Walter Cazzola, Massimo Ancona, Fabio Canepa, Massimo Mancini, and Vanja Siccardi. Enhancing Java to Support Object Groups. In Proceedings of the 3rd Conference on Recent Object-Oriented Trends (ROOTS'02), *Bergen, Norway, April 2002.*

Enhancing Java to Support Object Groups

Walter Cazzola Massimo Ancona Fabio Canepa Massimo Mancini Vanja Siccardi

DISI-Department of Informatics and Computer Science
University of Genova, Italy
E-mail: {cazzola|ancona}@disi.unige.it

Abstract

In this paper we show how to enhancing the Java RMI framework to support object groups. The package we have developed allows programmers to dynamically deal with groups of servers all implementing the same interface. Our group mechanism can be used both to improve reliability preventing system failures and to implement processor farm parallelism. Each service request dispatched to an object group returns all the values computed by the group members permitting the implementation of both kind of applications. Moreover, these approaches differ both over computations failure and over the semantic of the implemented interface. Our extension is achieved enriching the classic RMI framework and the existing RMI registry with new functionalities. From user's point of view the multicast RMI acts just like the traditional RMI system, and really the same architecture has been used.

1. Introduction

Client/server object models focus their attention on improving portability, interoperability and reusability of distributed software components and applications. Unfortunately, most of them do not provide an adequate support for the development of reliable and high-available applications. This constitutes a major limitation for many modern industrial applications, for which requirements such as reliability and high-availability are gaining increasing importance. In the absence of any kind of systematic support, building applications capable to deal with partial failures such as process crashes or to subdivide the load-work among several processes is an error-prone and time-consuming task.

In order to overcome these difficulties, object groups [11] have been proposed. In the object group approach, many servers provide the same functionalities, — i.e., they are based on the same interface. Clients interact with object groups in a transparent way, as if they were a single, and non-replicated object. Objects forming a group cooperate in order to provide a reliable and high-available service to their clients. This cooperation is established through the facilities offered by a *group communication service* (GCS) [4, 9], that enables the creation of dynamic groups of objects that communicate through reliable multicast primitives. Objects forming a group are kept informed about the current membership of the group itself, that may vary at run-time due to accidental events such as failures and repairs, or to voluntary requests to join or disjoin the group. Object groups can also be used to gather objects which implement the same interface, but whose methods have a different semantics, to realize *processor farm parallelism* [10] or to prevent from logical failures [2].

Nowadays, Java and its class library are one of the most used frameworks to realize distributed applications. It well supports point-to-point communication through the native RMI mechanism, but it is not enough powerful to deal with the increasing necessity of reliability and availability of many enterprise applications. Apart from Filterfresh [3], Jgroup [13,14], and few others frameworks that offer a group-enhanced extension of the Java distributed object model, there is no attempt to support multi-point communications in Java. Both Filterfresh and Jgroup focus their efforts in realizing multi-point communications transparently from the client's point of view, i.e., handling an object group like a single entity carrying out its services and returning a single value as answer. Transparency and implementation simplicity are the main advantages of this approach, since clients obtain a single value as if they were invoking a method of a non-replicated object. However, they present a lose of flexibility, since the object groups cannot be used to implement, for example, computations following the *processor farm* model. Therefore, we have developed an extension to the Java communication model which supports object groups and multi-point method invocations with multiple return values.

The paper is organized as follows. Sections 2 and 3 recall some background about the group communication paradigm and the Java RMI. Sections 4, and 5 describe our

communication model and its implementation. Whereas, section 6 presents the framework at work on simple examples. Finally, conclusions and related work are considered in the last two sections.

2. The Group Communication Paradigm

Object groups (groups for short) [11] provide a mechanism to handle many objects as one. Groups are the key abstraction of group communications. A group is a collection of objects that share a common goal. This goal consists in offering improved quality services, i.e., both enhancing reliability and availability through replication and enhancing performances through duty redistribution and cooperative work. Usually, groups dynamically grow, i.e., objects join and disjoin a group at their own discretion.

During the last few years, several academical and commercial group communication frameworks are appeared [5, 16, 21]. Although the services provided by these systems present several differences, the key mechanisms underlying their architectures are the same: a group membership service integrated with a reliable multicast service. The objective of a group membership service is to keep members consistently informed about changes in the current membership of a group through the installation of views. The membership of a group may vary in consequence of requests to join or to disjoin a group, or to accidental events such as failures and repairs of both the computing system (member crashes and recoveries) and the communication system (network partitioning and merging). Installed views are composed by a collection of members and represent the perception of the group's membership that is shared by its members. A reliable multicast service has the task of enabling the members of a group to communicate by multicasting messages. Two members that install the same pair of views in the same order deliver the same set of messages between the installations of these views. This delivery semantics, called *view synchrony*, enables members to reason about the state of other members using only local information such as the current view composition and the set of delivered messages. Two classes of GCS have emerged: primary-partition [5] and partitionable [21]. A primary-partition GCS attempts to maintain a single agreed view of the current membership of a group. Members excluded from this view are not allowed to participate in the distributed computation. In contrast, within a partitionable GCS approach multiple agreed views may coexist in the system, each of them representing one of the partitions in which the network is subdivided. Primary-partition group communication services are suitable for non-partitionable systems, or for applications that need to maintain a unique state across the system. Partitionable systems are useful for applications that are able to take advantage of their knowledge about partitioning in order to make progress in multiple, concurrent partitions.

