Massimiliano Parlione

Remote Class Inheritance

Thesis Committee
Dott.ssa Gianna Cioni (Advisor)
Prof. Maurizio Lenzerini
Prof. Francesco Parisi Presicce

Reviewers
Prof. Kurt Geihs
Prof. Antonio Corradi
Abstract

The purpose of this thesis is to allow a programmer to develop a distributed application in Java [GJS96] almost in the same way he develops a traditional one. Our proposal relies on the idea that a class should be developed without considering whether it is remote or not. To realize this objective we propose an object model that makes transparent the remote nature of a class and its instances. This model simplifies the usage of remote objects and introduces, in the distributed scenario, a new programming technique called remote class inheritance.
Acknowledgments

I wish to thank my wife Rita for her help and support, Gianna Cioni and Marco Temperini for reviewing this thesis. Finally, a thanks goes to the designers of ObjectSpace Voyager universal ORB. Without this wonderful product this prototype would not have been possible.
Contents

1 Introduction .................................................. 1
  1.1 Object oriented programming .............................. 1
  1.2 Remote inheritance and remote method invocation ......... 3
  1.3 Current distributed architectures ......................... 4
  1.4 Class inheritance in previous proposals .................. 6
  1.5 The distributed hierarchy proposal ....................... 6
  1.6 Thesis structure ........................................... 7

2 An Introduction to Java ........................................ 9
  2.1 Java overview ............................................. 9
    2.1.1 Object Model ........................................ 9
    2.1.2 Packages and visibility ............................. 11
    2.1.3 Exceptions handling ................................ 12
    2.1.4 Garbage collector .................................... 13
    2.1.5 Multithreading ....................................... 13
    2.1.6 Under the hood: Java Virtual Machine and ClassLoader ... 14
    2.1.7 Applets .............................................. 16
    2.1.8 Security architecture ............................... 17
    2.1.9 Reflection mechanism ............................... 17
    2.1.10 Network Programming .............................. 18
    2.1.11 Summary ............................................ 18
  2.2 Java Reflection in detail .................................. 19
    2.2.1 Reflection Model ..................................... 21
    2.2.2 Data Conversions .................................... 23
    2.2.3 Applications ........................................ 24

3 Object Oriented Distributed Architectures in Java ............ 25
  3.1 Java IDL .................................................. 25
    3.1.1 The CORBA Architecture ............................ 26
    3.1.2 The Naming Service ................................ 28
    3.1.3 The Interface Definition Language ................. 28
    3.1.4 Summary ............................................ 30
  3.2 Java Remote Method Invocation ............................ 31
    3.2.1 RMI at work ........................................ 31
    3.2.2 Advanced RMI ....................................... 32
    3.2.3 The system architecture ............................ 33
3.2.4 Summary .............................................. 34
3.3 Java IDL versus Java RMI ........................... 35
3.4 ObjectSpace Voyager ............................... 36
  3.4.1 How it works ....................................... 37
  3.4.2 Summary ........................................... 38
3.5 Voyager versus Java IDL ........................... 38
3.6 Voyager versus Java RMI ........................... 39
3.7 Conclusions ............................................. 41

4 Inheritance in Distributed Architectures ........... 43
  4.1 Distributed Smalltalk .............................. 44
    4.1.1 Evaluating the solutions ..................... 45
    4.1.2 Possible solutions in Java .................... 46
    4.1.3 Summary ......................................... 47
  4.2 Global Object Space ............................... 47
    4.2.1 Object Model .................................... 48
    4.2.2 Application development ...................... 49
    4.2.3 The GOS scenario .............................. 49
    4.2.4 Remote Inheritance .............................. 50
    4.2.5 Summary ......................................... 51

5 Remote Class Inheritance among JVMs ............... 53
  5.1 The "Hello World!" example ...................... 54
  5.2 The distributed hierarchy architecture .......... 63
  5.3 The forwarder and receiver framework .......... 64
    5.3.1 The package di ................................ 67
    5.3.2 Forwarder Structure .......................... 70
    5.3.3 Receiver structure ............................ 80
    5.3.4 Forwarders and receivers at work .......... 86
  5.4 Remotizing a system class ....................... 89
  5.5 Remotizing a linked list ......................... 93
  5.6 Summary ............................................. 95

6 The tic compiler .......................................... 97

7 Conclusion ............................................... 103
  7.1 Further developments .............................. 105

A Class Diagrams in UML ................................. 107

B "Hello World!!" in Java IDL, RMI and Voyager. ... 109
  B.1 Java IDL ............................................ 109
  B.2 Java RMI ............................................ 113
  B.3 Voyager .............................................. 117
  B.4 Summary ............................................ 119

Bibliography .............................................. 121
Chapter 1

Introduction

The purpose of this thesis is to allow a programmer to develop a distributed application in Java [GJS96] almost in the same way he develops a traditional one. Our proposal relies on the idea that a class should be developed without considering whether it is remote or not. To realize this objective we propose an object model that makes transparent the remote nature of a class and its instances. This model simplifies the usage of remote objects and introduces, in the distributed scenario, a new programming technique called remote class inheritance. In this way, a distributed application can easily be developed according to the object oriented paradigm and without learning all the technicalities of the current object oriented distributed architectures. Notice that an easier way to develop and use a remote object does not imply that a programmer has to disregard its remote nature. Some precautions are still required when a distributed system is designed, since a remote object introduces some issues such as communication overhead and network failures. But, apart from simplifying the development of a distributed application, we have accomplished a full compliance to the object oriented paradigm that does not occur in any other architecture. In fact, currently, all the available object oriented distributed architectures are simply remote method invocation systems.

1.1 Object oriented programming

Object oriented programming [FiM95] is a practical and useful methodology that encourages modular design and software reuse. It is based on the notion of "object" which is essentially some data and a collection of operations that operate on that data. There are two flavors of object oriented languages: delegation-based\footnote{Self [ABCC93] is an example of delegation based language.} and class-based. According to the aim of this thesis we will consider exclusively the class-based group. In class-based languages such as C++, Java, Smalltalk and others, the implementation of an object is specified by its class. A class is intended to describe the structure of all the objects generated from it. In fact, a class defines a set of named operations, called methods, and a set of named fields, called variables. A programmer can define new classes (or types) and define their members i.e., their
methods and variables. Each object is an instance of one class and it is represented by a collection of instance variables and a collection of methods, as defined by the class. Within these methods the special identifier *self* refers to the host object. An instance is created through a specific method named constructor. A constructor is a particular kind of method automatically invoked at instantiation time. Usually there is also a method named destructor that is invoked when the object is released. Note that an instance can be allocated both in the heap and in the stack according to some programming languages. Other languages allow only to create instances in the heap. In fact, class-based object oriented programming languages can be classified according to the policy implemented to manage instances. Some languages, such as C++, expose the distinction between stack-allocated and heap-allocated objects. While other languages, such as Smalltalk and Java, hide this distinction and deal exclusively with heap-allocated objects. Nevertheless, a class can also be used without creating an instance since a programmer can define class-level variables\(^2\) and methods that can be used directly. These kinds of attributes and methods are usually called *static* members in the C++ and Java programming languages.

Usually methods and variables are accessed using the so-called “dot” notation. For instance, given a class \(C\) that declares an instance variable \(v\) ad an instance method \(m\), once we have created an object from this class and assigned it to a variable \(c\) of type \(C\), then both the members can be accessed in this way: \(c.m()\) to invoke the method and \(c.v\) to access the variable. Note that it is possible to have more than one method with the same name as long as all the homonymous methods have different parameter profiles, consisting of the number and types of their arguments and the types of their return values. This is the concept of method overloading since the right method is invoked according to the number and types of the actual parameters.

This short description of the basic features of a class gives only an overview to understand object oriented languages. In fact, the real power of object oriented programming relies on the following features: dynamic lookup, inheritance, subtyping, and encapsulation.

Some object oriented languages use a dynamic lookup to select the actual method when it is invoked. Dynamic lookup consists of executing the method body according to the run-time type identity of the receiver object, not its static type.

Inheritance is a language feature that allows new classes to be defined as increments to the existing ones. In this way a class can share common components by means of inheritance. If a class \(C\) (directly) inherits from a class \(P\), \(P\) is a parent of \(C\) and \(C\) is a child of \(P\). The importance of inheritance is that it saves the effort of duplicating (or reading duplicated) code, and that when one class is implemented by inheriting from another, changes in one affect the other. We have a so-called inheritance hierarchy when there are several classes in a sort of chain where a class can inherit from another class that in turn inherits from a different class, and so on, until we reach the root class. In this context the terms ancestor and descendant are used in the obvious way even though some authors prefer the traditional terms subclass and superclass. Besides adding new methods in a descendant class with respect to an ancestor class, one of the main advantages of inheritance is that a class can redefine existing methods of its superclass in order to supersede them. This feature is called method overriding. Unfortunately, almost all the commercial programming languages

\(^2\) Class variables can also be thought as fields shared by all instances of a class.
allow only to change the behavioral part of an overridden method and to invoke the same method defined in the ancestor by means of a variable super that references the superclass\(^3\) itself. In commercial languages, such as C++, Java and Smalltalk, it is not allowed to change the signature of a method in order to specialize it in the same way we specialize its behavioral part i.e., the code. There are two flavors of inheritance: single and multiple. With single inheritance the class hierarchy is a tree since each class can have only one superclass, while with multiple inheritance the hierarchy is a graph since a class can have more than one superclass. Although multiple inheritance gives more flexibility to the programmer, it introduces also some issues such as name conflicts and repeated inheritance. Name conflicts are raised when two or more ancestors of a given class have homonymous members, while repeated inheritance occurs when a class is an ancestor in more than one way. Therefore, to avoid these issues, only few programming languages support multiple inheritance while other languages like Java only support single class inheritance.

The basic principle associated with subtyping is substitutivity: if \( s \) is a subtype of \( t \), then any expression of type \( s \) may be used without type error in any context that requires an expression of type \( t \). Therefore, subtyping means that if some object \( Ob1 \) has all the functionality of another object \( Ob2 \), then it is possible to use \( Ob1 \) in any context expecting \( Ob2 \). The primary advantage of subtyping is that it permits uniform operations over various types of data. Note that subclasseing implies subtyping unless we use C++ private inheritance\(^4\) but subtyping does not necessarily implies subclassing as we will see with Java interfaces in the next chapter\(^5\). See [LIW93/1] and [LIW93/2] for further details.

Finally, encapsulation provides a way to protect from unauthorized access the members of a class. Usually a programmer can define an access specifier for each class member in order to create a barrier with respect to access attempts from the clients of a class. A class can have at least three level of protection for its members. A class can have some of its members accessible only from inside the class itself, other members accessible also from its descendant and some others accessible from outside the class. These protection levels are called private, protected and public in some languages such as C++ and Java.

### 1.2 Remote inheritance and remote method invocation

The purpose of remote method invocation (RMI) [BNO95] is to allow a program to invoke the methods of a remote object. This means that an object created on a remote host can be used as if it were local to its client. The RMI mechanism is the object oriented version of the Remote Procedure Call (RPC)[BiNi84] mechanism. The RPC system was conceived to build a distributed application using remote procedures and invoke them as if they were local to the callers. All the values passed to the formal

---

\(^3\)There could be variables that have the same feature but a different name according to the programming language considered.

\(^4\)With private inheritance all the methods and attributes of a class become private in its descendant.

\(^5\)Several classes can implement an interface without being in the same inheritance path. Everywhere that interface is required, it can be substituted by an instance of these classes.
parameters and the return value were managed by the underlying run-time support for the marshaling and unmarshaling operations. In this way, an application was able to pass values back and forth to a procedure without worrying about all the transmission issues. RPC represented a step toward more transparency for the programmer with respect to socket programming. RPC has evolved into RMI when the object-oriented paradigm was introduced in the distributed arena. Yet, these RMI systems, besides being quite difficult to learn and use for the average programmer, can only enable a class to receive remote method invocations. Nevertheless, the object-oriented paradigm relies not only on data encapsulation and method invocation but also on class inheritance. Unfortunately, class inheritance is a feature completely disregarded in these architectures, although it has an important role in the object-oriented paradigm since it allows systematic code reuse. In fact, a programmer cannot take advantage of this powerful mechanism when he moves from a traditional object-oriented paradigm to a distributed one. Note that class inheritance is not even supported in pure Java distributed architectures.

Currently, remote objects are seen exclusively as service providers. A client can access a service i.e., invoke a remote method, using somehow an interface that represents this service. Interfaces can be seen as a simplified variant of the concept of class. In fact, an interface is nothing more than a container of method signatures without code. There is, similarly to class inheritance, interface inheritance.

Some programming languages support natively interfaces. For instance, a Java class can implement any number of interfaces by giving a body to all the methods of each interface. On the other hand, languages that do not have interfaces usually can mimic them using abstract classes such as in C++ [Str98]. In current object-oriented distributed systems, an interface is used to describe and access the services offered by a remote object to its clients. Let us introduce some of these architectures to see how interfaces are used to build an object-oriented distributed application.

### 1.3 Current distributed architectures

The most popular architecture was conceived by a consortium of industries: the Object Management Group (OMG). The OMG has brought about the Common Object Request Broker Architecture (CORBA) standard [OMG99]; a language and platform independent architecture, available in several commercial implementations, that introduces among its features the remote method invocation facility.

