Walter Cazzola, Massimo Ancona, Fabio Canepa, Massimo Mancini, and Vanja Siccardi. Shifting Up Java RMI from P2P to Multi-Point. Technical Report DISI-TR-01-13, DISI, Università degli Studi di Genova, December 2001.

Shifting up Java RMI from P2P to Multi-Point

Walter Cazzola, **Massimo Ancona**,
Fabio Canepa, **Massimo Mancini**, and **Vanja Siccardi**

DISI, University of Genova, Italy
{cazzola|ancona}@disi.unige.it

Abstract

In this paper we describe how to realize a Java RMI framework supporting multi-point method invocation. The package we have realized allows programmers to build groups of servers that could provide services in two different modes: fault tolerant and parallel. These modes differ over computations failure. Our extension is based upon the creation of entities which maintain a common state between different servers. This has been done extending the existing *RMI registry*. From the user's point of view the multi-point RMI acts just like the traditional RMI system, and really the same architecture has been used.

1 Introduction

The Java framework already supports point-to-point (P2P) communication through the RMI mechanism. In the version shipped with the `jdk`, there is only the possibility for a peer-to-peer communication model. This does not fit well with the needs emerging from a distributed system, i.e. higher resources availability systems. Within this background we developed an extension to the usual communication model which permits a multi-point communication model. Multi-point communications help in dealing with many situations, e.g., servers failure tolerant, or data-parallel programming. However, multi-point communications among a client and many servers can be interpreted as many P2P communication between such a client and each of those servers. Results are collected into an array, which is returned to the client. A multi-point communication model treats the server side as a unique entity (like a *group*).

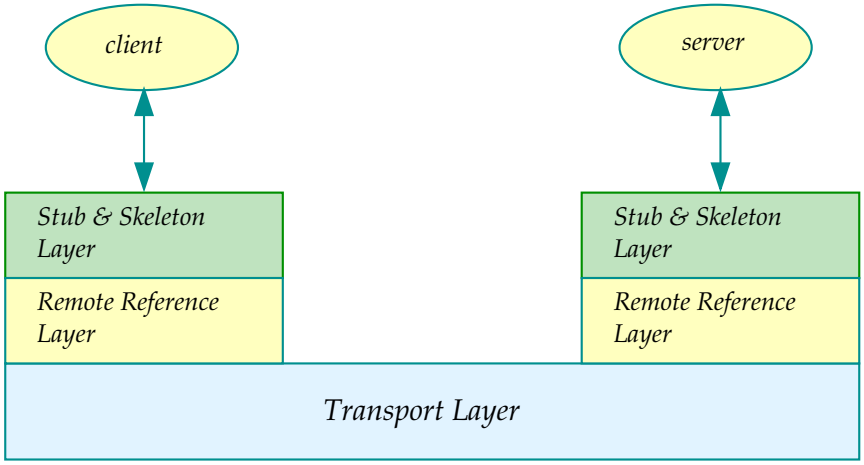
2 RMI architecture

2.1 Java Distributed Object Model: Overview

The Java RMI framework permits to build distributed applications where components communicate with each other by messages exchange. This is done coherently with the Java programming framework. Communication details are handled transparently by the framework exempting the software developer from directly handling the access to the resources of the remote host. A distributed application can be realized easily, keeping the focus only on the program logic and forgetting

all the low-level details needed for establishing a remote communication between two (or more) computers.

The RMI framework is composed of three layers of abstraction (see the figure below), which cooperate for the realization of the remote communication. Each layer is independent, and can be replaced without affecting the other layers.



Remote Method Invocation System

2.1.1 Stub & Skeleton Layer.

The *stub & skeleton layer* represents the interface between the components of the application and the rest of the RMI system. Fundamental components of the stub & skeleton layer are the *stub*s. They work on the client side playing the role of a proxy to the remote object implementation. Stubs are automatically created by compiling the related remote service, and keep an internal representation of the remote object. They handle remote method calls as defined by the remote interface they implement.

Here in detail their tasks:

- to marshal input data, preparing them to be transmitted on the communication channel,
- to forward the call towards the remote server, which will carry out the requested methods,
- to unmarshal the received data to the expected output format, and
- to return the result back to the client

The stub is a Java class generated by the `rmic` preprocessor out of a remote service implementation. The stub implements the same interface as the remote

object, and acts on the client side. In `jdk` version 1.1 there is the need for a complementary server side entity, called *skeleton*. Basically, skeleton duties consist of dispatching incoming requests from remote stubs to the appropriate remote objects implementations, and of passing the results back to the stubs. Its presence in the communication protocol used in `jdk` version 1.2 is no more necessary because the server side class hierarchy — e.g., `RemoteObject`, `UnicastRemoteObject`, and so on — used by the remote services are imbued with the same functionalities of the skeletons.