3. The Java Distributed Object Model

Java RMI is a distributed object model that maintains the semantics of the Java object model, making distributed objects easy to implement and use. Remote objects are characterized by the fact that their methods can be invoked from other Java virtual machines, potentially on different hosts. Given a remote object class, the set of its methods that can be remotely invoked is defined by one or more remote interfaces. Clients of a remote object never interact with the actual implementation class of this object, but only with a local surrogate object that presents the same set of remote interfaces.

The Java RMI architecture is illustrated in Figure 1 and consists of three layers: *stub & skeleton*, *remote reference*, and *transport* layer. Each layer is independent, and can be replaced without affecting the other layers.

3.1. Stub & Skeleton Layer.

The *stub & skeleton layer* represents the interface between applications and the rest of the RMI system: it has the task of marshaling and unmarshaling the invocation parameters and the return values. Clients invoke methods of a remote object through a stub, which plays the role of a proxy for the remote object. The stub implements the same remote interfaces of the remote object, and forwards each invocation request to the remote object through the remote reference layer. On the server side, a skeleton object dispatches the requests coming from the remote reference layer to the corresponding methods of the remote object. Both stubs and skeletons are described by Java classes generated by the `rmic` preprocessor out of a remote service implementation. In Java 2, the class `RemoteObject`, — i.e., the class extended by each remote object — carries out skeleton duties, so skeletons are no more needed.

3.2. Remote Reference Layer.

The *remote reference layer* is responsible for the semantics of the invocation. Objects defined in this layer realize the link with the implementation of the remote services. The current version of Java RMI includes only two unicast (point-to-point) invocation mechanisms: one relative to servers always running on some machine (`java.rmi.server.UnicastRemoteObject`), and one relative to servers that are activated only when one of their methods is invoked (`java.rmi.activation.Activatable`). To provide remote services, the class of a server has to extend either the class `UnicastRemoteObject` or `Activatable`.

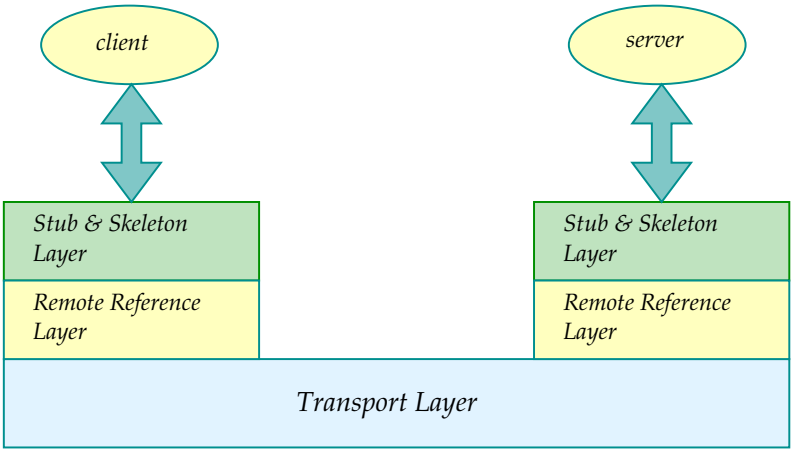

Figure 1. The Java RMI architecture.

3.3. Transport Layer.

The *transport layer* encapsulates all the low-level details such as connection (among JVMs) management and invocation request transmission. Communications among JVMs are done by TCP/IP connections by using a proprietary stream-based protocol called Java Remote Method Protocol (JRMP).

Each client, before invoking methods of a remote object, must obtain a stub for it. For this reason, the Java RMI architecture includes a repository facility called *registry* that can be used to retrieve remote object stubs by name. Each registry maintains a set of bindings

$$<\textit{name, remote object}>;$$

new bindings can be added using the `bind` method, whereas the `lookup` method is used to get the stub for a remote object registered under a certain name. Since registries are remote objects, the Java RMI architecture includes also a bootstrap mechanism to obtain registry stubs.

4. Object Group Abstraction in Java

Point-to-point remote communications are an adequate mechanism to model client-server applications. Notwithstanding that, there are requirements which cannot or are difficult to be achieved by using this communication model — e.g., services' reliability through server replication, or divide and conqueror algorithms.

Our project consists of extending the Java system by adding to the point-to-point RMI a one-to-many communication mechanism. In our communication model, service requests will be forwarded to several servers providing such services.

From the point of view of the end user, a multicast communication can be realized either:

- by signing each multicast communication with its targets, such as in `acast()`, `bcast()`, and `abcast()` of ISIS [5], or
- by hiding its multiple targets with a representative (the group referent), and by using a unicast-like primitive to establish a connection to such a referent.

A complete transparency may be achieved by providing an intrinsic mechanism which chooses a value to return to the client, e.g., by voting, on first arrived first served basis and so on. Unfortunately, this approach jeopardizes its usability hindering the programmer from using all the calculated values in his algorithms.