To develop an application in CORBA, a programmer has to create a bunch of files, using a specific compiler, from an interface describing the remote services (methods) that an object offers. Since the CORBA standard is language neutral, this interface should be written using the Interface Definition Language (IDL). From an IDL description, a compiler builds the base files according to the mapping implemented by the programming language or languages chosen to develop an application. We said languages because CORBA allows, for instance, to have a client written in Java and a server written in C++. Once generated, some of these files are moved to the client side while the others are transferred to the server side. They are used, together with the ORB core, to deliver requests and responses to an object that actually implements the requested services.
1.3. CURRENT DISTRIBUTED ARCHITECTURES

Language neutrality is the main advantage of CORBA and also its main disadvantage, since it forces to use the architectural neutral IDL language to describe the services of an object, allows to create a remote-enabled class only by means of the generated files and brings some limitations on the parameters that can be passed to remote methods.

Unfortunately, no CORBA implementation supports class inheritance or something that resembles in case the descendant and the ancestor are not located in the same environment. In this context a programmer can only extend IDL interfaces, before compiling them, and local classes.

To cancel some of the mentioned CORBA limitations due to its language neutrality, Sun has proposed Java RMI [WRW96], a CORBA-like architecture to develop object oriented distributed systems. The stance of Sun is the following: if we develop everything in Java then we do not need an architectural neutral object model like the one offered by IDL.

Indeed, a Java RMI programmer has to use only some classes offered in a specific package to remote-enable an object. Notwithstanding, also this architecture requires to use an external compiler to generate two files containing the proxies (named stub and skeleton) that manage the message delivery to a remote object. Moreover, Java RMI requires that a descendant of a predefined interface is implemented by the remote-enabled object.

Once the object has been created and registered with a naming service running on the server side, a client, using the same naming service, can obtain a stub that represents locally the remote object. Since this interface is also implemented by the downloaded stub, a variable of the same type can be used to invoke a method. Furthermore, Java RMI, since it is bound to Java, allows to use advanced features such as passing behavior (classes) to remote objects during a method invocation. Note that this feature is the object-oriented equivalent of the Remote evaluation mechanism (Rev) described in [StG90]. In conclusion, although Java RMI has an object model compatible with the Java object model, it provides some innovative features such as dynamic stub downloading and behavior delivery, it still does not allow to have a descendant of a remote class.

Subsequent to the introduction of Java, another kind of distributed architecture to build agent-based applications [WoJ94] have come of age. As far as Java based systems is concerned, this architecture has three main representatives: Aglets [Lan97], Concordia [WPW98], and Voyager [Gla98]. All these systems allow to build applications that enable objects to receive remote calls and to be moved to different places. Notwithstanding, even this kind of architecture does not offer a class inheritance mechanism.

In fact, class inheritance is one of the most neglected issues in all the known platforms that have been proposed for the development of object oriented distributed applications. In our knowledge none of the platforms available at the time of this writing such as all the commercial CORBA implementations, Microsoft DCOM [BrK], Sun's Java RMI and the numerous Agent Platforms such as IBM Aglets, Mitsubishi Concordia and, above all, ObjectSpace Voyager, allow class inheritance among remote classes.

In addition, all the mentioned approaches have another drawback, particularly the CORBA implementations. These platforms require to learn new APIs to build
distributed applications. Despite that, during the development stage, remote objects cannot be used in the same way local object are used.

1.4 Class inheritance in previous proposals

In the past, in our knowledge, only two attempts were made to address the class inheritance issue in a distributed system: Distributed Smalltalk (DS) [BEN87/1] and Global Object Space (GOS). Distributed Smalltalk did not succeed, its author stated that inheritance does not scale well in a distributed context with respect to object mobility. We believe that this was due only to some limitations of Smalltalk at that time\(^6\). On the other hand, GOS in [GRG96] proposes a solution that mimics the class inheritance mechanism. In GOS an object is divided into two parts: one implements the behavior and the other contains the state and the interface shown to the clients. The behavior part is named engine while the other part is the chassis. If a method is invoked on the chassis, then it will be attached to the corresponding engine that implements the required facility. When the method is run, it modifies the state in the chassis. After the completion of the method execution, the engine detaches in order to be ready for new method invocations. Allowing a chassis to be attached by several engines at the same time, as long as there is type compatibility, is how in GOS the inheritance feature is implemented. Note that this kind of inheritance, obtained by the composition of the engines, is a monotonous form of inheritance since it allows neither the method overloading and overriding nor attributes redefinition.

In other words, the GOS solution proposes an object model where all the objects are vertically partitioned into a chassis, that contains the state, and a behavioral part. In turn, the behavioral part of each object might be horizontally divided into several engines, each one containing only a fragment of the services offered by the object itself.

In our approach, we propose a different solution with a one to one correspondence between instances of a descendant and instances of its ancestor.

1.5 The distributed hierarchy proposal

Our proposal aims at giving the possibility to use single class inheritance even when ancestor and descendant are not located on the same place. In this way, it is possible to create distributed hierarchies of classes that specialize the behavior of their ancestors. In a pure Java environment this is equivalent to have ancestor and descendant on different Java Virtual Machines.

According to our proposal, a programmer has only to write a class without using a specific API for distributed computing. Afterwards, he uses a compiler, named distributed hierarchy compiler, to remote-enable this class. The distributed hierarchy compiler (dic) uses this class as a blueprint to automatically generate two new classes that act as a communication framework: a proxy to forward all the method invocations, another to receive them and invoke the actual method of an instance of the original class. The dic compiler uses extensively the Java reflection API to build

\(^6\)For a more updated version of Smalltalk see [IKM97].
1.6. Thesis structure

The thesis is organized in this way: chapter two offers an introduction to Java 2\(^7\), chapter three describes and compares in detail the architectures of Java IDL, Java RMI and Voyager. Chapter four offers an insight on the Distributed Smalltalk and GOS proposals. Chapter five is divided into three parts: the first one describes the usage of the DIC compiler by means of a simple example, the second one describes

\(^7\)The jdk 1.2 is available at the Java Web site http://java.sun.com/products/jdk.
thoroughly how the DIC compiler generates the communication framework to remote-enable, remotize, a class and how this framework works, while the last part shows how to remotize a Java system class and how to build a remote version of a linked list. Finally, chapter six presents the DIC compiler structure. There are also two appendixes. The first describes the notation for the class diagrams, while the second shows how to implement a simple application, the same implemented at the beginning of chapter four, according to Java IDL, Java RMI, and Voyager. In this way, it will be possible to compare each of these technologies with our proposal.

The reader that has a limited knowledge of the Java architecture should read chapter two. Particularly, he should focus on the exception mechanism and the reflection API in order to comprehend how our solution works. Similarly, if the reader does not have any development experience in at least one of the following architectures such as CORBA, Java RMI or Voyager, we suggest to read chapter three. In any case, we would recommend he reads the Voyager section since Voyager is the ORB used by forwarder and receiver to communicate.

The reader interested in the other previous proposals should refer to chapter four. Finally, chapter five can be used exclusively as tutorial on DIC if the reader is not interested in the structure of the forwarder-receiver framework. In this case, he should read only the first part of this chapter. Analogously, he should read chapter five only if he is interested in the implementation technicalities of the DIC compiler.
Chapter 2

An Introduction to Java

The Java programming language is a general-purpose object-oriented concurrent language that combines in a single environment some of the most innovative features available in Eiffel, Smalltalk and C++ with some others introduced by Sun. Since our proposal uses extensively some facilities available in Java, this description is required in order to simplify the comprehension of how the system works.

First of all, we give a bird’s eye view of Java 2 that was the latest version available at the time of this writing. Afterwards, we will describe the facilities available in some packages. In particular, we will focus on those packages utilized in our prototype such as the Core Reflection API.

2.1 Java overview

In this section we will give a short description of the Java object model, how exceptions are managed, and how its garbage collector works. Subsequently, the notion of Java Virtual Machine and the main features of the Class Loader will be described. Finally, we will introduce a synopsis of its security model, an explanation of the Applet role to bring object computing to the Web and what Java offers to develop networked applications.

2.1.1 Object Model

The design of the Java programming language is based on a simple and elegant object model. In Java a class is a template for creating objects, or instances. It defines data attributes, called instance variables, and operations, called methods, of its instances. A class is also a run-time repository for shared data and operations called static variables and methods. Each class implicitly defines a homonymous type. Classes may inherit from other classes, thus extending the set of data attributes and methods of the objects they define. Java only allows single inheritance, that is, a class may extend a single parent. An extending class can override inherited methods as long as they are not final, but can only shadow inherited instance variables. Since constructors are not members, they are not inherited by subclasses. If an object is held in a superclass variable and the class of the object overrides a method, the underlying system runs on
request the most specialized method according to the run-time type of the assigned object. In object oriented parlance this behavior is called dynamic method lookup [Tem90].

The Java object model has another interesting feature, a programmer can define a class as a member of another class to enforce their relationship. Such a class is called a nested class. As a member of its enclosing class, a nested class has a special privilege: It has unlimited access to its enclosing class members, even if they are defined private. Like other members, a nested class can be defined static (or not). A non static nested class is called an inner class.

Both class and instance variables can have static initializers. Initializers are values assigned respectively to their corresponding class variables when a class is first loaded, and to the instance variables when an instance is created. Besides, a class definition may also include instance initializers. An instance initializer is like a constructor: it runs when an instance of the class is created, but it runs before any other defined constructor.

Java also provides interfaces. Interfaces are named sets of method signatures which implicitly define homonymous types. A class may implement any number of interfaces i.e., provide implementations for all methods of these interfaces. Interfaces introduce multiple subtyping in Java.

As in other programming languages, class and interface members have a visibility modifier. The visibility modifier can be selected among the private, protected, public and the so-called friendly qualifier. According to the specified qualifier, the access to class members from other classes can be more or less restrictive.

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>(from java.lang)</td>
</tr>
<tr>
<td>+clone() Object</td>
</tr>
<tr>
<td>+equals(java.lang Object): boolean</td>
</tr>
<tr>
<td>+finalize()</td>
</tr>
<tr>
<td>+getClass(): Class</td>
</tr>
<tr>
<td>+hashCode(): int</td>
</tr>
<tr>
<td>+notify()</td>
</tr>
<tr>
<td>+notifyAll()</td>
</tr>
<tr>
<td>+toString(): String</td>
</tr>
<tr>
<td>+wait(long)</td>
</tr>
<tr>
<td>+wait(long, int)</td>
</tr>
<tr>
<td>+wait()</td>
</tr>
</tbody>
</table>

Figure 2.1: Object

All the classes in Java are descendants of a common superclass: Object. This class defines the basic state and behavior that all class instances must have, such as the ability to clone itself, to compare itself to another object, to convert to a string, to clean up an object before it is garbage collected, to wait on a condition variable, to notify other objects that a condition variable has changed, and to return the class of the object.

This is accomplished by two groups of methods. The first group includes the methods clone, the pair equals-hashcode, toString and finalize. The second group contains the methods wait, notify with its variant notifyAll, and getClass. A class
2.1. JAVA OVERVIEW

can override only the first group of methods since the others are defined final. Note that Java handles class instances by reference, for this reason we cannot compare or copy two classes directly. In the first case, we would only compare their reference and not their members. In the second case, we would obtain two variables that point to the same instance and not an independent copy of the object. Obviously, Object offers only the basic behavior for these methods, it is up to the programmer to override them according to the class structure. Nevertheless, the default behavior of the method clone is to raise a CloneNotSupportedException. In this case, only classes that implement the Cloneable interface may be cloned in order to allow an object to create a copy of itself.

Java has only a few primitive types such as boolean, char, byte, short, int, long, and double. All the other data types are objects and arrays. These types are often called reference types because they are handled by reference.

In a class, the method parameters and the return values are passed by-value. Therefore, the objects are actually passed and returned by reference while the primitive values are passed by value. A primitive value can be passed by reference only using a wrapper class.

2.1.2 Packages and visibility

Classes and interfaces can be grouped in packages. If a programmer declares some classes in a package, these classes are physically located under a directory tree whose components are the same as in the package name. For instance, given the package com.foo.pkg the corresponding directory tree is com\foo\pkg. To avoid name clashes Java proposes an Internet-wide naming scheme for the packages developed by other organizations. Thus, com.foo.pkg could be the package pkg developed by the commercial organization foo.

![Diagram of package structure](image)

Figure 2.2: package

Now, suppose that in com.foo.pkg there is a class A. If there were another package, say edu.uni.pac, containing another class A, then some precautions would be required to use both packages in the same application. Indeed, they can be used only with their full qualified name such as com.foo.pkg.A and edu.uni.pac.A.

Except for this particular case, usually the public classes in a package can be accessed from other packages without using their full qualified name, as long as the pack-
age name is in an import statement. That is, if we need an instance of `com.foo.pkg.A` we will write something like this:

```java
import com.foo.pkg.*;
...
A a=new A(); // an instance of com.foo.pkg.A
...
```

The rules followed by Java for access to packages, classes, and class members can be summarized in this way: All the elements of a package are accessible to one another. Only public classes belonging to a package are accessible within a different package. Class members are accessible from a different class within the same package as long as they are not private. Indeed, private members are accessible only within their own class. All the class members are accessible from another class in a different package whether they are public or they are protected and the other class is a subclass.