2.1.2 Remote Reference Layer.

The *remote reference layer* defines and implements the remote invocation semantic. Objects defined in this layer realize the link with the implementation of the remote service. In traditional RMI there is only the possibility for a P2P communication between objects. As such, only one representation of the remote link is defined and implemented: the `UnicastRef` class which defines the semantic of P2P communication. To provide a P2P remote service, an object must simply extend the `UnicastRemoteObject` class.

2.1.3 Transport Layer.

The *transport layer* links the Java Virtual Machines (JVM) involved in the distributed application each other. The communication between JVMs is done by TCP/IP connections using a proprietary stream-based protocol called Java Remote Method Protocol (JRMP). With `jdk` version 1.3, it is possible to use a new version of the RMI library called RMI/IIOP which does not use JRMP protocol but the standard protocol IIOP allowing the integration between CORBA and RMI objects.

So far, we have talked of remote references, but we did not specify how to get such items. Talking at a different abstraction level we can introduce here the `Registry`, a name server maintaining associations between services names and remote stubs. The registry is an entity which runs on any machine hosting one or more remote services. Every service is locally bound to the registry. This is not mandatory since the naming service is a straightforward and easy way to obtain a remote service hook, but not the only way. We can pass remote services references from an object to another by using methods, or whatever means the environment supplies.

2.2 Interactions among RMI components in an hypothetic application.

❶ Definition of the remote interface which clients and servers refer to

❷ Definition of the server-side implementation of the service

❸ Generation of the stubs and skeletons by means of the `rmic` preprocessor

❹ Creation of a server that implements the remote service. Exportation and creation of the stub for the service

❺ Registration of the service in the local registry

❻ Lookup of a remote service. A client queries the registry for a particular service, the registry supplies the corresponding stub to the client

❼ Remote method invocation. The client invokes the method. The stub sets the connection with the remote server, carries out marshaling, request of the execution of the method, unmarshaling of the answer and finally returns the result to the client

3 Multi-Point RMI Overview

In the RMI system the P2P communication mechanism is suitable for modeling client-server applications. Notwithstanding that, there are requirements which cannot or are difficult to be achieved by this communication model — e.g., servers reliability and availability.

The goal of our project consists of extending the Java system by putting besides the P2P RMI a one-to-many communication mechanism. In this communication model, a service request is forwarded to several similar severs supplying that service. In our architecture a group of objects supplying the same service is defined and is accessible by a common tag. Every node in the network involved in the group knows all the required data for a distributed interrogation, so they are not centralized. Changes in the group are dynamically spread over the network.

3.1 Multi-Point RMI architecture

The multi-point RMI system, we have implemented, uses the same architecture as the traditional RMI. Hence, we continue to have the remote reference, the stub *&* skeleton, and the transport layers. A new layer, called *group layer*, has been put besides the remote reference, and the stub *&* skeleton layers.

3.1.1 Remote Reference Layer.

The remote reference layer has been extended to deal with the new remote invocation semantic. The new multi-point type for a service reference has been implemented by the `MulticastRef` class. Each instance of this class represents a group, i.e., a reference to a list of servers belonging to such a group.

3.1.2 Group Layer.

A group is a composite entity whose components are objects providing the same services. Each object joins to the group through specific primitives. Information about the group are kept and continuously updated by each registry in the system. Registries use some synchronization primitives to notify each other changes occurred to the group (for example, a member has stopped its services, a new member has joined in, and so on).

The result of a service provided by a group is the collection of the results that each member of the group provides for that service. Hence, the group interface and the interfaces of its members have the same declared service but they differ in the return values.

Thus, the interface describing the service provided by a member of the group will look like:

```
public interface RemoteInterface extends Remote {
    typename1 method_name1() throws RemoteException;
}
```

Whereas the group interface describing the services provided by the group will look like:

```
public interface RemoteInterface_Group extends Remote {
    typename1[] method_name1() throws RemoteException;
}
```

3.1.3 Stub & Skeleton Layer.

The stub & skeleton layers has been modified in order to handle the group interface. Whereas, the traditional scheme is characterized by the matching of the two interfaces, the multi-point scheme is not. At the stub & skeleton layer one has to deal with this, being the stub the meeting point between the two interfaces. Luckily for us, the only difference is represented by the return value. The `rmic` preprocessor is in charge of generating these extra stubs. Hence, to support the group our coding also affects this component.