We have preferred to follow a less transparent approach in our group abstraction implementation. From the programmers' point of view the mechanism used to invoke a remote method call remains unchanged both for unicast and multicast communications since, in case of a multicast communication, he asks the service to a representative of the group and not to each member, but he will receive back an array containing all the values computed by the members of the group instead of a single value only.

In the sequel, we show our framework and its realization. A detailed overview can be read in [7].

4.1. Object Groups Management.

A group is a composite entity whose components are objects providing the same services. Each object joins the group by invoking specific operations. Information about the group are kept and continuously updated by each registry involved in the group management. Registries

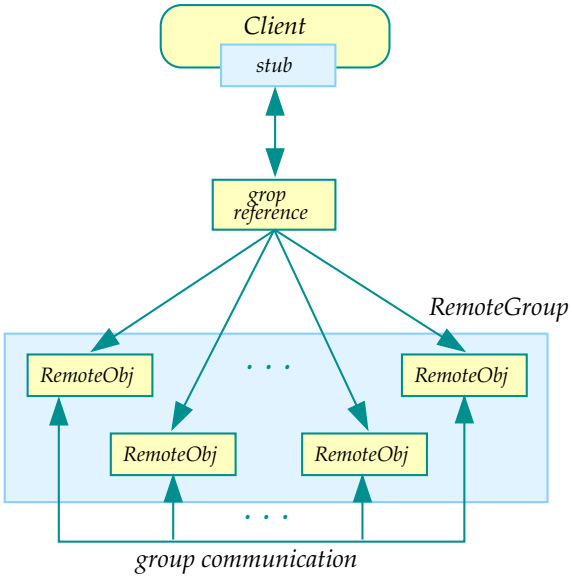

Figure 2. Architecture of an Object Group.

use some synchronization primitives to notify each other changes occurred to the group structure, e.g., when an object joins or disjoins the group.

In the sequel of this section, we show how to deal with object groups and multicast remote invocations in Java.

4.1.1. Members and Group Interfaces.

Every object belonging to a group provides the same services of the other members, these services are described by a Java interface. Member interfaces look as follow:

```
public interface Member extends Remote {
  typename1 method1() throws RemoteException, ...;
                    [...]
}
```

Analogously, services provided by the group are described by an aggregative interface, which reassembles the interfaces implemented by each object belonging to that group.

```
public interface Member_Group extends Remote {
  typename1[] method1() throws RemoteException, ...;
                    [...]
}
```

Group interfaces differ from member interfaces in the values returned by the specified services. Services provided by a group, as shown in the above interface, return the collection of all values computed by the group members serving the analogous services.

4.1.2. Members Definition & Creation.

Group members, like any remote server, have to extend the `RemoteObject` class or one of its extensions. Extending one of these classes provides objects with the mechanisms to serve remote invocations. In particular, each object has to extend the class `MulticastRemoteObject` to be able to serve requests appointed to a group.

```
public class Member1 extends MulticastRemoteObject
      implements Member {
                    [...]
}
```

Group members are created, as any remote server, by using the primitive `new`.

4.1.3. Group Creation and Membership.

Groups have to be created and registered to the `rmiregistry` as any remote server. They use a special instance of the RMI registry, called *multicast RMI registry*. This registry deals with group management, e.g., creation, joining and so on. Group creation takes two steps:

❶ the remote object asks (a call to `createNewGroup`) the multicast RMI registry for the creation of a new group, then

❷ the new group is registered in the multicast RMI registry as a group composed only of the requiring object.

The multicast RMI registry stores for the just created group the association «*GroupName*, *stub*», where *stub* represents the *group reference* (see Fig. 2).

After group creation, remote objects can ask to a registry to join (or to disjoin) the group using the method `join` (or `disjoin`). The multicast registry notifies the changes to all the other registries hosting a member of the group. After the notification each registry updates its databases.

```
Member m1, m2;
MulticastNaming.createNewGroup(GroupURL, m1);
MulticastNaming.join(GroupURL, m2);
```

4.1.4. Getting a Service.

As for the unicast RMI, clients to ask a group for services do not directly interact with it but with a local representative. This representative masks the real connection to the remote servers. Clients can get a representative of the group through a call to `MulticastNaming.lookup`.

```
Member_Group gr_ref;
gr_ref = (Member_Group)MulticastNaming.lookup(URL);
```

From user's point of view, the *bind/lookup* mechanism works similarly to the standard approach. They differ only for the meaning of the `URL` argument. In the classic approach, it directly links to the URL of the remote server, whereas in our framework it links to the remote object which has created the group. However, getting a service from a group is transparently carried out as in the unicast remote invocations.

```
typename1[] results = gr_ref.method1();
```

Figure 2 reassembles how the remote method invocation takes place. The stub dispatches the calls using its reference layer. The group reference (layer) forwards the request to the members of the group, collects the results and returns them to the client in form of an array as specified by the group interface.

