# Adding Group Communication to Java in a Non-Intrusive Way Using the Ensemble Toolkit

Bela Ban
Dept. of Computer Science
Cornell University
bba@cs.cornell.edu

Nov 11 1997

# Contents

# Chapter 1

# Introduction

This paper describes a possible approach to merge the work done at Cornell (Isis, Horus, Ensemble) [Bir96] with the Java environment.
Java is currently lacking group communication facilities, communication both on the socket- and RMI-level [RMI96] is only one-to-one rather than one-to-many[1]. Adding group communication to Java would greatly enhance the range of applications that can be built in Java. Applications that have to be highly reliable and robust have to-date to be created using traditional languages such as C/C++. Merging the proven frameworks developed at Cornell with the power and dissemination of the Java environment would combine the best of both worlds. Additionally, the visibility of the Ensemble toolkit would be increased. As a side effect, important knowledge can be gained with respect to integration of Ensemble into existing environments, possibly in a non-intrusive way. The lessons learned may lead to a better understanding of what is required of high-level toolkits / frameworks for group communication.
The approach outlined below consists of (1) a class compiler which takes existing Java classes and enhances them with group communication capabilities, (2) a generic (reflective)[Gra97] Java dispatcher which accepts any method to be executed on a Java object, performs the operation and returns the result, and (3) the Ensemble runtime system which has been adapted to Java[2].

[1] Disregarding MulticastSockets which ...
[2] This involves either a complete Java system, or access via JNI.

# Chapter 2

# Implementation

## 2.1 Class Compiler

The task of the class compiler is to take an existing Java class file, copy it, parse it and generate an enhanced version (either `.java` or `.class`) which contains essentially the Maestro methods in addition to the original methods. The modified class can be used in the same way as the original, but will use Ensemble to send methods to a *group of receivers* instead of just a single receiver. If the group view is empty (that is, the object to which the request is sent is the only member of the group), then Java's Core Reflection API is used to forward the request to the method's implementation in the same object.
An example of a Java class is shown in fig. 2.1.
It contains 2 methods, `foo` and `bar`. The class compiler now parses the class file[1]

[1]It can use either the Core Reflection API, or read the class file's structure directly, as described in the JVM specification.

```
class X {
    public void foo() {
        // implementation
    }
    public String bar() {
        // implementation
    }
}
```

Figure 2.1: Class X

and generates a copy of the class file (shown in source code form in fig. 2.2), annotated with additional members and methods, in a different directory. That directory has to be listed somewhere in the `CLASSPATH` environment variable *before* the directory containing the original classes. This is to ensure that the modified classes are to be used. (Using this scheme makes it easy to switch back to the original classes by setting the `CLASSPATH` variable accordingly.)

```
class X {
    public  void foo() {
        // stub, calls Send()
    }
    private void foo_impl() {
        // original implementation
    }
    public String bar() {
        // stub, calls Send()
    }
    public String bar_impl() {
        // original implementation
    }

    /* Methods and members added by class compiler for Ensemble. */
    /* No exact signatures given: */
    private Vector view;
    void Send() {}
    void Receive() {}
    void AddMember(X x) {}
    void RemoveMember(X x) {}
    void SetState() {}
    void GetState() {}
    void ViewChange(View new_view) {}
    void HeartBeat() {}
}
```

Figure 2.2: Modified class X

All the member variables are copied to the new class file. For each method `X`, 2 methods are present in the new file: `X` and `X_impl`. `X` does not contain the implementation, but is instead a stub method which uses the Ensemble runtime to dispatch a request to all members of the group. `X_impl` is actually the original method, copied to the new file. The renaming ensures that existing clients of class `X` continue working using the same methods, but instead of calling a normal method, they are calling the group-aware method. The byte code for a method can be copied to another file by just looking up the method in the class file (the layout is defined in the JVM specification) and copying its body.[2] In the example, method `foo` will call `foo_impl` only when its view is empty (thus effectively behaving like the original class).[3] In all other cases it will use Ensemble to package the request in the form of an RMI or IIOP message, send it to all members of the group, receive the response, unpackage it and return the (first) result value (all other responses are discarded). Of course, handling of all responses could be made visible by adding another method for `X` which returns a list of results rather than just a single result (see [Maf95] for details of response collection). However, this would force applications that need explicit response collection to be modified to access the new method.