### 2.1.3 Exceptions handling

Java has a simple but effective exception mechanism. In Java the exceptions follow the termination model. A statement which raises an exception causes a non local control flow branch to the first handler of this type of exception. The `try/catch/finally` statement represents the exception handling mechanism. `Try` establishes a block of code that needs to have its exceptions handled. The `try` block is followed by zero or more `catch` clauses to handle specified types of exceptions. The catch clauses are optionally followed by a `finally` block that contains code guaranteed to be executed.

```java
try
{
// Some code
}
catch (SomeException se)
{
// handle an exception object se of type SomeException
// or of any sub-type
}
catch (AnotherException se)
{
// handle an exception object se of type SomeException
// or of any sub-type
}
finally
{
// other code that is always executed
}
```

The exceptions are divided into two groups: checked exceptions and Runtime exceptions. Only the first group requires to be explicitly handled in a `try-catch` block. Alternatively, they can be specified in the `throws` part of a method signature in order
to delegate the exception handling to the caller. Such a \textit{throws} clause might look like this:

```java
public void myOwnMethod(int arg) throws MyOwnException
{
    // Some code
}
```

While Runtime exceptions do not have to be explicitly managed since this group of exceptions represents problems that are detected by the runtime system. These exceptions include arithmetic exceptions, pointer exceptions and indexing exceptions. Since Runtime exceptions could occur everywhere and could be numerous, the benefit of catching or specifying them would be canceled by the burden that their usage would introduce. For this reason a Java programmer has different obligations with a checked exception and a Runtime exception.

### 2.1.4 Garbage collector

Java has an automatic garbage collection that allows the programmers not to be concerned about complex and error-prone memory allocation schemes.

The garbage-collector frees the memory used by objects that are no-longer needed using a mark-sweep approach. It scans dynamic memory areas for objects and marks those that are referenced. After all possible paths to objects are investigated, unmarked objects (unreferenced objects) are known to be garbage and are collected.

Before an object gets garbage-collected, the garbage collector gives the object an opportunity to clean up after itself through a call to the object’s \textit{finalize} method. This process is known as finalization.

### 2.1.5 Multithreading

Java has a built-in support for multithreading and provides a set of basic methods that makes dealing with threads much easier. Indeed, the Java programming language provides a mechanism for synchronizing the concurrent activity of threads, in order to allow the coding of programs that, though concurrent, still exhibit deterministic behavior. To synchronize threads the language uses monitors, a mechanism for allowing one thread at a time to execute a region of code. The behavior of monitors is explained in terms of locks. Java associates a lock with each object.

There is a \textit{synchronized} statement that performs two special actions relevant only to a multithreaded operation:

- After computing a reference to an object but before executing its body, it locks a lock associated with the object.
- After execution of the body is completed, either normally or abruptly, it unlocks that same lock. As a convenience, a method may be defined \textit{synchronized}; such a method behaves as if its body were contained in a synchronized statement.

The language also provides the methods \textit{wait}, \textit{notify}, and \textit{notifyAll}, belonging to the class \textit{Object}, to support an efficient transfer of control from one thread to another.
2.1.6 Under the hood: Java Virtual Machine and ClassLoader

Java is portable and platform independent since a source code for a class is compiled into a byte-code representation according to the class file format specified in [LIY99]. The representation of an individual class is referred to as a class file. The main characteristic of byte-code is its platform independence. The byte-code is interpreted into machine and operating system specific calls once it is running in a so-called Java Virtual Machine, hereafter abbreviated as JVM\(^1\).

![Java Platform](image)

Figure 2.3: Java Platform

However, byte-code sequences are only part of what a JVM needs to execute a program. A class file contains also symbolic references to fields, and names of other classes. We will see what it means later.

Notice that while the byte-code is platform independent, the Java Virtual Machine is platform specific. A specific implementation of the JVM is required for each platform that wants to execute class files. It is up to the JVM to use the system calls of the host operating system in order to actually run the loaded classes.

Moreover, classes are linked in as required and can be downloaded from across the network even though the incoming code is verified before being passed to the interpreter for execution. Both the loading and verifying operations are carried out by a specific class known as the `ClassLoader`. The purpose of a class loader is to dynamically load classes on the Java platform. This platform relies on classes as the basic unit of software distribution. In particular, the classes are introduced into the Java environment when they are referenced by name in a class that is already running. Notice that this machinery works for every class except for the first class to be loaded. The latter must declare a static method named `main` in order to specify an execution entry point such as the following application:

```java
public class MyProgram{
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
} // MyProgram
```

\(^1\)Other languages such as TCL[Ous94] use an interpreter without compiling the source code to an intermediate byte-code format.
Typing at the command prompt `javac MyProgram.java`, the class file `MyProgram.class` is created. The interpreter can load this class and execute it every time a user gives the command `java MyProgram`.

A more peculiar aspect of class loading is that a class file does not have to be stored in an actual file to be loaded. It can reside in a memory buffer, or even be obtained from a network stream. All Java virtual machines have an embedded class loader named primordial class loader. It is somewhat special because the JVM assumes that it has access to a repository of trusted classes which the JVM can run without verification. These classes are located in the directories included in a path stored in the CLASSPATH environment variable\(^2\). Usually this path contains the current directory and few other directories where the classloader seeks all the requested classes and possibly loads them.

The primordial class loader loads classes exactly in two cases: when a class bytecode is executed for the first time, for instance during a class instantiation such as:

```java
{ ...
    Dummy d;
    d=new Dummy();
    ...
}
```

and when a class bytecode makes a static reference to a class such as `System.out` where the usage of a static variable `out` requires to load the system class `PrintStream`.

After a class is loaded, it must be linked and initialized. The linking process can be divided into three different activities: verification, preparation, and resolution of symbolic references.

- Verification checks that the loaded representation of a class is well formed, with a proper symbol table. If a problem is detected during verification, an error is thrown.

- Preparation involves allocation of static storage and any data structures that are used internally by the virtual machine, such as method tables.

- Resolution is the process of checking symbolic references from a class to other classes and interfaces, by loading the other classes and interfaces that are mentioned and checking that the references are correct. In this way symbolic references are resolved to actual types.

If the linking process has succeeded, a class must be initialized. Initialization consists of execution of any class variable initializers and static initializers of a class, in textual order. But before a class can be initialized, its direct superclass must be initialized, as well as the direct superclass of its direct superclass, and so on, recursively. Instead, the initialization of an interface consists of executing the initializers for fields defined in the interface. Finally, after completion of the initialization process, a class can be instantiated and its methods or instance variables used.

An interesting scenario is when a Java application uses different kinds of class loaders to manage various software components. Indeed, in a Java application both user-defined class loaders and the primordial class loader can coexist. Although all

\(^2\)The CLASSPATH may contain not only directories but also archives of compressed class files.
the system classes can only be loaded by the primordial class loader, user-defined class loaders can be used to load classes originating from user-defined sources. Since in the JVM a class type is uniquely determined by the combination of the class name and the class loader, we can even load classes having the same name provided we use different class loaders.

Formally, class loaders provide a mechanism to create multiple separate namespaces. What’s more, each class contains a reference to its defining loader and each loader refers to all the classes it defines. For this reason, when a class loader is garbage-collected, all its connected classes are unloaded. Notice that in Java 1.2, all code, regardless of whether local or remote, can be subject to a security policy.

2.1.7 Applets

One of Java’s primary design goals is to create applets, which are little programs that run inside a Web browser. They adhere to a set of conventions. For instance, they do not have a main method, or other single entry point. Instead, they are subclasses of the Applet class that in case override a number of standard methods. At appropriate times, under well-defined circumstances, the Web browser invokes these methods. Applets are included in HTML pages by means of the HTML tag <Applet>. This tag reserves a space on the page for the Applet. When a Web browser loads a document containing the applet tag, it downloads the applet bytecode and executes it.

![Dynamic loading diagram](image)

Figure 2.4: Dynamic loading

Although in the early days of Java only the HotJava browser developed by Sun supported Java, currently all the most popular browsers have bundled a JVM that can run applets.

In this way, applets are a powerful tool in supporting client-side programming. Nevertheless, they are also a major issue for the Web programming since all the browsers at the time of this writing do not support Java 1.2. Currently, all the
available browsers support, at most, the 1.1.5 Java release. What makes this scenario worse is Microsoft Internet Explorer, since it does not support some packages such as Java RMI. In this way, the usage of Java on the client side is partially jeopardized by this lack of compliance of the browsers' JVM.

Because they must be safe, applets are limited in what they can accomplish unless differently specified. Indeed, most browsers install a security manager, so applets typically run under the scrutiny of this security manager.

Such an applet is not allowed to access resources unless the security policy in effect explicitly grants permission to do so.

2.1.8 Security architecture

The security policy defines the set of permissions available for code from various signers or locations and can be configured by a user or a system administrator. Each permission specifies a permitted access to a particular resource, such as read and write access to a specified file or directory or connect access to a given host and port.

The original security model provided by the Java platform, known as the "sandbox" model, existed in order to provide a very restricted environment in which to run untrusted code obtained from the open network. In the sandbox model local code is trusted to have full access to vital system resources, such as the file system, but downloaded remote code (such as an applet) is not trusted and can access only the limited resources provided inside the sandbox. A security manager is responsible for this. The security manager is a class that allows applications to implement a security policy. It allows an application to determine, before performing a possibly unsafe or sensitive operation, what the operation is and whether it is being attempted in a security context that allows the operation to be performed. The application can allow or not the operation.

Later, JDK 1.1 introduced the concept of a "signed applet". A digitally signed applet is treated like local code, with full access to resources, if the public key used to verify the signature is trusted. Unsigned applets are still run in the sandbox.

In the current model, JDK 1.2, the runtime system organizes code into individual domains, each of which encloses a set of classes whose instances are granted the same set of permissions. A domain can be configured to be equivalent to the sandbox, so applets can still be run in a restricted environment if the user or the administrator so chooses. Applications run unrestricted, as before, by default but can optionally be subject to a security policy.

2.1.9 Reflection mechanism

Another interesting facet of the object model in Java is the reflection mechanism. The Java reflection API is useful because it allows an executing Java program to examine or "introspect" upon itself, and manipulate internal properties of the program. In particular, it supports dynamic retrieval of information about classes, interfaces, and objects by name, and allows for their manipulation within an executing Java program in the current Java Virtual Machine.

For example, it's possible for a Java class to obtain the names of all its members and display them.
This feature is extremely powerful and has no equivalent in other conventional languages such as C, C++, Fortran, or Pascal. Since reflection has an important role in the development of the JIC compiler it will be thoroughly described in section 2.2.

2.1.10 Network Programming

The Java support for network programming comes in the form of classes that can deal directly with sockets so that the connections to servers can be opened. In particular, a socket is one end-point of a two-way communication link between two programs running on the network. Socket classes are used to represent the connection between a client program and a server program. The java.net package provides two classes—Socket and ServerSocket—that implement the client side of the connection and the server side of the connection, respectively.

There are also classes to parse network data (dealing with differences between hardware platform data representation) and to send full Java objects over the wire. Strictly speaking, the first collection of classes implements the so-called streams, while the other group implements the so-called Object Serialization functionality.

The java.io package contains a collection of stream classes that support these algorithms for reading and writing. On the other hand, Object Serialization [Sun97/4] supports the encoding of objects, and the objects reachable from them, into a stream of bytes; and it allows the complementary reconstruction of the object graph from the stream. The default encoding of objects protects private and transient data, and supports the evolution of the classes. A class may implement its own external encoding and is then solely responsible for the external format. Serialization is used for lightweight persistence and for communication via sockets or, as we will see below, for passing parameters in a Remote Method Invocation (RMI).

In addition, Java supports natively two distinct architectures for distributed object oriented computing. The first one is named Java IDL and it implements a CORBA 2.0 specification. Whilst, the other is Java RMI which proposes a proprietary but pure approach to distributed objects.

We will describe them thoroughly in a specific chapter. Notice that there are other commercial third party CORBA implementations such as OrbixWeb, VisiBroker and so on, that support the Java programming language. Moreover, there is a new breed of architectures developed in Java called Agent platforms. These platforms use in an extensive way Java’s main feature: dynamic class loading. The most famous among these platforms are IBM Aglets and Mitsubishi Concordia. There is also the Objectspace Voyager product that could be presented as a hybrid architecture since it supports both distributed object oriented computing and mobile agents computing.

2.1.11 Summary

With the network programming section we conclude this overview. Notice that we have disregarded some Java packages because they do not play a significant role in our proposal. Just to mention a few that have a main role in the so-called Java for Enterprise, we have neglected the Java Database Connectivity (JDBC) and JavaBeans

---

3See [ADJ96] for further details on an implementation of a persistent version of Java.
packages. In a nutshell, JDBC provides a uniform access to a wide range of relational databases while JavaBeans represents the Java component architecture.

2.2 Java Reflection in detail

The Java Core Reflection API [SUN97/2] has a key role in our DTC compiler and in the generated runtime support. For this reason its features will be thoroughly explained.