5. Multicast RMI & Groups: the Architecture.

One of our objectives has been the design of an architecture which supports object groups without affecting the original architecture of Java RMI. To do that, we have extended the structure described in section 3 putting besides the remote reference, the stub & skeleton, and the transport layers, a new layer, — called *group layer*. The group layer deals with group abstractions and the multicast remote method invocations.

A description of the reorganized architecture and an overview of the classes involved in such a reorganization is presented in the following sections.

5.1. Stub & Skeleton Layer.

The stub & skeleton layers has been modified introducing stubs representing remote groups. In our framework we have two kinds of stubs: *unicast* and *multicast*. The former is the stub used for unicast remote invocations, e.g., they are used to ask a single group member for services. Whereas, the latter is the group representative. As in the unicast RMI, the `rmic` preprocessor is still in charge of generating stubs and skeletons.

5.1.1. Multicast `rmic`.

To allow a uniform access, the generated stub will refer to a `RemoteRef` object for both remote objects and object groups. When the stub represents a group an instance of `MulticastRef` is used instead. Hence, we have modified the `rmic` preprocessor to also generate stubs having arrays as return values of group services. This extension has involved the `java.rmi.rmic` package, and the `Generator` class which is really in charge to generate stubs and skeletons.

The `Generator` class.

A `Generator` object generates the source code for the remote server stub, by parsing the source code. Basically, for each remote interface `X` the compiler looks for another interface, called `X_group` in the current path. If found, we are in the case of a group interface and the attribute `isMulticast` is set. This attribute is the key mechanism leading the generation of a *unicast* stub rather than a *multicast* one.

A multicast stub differs from a unicast stub in the return values of its methods. In fact, it has to gather and to return an array of values rather than a single value.

```
if (! returnType.isType(TC_VOID)) {
  if (! isMulticast ) p.p("Object $result = ");
  else p.p("ArrayList $result = (ArrayList) ");
}
```

We have also adapted the generation of the hash key which indexes remote method invocations in order to handle the type mismatch between group and member interface. The hash key changes with the method prototype. We have changed the method prototype (because of the return value), hence this change has been reflected in the hash key generation as well.

5.2. Remote Reference Layer.

The remote reference layer has been extended to deal with the new remote invocation semantic, i.e., it forwards each request for a group service to each member of the group. This automatic forwarding mechanism has been realized by changing the unicast `RemoteRef` reference generated by the `rmic` with an instance of the `MulticastRef` class. Each instance of this class effectively represents a group, i.e., a reference to a list of servers belonging to such a group.

5.2.1. The Class `MulticastRef`.

`MulticastRef` implements the multicast client-side group remote reference. The class stores the list of remote references of the objects composing the group.
`invoke` is the main method of the class. Its behavior consists of asking each member of the group for serving the given method (code from line 6 to line 15).

```
public Object invoke(Remote obj, Method method,
        Object[] params, long op) throws Exception {
  List result = new ArrayList();
  Iterator gref = ref.gref();
  while (gref.hasNext()) {
    LiveRef sref = (LiveRef)gref.next();
    Connection c = sref.getChannel().newConnection();
    RemoteCall call =
        new StreamRemoteCall(c, sref.getObjID(), op);
    ObjectOutput out = call.getOutputStream();
    marshalCustomCallData(out);
    Class[] types = method.getParameterTypes();
    for (int i = 0; i < types.length; i++)
      marshalValue(types[i], params[i], out);
    call.executeCall();
    Class rtype = method.getReturnType();
    if (rtype == void.class) {
      result.add(null);
      continue;
    }
    ObjectInput in = call.getInputStream();
    Object returnValue = unmarshalValue(rtype, in);
    sref.getChannel().free(c, true);
    result.add(returnValue);
  }
  return result;
}
```

Its return value is an object which contains the collection of the results of each remote method invocation (code from line 16 to line 24).

As default behavior, a `RemoteException`[1] is thrown if *at least one of the calls fails*. A different fault-tolerant behavior is realized using a `MulticastRefFaultTolerance` reference. `MulticastRefFaultTolerance` extends `MulticastRef` throwing a `ZombieGroupRemoteException`,[1] if *all the calls fail*.

To marshal and unmarshal data appointed to the communication channel, the serialization mechanism has been extended overriding methods `writeExternal` and `readExternal`. The method `writeExternal` serializes the object group on the stream. At the beginning, it serializes the number of servers belonging to the group (*the size of the group*); then all the remote references. The method `readExternal` deserializes the object from the stream. It reads the number of remote references belonging to the group then reads all the remote references.

5.2.2. The Class `MulticastRemoteObject`.

The class `MulticastRemoteObject` defines a composite remote object whose references are valid only while the server process is alive. This class supports the multicast active object references (invocations, parameters, and results) using TCP streams. There are two kinds of behavior supplied by the `MulticastRemoteObject`:

- *parallel*, which implements the semantic of a parallel process execution (default).
- *fault_tolerant*, which implements a fault tolerant behavior.

Operations carried out in *parallel* mode fail and throw an exception when one or more members of the group fail. Whereas operations carried out in *fault_tolerant* mode fail only when no server in the group can return an answer.