The private member `view` is a list of all members of the group. It will be updated whenever an object joins or leaves the group (this is ensured by the Ensemble runtime system).

Method `Send` is called by the stub methods which marshal the arguments into an RMI or IIOP packet which is then delivered to Ensemble (actually calling Ensemble's SEND function). The `Receive` method blocks while waiting for data from Ensemble (using the RECV function). When data is received, it will be converted to a response in the stub methods and the first response will be returned to the caller.

Method `AddMember` and `RemoveMember` can be called by the client directly to add / remove members to / from the group. This will trigger a view change in all members of the group, effectively calling `ViewChange` in all member objects. The difference between the former 2 methods and `ViewChange` is that the latter is always called by the runtime system, as opposed to the former methods which are always invoked by a client.

To obtain the state of existing members or set the state of a new member to the one of the group, methods `GetState` and `SetState` are called by the runtime to obtain or set state.

`HeartBeat` is used by the runtime to check for failures, e.g. crashed objects, network partitions etc.

Since the implementation of most of the Ensemble-related methods are generic,

[2] The technique of modifying class files was discovered by the author first in ODI's PSE Java ODBMS. Here, an object is made persistence-capable by modifying its inheritance structure and adding methods that handle persistence. The generated class (file) has the same name as the original one and has to be found in the `CLASSPATH` *before* the original class.

[3] For a more detailed discussion see section 2.2.2.

they could probably be moved to a separate class and each object would delegate this functionality to an instance of that class (*strategy* pattern [GHJV95]). This allows (1) to reduce the number of additional methods in an annotated class and (2) to replace the strategy object at runtime with a different one, effectively modifying the semantics of Ensemble.

### 2.1.1 Use Example

```
Directory dir, local_dir;
dir=Naming.lookup("//adl:5555/myDir"); // returns annotated proxy
dir.foo(); // unicast msg sent to single remote object

// Causes a ViewChange() msg to be sent to myDir and myDir2:
dir.AddMember(Naming.lookup("//bal:6665/myDir2"));

dir.foo(); // msg sent to 2 remote instances, returns 1st rsp.
local_dir=new Directory(); // creates local instance

// SetState() is called on local_dir
// Causes ViewChange() msg to be sent to remote objects myDir
// and myDir2 and local object local_dir
dir.AddMember(local_dir);

dir.foo(); // Sent to myDir, myDir2 and local_dir
```

Figure 2.3: Class X

## 2.2 Architecture

The architecture is shown in fig. 2.4. It will be discussed in the following sections.

### 2.2.1 Dispatcher

The dispatcher object waits for data to arrive on one of its sockets. When data arrives, it checks whether it is a response. In this case the response is returned to a caller waiting for it (and thus blocked). When the data is a request, then the

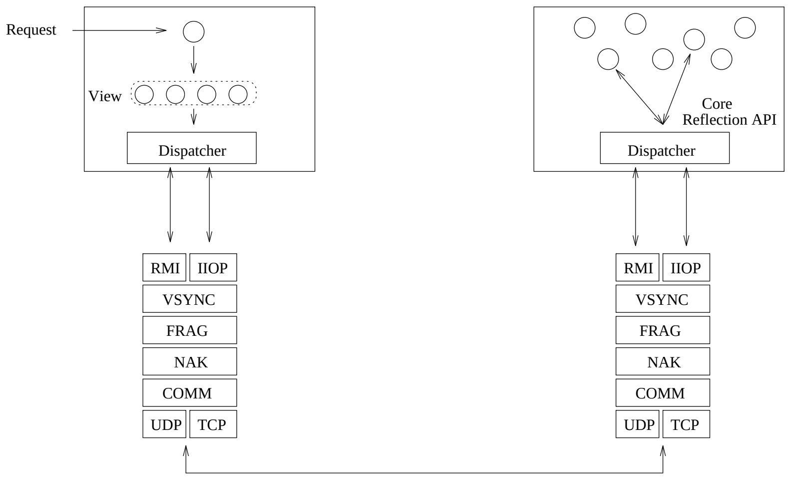


Figure 2.4: Overview of Architecture

request is unmarshalled and dispatched (on a separate thread) to the corresponding method, using the Java Core Reflection API.[4] Therefore, the dispatcher can be used both in a client and a server. In the client, the dispatcher invokes methods such as `HeartBeat`, `ViewChange` or `SetState`, as directed by the Ensemble runtime.

