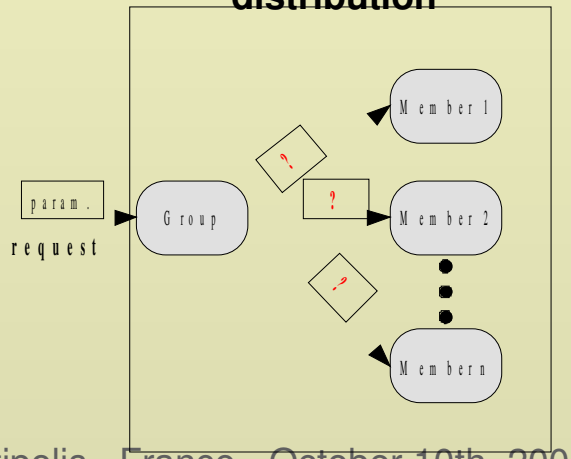
Group Semantics: policies (2)

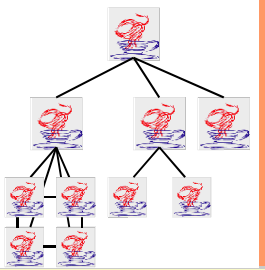
- Request mapping
  - One:
    - The request is delivered to one member
  - Fixed:
    - The request is delivered to a specified number of members
  - All:
    - The request is delivered to all the members
- Input parameters distribution
  - Scatter:
    - The parameters are split in several parts each one assigned to a member
  - Broadcast:
    - The parameters are sent to all the members

Request mapping



Input parameters distribution

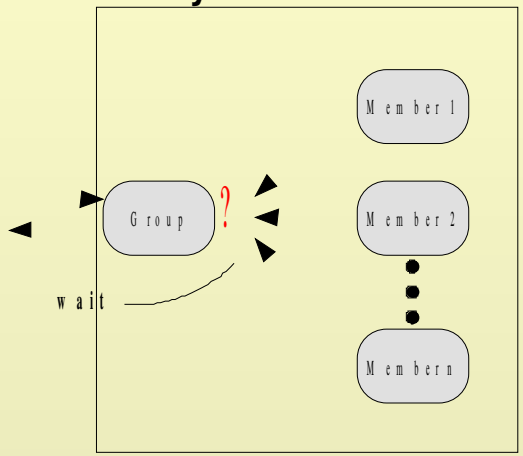




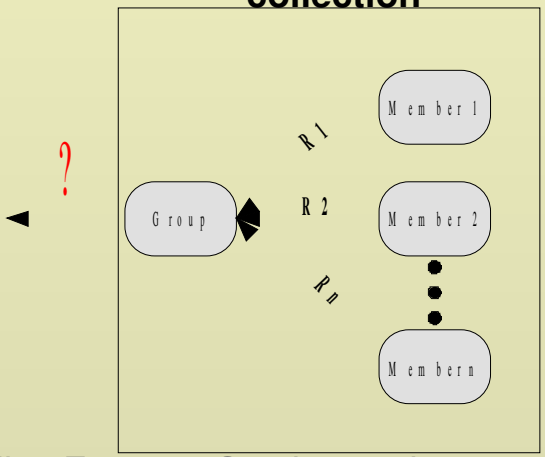
Group Semantics: policies (3)

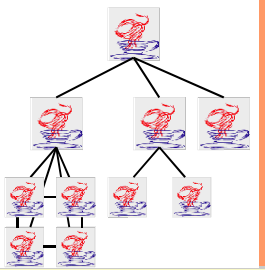
- Synchronization
  - All:
    - The overall result will be created only after the results from all the members
  - Majority:
    - The overall result will be created only after the results from the majority of the members
  - One:
    - The overall result will be created after the result of a member
  - Fixed:
    - The overall result will be created only after the results from a fixed number of members
- Output parameters collection
  - Gather:
    - The parameters are collected in an aggregate object whose parts are ordered according to the sequence of invocations
  - Merging:
    - The parameters are collected in a single structure according to some policy

