# Object-Oriented Distributed Computing Based on Remote Class Reference

Yan Huang[1], David W. Walker, and Omer F. Rana
Department of Computer Science
Cardiff University
PO Box 916
Cardiff CF24 3XF
United Kingdom
{Yan.Huang,David,Omer}@cs.cf.ac.uk

**Abstract**

Java RMI and Jini provide effective mechanisms for implementing a distributed computing system. Recently many numeral libraries have been developed that take advantage of Java as an object-oriented and portable language. The widely-used client-server method limits the extent to which the benefits of the object-oriented approach can be exploited because of the difficulties arising when a remote object is the argument or return value of a remote or local method. In this paper this problem is solved by introducing a data object that stores the data structure of the remote object and related access methods. By using this data object, the client can easily instantiate a remote object, and use it as the argument or return value of either a local or remote method.

## 1 Introduction

Java RMI and Jini extend the Java object model beyond a single Java virtual machine (VM) so that object methods can be invoked remotely by other VMs over the network. Java RMI[8] and Jini[3, 9] technology, and the portability of the Java language, make the Java language well-suited for scientific network-based distributed computing[6, 13]. Several numeral libraries, such as LAPACK[10], the BLAS[1], and LINPACK[11], which were originally written in C or Fortran, have now been implemented in Java[2, 7]. The Java interface to the commonly-used Message Passing Interface (MPI) makes parallel computing easier over networked Java VMs. These Java-based implementations automatically benefit from the objected-oriented programming model that makes them robust, maintainable, and reusable[12].

[1]Corresponding author

The NetSolve[14] system, which is written in C, supports distributed scientific computing through a traditional server-client model. Suppose, for example, that a NetSolve client wishes to compute the inner product of two double-precision vectors using the library routine `ddot()` which is installed on a server[2]. The client sends a request to the server to do the computation. After `ddot()` has been executed on the server the resulting scalar inner product is returned to the client. In this scenario the numerical libraries are in C or Fortran, which are not object-oriented languages, and the code and the data are kept distinct. Both the client and server understand the vector data type, although the routine `ddot()` needs to be installed only on the server.

The situation is different for an objected-oriented client-server model. Suppose we want to do the same thing as described above for NetSolve, but using pure Java and RMI technology. In an objected-oriented world everything is about objects, and the code and data are merged into a single entity - an object. In this example, the vector data structure and the method `ddot()` are included in an object called *Vector*. `Vector.ddot()` takes a *Vector* as its argument, evaluates the dot product of the input *Vector* with itself, and returns a value of data type double. In this scenario, both the server and client must have the definition of *Vector* to ensure the remote method invocation can be executed. This obviously will result in some inconvenience and repeated work for the developer and the user.

This paper will focus on this problem of objected-oriented distributed computing, and demonstrates an approach that solves it.

## 2 Limitations of Remote Object Invocation

Assume we have a class called *Matrix* and an instance of *Matrix* called `matrix`. Consider the following two situations:

1. The client side has the definition of *Matrix*.
2. *Matrix* is defined on the server side but not on the client side. The client can call its methods using remote method invocation.

An obvious difference between these two situations is in the first standalone case, the client has the definition of the class *Matrix*, but in the

[2] `ddot()` is used for simple illustrative purposes. The fact that it is $O(n)$ in both computation and data movement actually makes it a rather unsuitable candidate for remote execution.

second distributed case it does not. In the distributed case, a client can get a reference to the remote object, but there are several things that a client application cannot do in a distributed environment[4]:

1. The client cannot control the construction of the remote object since this is instantiated on the server.

2. The data elements in the remote object are inaccessible even if they are public, unless there are public `set()` and `get()` methods defined to expose them to other remote objects.

3. The client cannot call a method of a remote object if the method has an instance of a remote class as one of its input parameters or its return value. Here a "remote class" is a class whose definition is located in another Java VM and so is not available locally to the client.