### 2.2.2 Stub- and Implementation Objects

*Stubs* are proxy references to objects residing in different address spaces, for example in a different process on the same machine, or on a different machine. They do not implement the object's functionality, but forward all method call to the remote object. It is *implementation objects* that actually provide the functionality.
Each object annotated by the class compiler has a tag saying whether it is a stub or an implementation (for example by adding a field to the annotated object). When for example the naming service returns an object, it will tag it as being a stub. When an object is a stub, it has also the OID of the remote object which describes where to find the remote object.
Depending on whether an object is a stub or an implementation, method invocations are treated differently: if the object is an implementation, calling method `foo` would call `foo_impl` directly.[5] If the object is a stub, `foo` would send a remote request to the implementation object. This has the advantage that there need not

[4] The dispatcher object has roughly the same functionality as a CORBA object adapter.

[5] As discussed in section 2.1, the original method `foo` was renamed to `foo_impl` and method `foo` was generated by the class compiler.

be any differences between stubs and implementations[6], and the same object can be used in a server as an implementation, and in a client's address space as a stub. This allows for example other objects in a server to invoke methods of the implementation in the same address space, which effectively calls the implementation of the functionality instead of sending a message to a remote object as in the case of stubs.
Of course, when a stub is returned as result of a call (e.g. `Naming.lookup`), then the state of the implementation object in the server for which the stub is created, does not need to be transferred over the network. This is so because access of fields of a stub object should – preserving the semantics of remote access – cause the state of the *remote object* to be returned, rather than that of the local stub, which is technically not possible.[7] The actual RMI implementation forbids access of remote state (through public attributes) by requiring a remote object to be a Java interface which is not allowed to have any attributes.
Therefore, remote state should only be accessed through accessor methods such as `getFoo` and `setFoo`. These could either be generated for each attribute (with the danger that the user has already written such methods), or it could be prescribed that no public attributes must be available in remote objects.

**Summary of Changes**

1. Class Modification
   (a) Addition of tag (e.g. a `BitField` to each class, specifying the type of object, e.g. stub or implementation.
   (b) Addition of a field to each class storing the OID of the remote object in case of a stub. Note that for saving space, e.g. in case of implementations, the OID field could be stored 'externally', e.g. in a hashtable (keys = object hash codes, values = OIDs), similar to CORBA's Property Service.
2. Additional Methods
   (a) `asImpl()` Generates a full copy of the object, containing all its state. This is used when the object at hand is a stub, but we need its full state. The method will access the remote object, and return a clone

[6] Actually, stubs and implementations are *roles* that can be changed by flipping the tag denoting the role.

[7] This would cause changes to the Java compiler. Note that in languages with a meta object protocol (such as CLOS), attribute access could be redirected to remote objects using the MOP.

of it. In case the object is an implementation, `asImpl` is the same as `clone`.

(b) `asStub()` Used when a stub for an implementation object is to be returned, e.g. by a naming service to a remote client. Calling the method on a stub simply returns the stub.

The default parameter passing semantics is to pass objects as stubs when they are user-defined, and as values when they are built-in (such as strings, sequences, integers etc). The 2 methods described above allow to change the default parameter passing semantics.

3. Method Invocation

   (a) Method `foo_impl` contains original functionality

   (b) Method `foo` is generated by class compiler. If the object is an implementation, it calls `foo_impl`. In case of a stub, it uses the dispatcher to send the request to the remote object.