The default behavior is set to *parallel*, when the mode is not specified calling the constructor of the remote servers. Objects that should be part of a group have to extend the `MulticastRemoteObject` class. If the object does not extend `MulticastRemoteObject` and notwithstanding that, it would be part of a group, it has to provide by itself the correct semantics of the hashCode, and of method `equals`. It has also to override the `toString` method inherited from the `Object` class, so that it behaves appropriately for both remote objects and group of remote objects.

`exportObject` is the main method of this class. It exports the remote object passed as argument.

```
public static Remote exportObject(Remote obj, int port)
                                    throws RemoteException {
  Object[] args =
    new Object[] {new Integer(port), new String(mode)};
  return exportObject(obj, "MulticastServerRef",
                                    portParamTypes, args);
}

private static Remote exportObject(Remote obj,
        String refType, Class[] params, Object[] args)
                                    throws RemoteException {
  String refClassName = "sun.rmi.server." + refType;
  Class refClass = Class.forName(refClassName);
  Constructor cons = refClass.getConstructor(params);
  ServerRef serverRef = cons.newInstance(args);

  if (obj instanceof MulticastRemoteObject)
    ((MulticastRemoteObject)obj).ref = serverRef;

  return serverRef.exportObject(obj, null);
}
```

To render it available to receive the incoming calls, it builds and returns a `RemoteStub` using the method `exportObject` of the class `MulticastServerRef` (line 20 of the reported code).

[1] Of course, if the remote object fails raising an exception this one is propagated to the client instead.

5.2.3. The Class `MulticastServerRef`.

`MulticastServerRef` implements the server side part of the remote reference layer for remote objects exported with the `MulticastRef` reference type. This class has only the attribute `mode` which specifies the behavior realized by the group: *parallel* or *fault_tolerant*. The attribute `mode` is set to *parallel* if not otherwise specified. To allow remote access to the object, the method `exportObject` builds the remote stub for the class starting from the `MulticastRef` class if `mode` is set to *parallel* and from the `MulticastRefFaultTolerance` class if `mode` is set to *fault_tolerant*.

5.3. Group Layer.

The *group layer* represents the kernel of our extension. Its main duty consists of providing all the needed tools for handling with groups, i.e., the multicast RMI registry, with a naming and a locating mechanism.

5.3.1. The Class `MulticastNaming`.

The class `MulticastNaming` — analogously to the Java class `Naming` — provides methods for storing and retrieving references to object groups in/from the remote registry.

Binding a name to a remote object means associating a name with it. Such a name will be used to look up the object. A remote object can be associated with a name by using the methods `bind` and `rebind` of the class `MulticastNaming`. When the exported object is a `UnicastRemoteObject`, the name represents simply its service label. When the exported object is a `MulticastRemoteObject`, the name represents the group service label.

Once a remote object is registered with the RMI registry on the local host, callers from a remote host can look up the object by name (using the method `lookup`), get its reference, and then invoke its methods (as seen in section 4). If the object represents a group, its methods will return an array of values, containing the result of the method invocation on each server. A registry can be shared by all the servers running on the host or each server may create and use its own registry. Methods of this class use services supplied from the registry defined in the interface `MulticastRegistry` and implemented by the class `MulticastRegistryImpl`. New methods have been added for dealing with groups: `createNewGroup`, `join` and `disjoin`. Moreover, the class `MulticastNaming` provides methods to access a remote object registry using URL-formatted names to specify in a compact format both the remote registry and the name for a remote object.

5.3.2. The interface `MulticastRegistry`.

Our framework comes with a simple remote object registry interface, `MulticastRegistry`, which provides methods for storing and retrieving remote object and group references. This interface contains both the methods defined by `UnicastRegistry` and some other methods needed to deal with groups.

```
public interface MulticastRegistry extends Remote {
  public Remote lookup(String name)
     throws NotBoundException, AccessException;
  public void bind( String name, Remote obj)
     throws AlreadyBoundException, AccessException;
  public void unbind( String name)
     throws NotBoundException, AccessException;
  public void rebind ( String name, Remote obj)
     throws AccessException;
  public String [] list () throws AccessException;
    // multicast RMI registry part
  public void update( String Name, Remote obj )
     throws NotBoundException, MalformedURLException;
  public void sync( String Name, Remote obj )
     throws NotBoundException, AccessException;
  public void disjoin ( String name)
     throws GroupNotBoundException, AccessException;
}
```

Typically a registry exists on every node running remote servers. Every server belonging to a group *must* register to a multicast registry. A multicast RMI registry is also unicast compliant, in the sense that it deals with unicast remote method invocations as well.

Every registry contains a database that maps group names to the objects belonging to that group. Initially, the database of a registry is empty. A server stores its services in the registry prefixing (but it is not mandatory) their name with the package name to avoid name collisions.

To create a multicast registry, the programmer can invoke the method `LocateMulticastRegistry.createRegistry`. Instead to get a reference to a remote object registry, the programmer can invoke the method `LocateMulticastRegistry.getRegistry`.

Methods `lookup` and `bind` are defined to carry out the `lookup`, `join`, and `disjoin` operations defined in the class `MulticastNaming`.

When a server joins (or disjoins) a group all the registries keeping entries for the group need to be informed of the change. This is implicitly done by methods `update`, and `sync`.