3.1.4 Transport Layer.

The transport layer has not been modified.

3.2 The Multi-RMI API

In this section, we show, on snippets of code, the API to handle group creation and multi-point interactions. We have extracted the snippets from the banking system described in section 5.

3.2.1 Definition of Server Side Interface.

Description of the services provided by remote servers.

```
public interface Account extends Remote {
    // This method returns the current balance of the account
  Integer balance() throws RemoteException;
}
```

An interface like this has to be implemented by each object which wants to provide remote services.

3.2.2 Definition of the Group Interface.

Services provided by a group have to be described by an aggregative interface, which reassembles the interfaces implemented by each object belonging to that group.

```
public interface Account_Group extends Remote {
    // This method returns all the balance of the remote servers belonging to the group
  Integer [] balance() throws RemoteException;
}
```

3.2.3 Server Definition & Creation.

Each object which will be part of a group has to extends the `MulticastRemoteObject` (as shown in the following snippet of code). At this point, only information about the server which has created the stub are kept at the remote reference.

```
public class Bank1
      extends MulticastRemoteObject
      implements Account {
            [...]
}
```

Remote servers are created in the usual way.

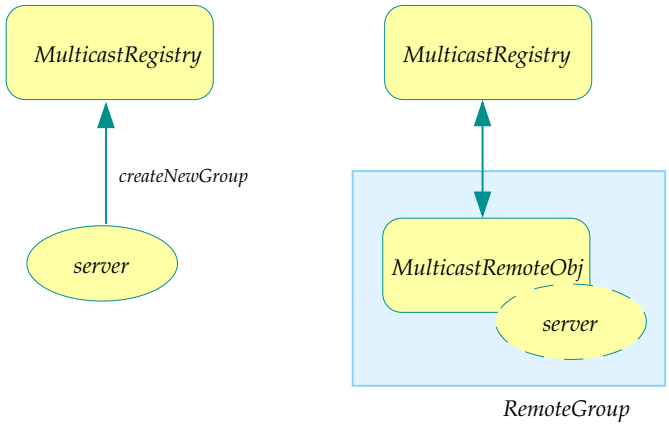
```
Bank1 bank = new Bank1();
```

3.2.4 Group Creation.

The group is registered in the local registry of the server with the association «`newGroupName, Stub`» through a call to the `createNewGroup` method.

```
java.rmi.MulticastNaming.createNewGroup("rmi://"+host+"/Bank", bank);
```

Where `host` represents the host of the server which creates the group, and `rmi://host/Bank` is the URL where the first reference of the group is located. Hence, a group creation takes two steps:



a. group creation *b. group registration*

Figure 1: **How a group is created.**

❶ a server asks the multicast registry for the creation of a new group (Fig. 1.a), then

❷ the new group is registered in the multicast registry as a group composed only of the server originating it (Fig. 1.b).

3.2.5 Look up of a Group Service.

Clients get a representative of the group through a call to the `MulticastNaming.lookup` method. Through this representative the client accesses to the group services.

```
public class Client {
    static Account_Group bank;

    public static void main(String[] args) {
                                [...]
        bank = (Account_Group) MulticastNaming.lookup("rmi://"+server+"/Bank");
                                [...]
    }
}
```

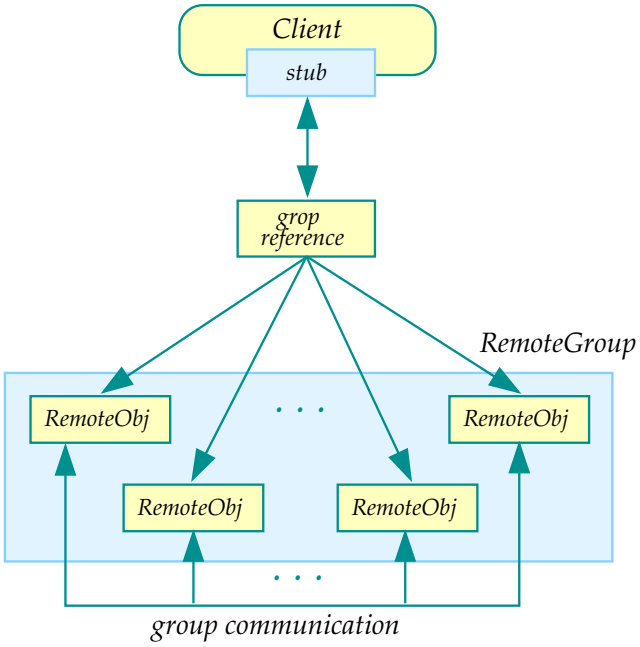


Figure 2: **How a request to a group takes place.**

The representative, that the client gets, is directly bound to the server that has registered the group.