This paper addresses each of these difficulties. In Section 4, we describe how "wrappers" can be used to control locally the construction of a remote object. This approach converts a constructor into a normal public method. The problems of accessing the data elements of a remote object, and of calling a method of a remote object when it has an instance of a remote class as an input parameters or the return value, is tackled in Section 5. Here an additional class, known as a "data class", is used to maintain a local copy of the data state of the remote object. In Section 6 a description is given of how Jini has been used to apply the ideas discussed in the preceding sections to implement a truly object-oriented approach to programming in a distributed heterogeneous environment. Some concluding remarks are presented in Section 7.

## 3 An Example

In this section a simple example is given to illustrate the difficulties encountered if we want to take full advantage of a third-party class package located on a remote host in a distributed computing environment.

The third-party package we will use is a Java matrix package called JAMA[5]. We have simplified the code examples so they are easy to understand. Our simple example will use just two JAMA classes: *Matrix* and *LUDecomposition*. The simplified class definitions of *Matrix* and *LUDecomposition* are shown in Fig. 1.

In the non-distributed, standalone case, the user has the package installed on their machine, and their application code will be similar to that shown in the piece of code in Fig. 2.

```
public class Matrix implements
 java.io.Serializable {
  private double[ ][ ] A;

  public Matrix (double[ ][ ] A) {
    ...
    //context of the constructor
  }
  public void Matrix( ){}

  public void setA(double[ ][ ] A) {
    this.A=A;
  }
  public double[ ][ ] getA( ) {
    return this.A;
  }
  public void print ( ) {
    ...
    //code for print out the matrix.
  }
  ...
  //Other public methods.
}
```

```
public class LUDecomposition implements
 java.io.Serializable{
   private double[ ][ ] LU;
   public LUDecomposition (Matrix A) {
      ...
      // LU decomposition A=LU. The results
         are written into one array LU.
   }
   public LUDecomposition(){}
   public void setLU(double [][] LU)
      {this.LU = LU;}
   public double[][] getLU(){return this.LU;}
   public Matrix getL () {
      ...
     //return the unit lower-triangular
   }
   public Matrix getU () {
      ...
      //return the upper-triangular
   }
   ...
   //Other public methods.
}
```

Figure 1: Simplified code of *Matrix* and *LUDecomposition*.

```
public class SimpleExample{
   public static Matrix matrixMaker(int n) {
      double[][] A = new double[n][n];
      for(int i=0; i<n; i++)
         for(int j=0; j<n; j++)
             M[i][j] = (i+j)%n;
      return new Matrix(A);
   }
   public static void main(String argv[]){
      Matrix M = matrixMaker(100);
      LUDecomposition LU = new LUDecomposition(M);
      Matrix L = LU.getL();
      L.print();
   }
```

Figure 2: A simple application executable only on a standalone machine

Everything works fine on a standalone machine, but difficulties arise when we want to distribute the computation. In this case, the user wants most of the computational work to be done on remote servers thereby reducing the load on the local client. Suppose the JAMA package is not installed on the local client. There may be several reasons for this:

1. To save local storage space, or avoid having to install and maintain multiple copies of the same code.

2. To save the time to download and install the package.

3. To keep secret the internal details of the implementation of the package, or to maintain control over the software.

Henceforth we shall assume that the JAMA package is installed on the server but not on the client. Now we examine how the client can perform remote computations on the server.

# 4 Using RMI and Jini for Remote Invocation

To use Java RMI or Jini to perform remote computations using JAMA we need to build up the following classes:

The interface classes: *IMatrix* and *ILUDecomposition*.

The implementation classes: *MatrixImpl* and *LUDecompositionImpl*.

The distribution of all the necessary files is shown in Table 1.

| Server | Client |
|---|---|
| IMatrix.java | IMatrix.java |
| ILUDecomposition.java | ILUDecomposition.java |
| MatrixImpl.java | SimpleExample.java |
| LUDecompositionImpl.java | |
| Matrix.java | |
| LUDecomposition.java | |

Table 1: Distribution of files between server and client.