5.3.3. The Class `LocateMulticastRegistry`.

`LocateMulticastRegistry` is used to get a reference to a registry on a particular host (method `getRegistry`), or

to create a registry that accepts calls on a specific port (method `createRegistry`). The registry is a simple `UnicastRemoteObject`: the difference between the standard `LocateRegistry` is that this registry loads the class `MulticastRegistryImpl` and not the class `RegistryImpl`.

6. Object Groups at Work.

Object groups and multicast communications help in dealing with many situations, e.g., fault tolerant servers, or data-parallel programming. In this section we will face some simple examples showing how to use our approach to develop applications and their features.

6.1. Fault Tolerant Servers.

A classical object group application consists of managing *system & software fault tolerance* through either object replication [8, 19] or versioning [2]. In this kind of application, groups are used to improve the reliability of the provided services. They mask the fact that the server is replicated (or versioned) insuring the client against server failures. Many frameworks providing object groups support (e.g., ISIS [5], and Totem [16]), focus their efforts in this direction limiting groups potentiality.

As explained, our approach to group communication is a little bit less transparent than the other approaches rendering available all the computed values to the client. A similar approach does not prevent the programmer from realizing a group of replicas or versions improving servers reliability.

Both approaching software and system fault tolerance is quite simple. They differ only on the algorithms the group members implement. Software fault tolerance has to prevent from logical (both designing and programming) errors whereas system fault tolerance has to prevent system errors, e.g., host crashes. Hence, the latter will replicate the server, whereas the former will use two different versions of the server. However, both serve the same group interface. By example:

```
public interface server extends Remote {
    typename1 methodname1() throws RemoteException;
}

public interface server_group extends Remote {
    typename1[] methodname1() throws RemoteException;
}
```

A different approach will be taken with the implementation of the group members. If they are replicas we will have just a class implementing all the group members, whereas if they are versions we will have a class for each group member. By the way, this one is not the right place where to face the implementation of versions and replicas. We have more interest in showing how the client will deal with the multiple answers returned by the group.

6.1.1. Versioning.

The replicated servers implement the same services, but with different code in order to spare clients from logical errors. Each version returns its computed value, the client will receive them, then it decides that the most frequent value is the correct one (through the method `voting`).

```
public class Client {
  public static server_group grp;

  private static typename1 voting(typename1[] res) {
    Hashtable resCount = new Hashtable();
    int mostCommon = 0;

    for (int i=0; i<res.length; i++) {
      Integer val;
      if ( (val = resCount.get(res[i])) == null)
        resCount.put(res[i], new Integer(1));
      else resCount.put(res[i],
                new Integer(val.intValue() + 1));
      if (resCount.get(res[mostCommon]).intValue() <
        resCount.get(res[i]).intValue()) mostCommon = i;
    }
    return res[mostCommon];
  }

  public static void main(String[] args) {
    try {
      grp = (server_group)MulticastNaming.lookup(URL);
      typename1[] r = grp.methodname1();
      System.out.println("I vote for: "+voting(r));
    } catch (Exception e) {
      System.out.println("Client: "+e.getMessage());
      e.printStackTrace();
    }
  }
}
```

6.1.2. Replication.

All group members implement the same services with the same code. Replicas have the task to spare clients from system failures, i.e., from the impossibility of getting the service result. Our approach allows the group to specify this kind of operating mode.

```
public class Member1 extends MulticastRemoteObject {
  public Member1() { super( fault_tolerant ); }
}
```

All the computed values are supposed to be the same and it is not important which result has to be considered the right one. Hence, the client considers the first, i.e., the one indexed by zero, returned value as correct.

```
public static void main(String [] args) {
  try {
    grp = (Member_Group)MulticastNaming.lookup(URL);
    typename1[] r = grp.methodname1();
    System.out.println ("The result is: "+r[0]);
  } catch (Exception e) {
    System.out.println ("Client: "+e.getMessage());
    e.printStackTrace ();
  }
}
```

6.2. Distributed MergeSort.

To prove that our multicast remote method invocation is not limited to handle services' fault tolerance, we show how to use it to sort a bulky array, parallelizing the classical *mergesort* algorithm [22].

The sequential mergesort algorithm is based on a divide and conqueror approach, the array to be sorted is split into two slices and then again up to have slices composed of only two elements at most. Then, slices are sorted and merged backward into a single array again.

A simple parallelization of this algorithm consists of entrusting each slice to an object group. Each member of the group work on an half array, slicing it again and demanding to another group the job. When the slice can not be split again the object sorts the array and returns it to the calling object which gathers and merge the resulting slices.