3.2.6 Requesting a Service Provided by a Group.

Getting a service from a group is carried out through the group representative similarly to a P2P remote invocation.

```
public class Client {
    static Account_Group bank;

    public static void main(String [] args) {
                    [...]
        Integer [] balances = bank.balance (); // asking the state of accounts
    }
}
```

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. If a failure happens, the behavior depends on the semantic adopted for that multi-point communication (see section 4.1).

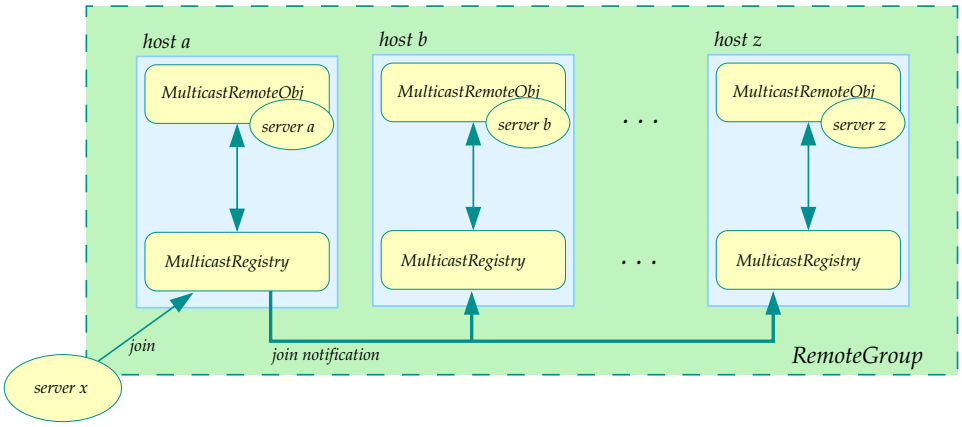


Figure 3: **The remote server `X` asks to join a group.**

3.2.7 Join to or Disjoin from a Group.

Servers ask to a registry to join to (or to disjoin from) a group using the method `join` (or `disjoin`). Such a registry notifies the changes to all the other registries hosting a member of the group. After the notification each registry updates its databases.

```
Bank bank = new Bank();
java.rmi.MulticastNaming.join("rmi://"+remoteserver+"/Bank", bank);
java.rmi.MulticastNaming.disjoin("rmi://"+remoteserver+"/Bank", bank);
```

Figure 3 shows how a server joins to a group.

4 Multi-point RMI interfaces and classes

Here follows a general explanation of the classes that have been added or modified in order to implement the multi-point RMI. Classes are described following the order given in section 3.

4.1 Remote Reference Layer

4.1.1 The `MulticastRemoteObject` class.

The `MulticastRemoteObject` class defines a composite remote object whose references are valid only while the server process is alive. The `MulticastRemoteObject` 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.

- *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 are down. Whereas operations carried out in *fault_tolerant* mode when no server in the group can give an answer.

If the mode is not specified calling the constructor of the remote servers, the default behavior is set to *parallel*. 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, then, it has to provide by itself the correct semantics of the hashCode, `equals`, and `toString` methods inherited from the `Object` class, so that they behave appropriately for remote objects and group of remote objects.

The main method of this class is:

```
RemoteStub exportObject(Remote obj)
```

which exports the remote object passed as argument. To render it available to receive the incoming calls, using an anonymous port, it builds and returns a `RemoteStub` using the method `exportObject` of the class `MulticastServerRef`.

4.1.2 The `MulticastServerRef` class.

`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_tolerance*. When `mode` is not specified, this attribute is set to *parallel* as a default. In order 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_tolerance*.

4.1.3 The `MulticastRef` class.

MulticastRef implements the multicast client-side group remote reference, with the *parallel* semantic. The class contains the list of server references composing the group. The list is stored in the attribute `ref`.