This package provides a small, type-safe, and secure API that supports introspection about the classes and objects in the current Java Virtual Machine. The ability to examine and manipulate a Java class from within itself may not seem so innovative, but in other programming languages this feature simply does not exist. For example, there is no way in a Pascal, C, or C++ program to obtain information about the functions defined within that program.

The Core Reflection API is in a subpackage of java.lang named java.lang.reflect. This avoids compatibility problems caused by Java’s default package importation rules. If permitted by security policy, the API can be used to:

- construct new class instances and new arrays
- access and modify fields of objects and classes
- invoke methods on objects and classes
- access and modify elements of arrays

Fundamentally, the Reflection API consists of two parts: classes that represent the various parts of a class, and a means to extract these parts in a safe and secure way.

The main role in this package is played by the class Class. Each Java class has an associated unique instance of this class. A class representation may be obtained by calling the method getClass(), inherited from the root class Object. An instance of Class can be manipulated by a running Java program. The programmer cannot create such instances since Class has no public constructors; instead they are constructed by the JVM when classes are loaded. Moreover, the reflection API is symmetric, which means if you are holding a Class object, it is possible to obtain its members, and if having one of the members, it is possible to ask which class defined it.

Part of the definition of Class, relevant to our discussion, is shown in figure 2.5. Some methods and details of exceptions have been omitted to save space.

The Reflection API defines the following classes and methods:

- Three classes –Field, Method, and Constructor– that reflect class, interface members and constructors. These classes provide a reflective information about the underlying member or constructor and a type-safe means to use the member or constructor to operate on Java objects.

- Some methods of class Class that provide for the construction of new instances of the Field, Method, and Constructor classes.

- One class, Array, that provides methods to dynamically construct and access Java arrays.
public final class Class {
  static Class forName(String className)
  static Class forName(String name, boolean initialize, ClassLoader loader)
  Class[] getClasses()
  ClassLoader getClassLoader()
  Class getComponentType()
  Constructor[] getConstructors()
  Class[] getDeclaredClasses()
  Constructor[] getDeclaredConstructors()
  Field[] getDeclaredFields()
  Method[] getDeclaredMethods()
  Class getDeclaringClass()
  Field getField(String name)
  Field[] getFields()
  Class[] getInterfaces()
  Method[] getMethods()
  int getModifiers()
  String getName()
  Package getPackage()
...
  Class getSuperclass()
  boolean isArray()
  boolean isAssignableFrom(Class cls)
  boolean instance(Object obj)
  boolean isInterface()
  boolean isPrimitive()
  Object newInstance()
  String toString()
}

Figure 2.5: the class Class
A utility class, *Modifier*, that helps decode Java language modifier information about classes and their members.

There are also some classes belonging to the `java.lang` package that support reflection. These classes are:

- Two classes, *Byte* and *Short*. These classes are subclasses of the class *Number*, and are similar to the class *Integer*. Instances of these classes serve as object wrappers for primitive values of type *byte* and *short*, respectively.
- Some objects, instances of the class *Class*, to represent respectively the primitive Java types *boolean*, *byte*, *char*, *short*, *int*, *long*, *float*, and *double*, and the keyword *void*, at run-time.
- An uninstantiable placeholder class, *Void*, to hold a reference to the *Class* object representing the keyword *void*.

To see how reflection works, consider this simple example: a class *DumpMethods* that prints all the public methods of class that receives in input.

```java
import java.lang.reflect.*;
public class DumpMethods {
    public static void main(String args[]) {
        try {
            Class c = Class.forName(args[0]); // loads the class
            Method m[] = c.getDeclaredMethods();
            for (int i = 0; i < m.length; i++)
                System.out.println(m[i].toString());
        }
        catch (Throwable e) {
            System.err.println(e);
        }
    }
}
```

For an invocation of:

```bash
class DumpMethods java.util.Stack
```

It prints the name of all the public methods of the system class *Stack*.

### 2.2.1 Reflection Model

The three classes *Field*, *Method*, and *Constructor* are *final*. Only the Java Virtual Machine may create instances of these classes; these objects are used to manipulate the underlying objects; that is, to:

- get reflective information about the underlying member or constructor
• get and set field values
• invoke methods on objects or classes
• create new instances of classes

As mentioned before the main classes of the Java reflection are Field, Method and Constructor. Let us describe them.

All the classes Field, Method and Constructor implement the Member interface. The methods of Member are used to query a reflected member for basic identifying information. Identifying information consists of the class or interface that defined the member, the name of the member itself, and the Java language modifiers (such as public, protected, abstract, synchronized, and so on) for the member.

A Field object represents a reflected field. The underlying field may be a class variable (a static field) or an instance variable (a non-static field). Methods of class Field are used to obtain the type of the underlying field, and to get and set the underlying field’s value on objects.

A Method object represents a reflected method. The underlying method may be an abstract method, an instance method, or a class (static) method. The methods of class Method are used to obtain the formal parameter types, the return type, and the checked exception types of the underlying method. In addition, the invoke method of class Method is used to invoke the underlying method on target objects. Instance and abstract method invocation uses dynamic method resolution based on the target object’s run-time class and the reflected method’s declaring class, name, and formal parameter types. Static method invocation uses the underlying static method of the method’s declaring class.

A Constructor object represents a reflected constructor. Methods of class Constructor are used to obtain the formal parameter types and the checked exception types of the underlying constructor. In addition, the newInstance method of class Constructor is used to create and initialize a new instance of the class that declares the constructor, provided the class is instantiable.

To decode Java language modifiers for classes and members the Reflection APIs includes the Modifier class that is an uninstantiable class. The language modifiers are encoded in an integer, and use the encoding constants defined by The Java Virtual Machine Specification [LIY99].

A special kind of class is the Array class since it is an uninstantiable class that exports class methods to create Java arrays with primitive or class component types. Methods of class Array are also used to get and set array component values.

Finally, there are nine Class objects that are used to represent the eight primitive Java types and void at run-time. (Note that these are Class objects, not classes.) The Core Reflection API uses these objects to identify the following:

• primitive field types
• primitive method and constructor parameter types
• primitive method and constructor parameter types
• primitive method return types
2.2. JAVA REFLECTION IN DETAIL

The Java Virtual Machine creates these nine Class objects. They have the same names as the types that they represent. The Class objects may only be referenced via the following public final static variables java.lang.XXX.TYPE, where XXX can be of the following wrappers such as Boolean, Character, Byte, Short, Integer, Long, Float, Double, and Void. In particular, these Class objects are not accessible via the forName method of class Class.

2.2.2 Data Conversions

Certain methods in the reflection package perform automatic data conversions between values of primitive types and objects of class types. These are the generic methods for getting and setting field and array component values, and the methods for method and constructor invocation.

There are two types of automatic data conversions. Wrapping conversions convert from values of primitive types to objects of class types. Unwrapping conversions convert objects of class types to values of primitive types. Additionally, field access and method invocation permit widening conversions on primitive and reference types. These conversions are documented in [GJS96].

A primitive value is automatically wrapped in an object when it is retrieved via Field.get or Array.get, or when it is returned by a method invoked via Method.invoke. Similarly, an object value is automatically unwrapped when supplied as a parameter in a context that requires a value of a primitive type. These contexts are:

- Field.set, where the underlying field has a primitive type.
- Array.set, where the underlying array has a primitive element type.
- Method.invoke or Constructor.newInstance, where the corresponding formal parameter of the underlying method or constructor has a primitive type.

Table 2.1 shows the correspondences between primitive types and class (wrapper) types.

<table>
<thead>
<tr>
<th>Primitive type</th>
<th>Reference type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>java.lang.Boolean</td>
</tr>
<tr>
<td>char</td>
<td>java.lang.Character</td>
</tr>
<tr>
<td>byte</td>
<td>java.lang.Byte</td>
</tr>
<tr>
<td>short</td>
<td>java.lang.Short</td>
</tr>
<tr>
<td>int</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>long</td>
<td>java.lang.Long</td>
</tr>
<tr>
<td>float</td>
<td>java.lang.Float</td>
</tr>
<tr>
<td>double</td>
<td>java.lang.Double</td>
</tr>
</tbody>
</table>

Table 2.1: Association between Primitive types and Reference Types

A method that is declared void returns the special reference null when it is invoked via Method.invoke. The reflection package permits the same widening conversions at run-time as permitted in method invocation contexts at compile time. These conversions are defined in The Java Language Specification, section 5.3 [GJS96].
2.2.3 Applications

The Core Reflection API accommodates three categories of applications:

- One category is comprised of applications that need to discover and use all of the public members of a target object based on its run-time class. These applications require run-time access to all the public fields, methods, and constructors of an object. Examples in this category are services such as Java Beans, and lightweight tools, such as object inspectors. These applications use the instances of the classes Field, Method, and Constructor obtained through the methods getField, getMethod, getConstructor, getFields, getMethods, and getConstructors of class Class. For instance, JavaBeans software components can be manipulated visually via a builder tool. The tool uses reflection to obtain the properties of Java components (classes) as they are dynamically loaded.

- The second category consists of sophisticated applications that need to discover and use the members defined by a given class. These applications need run-time access to the implementation of a class at the level provided by a class file. Examples in this category are development tools, such as debuggers, interpreters, inspectors, and class browsers, and run-time services, such as Java Object Serialization. These applications use instances of the classes Field, Method, and Constructor obtained through the methods getDeclaredField, getDeclaredMethod, getDeclaredConstructor, getDeclaredFields, getDeclaredMethods, and getDeclaredConstructors of class Class.

- Finally, the third category includes applications that use the availability of information on the structure of classes via Class instances in order to allow programs to generate new programs based on analysis of existing objects [Tre98]. This is our case. Indeed, a complete description of the class of an object may be built up by invoking the object's getClass method, followed by invoking the various class methods to obtain representations of the class's fields, methods, constructors, superclass and interfaces. These representations in turn provide methods that allow their structure, for example the types of fields and the parameter types of methods, to be discovered. Where such components are object types the process can be continued recursively. Given the ability to discover the structure of a class, it is possible to write new code fragments or even new classes.

We will see later how all the features of the Reflection package are used in order to reflect all the methods of a class and create two new classes, named forwarder and receiver, that have roughly the same interface of the original class. Besides, the reflection API is also used, at run-time, by the receiver to forward all the remote method invocations to the corresponding method of the original class.
Chapter 3

Object Oriented Distributed Architectures in Java

With the advent of Java and its modern language features, described previously, like threads, reflection, serialization and garbage collection, it is possible to build robust distributed object systems that are simple to learn and use. Sun RMI and ObjectSpace Voyager are two examples of this new approach, and the shortcomings of CORBA for Java are gradually being ironed out as well. All of the distributed object systems mentioned run or will eventually run over the IIOP\(^1\) protocol, which could help bring about a transition from HTTP\(^2\) to IIOP on the Internet. Only time will tell whether the simplicity of HTTP or the power of IIOP will prevail as the protocol of choice for the Internet.

In this chapter we will describe the main features of a specific CORBA implementation for Java included in the Java JDK named Java IDL and the two most popular pure Java solutions, Sun Java RMI and Objectspace Voyager.

All these approaches for the development of distributed applications will be compared with each other. The comparison will focus on the services offered and their usability for the average programmer. In the final part of this chapter we will explain why, in our opinion, Voyager has the most innovative features with respect to the other architectures and why it is simpler to learn and use than Java IDL and Java RMI.

3.1 Java IDL

The Java IDL API [McC98] provides an interface between Java programs and distributed objects and services built using the Common Object Request Broker Architecture (CORBA). CORBA is a standard defined by the Object Management Group (OMG). It describes an architecture, interfaces, and protocols that distributed objects can use to interact with each other.

\(^{1}\) Internet Inter-ORB Protocol.
\(^{2}\) HyperText Transfer Protocol.
CHAPTER 3. OBJECT ORIENTED DISTRIBUTED ARCHITECTURES IN JAVA

Part of the CORBA standard is the Interface Definition Language (IDL), which is an implementation-independent language for describing the interfaces of remote-capable objects. There are standard mappings defined by the OMG for converting IDL interfaces into C++ classes, C code, and Java classes, among other things. These generated classes use the underlying CORBA framework to communicate with remote clients and give the basis for implementing and exporting user-defined distributed objects. Java IDL is an implementation of the standard IDL-to-Java mapping, compliant with the CORBA 2.x specification, and is provided by Sun with the standard Java JDK in the org.omg.CORBA and org.omg.CosNaming packages and their subpackages.

Java IDL provides a way to access remote objects over the network. It also provides the tools you need to make your objects accessible to other CORBA clients. If a Java class is exported using Java IDL, it is possible to create an instance of that class and publish it through a naming/directory service. A remote client can find this object, call methods on it, and receive data from it, just as if it were running on the client’s local JVM.

The CORBA standard is extensive. In addition to the basic remote object architecture and the syntax of IDL, it also includes specifications for several distributed object services, like an object naming service, a security policy service, and persistent object services.

We'll cover the basic features of the CORBA architecture and the IDL syntax. We'll also look at the Naming Service, which is the key to almost every CORBA application because it provides a standard way to find remote CORBA objects on the network.