The first thing we have to think about is how to instantiate a remote object in this case. Both the classes for which we want to remotely obtain an object have a non-trivial constructor. In *Matrix*, the constructor takes an array as input and stores it into a private data element $A$. In *LUDecomposition*, the constructor is the main part of the class in the sense that it

does most of the computing work - it takes a *Matrix* as input, computes its LU decomposition, and obtains two matrices $L$ and $U$ as a result which are stored in a data element: an array with $L$ and $U$ below and above the main diagonal, respectively. In the Java distributed object model, the server side instantiates the object, and a client is able to get a reference to this remote object. Because the client sees only a stub for the original object and the stub implements only the remote methods[15], the client can only invoke the methods of the remote object; it cannot contribute to the construction of the object. However, sometimes it is necessary for the client to build a remote object using the data provided by the client, as in the example code:

```
double[ ][ ] A = new double[n][n];
...
return new Matrix(A);
```

The client code needs to build a remote object by passing an array $A$. The way we solve this difficulty is to make the constructor callable by the remote client by changing the constructor into a normal exposed method. We want to avoid having to change the original code because it may belong to a third party, so here the object implementation classes *MatrixImpl* and *LUDecompositionImpl* are not derived directly from *Matrix* and *LUDecomposition* by modifying their original code, but instead play the role of "wrappers" for the original classes by taking an instance of the remote class as a data element and then providing the interfaces to all the public methods provided in the original classes.

An example of the wrapper (or implementation class) *MatrixImpl* for the original class *Matrix* is shown in Fig. 3.

```
Public class MatrixImpl implements Imatrix {
   Matrix itself;
   public MatrixImpl(){}
   public void MatrixConstructor(double[][]A) {this.itself = new Matrix(A);}
   public void print () {this.itself.print();}
   ... //Other public methods
}
```

Figure 3: A wrapper class for *Matrix*.

In this code, the method `MatrixConstructor(double[][] A)` generates an instance of *Matrix* by passing the input data of the method to the *Matrix* constructor, and then stores the new matrix as a *Matrix* type data element called *itself*. By calling this remote method, the client is able to build the

object as required. In fact, the remote object whose methods are exposed here to a client is not an instance of *Matrix* any more, but an instance of *MatrixImpl*, which has an instance of *Matrix* as its data member. The architecture of this mode of use is described in Fig. 4, and the code is as follows:

```
obj = ...; /*Assume the client gets a reference to
             the remote object MatrixImpl.*/
double [][] A = ...; //assign value to array A
obj.MatrixConstructor(A);
obj.print();
```

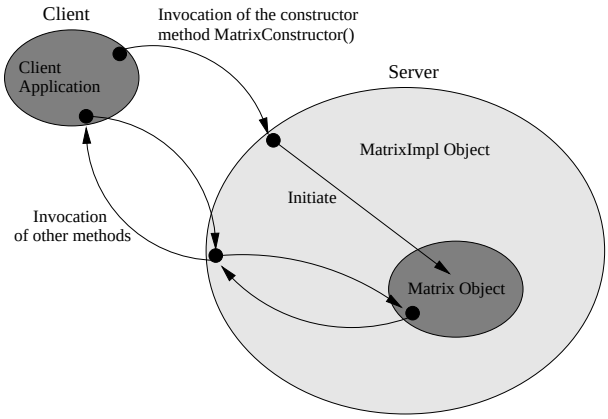


Figure 4: The client remotely controls the construction of a remote object.