`invoke` is the main method of the class. Its behavior consists in executing the remote method call on every server in the group. Its return value is an array containing the results of each remote method invocation. A `RemoteException` is thrown if *at least one of the calls fails*. If the remote invocation throws an exception, then an application-level exception is also thrown. To marshal the data for 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 beginning, it serializes the number of

servers belonging to the group (*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.

4.1.4 The `MulticastRefFaultTolerance` class.

`MulticastRefFaultTolerance` implements the multicast client-side group remote reference with the *fault-tolerant* semantic.

This class extends `MulticastRef` overriding the method `invoke`. The return value is again the array containing the results of each method invocation. The difference with the parent class lies in the exception mechanism. `MulticastRefFaultTolerance` throws an exception, namely a `ZombieGroupRemoteException`, if *all the calls fail* or an application-level exception if the remote invocation throws an exception.

4.1.5 The `UnicastServerRef` class.

This class is identical to the P2P version, but for scoping troubles some fields and methods have been defined as **protected** instead of **private**.

4.2 Group Layer.

4.2.1 The `MulticastNaming` class.

The `MulticastNaming` class — analogously to the P2P `Naming` class — provides methods for storing and retrieving references to remote objects in the remote registry.

Binding a name to a remote object means associating a name for it. Such a name will be used to look up the object. A remote object can be associated with a name using `bind` or `rebind` methods of the `MulticastNaming` class. 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 `lookup` method), get its reference, and then invoke its methods. 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 servers running on a host or an individual server process may create and use its own registry if desired. Methods of this class use services supplied from the registry defined in the `MulticastRegistry` interface and implemented in the `MulticastRegistryImpl` class. New methods have been added for dealing with groups: `createNewGroup`, `join` and `disjoin` methods. Moreover, the `MulticastNaming` class 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.

4.2.2 The `MulticastRegistry` interface.

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 the methods defined by `UnicastRegistry` and some other methods needed to obtain the multicast behavior. Typically a registry exists on every node running remote servers. Every server belonging to a multicast group *must* register to a multicast registry. A multicast registry is also unicast compliant, in the sense that it can handle P2P calls as well.

A registry on a particular node contains a database that maps group names to the object belonging to the 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 reduce name collisions.

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

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

When a server is joined (or disjoined) to a group all the registries keeping entries for the group need to be informed of the change. This is accomplished by the method `update`.

4.2.3 The `LocateMulticastRegistry` class.

`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`.

4.3 Stub & Skeleton Layer

No modifications have been done to the standard classes of the stub & skeleton layer. The Remote reference inside the stub is a `RemoteRef` object as in the *classic* RMI framework. At run-time, its subclass `MulticastRef` is used. The only modifications we needed to do are related to the `rmic` preprocessor, which must generate stubs having arrays as return values of group services. To do that, we have modified some parts of the `Generator` class.

4.3.1 The `Generator` class.

A Generator object generates the Java source code of the stub class for a remote server, parsing its source code. We have modified the generator to enable it to deal with multi-point RMI generating the appropriate stubs.

We have introduced the attribute `isMulticast` to distinguish when to generate a *classic* stub instead of a *multi-point* one. Its value is internally handled in order to spare the user to set it manually. Basically, for each remote interface `X` the compiler look for a parent interface `X_group` in the current path. If found, we are in the case of a group interface and the attribute `isMulticast` is set. If not, we continue in the normal way.

Another issue we had to tackle was the generation of the hash key indexing remote method invocations. The hash key changes with the method prototype. We changed the method prototype (because of the return value) so we had to reflect this change in the hash key generation as well.

5 Multi-Point RMI at Work.

As an example of using the multi-point RMI library, an application which models a simple banking system has been implemented. A bank supplies some services like money withdrawal and deposit on a bank account. Due to the intrinsic necessity of these operations to be reliable we have chosen to replicate the bank server. In particular this application shows an example of *software fault tolerance*. The replicated bank objects implement the same services, but with different code in order to spare clients from logical errors. Each version of the bank returns its balance for the involved account, and the client will receive, as correct balance, the most frequent value (i.e., by voting).