### 2.2.3 Group Service

## 2.3 Dispatcher

The dispatcher has 3 major tasks:

1. Generation of an RMI request when called by an annotated object's `Send` method (see fig. 2.2). Encapsulation of RMI request in Ensemble and sending of request to remote dispatcher.

2. Reception of requests and stripping of Ensemble-related headers/trailers. Dispatching of RMI request to RMI functionality.

3. Ensemble-related activities. E.g. addition of a new member to a group triggers view changes which are propagated to all member objects of the group. This will essentially invoke a method on all objects in the group (`ViewChange`). Another example is a periodical heartbeat message sent to all members of the group.

   The dispatcher has to take care that these messages are exchanged between the corresponding objects in parallel to its other tasks.

Generation/dispatching of RMI request and wrapping/unwrapping in/from Ensemble is done using special Ensemble *layers* [Bir96] for RMI, as shown in fig. 2.4.[8] Using the partly opened-up design of RMI should allow to 'slide in' the Ensemble functionality between a Java RMI method invocation and its reception by the remote server (dispatcher).

### 2.3.1 RMI and IIOP Layer in the Ensemble Stack

[8] Note that additional layers might be written to accommodate CORBA's IIOP.

# Chapter 3

# RMI Revisited

This chapter describes how RMI can be implemented in a non-intrusive way.[1] The current implementation has quite some requirements to enable distributed Java objects:

- Any object that wants to enable its methods for remote execution has to be specified as an interface `X`, derived from `java.rmi.Remote`.
- The class implementing the interface has to a) provide functionality for the methods by implementing interface `X` and b) subclass `UnicastRemoteObject`.
- All parameters and return values of remote methods have to implement interface `Serializable` in order to be marshaled / unmarshaled to / from a byte stream over the network. There is no actual need to do this since the type of objects is known and the Core Reflection API could be used to do this task.[2]
- Besides the normal exceptions, all methods also have to take care of `RemoteException`s.
- Client stubs and server skeletons have to be generated using Java's `rmic` compiler.
- An implementation can provide implementations for multiple interfaces, that is, it can provide functionality for multiple remote objects, or it (this is the normal case) will implement only a single interface. Therefore, only 1 object will be served most of the time by a Java server, which is not economical.

[1] This chapter may be relocated / integrated with a different section later on.
[2] The Core Reflection API is actually used, so that decision is even more questionable !

A better solution would be to provide a single server (similar to CORBA's BOA), serving all Java objects. Also, the implementation has to actively create and register an object. For scalability and manageability purposes, it would be better to be able to create and activate/deactivate objects in a server from the client. For example, when the client requests the creation of an instance of class `X`, the server (the dispatcher) would load class `X` (if not yet known), create an instance and return a stub for it to the caller.

A method to improve/eliminate these points is described below. It is the technology underlying the proposal discussed in the previous chapter to add group communication to Java in a non-intrusive way.

## 3.1 A Non-Intrusive Implementation of RMI

### 3.1.1 Eliminating the Need for Client Stubs: Using the Class Compiler

Using the approach described in section 2.1, we can eliminate the need to generate client stubs for classes that want to do RMI. Using the class compiler, an existing Java class (.class file) is copied and the copy annotated with remote behavior. Each public method `foo` is renamed to `foo_impl` and a (stub) method `foo` is created which performs the remote method call. It will marshal its arguments onto an output stream, dispatch the operation to the remote Java server[3], unmarshal the return value and return it to the caller.
Although this approach is more elegant than the client stub approach used by `rmic`, it is more complex as JVM byte code has to be read from class files, modified and output to another class file. Therefore, as a first step, we might still go with `rmic`'s approach and generate a client stub. Later, the class compiler approach can replace this approach without changing any client code.