Synchronization



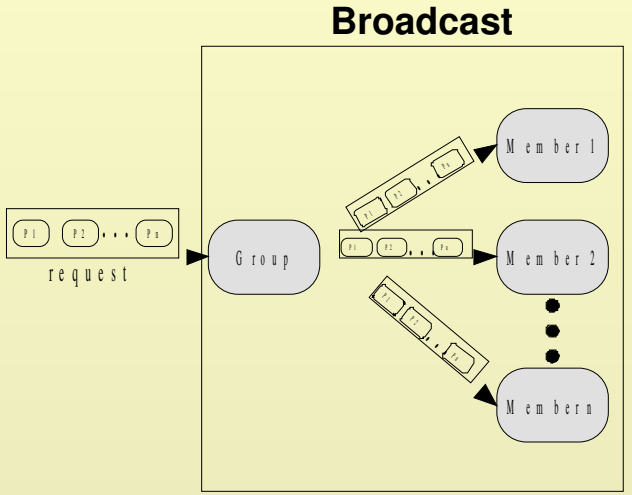
Output parameters collection

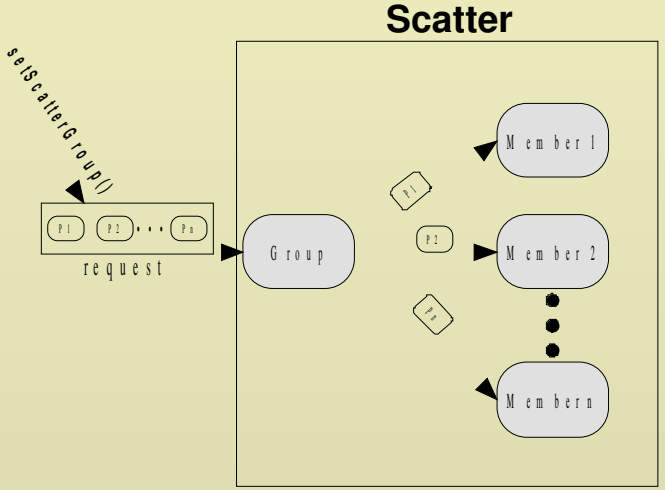


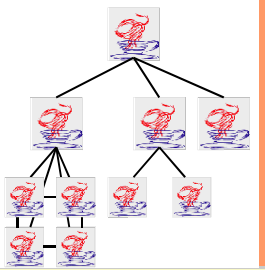


ProActive groups

- ProActive groups provides:
  - A default request mapping policy: *all*
  - A default input parameters distribution policy: *broadcast* + the possibility to change in *scatter* through the invocation of setScatterGroup()
  - A default output parameters collection policy: *gather*
  - Several synchronization policies explicitly managed by using the result

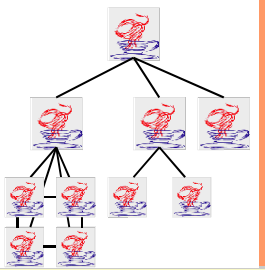






Group Architecture

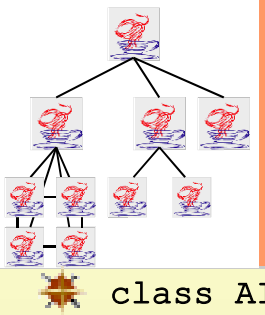
```
interface OutputCollectionSemantic {
  Object manageOutput(MethodCall mc, Vector futures);
}
```



Applying semantics to methods

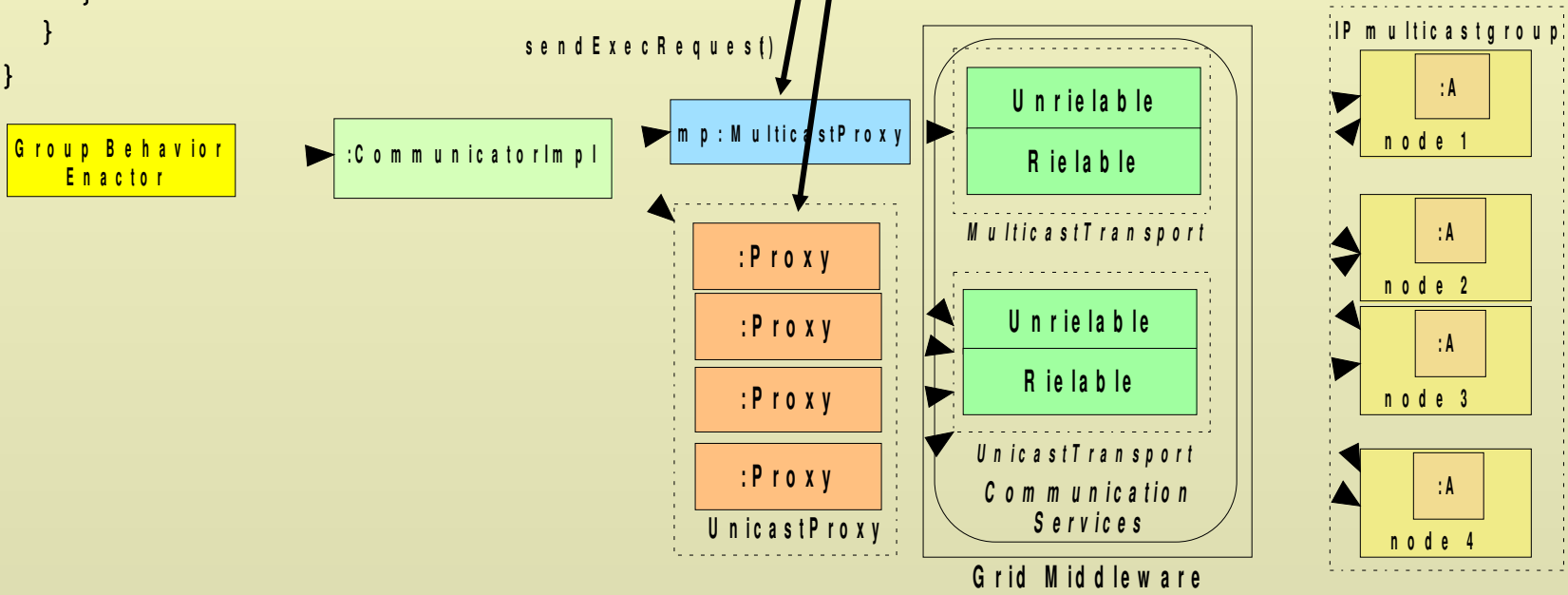
- Consider for example a class for a group:

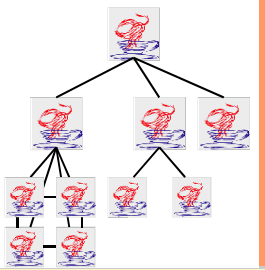
```
class A {
  public A() {}
  public C method1 (Object input) {}
  public C method2 (Object input) {}
  public C method3 (Object input) {}
}
class AMappings extends Mappings {
  public Vector getMembers(MethodCall mc, Vector ml) {
    if (<mc==method1>) { one(ml, 0);}                    // One
    else if (<mc==method2>) { fixed(ml, new int[]{0,1}); } // Fixed
    else if (<mc==method3>) { all(ml); }                   // All
    return mapped;
  }
}
class AInputs extends Inputs {
  public Vector manageInputs(MethodCall mc, Vector ml,
                 Communicator comm) {
    if (<mc==method2>) { scatter(mc.getParameter(0), ml); } // Scatter
    else if (<mc==method3>) {
                       broadcast(mc.getParameter(0), ml);  // Broadcast

    }
  }
}
```



Selecting a transport layer

```
class AInputs extends Inputs {
  private Parameters par = new Parameters();
  public Vector manageInputs(MethodCall mc, Vector ml,
                Communicator comm) {
    if (<mc==method2>) { scatter(mc.getParameter(0), ml);        // Scatter
       Parameter p = new Parameter("reliability", "reliable"); par.add(p);
       comm.setLogicalCommunication("unicast", par); }
    else if (<mc==method3>) { broadcast(mc.getParameter(0), ml); // Broadcast
       Parameter p = new Parameter("reliability", "unreliable"); par.add(p);
       comm.setLogicalCommunication("multicast", par);
    }
  }
}
```

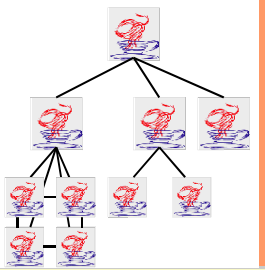




Coding the semantics (1)

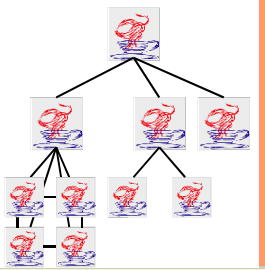
- Known semantics can be collected in abstract classes:
  - RequestMappingSemantic

```
abstract class Mappings implements RequestMappingSemantic {
  protected Vector mapped = new Vector();
  protected void one(Vector ml, int i) {
    mapped.add(ml.elementAt(i));
  }
  protected void fixed(Vector ml, int[] i) {
    for (int j = 0; j < i.length; j++)
      mapped.add(ml.elementAt(i[j]));
  }
  protected void all(Vector ml) {
    mapped = ml;
  }
  abstract Vector getMembers(MethodCall mc, Vector ml);
}
```

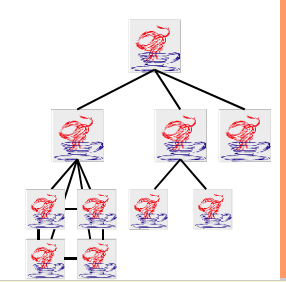


Coding the semantics (2)

- Known semantics can be collected in abstract classes:
  - InputDistributionSemantic
-
```
abstract class Inputs implements InputDistributionSemantic {
  protected Vector scatter(Object par, Vector ml) {
    if (par.getClass().isArray()) {
      Object[] o = (Object[])par;
      Class c = par.getClass().getComponentType();
      Object part = null;
      int size = ml.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]);
        ml.add(k, part);
      }
    } else if . . .
  }
  protected Vector broadcast(Object par, Vector ml) {
    for (int i = 0; i < ml.size(); i++)
      ml.add(i, par);
    return ml;
  }}
```

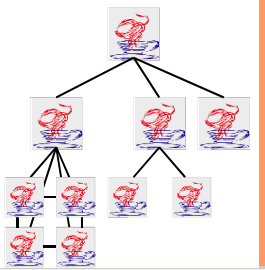


Reliable Multicast for ProActive Groups



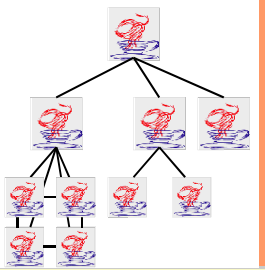
Reliable Multicast Protocols: problems

- Sender-initiated schemes
  - A critical issue is the reduction of the number of feedback messages that are returned to the sender
- Receiver-initiated schemes
  - Avoid ACK implosion by sending feedbacks only when a receiver notices a loss (negative acknowledgement – NACK)
- However, also in this case a large number of NACKs can be sent to the sender when the number of receivers grows
- To control the NACK implosion problem more sophisticated mechanisms are needed