3.1.1 The CORBA Architecture

At its core, the CORBA architecture for distributed objects shares many features with the architecture used by Java RMI, as we will see in the next section. A description of a remote object is used to generate a client stub interface and a server skeleton interface for the object. A client application invokes methods on a remote object using the client stub. The method request is transmitted through the underlying infrastructure to the remote host, where the server skeleton for the object is asked to invoke the method on the object itself. Any data resulting from the method call (return values, exceptions) is transmitted back to the client by the communication infrastructure.

But that’s where the similarities between CORBA and RMI end. CORBA was designed from the start to be a language-independent distributed object standard, so it is much more extensive and detailed in its specification than RMI is. For the most part, these extra details are required in CORBA because it needs to support languages that have different built-in features. Some languages, like C++, directly support objects, while others, like C, don’t. The CORBA standard needs to include a detailed specification of an object model so that non-object-oriented languages can take advantage of CORBA. Java includes built-in support for communicating object interfaces and examining them abstractly (using Java bytecodes and the Java Reflection API). Many other languages do not. So the CORBA specification includes details about a Dynamic Invocation Interface and a Dynamic Skeleton Interface, which can
be implemented in languages that don’t have their own facilities for these operations. In languages that do have these capabilities, like Java, there needs to be a mapping between the built-in features and the features as defined by the CORBA specification.

The core of the CORBA architecture is the Object Request Broker, as shown in Figure 3.1. Each host involved in a CORBA application must have an ORB running in order for processes on that machine to interact with CORBA objects running in remote processes. Object clients and servers make requests through their ORBs; the ORB is responsible for making the requests happen or indicating why they cannot.

The client ORB provides a stub for a remote object. Requests made on the stub are transferred from the client’s ORB to the ORB servicing the implementation of the target object. The request is passed onto the implementation through its skeleton interface.

![Image of CORBA architecture](image)

**Figure 3.1: Basic CORBA architecture**

Version 2.0 (and later) of the CORBA standard includes specifications for inter-ORB communication protocols that can transmit object requests between various ORBs running on the network. The protocols are independent of the particular ORB implementations running at either end of the communication link. An ORB implemented in Java can talk to another ORB implemented in C, as long as they’re both compliant with the CORBA standard and use a standard communication protocol. The inter-ORB protocol is responsible for delivering messages between two cooperating ORBs. These messages might be method requests, return types, error messages, etc. The inter-ORB protocol also deals with differences between the two ORB implementations, like machine-level byte ordering and alignment. A CORBA application developer shouldn’t have to deal directly with the low-level communication protocol between ORBs. If he wants two ORBs to talk to each other, he just needs to be sure that they both speak a common, standard inter-ORB protocol.

The Internet Inter-ORB Protocol (IIOP) is based on TCP/IP. TCP/IP is by far the most commonly used network protocol on the Internet, so IIOP [OMG99] is the most commonly used CORBA communication protocol. There are other standard CORBA protocols defined for other network environments, however. The DCE Common Inter-ORB Protocol (DCE-CIOP), for example, allows ORBs to communicate...
on top of DCE-RPC.

### 3.1.2 The Naming Service

The CORBA Naming Service provides a directory naming structure for remote objects. The tree always starts with a root node, and subnodes of the object tree can be defined. Actual objects are stored by name at the leaves of the tree. The fully qualified name of an object in the directory is the ordered list of all of its parent nodes, starting from the root node and including the leaf name of the object itself.

Each branch in the directory tree is called a naming context, and leaf objects have bindings to specific names. The `org.omg.CosNaming.NamingContext` interface represents each branch in the naming directory. Each NamingContext can be asked to find an object within its branch of the tree by giving its name relative to that naming context. It is possible to get a reference to the root context of the naming directory from an ORB using the `resolve_initial_references()` method. The standard name for the Naming Service is "NameService", so the following code snippet gets the root NamingContext:

```java
ORB myORB = ORB.init(...);
org.omg.CORBA.Object nameRef =
    myORB.resolve_initial_references("NameService");
NamingContext nc = NamingContextHelper.narrow(nameRef);
```

Note that we have to narrow the Object reference to a NamingContext reference using the `NamingContextHelper.narrow()` method. Even though Java has a cast operation in its syntax, there's no guarantee in the Java IDL binding that the object reference returned by the `resolve_initial_references()` method is the correct type, since there's no guarantee that the local environment has access to the language-specific definition of the object’s interface.

### 3.1.3 The Interface Definition Language

The Interface Definition Language provides the primary way of describing data types in CORBA. IDL is independent of any particular programming language. Mappings, or bindings, from IDL to specific programming languages are defined and standardized as part of the CORBA specification. At the time of this writing, standard bindings for C, C++, Smalltalk, Ada, COBOL, and Java have been approved by the OMG.

The central CORBA functions, services, and facilities, such as the ORB and the Naming Service, are also specified in IDL. This means that a particular language binding also provides the bindings for the core CORBA functions to that language. Sun's Java IDL API follows the Java IDL mapping defined by the OMG. This allows you to run your CORBA-based Java code in any compliant Java implementation of the CORBA standard, provided you stick to standard elements of the Java binding. Note, however, that Sun's implementation includes some nonstandard elements.

Let us give a quick overview of writing a CORBA interface in IDL. The syntax of both Java and IDL were modeled to some extent on C++, so there are a lot of similarities between the two in terms of syntax. Interfaces in IDL are declared much like classes in C++ and, thus, classes or interfaces in Java. Here's a complete IDL
example that declares a module within another module, which itself contains several interfaces:

```java
module DS {
    module services {
        interface Server {
            readonly attribute string serverName;
            boolean init(in string sName);
        };

        interface Printable {
            boolean print(in string header);
        };

        interface PrintServer : Server {
            boolean printThis(in Printable p);
        };
    };
}
```

The first interface, `Server`, has a single read-only string attribute and an `init()` method that accepts a string and returns a boolean. The `Printable` interface has a single `print()` method that accepts a string header.

Finally, the `PrintServer` interface extends the `Server` interface (hence inheriting all its methods and attributes) and adds a `printThis()` method that accepts a `Printable` object and returns a boolean. In all cases, we’ve declared our method arguments as input-only (i.e., pass-by-value), using the `in` keyword.

Once described these remote interfaces in IDL, a programmer needs to generate Java classes that act as a starting point for implementing those remote interfaces in Java using an IDL-to-Java compiler. Every standard IDL-to-Java compiler generates the following Java classes from an IDL interface:

- A Java interface with the same name as the IDL interface. This can act as the basis for a Java implementation of the interface (but a programmer has to write it, since IDL doesn’t provide any details about method implementations).

- A helper class, whose name is the name of the IDL interface with "Helper" appended to it (e.g., `ServerHelper`). The primary purpose of this class is to provide a static `narrow()` method that can safely cast CORBA Object references to the Java interface type. The helper class also provides other useful static methods, such as `read()` and `write()` methods that allow you to read and write an object of the corresponding type using I/O streams.

- A holder class whose name is the name of the IDL interface with "Holder" appended to it (e.g., `ServerHolder`). This class is used when objects with this interface are used as out or inout arguments in remote CORBA methods. Instead of being passed directly into the remote method, the object is wrapped with its holder before being passed.
CHAPTER 3. OBJECT ORIENTED DISTRIBUTED ARCHITECTURES IN

JAVA

When a remote method has parameters that are declared as out or inout, the method
has to be able to update the argument it is passed and return the updated value.
The only way to guarantee this, even for primitive Java data types, is to force out
and inout arguments to be wrapped in Java wrapper classes, which are filled with the
output value of the argument when the method returns.

The idltojava tool provided by Sun can also generate two other classes:

- A client stub class, called _interface-nameStub, that acts as a client-side im-
  plementation of the interface and knows how to convert method requests into
  ORB requests that are forwarded to the actual remote object. The stub class
  for an interface named Server is called _ServerStub.

- A server skeleton class, called _interface-nameImplBase, that is a base class for
  a server-side implementation of the interface. The base class can accept requests
  for the object from the ORB and channel return values back through the ORB
  to the remote client. For instance, the skeleton class for an interface named
  Server is called _ServerImplBase.

So, in addition to generating a Java mapping of the IDL interface and some helper
classes for the Java interface, the idltojava compiler also creates subclasses that act
as an interface between a CORBA client and the ORB and between the server-side
implementation and the ORB.

3.1.4 Summary

In order to draw our conclusions, let us see what a programmer has to do in order to
implement a distributed application using the basic facilities of Java IDL. First, he
has to write the IDL specification of the remote services he wants to offer. Second, he
compiles these IDL specifications in order to create the corresponding Java interface,
the helper class, the holder class, stub and skeleton. Finally, he has to write a class
on the client side and another on the server side.

On the server side, he has to create a subclass, the actual server class, of the
_interfacenameImplBase class in order to provide the actual implementation of the
methods. In turn, an instance of this class has to be registered, using a reference to a
local or remote ORB, with the local CORBA Naming service under a specific name
register.

On the client side, a Java client gets a reference to a remote object using the
naming service. The reference obtained is actually a stub that requires to be narrowed,
using the helper class, in order to assign it to an interface variable. This interface
corresponds to the Java interface generated by the IDL compiler. Now, the client is
ready to invoke the remote methods by means of this variable.

In our opinion, this approach is cumbersome, makes difficult to create distributed
applications that can pass complex objects during remote method invocation and,
above all, class inheritance among remote classes is not supported.
3.2 Java Remote Method Invocation

Java RMI provides a light CORBA-like functionality based on a protocol called the Java Remote Method Protocol (JRMP).

From the programmer’s point of view using Java RMI roughly consists of defining method calls in a server object and calling them from a client object. In this way, a server object defines an interface which can be used to access it outside of the current Java Virtual Machine and on another JVM that might or might not be running in the same host. As with Java IDL, there is an RMI code generator (RMIC) that creates a stub and a skeleton. For instance, given a remote object Impl, the stub generated is Impl_Stub, and the skeleton is Impl_Skel. Both have the same set of methods that are defined in the interface ImplInt implemented by Impl. This interface declares all the methods that will be remotely invoked.

3.2.1 RMI at work

When a client makes a remote method call, what is called is not the method coded, but a method of an automatically generated class that implements an interface, say ImplInt. An RMI stub is a client class that implements this interface; a skeleton is a server class that unmarshals (converts from a stream) incoming data, then calls the actual method implementation.

For a client to locate a server object for the first time, RMI depends on a naming mechanism called RMIRegistry (a daemon program) that runs on the server machine and holds information about available server objects. A server program that wants to make an object remotely invokable must first export the object.

When a client program connects to an RMI server and calls Naming.lookup(), it returns what appears to be an instance of the remote object class. Indeed, it has the same behavior although it is remote. Notwithstanding, it is actually a different class type that also implements the remote interface defined and incorporates the additional behavior needed by the RMI protocol. From then on, all remote object references received are the result of remote calls since the client program is always dealing with a proxy (stub) object. The duty of the stub is to pass calls remotely. Java RMI server objects are named using URLs and for a client to acquire a server object reference, it should specify the URL of the server object. The URL is composed by the host-name, the port number and the name assigned to the server object.

Notice that a Java RMI client can download a stub only if there is an http-server running on the host where the server object is active. Obviously, the stub must be located in a public directory of the http-server.

On the server side, the skeleton interprets the incoming stream, to determine which method of the server to call. It then makes the call, supplying parameters it obtained from the stream. Once in the server implementation code, all the calls are local unless remote callbacks are made.

Notice that callbacks, from a remote server object to its clients, are an interesting feature of Java RMI. They can be made by providing the server object with a remote reference to another server. Alas, this technique implies the risk of deadlock. This risk can be avoided by creating a dedicated thread to accept the callback in the client.

For an object to accept remote requests it must implement an interface that ex-
tends `java.rmi.Remote`, the interface exposes a set of methods, which are indicative of the services offered by the server object. The server object must be exported by calling the method `exportObject()` of the `UnicastRemoteObject` class. Alternatively, the remote object extends the class `UnicastRemoteObject()` from the package `java.rmi.server`, since this class automatically exports instances of itself through its constructor calling the method `exportObject(this)`, and also implements the requested behavior for a remote object through the method `equals()` and `hashCode()`.

```
 Interfaces
   Remote --> RemoteObject
   RemoteServer --> RemoteException

 Classes
   activation --> UnicastRemoteObject

 Figure 3.2: RMI Classes
```

The RMI system provides the usual two ways of passing objects in a remote call. Nevertheless, in this context, passing by reference and by value has a slightly different meaning with respect to the standard pass by-value and pass by-reference of the Java object model. The object is passed by reference when a remote reference to an object is transferred from the host to the client or vice versa. In actuality, a lightweight reference object is passed (the stub object), which contains enough information for the recipient to identify the instance of the host where the actual object resides. An object is passed by remote reference if it extends `java.rmi.UnicastRemoteObject`, which provides the behavior needed for the object to support remote referencing. All the other parameters are passed by-value.

To pass the parameters by-value Java RMI relies heavily on Java Object Serialization, which allows objects to be marshaled (or transmitted) as a stream. Objects that are serializable must implement `java.io.Serializable`. The Serializable interface is just a flag to the system that a type is serializable. Many standard Java classes are serializable by default. This parameter is completely transferred including all the objects referenced inside. The recipient receives a complete copy of the object, referenced objects included; the Java reference that is obtained is a handle to this local copy. Note that the programmer can control what is passed and what is not; he can use the keyword `transient` to mark references that are volatile or unserializable and that should not be traversed when copying an object for transfer.