### 3.1.2 Eliminating the Need for Server Skeletons: Introducing the Reflective Dispatcher

A server skeleton class generated by `rmic` essentially receives requests from a client via the RMI protocol. Its task is to unmarshal the arguments, find the correct method, dispatch to it, marshal the return value and write the marshaled value to

[3]Wrapping RMI in Ensemble, as described in 2.3.

the output stream.[4] There exists one skeleton per Java class.
The process of marshaling / unmarshaling, and dispatching to the correct method can be done in a more generic way *at runtime* using Java's Core Reflection API [Sun96]. This functionality would be similar to a CORBA BOA and could be located in a dynamic *dispatcher* process. A dispatcher demultiplexes requests coming in over a socket and dispatches them to the correct objects and methods within them using entirely the Core Reflection API, and no compile-time generated code.
When a requests come in to create a new instance of a class (containing the class-name (`"X"`) and arguments), a local instance of the class `X` would be created. (`X` must have been previously annotated (see above) !) The annotated instance (a stub for it) would then be returned.
When a remote method call (e.g. `"foo"`) to the newly created instance is received, then the dispatcher – knowing that the class has been annnotated and it can therefore not call the *stub method* `foo` – will call method `foo_impl` which contains the implementation of the method.
Introducing a dynamic, reflection-based, dispatcher eliminates the need to have one skeleton per Java class. Also, a dispatcher can – contrary to Java RMI server processes which have to perform creation and registration of remote instances themselves at startup – create a new instance only when asked to do so by the client, and not at startup of the server. Having only one dispatcher process instead of multiple Java servers reduces management and memory requirements and simplifies the whole process.
A similar approach used to implement a reflective Java ORB is described in [Gra97].

### 3.1.3 Eliminating the Need to Implement Interface `Serializable` For Parameters

It seems that the main reason for requiring a remote object to implement interface `Serializable` is to allow an implementor to override the 2 methods defined in that interface for marshaling/unmarshaling the remote object's state. (The default implementation for marshaling/unmarshaling uses Java's metadata API.)
However, if the Java framework for remote communication would be opened up (using reification, reflection, Open Implementation concepts, cf. 4), then an implementor could easily modify a class' marshaling/unmarshaling behavior without needing to implement interface `Serializable`.

[4] The code for client stubs / server skeletons is quite simple and can be generated by calling `rmic` with option `-keepgenerated`.

### 3.1.4 Eliminating the Need to Modify the Inheritance / Implementation Tree

Using the annotation approach, no class would have to implement interfaces `Remote` or `UnicastRemoteObject` respectively. The desired methods would be defined by the class compiler and invoked by the dispatcher. `(* More details ! *)`

# Chapter 4

# Towards a Reflective Framework for Object Communication

`*Complete*`
As in the case of annotating classes for group communication, there will be other cases where annotation is desired, resulting in 'multiply annotated' classes which is probably undesirable. We have to look for ways in which the concept of annotation can be generalized.
The Java RMI framework should be 'opened up' (e.g. reflection, Open Implementation (OI, Xerox)) to accomodate non-intrusive changes/additions to remote communication, e.g.

- Remote method invocation
- Group communication
- Persistence
- QoS / policy enforcement
- Transport semantics (e.g. ATM)

Main idea: adding group communication to Java is just *one* way of modifying the remote communication semantics. There will be other modification requirements. To accommodate them, the remote communication framework of Java should be reified which allows implementors to replace/extend certain pieces. Procedure: analysis of several change requirements (e.g. those above), synthesis of major modifications and design of framework.
Changes might occur at compile time vs. runtime, class level vs. instance level, attributes vs. methods etc.
Can before- around and -after methods help ? (CLOS) [Ste90]

# Bibliography


[Bir96] Kenneth P. Birman. *Building Secure and Reliable Network Applications*. Manning Publications Co., 1996.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gra97] Ennio Grasso. Jrb: A reflective orb. Technical report, CSELT, 1997.

[Maf95] Silvano Maffeis. *Run-Time Support for Object-Oriented Distributed Programming*. PhD thesis, Institute of Computer Science, University of Zurich, 1995.

[RMI96] Sun Microsystems Inc. *Java Remote Method Invocation Specification*, 1.1 edition, November 1996. Draft.

[Ste90] Guy L. Steele. *Common LISP. The Language*. Digital Press, 1990.

[Sun96] Sun Microsystems Inc. *Java Core Reflection. API and Specification*, October 1996.