Hence, the application is composed of many groups ($\log_2 n$ groups, where n is the dimension of the array) composed of two members. Each member implements the interface `MSUInterface`, whereas the groups is described by `MSUInterface_group`.

```
public interface MSUInterface extends Remote {
  Integer [] MergeSort(Integer[] a) throws RemoteException;

public interface MSUInterface_group extends Remote {
  Integer [][] MergeSort(Integer[] a) throws RemoteException;
```

The class `MergeSortUnit` describes the objects belonging to the group which realizes the mergesort algorithm. The class constructor links the just created instance to the group which will handle the next step of the algorithm. `MergeSort` is a multicast method, i.e., defined in the group interface and activated by a multicast remote method invocation. It ignites a new step in the algorithm, slicing the array and forwarding it. It also merges the slices.

```
public class MergeSortUnit extends MulticastRemoteObject
                           implements MSUInterface {
  private boolean higher = false;
  private MSUInterface_group wks;

  public MergeSortUnit(String [] v) {
    if (v[0].equal(HIGHER)) higher = true;
    try {
      wks =
        (MSUInterface_group)MulticastNaming.lookup(v[1]);
    } catch(Exception e) {
      System.out.println ("Group not found!");
      e.printStackTrace ();
    }
  }

  private Integer [] merge(Integer a[], Integer b[]) {...};
  private Integer [] extract( Integer a[], int d1, int d2) {...};

  public Integer [] MergeSort(Integer[] a)
                              throws RemoteException {
    if (a.length > 2) {
      int dimSlice = a.length /2;
      if (!higher) dimSlice += a.length%2;
      Integer [] slice = new Integer(dimSlice);
      if (higher) slice = extract (a, 0, dimSlice);
      else slice = extract (a, a.length /2+1, a.length);
      Integer [][] res = wks.MergeSort(slice);
      a = merge(res[0], res[1]);
    }
    if ((a.length == 2) & (a[0] > a[1])) {
      int tmp = a[0];
      a[0] = a[1];
      a[1] = tmp;
    }
    return a;
  }
}
```

Each group member plays both the client and the server role. It is a server since it provides the service `MergeSort` to other groups and is a client because to carry out such a service needs the same service from another group. In the `main`, the object joins the corresponding group. Groups are created by the first group which receives the array to be sorted.

```
public static void main(String [] args) {
  System.setSecurityManager(new RMISecurityManager());
  String server = args [0];
  try {
    MergeSortUnit MSUnit = new MergeSortUnit(args);
    java.rmi.MulticastNaming.join(args [1], MSUnit);
  } catch (Exception e) { ... }
}
```

	msec	speed up		msec	speed up
Java RMI	160	1	Filterfresh[2]	210	0.76
JP/KaRMI	132	1.21	Jgroup	166	0.96
Multi-RMI	153	1.05			

[2] Data have been extrapolated from results reported in [3].

Table 1. Performances evaluation.

6.3. Performance Evaluation.

At the moment, we have carried out only some simple tests to estimate the overall performance of our system. Our tests consist of building a group with a fixed number of members (in Table 1 we report only the case of two members) offering a null service, i.e., implementing a method which does nothing. Then we have estimated how long that service takes to be served. Beyond Jgroup and Filterfresh — that support object groups (see section 7) —, we have also involved the standard Java unicast remote method invocation and JavaParty/KaRMI [17, 18], which implements an efficient RMI mechanism for Java, in our tests. When the tested framework did not support the object group abstraction we have modified the test considering a client which sequentially asks two or more servers for the null service.

Table 1 summarizes our measurements. The speed up ratio is related to standard Java RMI. The slowing down of Jgroup, and Filterfresh is due to several reasons: they have a decision phase before returning the answer, more latency in request propagation to each member, they realize reliable communications, and so on. From this simple test emerges that the choice of demanding both the selection and the computation of the service result to the client could be the choices improving the performance of dealing with object groups.

7. Related Work

In the last few years, the problem of integrating the group communication paradigm with distributed object technologies such as Java RMI [20] has been the subject of intense investigation. Many Java-based frameworks supporting object groups have been developed. FilterFresh [3], iBus [12], and Jgroup [13, 14] are significant results due to those investigations.

iBus [12] is a commercial product written in Java and aimed at supporting intranet applications such as content delivery systems, groupware and fault-tolerant client/server systems. Its architecture does not integrate the group communication paradigm with the standard Java RMI architecture; instead, it is based on the concept of multicast channels mapped on IP multicast groups. Clients can subscribe to multicast channels and can push and pull messages over the subscribed channel.

Filterfresh [3] and Jgroup [13, 14] share the same goal, i.e. the integration of the group communication paradigm with the Java distributed object model. Due to the constraints inherent to Java RMI, the approaches they follow are similar: both offer a reliable invocation mechanism for remote object groups composed by a collection of remote objects that cooperate through a GCS, and a distributed implementation of the RMI registry.

Due to the fact that their main goal consists of using object groups to provide a reliable communication system, both Filterfresh, and Jgroup consider only object groups composed of replicas and not of cooperating objects. Hence, they get more transparency from client's point of view, which handles the group as a single remote object, losing on flexibility, the client does not get all the values elaborated from each group member, hindering, for example, the processor farm parallelism.

Jgroup has also been integrated with the Jini technologies [1] with the same goals getting similar achievements and limitations [15].