Now consider the following line of code:

```
LUDecomposition LU = new LUDecomposition(M);
```

Here, $M$, a local instance of a remote class, is input to the constructor of the server side object *LUDecomposition*. As in the earlier case of the remote *Matrix* constructor, we build an instance of *LUDecompositionImpl* which acts as a wrapper object for the real *LUDecomposition* object, or like a bridge between the client and the actual object. Any action the client wishes to perform on the real object will go through the wrapper object, and the wrapper object in turn will initiate the operations on the real object. By introducing the object *LUDecompositionImpl*, we change the code into the following:

```
ILUDecomposition obj = ...;
// Get the reference to the remote Object of LUDecompositionImpl.
LUDecomposition LU = obj.LUDecompositionConstructor(M);
```

Now we are able to call a remote constructor remotely, but if we try to compile the code, an error will occur because the *Matrix* and *LUDecomposition* classes cannot be found. We still have an unsolved problem – how do we initiate a local instance of a remote class? In fact, it is impossible to initiate a local instance of a remote class. We can only get an instance of a class that is defined locally. So if the definitions of *Matrix* and *LUDecomposition* are located remotely (i.e., on the server side) the client side cannot directly instantiate them. We examine this problem in the next section.

# 5 Implementation of Remote Class References

In order to implement remote method invocation, both client and server share an interface definition that specifies the remote interfaces of the remote object, so the client is assumed to know about the exposed methods of the remote class. We can change the constructor into a general public method (see Section 4) making it callable as a normal remote method, so we can assume the client side has all the interfaces of the exposed methods and the constructor methods specified in its interface class. A general class contains three parts – the data, the methods, and the constructors – so now the only thing left in a remote class that is still unknown on the client side is the data structure.

Suppose a client can quite easily find out the data structure defined in a remote class so that it is possible to have the following class shared by both client and server (see Fig. 5).

```
Public class MatrixData {
  Private double[][] A;
  Public MatrixData(){}
  Public void setA(double[][] A){ this.A = A; }
  Public double[][] getA(){ return this.A; }
}
```

Figure 5: A data class of class *Matrix*.

Comparing class *MatrixData* with class *Matrix* we see they contain the same data structure, namely `double[][] A`, the public `set` and `get` methods to make the data accessible, and an empty constructor. As the name of the

class *MatrixData* suggests, the main purpose of this class is to define the data structure of class *Matrix*. For the sake of convenience, we call this sort of class a "data class", and an instance of a data class a "data object".

We need also to create a data class *LUDecompositionData* for the remote class *LUDecomposition*.

Before we start to change the client application code, we need to add four methods to the wrapper class file `MatrixImple.java`, as shown in Fig. 6.

```
public void MatrixConstructor(MatrixData dataobject){
   This.itself  = dataclassToRemoteclass(dataobject);
}
public Matrix dataclassToRemoteclass(MatrixData dataClass){
   Matrix remoteClass = new Matrix();
   remoteClass.setA(dataClass.getA);
   return remoteClass;
}
public MatrixData remoteclassToDataclass(Matrix  remoteClass){
   MatrixData dataClass = new MatrixData();
   dataClass.setA(remoteClass.getA);
   return dataClass;
}
public MatrixData getDataObject(){
   return this.remoteclassToDataclass(this.itself);
}
```

Figure 6: Four methods are added to `MatrixImple.java`

Methods `dataclassToRemoteclass()` and `remoteclassToDataclass()` implement the data flow between the data object and its remote object. `MatrixConstrutor(Matrixdata dataobject)` is used to construct a remote object of type *Matrix* by copying data from a *MatrixData*. `GetDataObject()` returns a data object of type *MatrixData* by extracting data from a *Matrix*. In addition, we need to add these interfaces to the interface class *IMatrix* so the corresponding methods can be called remotely. Similarly, we need to add the same methods to *LUDecompositionImpl* and *ILUDecomposition*.

Assume we have the following code in the original client application:

```
Matrix M = matrixMaker(100);
M.print();
```

The process necessary to make it work in a distributed way is described in four steps.