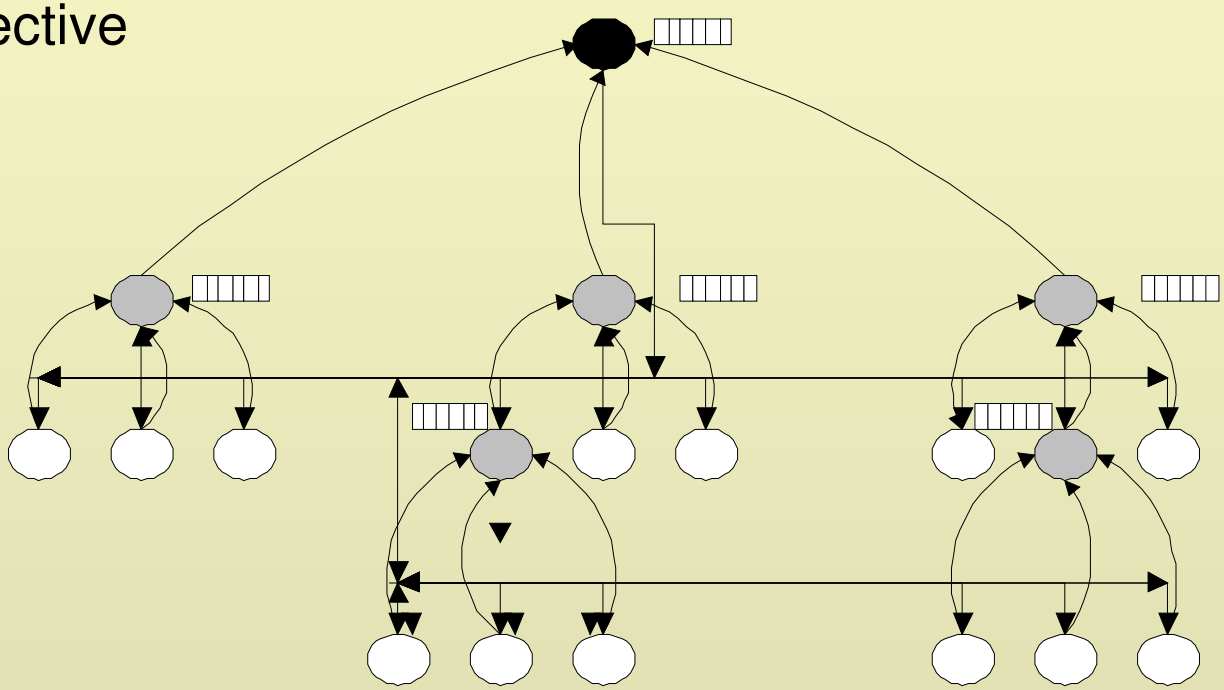
Reliable Multicast Protocols: solutions

- NACK suppression
  - SRM (*Scalable Reliable Multicast*)
- Local recovery
  - LRMP (*Light-weight Reliable Multicast Protocol*)
- Local recovery + static tree-based topology
  - RMTP (*Reliable Multicast Transport Protocol*)
- Local recovery + dynamic tree-based topology
  - TRAM (*Tree-Based Reliable Multicast*)

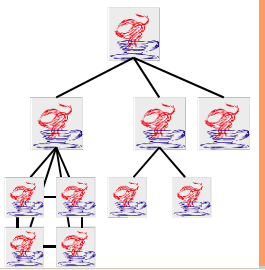


TRAM details

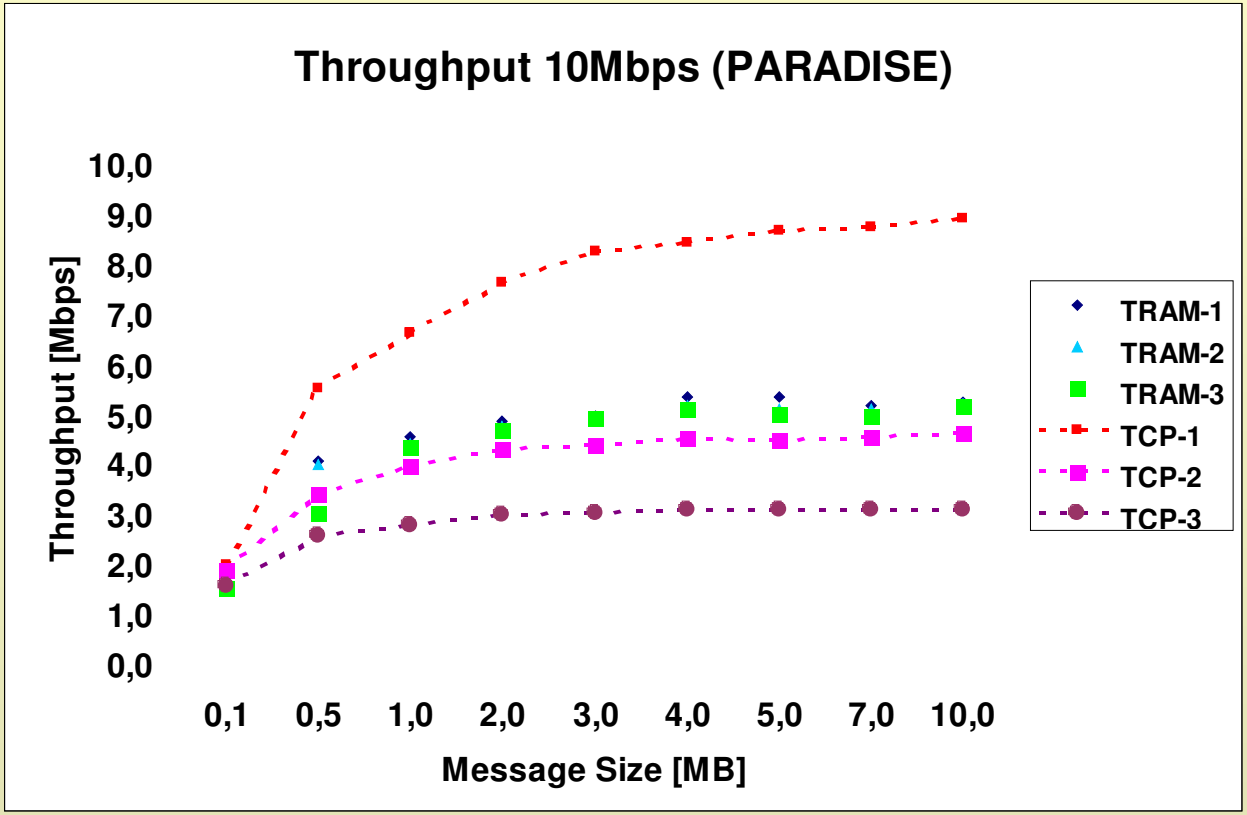
- Hierarchy is a key concept for ensuring scalability to reliable multicast protocols
- Therefore, among the discussed protocols we have considered TRAM for our objective