8. Conclusion and Future Works

At the moment, we have a working tool which provides a mechanism for Java to support object groups, groups communications, and task farm parallelism. Our package can be downloaded from `http://www.disi.unige.it/person/CazzolaW/sw/multi-rmi.tar.gz`. The tool is simple and completely based on features available since Java version 1.2. As future works we are interested in moving our package to support Java proxies available since version 1.3 and nonblocking communication primitive available since version 1.4. Proxies help in rendering more clean the implementation freeing programmers from using a non standard `rmic` tool to compile stubs and skeletons (avoiding also problems with Java class dynamic loading and security checking). Moreover, by using nonblocking communications improve performances because we can broadcast the invocation to every group member at the same time. We will also improve the transparency of the approach in case of object groups used to provide high available and reliable services (i.e., services based on object replication) like Jgroup and Filterfresh. We are also integrating this multicast remote method invocation with our reflective middleware mChaRM [6].

References

[1] K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification*. The Jini Technology

Series ... from the Source. Addison-Wesley, Reading, Massachusetts, 1999.

[2] A. Avižienis. The N-Version Approach To Fault Tolerant Software. *IEEE Trans. Softw. Eng.*, 11(12):1491–1501, Dec. 1985.

[3] A. Baratloo, P. E. Chung, Y. Huang, S. Rangarajan, and S. Yajnik. Filterfresh: Transparent Hot Replication of Java Server Objects. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'98)*, Santa Fe, New Mexico, Apr. 1998.

[4] K. P. Birman. The Process Group Approach to Reliable Distributed Computing. *Commun. ACM*, 36(12):36–53, Dec. 1993.

[5] K. P. Birman and R. Van Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, 1994.

[6] W. Cazzola. mChaRM: Reflective Middleware with a Global View of Communications. *IEEE Distributed System On-Line*, 3(2), Feb. 2002. ISSN: 1541-4922. Available at `http://dsonline.computer.org/middleware/articles/dsonline-mcharm.html`.

[7] W. Cazzola, M. Ancona, F. Canepa, M. Mancini, and V. Siccardi. Shifting Up Java RMI from P2P to Multi-Point. Technical Report DISI-TR-01-13, DISI, Università degli Studi di Genova, Dec. 2001. Available at `http://homes.dico.unimi.it/~cazzola/cazzolawbib-by-year.html`.

[8] M. Chérèque, D. Powell, P. Reynier, J.-L. Richier, and J. Voiron. Active Replication in Delta-4. In *Proceeding of the 22nd IEEE International Symposium on Fault Tolerant Computing (FTCS-22)*, pages 28–37, Boston, Massachusetts, USA, July 1992. Computer Society Press.

[9] R. Guerraoui, P. Felber, B. Garbinato, and K. Mazouni. System Support for Objects Groups. In *Proceedings of the 13th ACM Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, volume 33(10) of *Sigplan Notice*, Vancouver, British Columbia, Canada, Oct. 1998.

[10] A. J. G. Hey. Experiments in MIMD Parallelism. In E. Odijk, M. Rem, and J.-C. Syre, editors, *Proceedings of Parallel Architectures and Languages Europe (PARLE'89)*, LNCS 366, pages 28–42, Eindhoven, The Netherlands, June 1989. Springer-Verlag.

[11] L. Liang, S. T. Chanson, and G. W. Neufeld. Process Groups and Group Communications: Classifications and Requirements. *IEEE Computer*, 23(2):56–66, Feb. 1990.

[12] S. Maffeis. iBus - The Java Intranet Software Bus. Technical report, Olsen and Associates, Apr. 1997.

[13] A. Montresor. A Reliable Registry for the Jgroup Distributed Object Model. In *Proceedings of the Third European Research Seminar on Advances in Distributed Systems (ERSADS '99)*, Madeira, Portugal, Apr. 1999.

[14] A. Montresor. The Jgroup Reliable Distributed Object Model. In *Proceedings of the 2nd IFIP WG 6.1 Int'l Working Conference on Distributed Applications and Interoperable Systems*, Helsinki, Finland, June 1999.

[15] A. Montresor, R. Davoli, and Ö. Babaoğlu. Enhancing Jini with Group Communication. In *Procedings of the ICDCS Workshop on Applied Reliable Group Communication (WARGC 2001)*, Phoenix, Arizona, USA, Apr. 2001.

[16] L. E. Moser, M. P. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System. *Commun. ACM*, 39(4):54–63, Apr. 1996.

[17] C. Nester, M. Philippsen, and B. Haumacher. A More Efficient RMI for Java. In *Proceedings of ACM 1999 Java Grande Conference*, pages 152–157, San Francisco, California, June 1999.

[18] M. Philippsen and M. Zenger. JavaParty - Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.

[19] N. A. Speirs and P. A. Barrett. Using Passive Replicates in Delta-4 to Provide Dependable Computing. In *Proceeding of the 19th IEEE International Symposium on Fault Tolerant Computing (FTCS-19)*, pages 184–190, Chicago, NJ, USA, June 1989. Computer Society Press.

[20] SUN Microsystems. Java™ Remote Method Invocation - Distributed Computing for Java. White paper, SUN Microsystems, 1998. Internet Publication - `http://www.sun.com`.

[21] R. Van Renesse, K. P. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Commun. ACM*, 39(4):76–83, Apr. 1996.

[22] N. Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, Englewood Cliff, 1st edition, 1976.