### 3.2.2 Advanced RMI

Java RMI allows to reactivate a remote object. In fact, if a client uses a remote reference and the connection from its stub to the target skeleton does not exist any
more, the corresponding server object can be reactivated, assuming that is activable (via the exportObject() method of the Activable class). In Java RMI an activable object ImplAct can register with the RMD (RMI Daemon) running at its node, and consequently a remote reference to ImplAct will also contain information. If the remote reference is used and the corresponding connection to the target skeleton does not exist any more, the RMD indicated in the reference is contacted instead; the RMD reactivates the server object and provides the client proxy with an updated remote reference.

One of the most interesting and least known features of Java RMI is the use of so-called behavior objects. They represent the capability to pass true objects (data and code) between virtual machines without having to distribute the supporting class files. This is achieved by the Java RMI's built-in capability to dynamically load class files from remote machines to support passing behavior objects as method parameters of remote method calls.

A behavior object is an instance that implements an interface, where the interface is used to declare a remote method formal parameter. All that the object knows about the parameter value is that it implements an interface with some methods available. In this way every object that implements this interface can be passed as actual parameter when the remote method is called. In a nutshell, behavior objects provide a powerful mechanism for defining extensible, distributed object-oriented frameworks. Moreover, they can be used to build mobile objects systems [Tsv97]. In fact, passing as parameters objects whose class extends the Thread class and calls the run method on their arrival is a simple yet powerful way to build a mobile agent. Unfortunately, only the state can be transmitted among remote objects. At the time of this writing, there is only a prototype that attempts to freeze the execution thread of a behavior object, send it and restore on another JVM, see [Fun98] for further details.

### 3.2.3 The system architecture

The RMI framework is divided into three layers (top-down): the Stub/Skeleton layer, the Remote Reference Layer and the Transport layer.

![Figure 3.3: RMI architecture](image-url)
The Stub/Skeleton layer is responsible for transmitting data to and receiving data from the remote reference layer. The Remote reference layer is responsible for carrying out method invocations. The transport layer is responsible for connection setup, connection management and keeping track of and dispatching to remote objects. To summarize, the remote reference layer and the stub/skeleton layer implement the Broker for the RMI system.

Let us see what happens under the hood of the RMI framework during the life cycle of a distributed application. The export operation implies the instantiation of the skeleton object, the stub object, the reference object and the transport endpoint object in the server’s JVM. When the server object is bound via the bind or rebind operation, the stub, the client side reference object and the transport object are serialized and moved into the name server.

To invoke a remote object Impl, a client must first obtain a reference for it. This is achieved in one of two ways. The most common is when a client does a lookup operation on the name server RMIRegistry. This results in the instantiation of Impl Stub and other needed reference objects in the client’s JVM. The other way is when a client receives a remote reference to an object as an argument of an invocation. At the time the object reference is unmarshalled, the impl Stub and other related objects are instantiated to enable the client to remotely invoke Impl.

Java RMI performs a distributed garbage collection of remote objects together with the standard mechanism bundled in the JVM. The distributed garbage collector, running in the server’s JVM, keeps track of client references for the remote object Impl. In particular, client nodes lease Impl for a certain period of time and each client reference increments its reference count by one. If a client’s lease ends, the reference gets decremented and when this reference count becomes zero, Impl can be garbage collected.

Remote programs are multithreaded programs. A server program creates an instance of an object to service remote requests. When a request arrives, it runs (usually) in a new thread. Sun states that calls originating from different client virtual machines will execute in different threads. Other calls originating from the same client virtual machine might or might not execute in the same thread. This requires that all the methods are reentrant and guarded against simultaneous access to state objects. A way to achieve this is to make all remotely exported methods synchronized. Thus, each incoming request will have to wait for any prior request to complete. In case a higher rate of requests is expected, it is better to use a finer granularity in what is synchronized and what is not.

3.2.4 Summary

In conclusion, RMI provides an extremely powerful facility for creating truly distributed programs. Since Java/RMI relies on Java, it can be used on diverse operating system platforms as long as there is a Java Virtual Machine (JVM) implementation for that platform.

In fact, although Java RMI is not a strong mobile computing system like Telescript [Wht96] (since the objects that are moved are not closures and do not have their own thread of control, see [GhV97] for further details), it is not even a traditional RPC system in which data is moved to the site of computation. Indeed, Java RMI
makes essential use of the Java ability to move code around a network in a machine-independent and safe way. Such a system suggests a new approach of doing distributed computing that combines techniques from both traditional distributed computing systems and mobile computing systems. There is a drawback with this approach: Java Object Serialization is specific to Java, both the Java RMI server object and the client object have to be written in Java. Fortunately, this disadvantage will disappear with the new RMI over IIOP platform developed by Sun and IBM.

This approach combines the best of the two worlds: Java and CORBA. In fact, it allows to pass any serializable object by value like the normal Java RMI. Like CORBA, RMI over IIOP [SUN99/2] is based on open standards defined by the OMG. In this way, the integration of legacy applications is simplified. Components written in other languages such as C++, Smalltalk, and all the CORBA compliant languages, can communicate with components running on the Java platform. Alternatively, it is possible to use the JNI (Java Native Interface) [Sun97/3] to build distributed systems that communicate with existing servers written in other programming languages.

Notice that RMI, as Java IDL, requires the programmer to learn new APIs to build a distributed application and it does not support class inheritance between remote classes.

3.3 Java IDL versus Java RMI

It’s easy to see that RMI and Java IDL technologies are fairly similar. Like RMI, Java IDL provides a way to access remote objects over the network. Unlike RMI, however, objects that are exported using CORBA can be accessed by clients implemented in any language with an IDL binding. It is also obvious that each one has several inherent advantages that the other will never be able to match. Java RMI has a huge advantage in the ease-of-use category because the developer assumes from the start that the client and server will both be written in Java. When this assumption is made, the full capabilities of the Java language can be exploited.

For instance, in Java RMI we can simply return a Vector object as a result of a method invocation. In the Java IDL example, a more complicated construct would be required. Meanwhile, Java IDL has the advantage of working with legacy code written in C++, COBOL, Ada, or Smalltalk and the added advantage of the CORBA distributed architecture and all its capabilities.

Another meaningful difference is that Java IDL does not attempt to perform distributed garbage collection, while Java RMI offers this service.

Finally, Java RMI does not need to do a narrow() operation, as Java IDL, when a client obtains a reference to a remote object. In fact, an RMI client can automatically download the bytecodes for a remote stub from the object server, if the class for the stub cannot be found locally. Only a simple cast is needed to use the stub through an interface. While, in CORBA, since it is a language-independent remote object scheme, there is no portable way to specify a remote object’s type when a client obtains a stub reference. As a result, the stub reference is initially represented by a basic ObjectImpl object that knows how to forward methods requests to its server object. Subsequently, the client application is forced to "cast" this stub to the correct local type, using the appropriate narrow() method. This means calling the narrow() method on the corresponding helper class. The narrow() method converts the reference, making a
CHAPTER 3. OBJECT ORIENTED DISTRIBUTED ARCHITECTURES IN JAVA

type-specific stub interface that also includes the remote object reference information needed to forward method requests to the actual object implementation.

For all these reasons, our early choice to develop the distributed hierarchy architecture was Java RMI.

3.4 ObjectSpace Voyager

The Voyager project started in mid-1996, ObjectSpace's first release of Voyager dates back to April, 1997. At the time of this writing ObjectSpace has just released the third version. Nevertheless, we will refer mainly to the second version used for the development of the DIC compiler. The Voyager ORB aims at simplifying distributed computing since it has the ability to create objects remotely\(^3\), to invoke their methods and to migrate them.

The ObjectSpace Voyager 2.0 system is capable of taking an existing Java class and "creating" it to create a new class with an identical interface. This special helper class, in Voyager terminology, is called a "proxy" class. This proxy is a subclass of a predefined Voyager class Proxy. It is created in the image of another class and seems just like the original to anyone who uses it. But there is a subtlety: a proxy object does no real work by itself. Instead, a proxy object, a sort of surrogate, uses network communications to create and remotely control an instance of the real class it represents. In other words, the proxy object acts like a mediator between the caller and the real object. It is through proxies that all remote methods calls are made. We can say that Voyager has remote-enabled a class without modifying the original class.

Voyager uses Java reflection to inspect a class and create the byte codes of a proxy class for it at runtime. Whenever Voyager needs a proxy class, it generates byte codes for it on the fly and uses the Java classloader to bring the new code into the JVM. In fact, the new classes need not even be written to disk. Voyager does not require to run external compilers such as RMIC as in Java RMI or Java IDL to create stub and skeleton. To support the creation and "housing" of remote objects requires that a Java virtual machine (JVM) be running on each actual machine involved. To provide this, Voyager comes with a server program built right in along with its other classes. It plays the same role as the ORB (object request broker) does in CORBA and Java RMI systems: creating and managing remote objects on behalf of the programs that use them over a network.

In Voyager an object can be created directly on the JVM where it will receive remote calls, in this case the programmer uses a naming service where an object created must be bound in order to be accessible from other JVMs. When a client wants to access this object, it must query the naming service where the object has been bound. The naming service returns a proxy to the object. This approach is almost the same as we have seen with Java RMI.

Voyager allows also to create an object at a specified location using Factory.create(). This factory method returns a proxy to the newly created object and generates dynamically the proxy class if it does not already exist.

Mainly Voyager relies on interfaces to invoke remote methods. In fact, provided that an object implements an interface whose name starts with the letter "II", the

\(^3\)In our knowledge, Voyager is the only commercial platform that has this feature.
proxy returned from an object creation implements this interface. In this way a remote method invocation can be executed using a variable having as type the same interface implemented by the proxy. The only drawback is the need to have the interface available locally. There is an alternative approach to invoke remote methods instead of using a variable whose type is the interface implemented by the class and the proxy. In fact, Voyager has a flexible communication facility that provides asynchronous, synchronous, and future remote method calls through its classes Sync, OneWay and Future. All of them have a method invoke that directs the call to the object represented by a Proxy instance received in input. In this way, we can invoke methods of classes that do not implement any interface.

3.4.1 How it works

Voyager uses Sockets, Reflection, Serialization, and Threads. Since the Java classes make working with TCP/IP sockets relatively easy, setting up a network connection between two programs is just a matter of enlisting two built-in Java classes: Socket and ServerSocket. With a connection established, Java makes input and output over that connection easy by using the same stream classes you would use for file I/O.

In addition, Java Reflection makes it possible to programmatically discover the member variables and methods of a class at runtime. Reflection can even be used to create objects and invoke methods dynamically using the special classes Constructor and Method. This advanced language feature makes it possible for Voyager to inspect the methods of an arbitrary class, generate a proxy class to match, and load up the
byte codes for the proxy class.

Serialization is the process of taking the member data of an object and representing it as a serial stream of bytes, usually for the purpose of storing the data in a file or database. Serialization, when combined with a socket connection can also be used to transmit the state of objects from one place to another. What's more, objects endowed with serialization capability can read in their state from a serial data stream, too. And best of all, it doesn't take any programming to get the benefits of serialization.

Moreover, Java is a threaded language. Threads relieve programmers from tricky input/output situations where multiple socket connections are involved, like when reading and writing need to take place simultaneously.

### 3.4.2 Summary

Distributed object computing tools like Voyager represent a new breed of powerful tools to build distributed applications. Voyager offers a complete set of classes, not directly available in other distributed architectures, that can be seamlessly integrated into programs. All of these features taken together pave the way for a new technique to develop distributed applications. For instance, Voyager 's messaging features if combined with the feature to migrate objects, from server to server, through their proxies and the possibility to access them remotely by other objects in an RPC-like fashion, represent, in our opinion, one of the most innovative and easy to use set of API currently available.

### 3.5 Voyager versus Java IDL

Developing CORBA applications using Voyager is considerably easier than with traditional CORBA implementations such as Java IDL for the following reasons, as shown in [Gla99/2]:

- **Voyager is Non-Intrusive**, while Java IDL requires to modify the original Java class in order to be accessed remotely. Voyager allows any class to be remote-enabled without modification, even third party classes without source.

- **Voyager does not need to create Stubs or Helper Classes**, while Java IDL requires the programmer to manually run the utility IDLTOJAVA to produce the various stub and skeleton classes that are needed by their runtime environment. This tedious step must often be repeated in order to keep these auxiliary files up-to-date. Voyager generates all the distributed glue at runtime so no utilities or helper files are needed.

- **Voyager uses an ultra-light client.** Indeed, while Applets that wish to communicate remotely must include a client-side ORB, traditional CORBA implementations are too big (700K+) to download in an applet. The Voyager client is tiny (15K) and includes a complete client-side ORB that can connect and talk to any combination of CORBA, RMI, and DCOM remote objects. Applets can use the ultra-light client with almost no increase in download time.
• Voyager has a simplified Access to Naming Service. Indeed, the Voyager universal namespace API allows access to a CORBA object in any CORBA naming service with just one line of code. Traditional CORBA implementations require several lines of code to achieve the same goal.