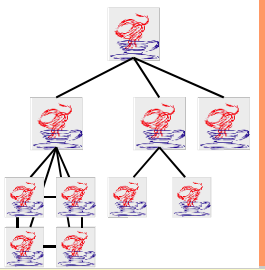
Sender, Group Head
Receiver, Group Head
Receiver, Group Member
Groups
Data Cache
Multicast Data Message
Unicast Ack Message
Multicast Local Repair (Retransmission)



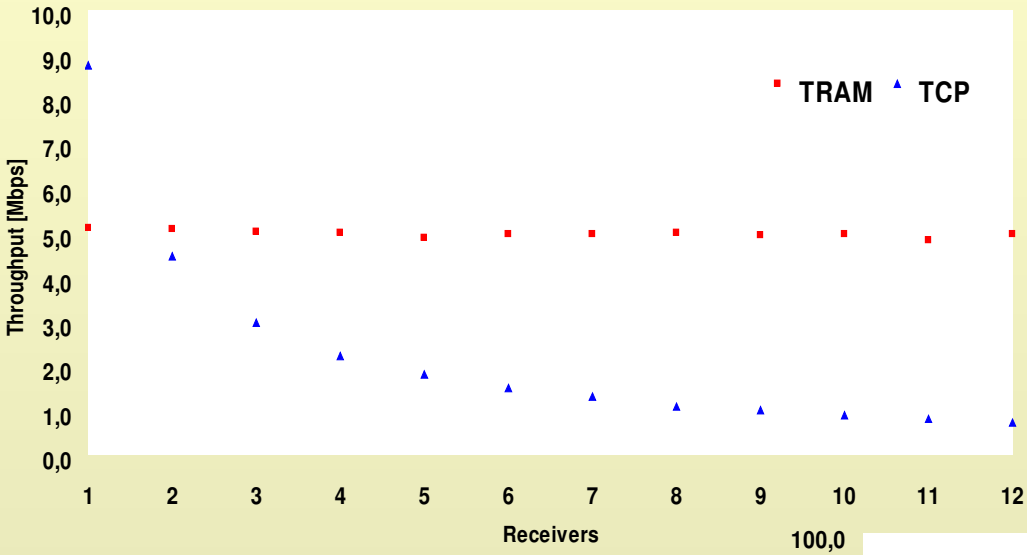
TRAM and TCP throughputs



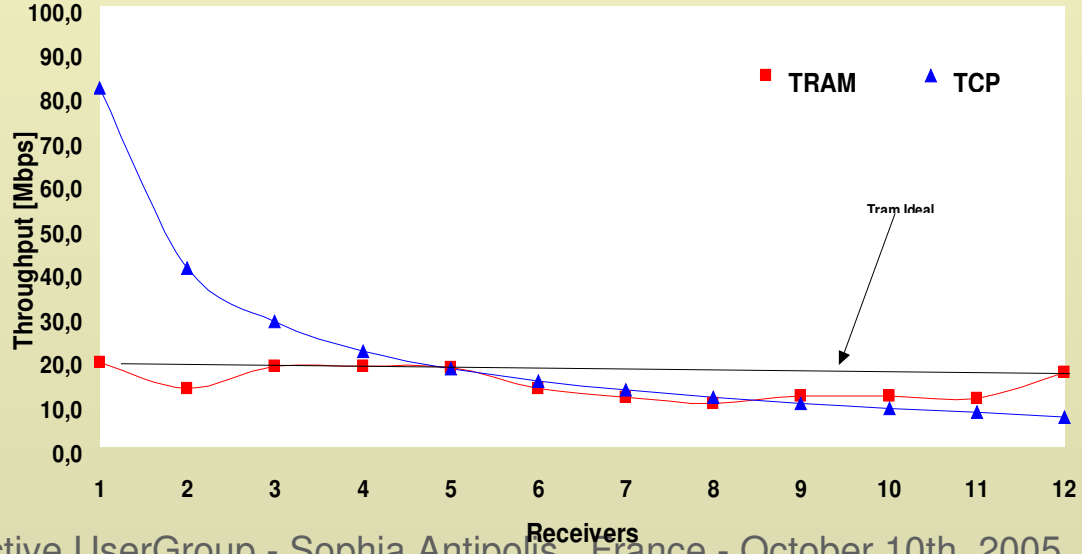
- Pentium II dual-processor
- Hub 10Mbps
- With an optimal setting of TRAM configuration parameters

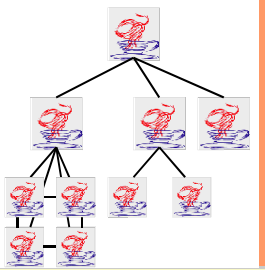


TRAM and TCP scalability (1)