1. In place of getting a local instance of a remote class we get a data object of its data class:

```
MatrixData M = matrixMaker(100);
```

We need to change the definition of routine `matrixMaker()` so that its return type is *MatrixData*, rather than *Matrix*.

2. We get a reference to the remote wrapper object, *MatrixImpl*, of the *Matrix* by any remote object invocation technique, such as RMI or Jini.

3. We transfer the data from data object $M$ to the remote object by calling remote method `dataclassToRemoteclass()`. Now the remote object has the same data state as the data object.

4. We send a request to the remote object to invoke the method `print()`.

Here the data object plays the role of a data state saver for the corresponding remote object. A remote object can have more than one data object. If an application needs more than one remote object of the same type, as happens quite often, several data objects are needed but only one remote object is necessary. The single object can work like multiple objects by copying data to and from its data objects. This reduces the network traffic and the overhead of looking up the same service throughout the distributed environment.

Next we consider the case in which an instance of a remote class is input to, or returned from, a local or remote method:

```
Matrix M = matrixMaker(100);
LUDecomposition LU = new LUDecomposition(M);
Matrix L = LU.getL();
```

First we need to modify all the methods in the wrapper classes, which take remote class type objects as their input or return value, by having all the remote class type objects replaced by their data objects. In *LUDecomposition*, these methods are changed into the definition in Fig. 7.

In this implementation, all the remote classes are wrapped inside the wrapper classes, *LUDecompositionImpl*, and run on the server side. The client only has to deal with the data objects of these remote classes, thereby achieving the functionality of initiating a local instance of a remote class without having its definition available locally. Figure 8 shows the data transformations in the remote method invocation.

```
Public void LUDecompositionConstructor(MatrixData dataMatrix){
   MatrixImpl wrapperObject = new MatrixImpl();
   This.itself = new LUDecomposition(
                    wrapperObject.dataclassToRemoteclass(dataMatrix));
}
Public MatrixData getL(){
   MatrixImpl wrapperObject = new MatrixImpl();
   return wrapperObject.remoteclassToDataclass(this.itself.getL());
}
Public MatrixData getR(){
   MatrixImpl wrapperObject = new MatrixImpl();
   return wrapperObject.remoteclassToDataclass(this.itself.getR());
}
```

Figure 7: The modified methods in *LUDecompositionImpl*.

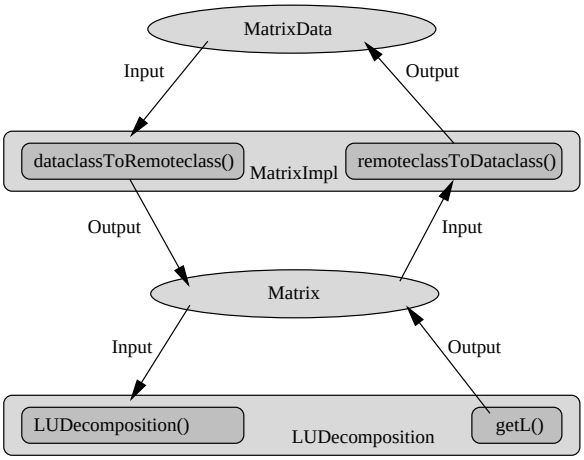


Figure 8: The transformation of objects in the remote method invocation.

# 6 A Client Application with Remote Class Referencing

In implementing the approach described in this paper Jini is used to build the distributed system.

So far we already have the following files: the interface definitions `IMatrix.java` and `ILUDecomposition.java`; the implementation of the interfaces `MatrixImpl.java` and `LUDecompositionImpl.java`; the data classes `MatrixData.java` and `LUDecompositionData.java`; the original classes `Matrix.java` and `LUDecomposition.java`; and, the application file `SimpleExample.java`. The distribution of the files between a client and a server is shown in Table 2.

| Server | Client |
|---|---|
| IMatrix.java | IMatrix.java |
| ILUDecomposition.java | ILUDecomposition.java |
| MatrixData.java | MatrixData.java |
| LUDecompositionData.java | LUDecompositionData.java |
| MatrixImpl.java | SimpleExample.java |
| LUDecompositionImpl.java | |
| Matrix.java | |
| LUDecomposition.java | |

Table 2: Distribution of files between server and client.