The remote object interface representing the bank is:

```
public interface Account extends Remote {
    // returns the balance of the account
  Integer balance() throws RemoteException;
    // withdraws «sum» from the account
  Boolean withdraw(int sum) throws RemoteException;
    // deposits «sum» in the account
  void deposit(int sum) throws RemoteException;
}
```

This interface describes the methods provided by all the versions of the `Bank`. As explained in section 3.2, to use the multi-point RMI facility we need both a server interface and a group interface.

```
public interface Account_Group extends Remote {
    // gathers and returns all the pretended balance of the account
```

```
    Integer [] balance () throws RemoteException;
      // withdraws «sum» from all the copies of the account
    Boolean[] withdraw(int sum) throws RemoteException;
      // deposits «sum» in all the copies of the account
    void deposit (int sum) throws RemoteException;
}
```

The interface `Account_Group` defines the same methods defined in the `Account` interface; the only difference is in the return values, that are arrays of the original type — e.g., Integer[] instead of Integer — containing the results of the computation of each server.

The following class represents the bank.

```
// this class realizes a bank, the balance of an account is calculated as the difference
// between the deposits and the withdraws.

public class Bank1 extends MulticastRemoteObject implements Account {
  private int depositSum;          // total deposited amounts
  private int withdrawSum;         // total withdrawn amounts

  // the constructor does not specify whether the object is or not fault tolerant,
  // so it is used with the default semantic: «parallel».
  public Bank1() throws RemoteException { super(); }

  // it returns the current balance of the account, calculated as:
  // «depositSum − withdrawSum».
  public Integer balance () throws RemoteException {
    return (new Integer(depositSum − withdrawSum));
  }

  // it withdraws «sum» euros from the account and updates the «withdrawSum» attribute.
  // it returns true if the current balance covers the requested withdraw.
  public Boolean withdraw(int sum) throws RemoteException {
    if ( (this.balance (). intValue () − sum) > 0) {
      withdrawSum += sum;
      return new Boolean(true);
    } else return new Boolean(false);
  }

  // it deposits «sum» euros in the account and update the «depositSum» attribute.
  public void deposit (int sum) throws RemoteException { depositSum += sum; }

  public static void main(String [] args) {
    String server = args [0];
    System.setSecurityManager(new RMISecurityManager());
    try {
      Bank1 bank = new Bank1();
      java.rmi.MulticastNaming.createNewGroup("rmi://"+server+"/Bank", bank);
```

```
        } catch (Exception e)    {
          System.out. println ("Bank1 error: " + e.getMessage());
          e. printStackTrace ();
        }
    }
  }
```

The class `Bank1` implements the normal behavior of a bank. `Bank2`, and `Bank3` extend `Bank1` giving an alternative implementation of its methods. We do not show them, because their code do not add information to the exposition.

The following code shows how a client accesses to the bank services. The client executes some deposits to and withdrawals from its account (lines 32, and 33), then it checks his statement of account invoking the method `balance` (line 35). The `BankClient` code also contains a *software failure detection* mechanism (the method `findRightResult` defined at line 8) which chooses the most common returned value from method `balance`.

```
                /* simple interactions with the banking system */
public class BankClient {
    // ≪bank≫ is the reference used to refer to the remote multicast object
    // that implements the ≪Bank_Group≫ interface.
  static Account_Group bank;

      /* it returns the most common value in ≪res≫ */
  private static Integer findRightResult ( Integer [] res ) {
    Hashtable resCount = new Hashtable ();
    int mostCommon = 0;

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

    // This method interacts with a remote multicast object of group ≪Bank≫.
    // It carries out some deposits and some withdrawals, then it asks the
    // statement of account. The right value is determined by majority .
  public static void main( String [] args ) {
    System.setSecurityManager(new RMISecurityManager());
    String server = args [0];
    try {
        /* looks up the multicast remote object representing a bank */
```

```
            bank = (Account_Group) MulticastNaming.lookup("rmi://"+args[0]+"Bank");
                /* it carries out some transactions */
            for (int i = 0; i < 10; i++) bank.deposit(i * 10);
            for (int i = 0; i < 5; i++) bank.withdraw(i * 2);
                /* ask for the statement of account */
            Integer [] balances = bank.balance();
                /* it determines which is the right statement of account */
            System.out.println("The statement of account is "+\
                               +findRightResult(balances)+" euros.");
        } catch (Exception e) {
            System.out.println("BankClient connection error" + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

6 Conclusion

From some naive tests we have discovered that our multi-point RMI mechanism carries out a remote request to a group of ten servers saving the 4% of time with respect to use ten P2P RMI. Our system is available at `http://www.disi.unige.it/person/CazzolaW/sw/multi-rmi.tar.gz`.

References

[1] jGuru. Fundamentals of RMI (Short Course). Available on `http://developer.java.sun.com/developer/onlineTraining/rmi/RMI.html`.

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