Homogeneous cluster





TRAM and TCP scalability (2)

P4_1.8:

P4
1,8 Ghz
256 MB
Windows XP
Java 1.5.0

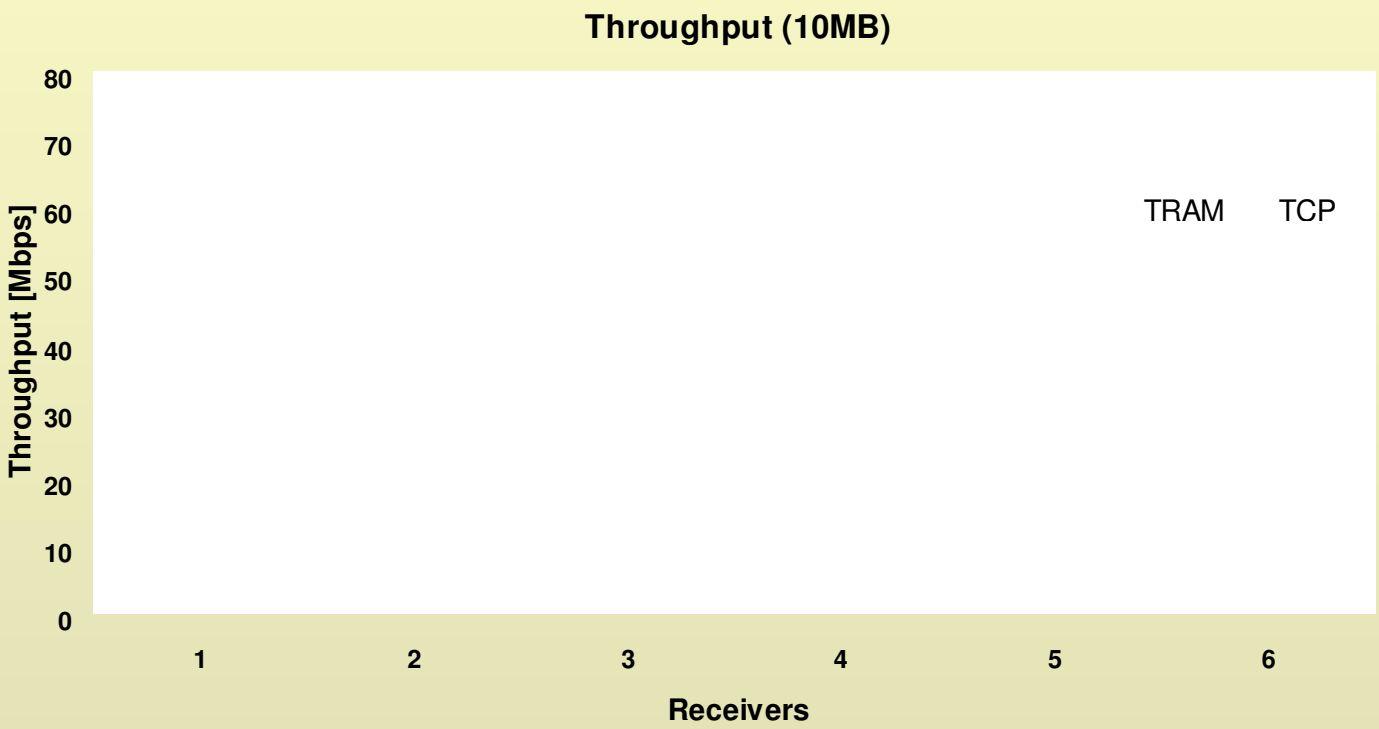
AMD_1.5:

AMD Athlon XP 1800+
1,5 Ghz
384 MB
Windows XP
Java 1.5.0

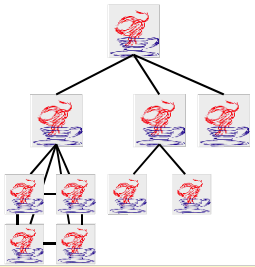
P4_2.0:

P4
2,0 GHz
512 MB
Windows Xp
Java 1.5.0

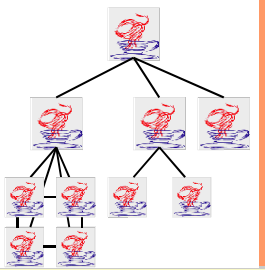
P4_2.4:



Heterogeneous cluster



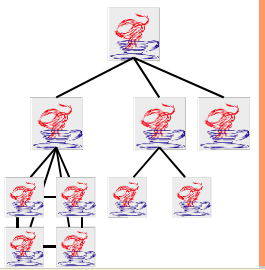
Case Study



Implementing the master/slave model with ProActive Groups

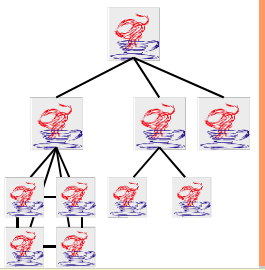
- Request Mapping
  - Request must be sent to **all** slaves
    - Slaves can be chosen in order to minimize the computation time
- Input Distribution
  - Constructor parameters can be **broadcasted** to the slaves
  - Method parameters must be **scattered** to the slaves
    - Slaves can receive different task sizes if the computational resources are heterogeneous
- Synchronization
  - **All** the results must be available to continue the computation
- Output Collection
  - Each result is **assembled** (merged) to form a single object