• Voyager exposes a simplified API for Dynamic Invocation. Indeed, the Voyager universal messaging layer allows dynamic invocation of a CORBA operation with just one line of code. Traditional CORBA implementations require many lines of code to achieve the same goal.

• Finally, the Voyager universal gateway allows a CORBA object to be exported to other non-CORBA environments and the protocol conversions are taken care of automatically.

3.6 Voyager versus Java RMI

Developing distributed applications using Voyager is considerably easier than with RMI. Here is a partial list of reasons excerpted from [Gla99/2]:

• Voyager is non-intrusive, while RMI requires to modify the original Java class in order to be accessed remotely. Voyager allows any class to be remote-enabled without modification, even third party classes without source code.

• Voyager does not require Stubs or Helper Classes, while RMI requires a programmer to manually run a utility, the rmic compiler, to produce the various stub and skeleton classes that are needed by their runtime environment. This tedious step must often be repeated in order to keep these auxiliary files up-to-date. Voyager generates all the distributed glue at runtime so no utilities or helper files are needed.

• Voyager supports checked and unchecked exceptions, while RMI only supports checked exceptions. This requires that a programmer must modify all methods in an interface to explicitly throw RemoteException. This inflexibility is particularly troublesome if we wish to remote-enable a third party class that cannot be modified. Voyager supports both checked and unchecked exceptions, allowing developers to choose on a per-method basis how they wish exceptions to be handled. If a method does not explicitly throw RemoteException, a Voyager-related exception is implicitly thrown as a RuntimeException.

• Voyager has a built-in HTTP server that allows remote classes to be loaded dynamically. Instead, using RMI, a separate HTTP server program must be executed. Voyager has built-in native support for HTTP, removing the need for this separate program.

• Voyager includes support for scalable multicast and publish-subscribe. You can publish a Java event on a specified topic to a distributed group of subscribers. Server-side filtering and wildcard matching of topics is supported. The RMI documentation mentions multicast briefly but RMI does not include an implementation of multicast or publish-subscribe.
Voyager is widely regarded as the most sophisticated platform for mobility and mobile agents. Voyager allows to move objects at runtime and create agents that continue to execute as they move. Messages sent to an object’s old location are automatically rerouted to the object’s new location and the client-side proxy is automatically updated to prevent subsequent unnecessary routing. RMI has no support for object mobility or mobile agents although it allows to pass behavior objects.

Voyager delivers a non-intrusive activation framework, while the RMI activation framework requires that the class of the activateable object is modified. The Voyager activation framework allows instances of any class to be activateable without modification.

![Universal Communication Architecture](image)

Figure 3.5: Universal Communication Architecture

Finally, although it does not matter in this context, Voyager includes full native support for CORBA and IDL, whereas the RMI-IIOP only includes support for a limited subset. Furthermore, Voyager includes support for Microsoft COM[GR197], whereas RMI does not.

Though, in our opinion, Voyager simplifies enormously the development effort, it still has the same disadvantage of the other architectures. Also Voyager requires to learn new APIs for the development of a distributed application like Java RMI and Java IDL and, unfortunately, it does not support class inheritance between remote classes.
3.7 Conclusions

During the first stages of our project we had chosen Java RMI to develop our prototype. We discovered Objectspace Voyager by chance when we were addressing some difficulties that would have required a greater coding effort with the RMI package than, as we discovered later, with Voyager itself. Nevertheless, we continued to use Java RMI while we started testing Voyager. After a while, we discovered all the features described above and almost immediately we changed our mind on the ORB to use.

The Voyager messaging features, its capability to create remote instances and its simple support for mobility have made the difference for us. For all these reasons we have chosen ObjectSpace Voyager as the ORB for the proposed distributed hierarchy framework. Our choice was right since the development time has been enormously shortened and simplified.
Chapter 4

Inheritance in Distributed Architectures

To our knowledge only two previous attempts were made to study inheritance in distributed object oriented systems. The first attempt dates back to 1987 and it would have not been classified as an approach to introduce class inheritance in the distributed context if the conclusions drawn by the author had not been so misleading during these years. This proposal introduces an extended version of Smalltalk named Distributed Smalltalk aimed at studying how to adapt all the facilities of an object oriented language in a distributed context. Among them, according to the author, is inheritance. In reality, the author addressed the issue of moving an object among several hosts running a Distributed Smalltalk system.

Since instances and classes are distinct entities in object-oriented languages, an object could be thought as if it were divided into two components: its state and its behavior. If an object is moved, this division raises several problems not solved properly in Distributed Smalltalk. In fact, the conclusion drawn was that inheritance and reactivity do not scale well in a distributed system. We think that this result was misunderstood by the scientific community since it does not imply that class inheritance is impossible in a distributed context. However, we will discuss in details the solutions proposed to solve the object mobility issue and in which way class inheritance effects them. Finally, we will present some hints to possible solutions, according to the current features available in the Java architecture, for some of the issues that Distributed Smalltalk left unsolved.

The second successful attempt is from the mid 90’s and is based on a commercial CORBA implementation extended in order to create a framework, called Global Object Space, to support the inheritance mechanism. In this approach an object is divided vertically into state and behavior. We will present thoroughly this approach underlying its advantages and disadvantages since it is the only solution available up to now.
4.1 Distributed Smalltalk

This work aims to implement a distributed version of Smalltalk, named Distributed Smalltalk, in order to improve this language with a group of new primitives. These primitives would allow object interaction in a distributed context and a better "understanding of the role of inheritance and reactivity in distributed systems by exploring how object behavior and state are shared among distributed objects".

For reactivity, Bennett means (see [BEN87]) "the degree with which objects in the system itself can be easily presented for inspection or modification". To clarify the concept let us give an example. According to this definition, Smalltalk is a complete reactive system, while Java could be classified as a sort of reactive system since it can inspect its classes, using the reflection API, but it could only modify them before loading.

Although the author added to Smalltalk some features such as remote method invocations, distributed garbage collector and so on, now commonplace in languages like Java, we will consider only its contribution to the problem of class inheritance in a distributed context. In a scenario where several Smalltalk systems interact with each other, the author tries to figure out how objects can acquire and share behavior when an object is moved from the host where it was created to another host. Obviously, the moved object should continue to work properly.

Since an object is moved from its class hierarchy to another class hierarchy, it could happen that its creating class is not available in the target hierarchy or there is a homonymous class that is completely different. To find a way to cope with these two problems, Bennett proposes the following solutions:

1. Disallow remote classes, requiring that classes and instances reside on the same node. This approach allows only to move objects to hosts having in their hierarchy a class compatible with the class used to instantiate the object.
2. Making classes immutable and allow only the creation of new subclasses.
3. Using a master copy of the class hierarchy, and caching copies of classes on nodes where they are needed.
4. Provide multiple class copies on every node, managing them as a replicated and distributed database.
5. Requiring that an object's class pointer always points back to the class from which it was created.
6. Provide a means for class pointers of objects that are being moved to point back to the creating class definition when appropriate, but allow class pointers to "splice-in" to the class hierarchy on the destination machine when possible.
7. Require that users mutually agree about compatibility between classes before allowing an instance of a class to be moved and spliced into a destination class hierarchy.

How Bennett decided to cope with the inheritance issue, with respect to object mobility, in its Distributed Smalltalk will be the topic of the remainder of this section. The
proposals listed above will be evaluated according to the author’s criteria of semantic clarity, ease of implementation, reactivity, mobility, performance, and reliability. Notice that Bennett did not try to create a mechanism to implement class inheritance among several Smalltalk enviroments.

### 4.1.1 Evaluating the solutions

The third and fourth choices, based on variants of a replication scheme, were rejected for the burden required to manage them. The seventh solution, moving an instance of a class to splice-in to the destination class hierarchy, was dropped for the unrealistic level of cooperation required between users to ensure the type compatibility with all the versions of the class. Indeed, this solution requires the availability of either the same class or a distinct yet compatible version of it inside another hierarchy. Only in this way a moved instance can become an instance of a local class as long as this class is compatible to the creating class.

The second proposal on disallowing a class modification was abandoned since it would have introduced an unnatural way of using immutable classes for a Smalltalk user. Moreover, Bennett rejects even the fifth solution for performance reasons raised from the point-back mechanism. Nevertheless, in the beginning of Bennett’s research activity, he chose a variant of the point-back solution described in the sixth approach: when an object is moved, either it points back to its remote classes or it joins the local hierarchy when possible.

The author states that the approach imagined with the sixth choice would have provided a single logical address space where all the objects were part of a common and distributed class hierarchy. Unfortunately, an unsolved issue arises out of the splicing-in policy for the classes of the objects moved. Bennett did not find out an appropriate criteria to "splice-in" the object to the classes of the remote class hierarchy and when to "point back" to the classes available in the source hierarchy.

This inability is due to the nature of the inheritance and subclassing mechanism. It is difficult to decide when two classes, not located in the same host, are compatible. Thereafter, the first simpler solution was preferred in [BEN87/1].

The first solution is the simplest since classes and objects have to be co-resident and there is only a weak support for the object mobility. Therefore, this solution implies that the scope of class inheritance is local to each host. Object mobility is allowed only when there are compatible classes available on different hosts.

In our opinion this is not a solution at all, since the problem is not solved but only disregarded. Indeed, Bennett postpones only the issue since later he proposes in [BEN90] and [BEN87/2] a combination of the sixth and second proposal for further research. In this case, all the classes are immutable and their instances point back or possibly splice-in to the local hierarchy. The usage of immutable classes allows to avoid the updating issue when a class gets modified and there are some copies in other hierarchies. In [BEN87/2] the author introduces a sketchy description of this alternative solution. Nevertheless, Bennett has not published other papers on this topic since 1990. We believe that he has moved his research interests on something different.

Maybe this is due to the conclusions he draws on the use of inheritance in distributed systems. Indeed, at the end of his thesis, Bennett states that the mechanism
of inheritance and reactivity do not appear to be well suited for building distributed systems with respect to object mobility. Note that from this sentence a lot of people have drawn the conclusion that class inheritance is not feasible when the classes are distributed among several hosts.

In his opinion, this negative result is due to the commitment not to change the character of the Smalltalk programming environment and to the awkward separation implied by the distribution of the object state (instances) and behavior (classes) on different hosts. Therefore, he claims that a prototype object model appears to be a better choice than an object model based on subclassing. The rationale of this position is in the tight coupling of state and behavior in a single entity called prototype. Indeed prototypes avoid the problem of having two different parts, class and state, to get an object. For instance, this is the approach followed by Obliq [Car95], a prototype-based object oriented language with support for mobility. In this way, a moving object contains everything it needs in order to continue to work properly.

Nevertheless, we do not agree with this stance. In our opinion this work has given a negative result because in the late 80’s the technology was less mature than nowadays. In fact, some of his proposals, although consider to move a class at runtime to a different host together with its instances, do not solve the problem of type compatibility when a class having the same name is still locally available. The general case of ensuring the compatibility of two classes is a well-known undecidable problem since it is equivalent to deciding whether or not two programs produce the same output for all inputs. A practical alternative would be the possibility to determine with a high degree of confidence that two objects are compatible.

Nevertheless, we believe that a simpler solution to this issue relies on using distinct namespaces instead of finding an algorithm that can assess with some confidence whether or not two classes are compatible. In this way even different versions of the same class can coexist on a single host. Probably in the Smalltalk version used by Bennet this feature was not available.

### 4.1.2 Possible solutions in Java

Bennett rejects some of his proposals since he was not able to develop them in Distributed Smalltalk; we will suggest how some issues could be solved in Java.

A solution proposed in Distributed Smalltalk is based on the idea to decouple an instance from its class so that an object points back to its creating class wherever it is. This approach implies that an instance, created on a host distinct from the host where its class is located, refers every time to its remote class when it has to execute a method. This decoupling in Java could be avoided since we can move and load a class everywhere we need to create its instances. Using Java dynamic nature and its communication features we can transmit both classes -the behavior- and object instances -the state- to remote hosts. To explain how it might work, we can give a more detailed description. In Java we can deliver instances and classes together. A class can be loaded from a remote JVM using a so-called network class loader, while its instances can be serialized from a program running on a given JVM and transmitted to a different one. When the target JVM receives this stream, it can restore all the instances from their serialized condition to their original state. The class loaded from the remote JVM is used to create these instances. What’s more,
4.2. GLOBAL OBJECT SPACE

loading classes with different class loaders puts them in distinct namespaces allowing
us to avoid compatibility problems.

Another solution that combines with the point-back mechanism suggests to use
local cached copies of a remote class for each hierarchy in order to improve the speed
of method lookup. Unfortunately, the cache usage requires to manage properly the
updating of all the copies when the original class is changed.

The problem of coping with the updating issue, raised from classes whose instances
are disseminated among several hosts, has not found a solution yet. At the moment,
even in Java this issue has not been solved completely. Indeed, this would allow to
change a loaded class at run-time without effecting the created instances. Unfortunately,
Java does not allow to replace a class with an updated version if we try to
reload the updated one with the same class loader. Nevertheless, somehow in [LIB98]
this issue has been addressed and partially solved. This paper, in one of its sections,
shows how to substitute a class already loaded provided it is not a system class and
it is not used directly in the code. Briefly, the solution conceived proposes to load
every new version of the class using a new class loader so that the class is added to a
different namespace. Afterwards, the loaded class is instantiated and stored inside an
instance variable of a wrapper object. This new instance replaces the old one built
using the old class. To allow this kind of substitution, the instance variable can be
defined either as a generic type Object or as an interface that the class implements.
In turn, this wrapper object receives these method invocations and uses the reflection
API to redirect them to the corresponding method in the instance built from the class
just loaded. Although this last solution seems to solve one of the problems raised by
Bennett, it has not been properly investigated yet.