We used two Unix machines running Solaris 2.6 and a Windows 2000 machine in our implementation. The Jini lookup service was run on one of the Unix machines, and the other Unix machine was used as the server providing the matrix computing service. The Windows 2000 machine was used as the client on which we initialize execution of the application. So on this machine we just install the data classes and the interfaces and make sure the original classes are not accessible to the client. We also executed a standalone case so that we can compare the results between the standalone case and the distributed case. Both cases gave the same output which indicates that the distributed case is running correctly, and that the approach described in this paper actually works. The complete code for the example application and instruction for its use are available from `http://www.cs.cf.ac.uk/User/Yan.Huang/research/remote.html`.

# 7 Concluding Remarks

By introducing the concepts of the data object and data class a programmer can build a true object-oriented distributed system without needing multiple copies of the definition of the remote objects on both clients and servers. There are also other benefits from the idea of using a data object to store the data state of the remote object.

1. Space is saved on the client. When a remote class reference is needed in a client application only one definition of the remote class is needed on the server - the client just needs to have its data class which is normally much smaller than the original class.

2. Space is saved throughout the network. If more than one client needs the same remote class reference only one copy of the class is needed on the server.

3. Network traffic is reduced. If a client needs more than one instance of the remote class only one reference to the remote object is necessary. In comparison with getting a reference to the remote object each time when a new remote object of the same type is needed, this can save the time spent looking for the service or object over the network.

We have demonstrated the feasibility of our approach by implementing a truly object-oriented distributed system across heterogeneous hardware platforms, operating systems, and software environments. Future work will develop tools to automatically convert large numerical libraries for use in such distributed environments, and will examine performance issues when our approach is applied to computational science applications.

# References


[1] BLAS (Basic Linear Algebra Subprograms), `http://www.netlib.org/blas/`.

[2] Ronald F. Boisvert and Jack J. Dongarra, "Developing Numerial Libraries in Java", ACM 1998 Workshop on Java for High-Performance Network Computing, March 1998.

[3] W. Keith Edwards, "Core JINI", 2nd Edition, Prentice Hall PTR 2001.

[4] Daniel Hagimont and Fabienne Boyer, "A Configurable RMI Mechanism for sharing Distributed Java Object" IEEE Internet Computing, Volume 5, Issue 1(2001), pp. 36–43.

[5] JAMA: A Java Matrix Package, `http://math.nist.gov/javanumerics/jama/`.

[6] Java Distributed Object Model, `http://java.sun.com/products/jdk/1.1/docs/guide/rmi/spec/rmi-objmodel.doc.html`.

[7] JavaNumerics, http://math.nist.gov/javanumerics/.

[8] Java Remote Method Invocation (RMI), http://java.sun.com/products/jdk/rmi/.

[9] Jini Network Technology, http://www.sun.com/jini/.

[10] LAPACK – Linear Algebra PACKage, http://www.netlib.org/lapack/.

[11] LINPACK, http://www.netlib.org/linpack/.

[12] Robert Martin, "OO Design Quality Metrics, An Analysis of Dependencies", http://www.ao.net/~juang/IntroJava/IntroJavaMain.html.

[13] Jose E. Moreira, Samuel P. Midkiff, and Manish Gupta, "From flop to megaflops: Java for technical computing", ACM Transactions on Programming Languages and Systems Volume 22 , Issue 2 (2000), pp. 265-295.

[14] NetSolve, http://www.cs.utk.edu/netsolve/.

[15] RMI and Object Serialization Frequently Asked Questions, http://java.sun.com/products/jdk/1.2/docs/guide/rmi/faq.html.