- Constructor invocations
  - The transport layer is **reliable multicast** (**TRAM**)
- Method invocations
  - The transport layer is **reliable unicast** (**TCP**)



Matrix multiply with native groups: code writing (1)

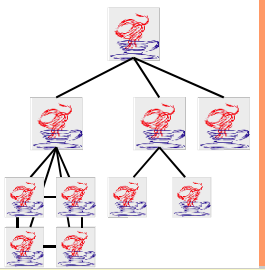
```
public class MatrixMultiply1 {
  public static void main (String args[]) {
    Matrix mRxGroup, mLxGroup result;
    Node[] nodes = null; // nodes list for slaves
    float[][] a, b;
    // def. of the left mat. a and right mat. b
    int totalRows = b.length;
    Object[] po = new Object[] {b};
    mRxGroup = (Matrix)ProActiveGroup.newGroup("Matrix", po,
                nodes);
    Object[] parts = createSubMatrices(a, nodes.length);
    Object[][] pars = new Object[nodes.length][];
    for (int i=0 ; i < nodes.length ; i++) {
      po = new Object[] {parts[i]};
      pars[i] = po; }
    mLxGroup = (Matrix)ProActiveGroup.newGroup("Matrix", pars,
               nodes);
    ProActiveGroup.setScatter(mLxGroup);
    Matrix gResult = mRxGroup.multiply(mLxGroup);
    Matrix result = reconstruction(gResult, totalRows);} }
```



Matrix multiply with native groups: code writing (2)

```
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; }
```

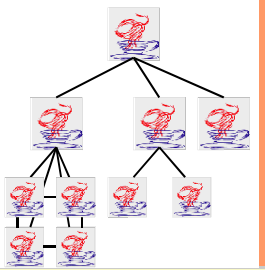
```
public Matrix reconstruction(Matrix group, int rows) {
   int index = 0;
   Matrix partial = null;
   int size =  ProActiveGroup.size(group);
   float[][] d = new float[rows][];
   for (int i=0 ; i < size ; i++) {
     partial = ((Matrix)(ProActiveGroup.get(group,i)));
     int  widthTmp = partial.getWidth();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



Matrix multiply with extended groups: code writing

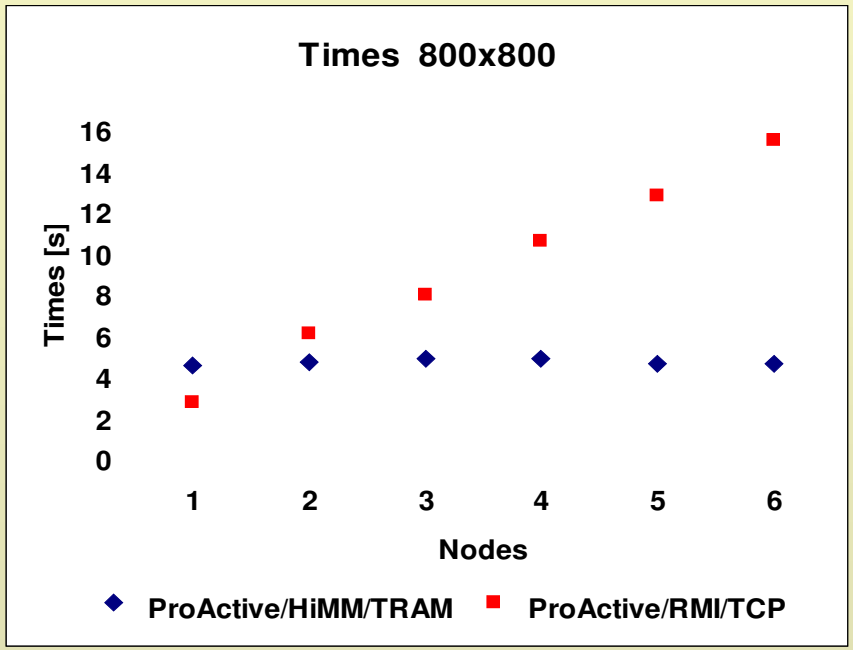
- The master/slave semantics and policies are provided by the class `MSGroupBehavior`

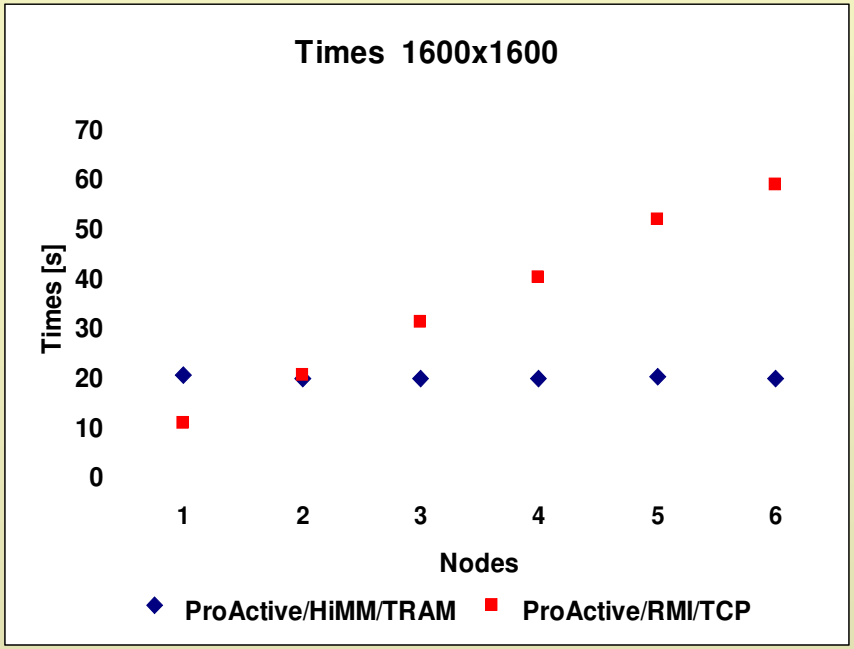
```
public class MatrixMultiply2 {
  public static void main (String args[]) {
    Matrix mRxGroup, result;
    Node[] nodesList = null;// nodes list for slaves
    float[][] a, b;// … def.left mat. a, right mat. b
    GroupBehavior msbeh = new MSGroupBehavior();
    Object[] po = new Object[] {b};
    mRx = (Matrix) ProActiveGroup.newGroup("Matrix",
           po, nodesList, msbeh);
    result = mRx.multiply(a);
    . . . // use of the result matrix
  }
}
```

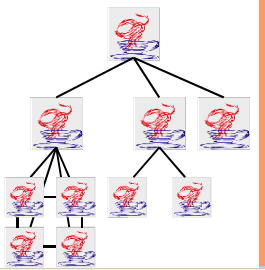


Performance of matrix multiply: data transfers

- Times for group members creation (transfers of right matrices)

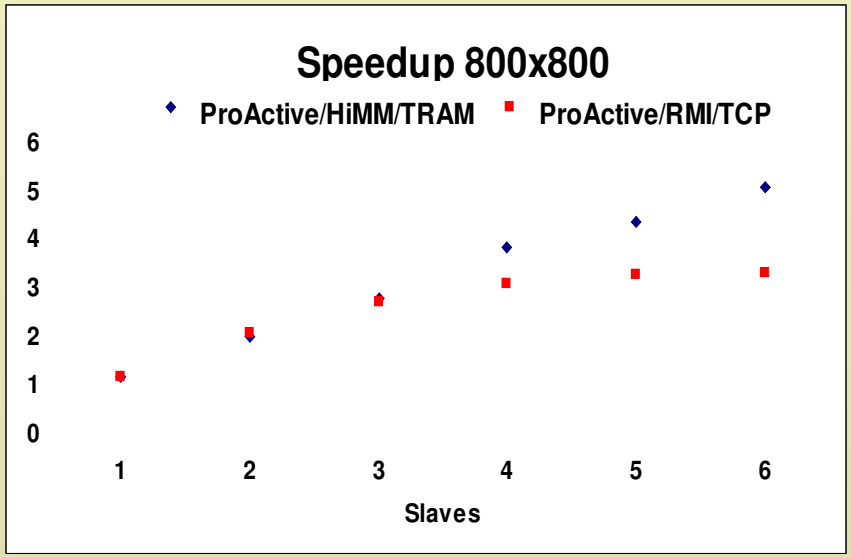


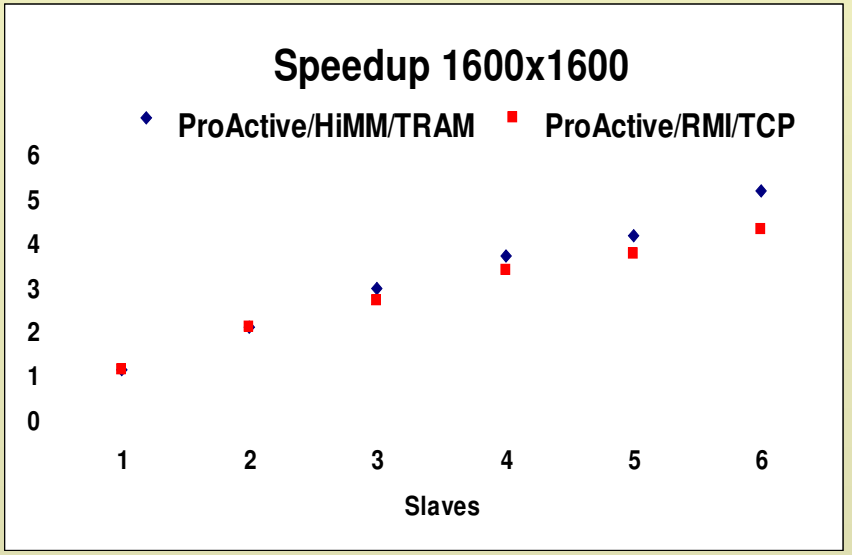


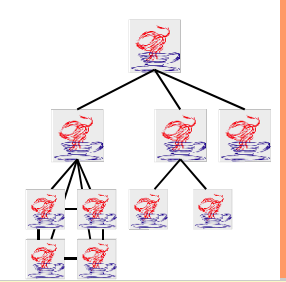


Performance of matrix multiply: overall computation

- Speedups
- In this case we consider
  - Group member creation (right matrix broadcasting)
  - Left matrix scattering
  - Product processing on each slave
  - Output collection







Future work

- Testing hierarchical groups over a large hierarchical network

- Implementing automatic group member allocation with resource management system based on QoS
  - This way in the m/s model, the number of slaves can be chosen dynamically on the basis of available resources and desired computation time