4.1.3 Summary

The approach followed by our system addresses the actual lack of class inheritance,
while Bennett addressed object mobility with respect to class inheritance. Unfortunately,
his failure has influenced negatively all further research activity. Nevertheless,
his results concern exclusively object mobility in Smalltalk and not class inheritance.
Moreover, as mentioned before, in Java, almost all the issues he left unsolved would
be solved using dynamic class loading and different namespaces. Moreover, the auto-
matic reloading of updated classes even when there are some active instances, could
be appropriate to improve our system.

4.2 Global Object Space

The work of H. Günder and K. Geihs addresses the problem of class inheritance in
a distributed system. Their proposal is based on a mechanism that implements a
sort of remote inheritance that supports the main properties of inheritance such as
systematic code reuse, extensibility and polymorphic substitution. The ORB used to
develop this system is DSOM, a specific CORBA implementation from IBM.
4.2.1 Object Model

The authors propose a conceptual decoupling of an object’s code (engine) from its data, identity and interface (chassis). It is as if we had the object split into two parts as shown in Fig. 4.1. A chassis represents the offered services and contains the operational interface, the state representation and all the additional information required by the engines. An instance of a chassis is created when an object is instantiated and works as a gateway between the client proxy and the engine. In turn, the object engine implements the object behavior, together with the runtime support for attaching, detaching to a chassis, reading, writing an object state and carrying out methods invocations. According to this architecture the engine can attach to the chassis, read the state, execute method calls, and write back the state changes to the chassis.

![Figure 4.1: Object decoupling](image)

This attachment operation can be achieved only if there is type compatibility between the two parts, like in a subtype relationship. The type compatibility is checked during the engine attachment operation to the chassis; this compatibility test is divided into two conformance tests, the behavioral one and the structural one. Let us give a short description of both:

The engine presents a syntactical description of the implemented services that is used to test the behavioral conformance. In this way, it is guaranteed that the engine has the methods that the chassis declares to the outside world.

The structural conformance controls the differences between the chassis’ attributes and the engine’s attributes. They must either have the same names or be coupled using a mapping function. Moreover, the conditions where some attributes are not available in both the parts are properly managed in order to avoid structural problems. For further details the reader can refer to [GRG95].

The structural conformance is accomplished comparing the engine state information with the actual state information of the attaching object.
4.2.2 Application development

A programmer that wishes to use the remote inheritance mechanism has to write an IDL description of the services that he wants to offer. Inside this description he should specify the interfaces that he wants to inherit. Obviously, the inherited interfaces must have their corresponding engines implemented somewhere in the distributed system. After having specified an interface, it is up to the programmer to add new methods and attributes to the inherited ones and to specify the additional state description. An interface compiler reads the description and generates a client proxy, an object chassis and an object engine. The proxy is generated for the clients to grant the access to an instantiated object.

It is important to emphasize that from the computational point of view the object is seen as before.

![Object computational viewpoint](image)

Figure 4.2: Object computational viewpoint

4.2.3 The GOS scenario

In spite of this sketchy survey of the work of H. Günder and K. Geihs, it is easy to understand that this approach of splicing the objects into two snippets allows their distribution. Indeed chassis and engine can reside on different nodes, giving the object a distributed character. Moreover, through the structural conformance test previously mentioned, the polymorphic engine substitutions is supported when an engine implements a subtype of the object chassis.

The whole scenario where the authors imagine using their model is an infrastructure named Global Object Space (GOS) that supports the cooperation between users and service providers based on the object model mentioned so far.

In this environment the clients request services through a trading component. The trading component satisfies the requests returning to these clients the references to objects that match the needed service. The clients use these references to access the objects to accomplish their needs. Nonetheless, the object can execute the code only by attaching to an engine available in a remote site. This operation can be done through an object repository that records the available objects, their type and location. Each engine queries this repository, before attaching to a chassis, for testing the type compatibility.
4.2.4 Remote Inheritance

The remote inheritance proposal, see [GRG96] for further details, works in this way:

- each engine implements only a subset of the methods defined by a single object,
- a chassis is enabled to attach to several engines at the same time.

Hence, an engine can attach to an object chassis that implements a subtype of its class. That is, these engines are the behavioral part of the ancestors of the present object. In this way the system mimics inheritance, as it is known in traditional object oriented programming. This is achieved by adding operations and subsequently engines to an object as shown in Fig. 4.4. The only restriction is in the form of inheritance

![Figure 4.3: Global Object Space](image)

![Figure 4.4: Runtime Constellation](image)

simulated since it is restricted to the monotonous one. The monotonous inheritance allows neither method overloading and overriding nor attributes redefinition.

Finally, in [GRG95] the authors claim that the notion of engine attaching could be a valid model in an environment that uses agents to accomplish distributed applications
4.2. GLOBAL OBJECT SPACE

instead of immobile engines. Nevertheless, in our knowledge, they did not address this
solution based on mobile engines. We think that this is due to the DSOM architecture
used to develop their prototype.

4.2.5 Summary

The main advantage of this work relies on the partition of interface and state from the
behavior. On the other hand, in our opinion, sending back and forth the state from
the chassis to the engines in order to invoke a method could create a communication
overhead. Indeed, every time a method has to be invoked, the whole state of the
chassis should be sent to the corresponding engine. What’s more, the whole state
–that might have been modified as a result of the method execution– must be sent
back in order to update the chassis.

Other disadvantages of this approach are: the immobile nature of the engines,
justified by the choice of using CORBA as middleware, the non availability in English
of a more detailed technical report, and the non availability of the prototype itself in
the public domain. This last consideration can be mitigated by the need to purchase
IBM DSOM in order to test the prototype. Unfortunately, DSOM is no longer sup-
ported as a product from IBM. Therefore, the GOS approach does not run anymore
on current systems.

Nevertheless, these considerations are actually drawn only from the papers pre-

tented by the authors during their research activity and a short conversation with

prof. Geihs during a conference break. We have not had the chance to use their

prototype.
Chapter 5

Remote Class Inheritance among JVMs

Let us suppose a programmer wants to develop a distributed application using Java and this application consists of several communicating objects running on different JVMs. After the design stage, he has to start developing the application with a distributed architecture. He can choose to use Java IDL, Java RMI, and so on, according to his needs and, above all, his experience in one of these architectures. Whatever the choice, when he develops his application he writes the code for each local class in a natural way. On the contrary, he is forced to use, at least, a specialized package to implement all the remote classes. Notwithstanding, he cannot even extend a class when the descendant is not local to the ancestor. We claim that this is not a full-fledged object oriented approach since class inheritance cannot cross the boundaries of a single JVM.

We propose a compiler named DIC, distributed hierarchy compiler, that allows to develop a distributed application similar to a stand-alone one. By means of the DIC compiler, a programmer has the freedom to develop his application as if it were made only of local classes. He can use both delegation and class inheritance without any restriction. When needed, he can remote-enable, we will say also remotize, a given class without modifying it. If the class has already been used in a legacy application, only a few changes are required in all its clients and its subclasses.

On the other hand, when a programmer creates an application from scratch, he has only to take the usual precaution of designing properly the remote classes and their interaction in order to consider both the communication overhead introduced by the distribution of classes among several Java Virtual Machines and the issues raised by network errors and host failures. Nevertheless, he does not have to learn new APIs to create it. For instance, he does not need to consider extending particular classes such as UnicastRemoteObject, using a registry and so on, as in the Java RMI package. Instead, he can write his code as if he developed a stand-alone application. Afterwards, it is up to the compiler to build a framework around each class designed to accept remote method invocations. The underlying framework will make it possible for a class to offer its services to remote clients as if this class were developed to do so. Even the exceptions are managed properly in order to have the same behavior.
Moreover, if we have a class that inherits from an ancestor class that needs to be installed on a different host, the proposed tool can automatically remotize this ancestor. Only a simple reorganization in the source code of the descendant is required to continue to work properly.

In addition, this tool allows to choose whether an instance of the original class has to compute on the remote server, where it was created, or locally at the caller. This alternative is useful when the class to remotize implements some specific services such as a GUI object that should run in the same JVM of the caller. In this case, it does not make sense to invoke methods whose effect is shown far away from the caller; a widget must be displayed locally unless specifically required.

We believe that these features are a breakthrough with respect to the other ORBs available at the time of this writing. Indeed, we have an effective way to build distributed applications with a full-fledged object oriented approach. The DIC compiler not only simplifies the remote method invocation usage but also paves the way to class inheritance among distributed classes. As a matter of fact, we have found a new form of inheritance, named remote class inheritance, that creates distributed inheritance hierarchies among classes located in different JVMs.

The advantage of this approach is twofold. First of all, it allows to build distributed applications using class inheritance. At last, even programmers that have no specific knowledge of any distributed architecture such as Java RMI, CORBA and so on, can develop a distributed application and convert a common stand-alone application into a distributed one.

This chapter is divided into three sections. The first one will show through a simple example how the DIC compiler can be used by a programmer to remotize a class. The second section will describe how the DIC compiler creates the framework to remotize a class and how this framework works. Finally, we will see a more meaningful usage of DIC in order to remotize a predefined Java class: ArrayList and to create a remotized version of a linked list.

### 5.1 The “Hello World!” example

The best way to introduce a new tool is usually by a simple example. We propose a variant of the classical “Hello World!” application where there is a class that prints a string, Hello World!, on behalf of another class.

Figure 5.1: Application Simple

<table>
<thead>
<tr>
<th>Simple (from default)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Simple</td>
</tr>
<tr>
<td>+main</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Message (from default)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+say</td>
</tr>
</tbody>
</table>

Suppose we have a class Message, see Fig 5.1, that defines only a public method, called say, that prints on the standard output a string msg received as parameter such as:
public class Message {
    public String say(String msg)
    {
        System.out.println(msg);
        return "I have just said: "+msg;
    }
} // Message

Now, suppose we have a console class Simple, see Fig 5.1, that creates in its main method an instance of Message and invokes its method say to print “Hello World!”.

public class Simple {
    public static void main(String[] args) {
        Message message;
        String m;

        message=new Message();
        m=message.say("Hello World!");
        System.out.println(m);
    }
} // Simple

After compiling the two classes, the working directory contains the source code files and their bytecode representation. Below is shown the content of the directory d:\projects\java\examples used during the development of this example:

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Size</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>15/10/99</td>
<td>9.19</td>
<td>717</td>
<td>Message.class</td>
</tr>
<tr>
<td>15/10/99</td>
<td>9.19</td>
<td>294</td>
<td>Message.java</td>
</tr>
<tr>
<td>15/10/99</td>
<td>9.19</td>
<td>686</td>
<td>Simple.class</td>
</tr>
<tr>
<td>15/10/99</td>
<td>9.18</td>
<td>345</td>
<td>Simple.java</td>
</tr>
</tbody>
</table>

Now we can run the application Simple that creates an instance of Message and calls the method say to print “Hello World!”. In turn, the return value of say is printed.

[d:\projects\java\examples]java Simple
Hello World!
I have just said: Hello World!

What we have seen is a traditional stand-alone application. Suppose we want to “remotize” the class Message without modifying its source code. That is, we want to create an instance of Message on a different JVM, remote or local it does not matter, and enable its method say to be invoked from the class Simple running on a local JVM. In this way, as long as the class Message allows to invoke its method from a remote caller, the application Simple roughly works as described previously for the stand-alone application. There is only one difference from the user’s standpoint, the message “Hello World!” is now printed on the standard output of the JVM running
the object of type `Message`, though the return message is still displayed locally to `Simple`.

To create a distributed version of this application a programmer can either choose one of the architectures described in the previous chapter or our DIC compiler.

Let us see what happens when a programmer wants to remotize the class `Message` with our tool.

![Diagram of remotization](image)

**Figure 5.2: Remotized Message**

When DIC receives input `Message` and an Internet address such as `localhost:9000`, it creates two new classes - `MessageReceiverProxy` and `MessageForwarderProxy` as shown in Fig.5.2- and compiles them. The Internet address is the location of the JVM hosting `Message`.

```
[d:\projects\java\examples] dic Message localhost:9000
```

```
== Creating 'MessageReceiverProxy.java' ==

== Executing 'javac MessageReceiverProxy.java'. ==

== Creating 'MessageForwarderProxy.java' ==

== Executing 'javac MessageForwarderProxy.java'. ==
```

After the completion of this step, the programmer must slightly modify `Simple`’s source code in this way:

```java
import di.*;             // added
public class Simple {

    public static void main(String[] args) {
        MessageForwarderProxy message; // changed
        String m;

        DlEngine ris=DlEngine.make(); // added
        message=new MessageForwarderProxy(); // changed
```
5.1. THE "HELLO WORLD!" EXAMPLE

```java
m=message.say("Hello World!");
System.out.println(m);
DIEngine.destroy(); // added

} } // Simple
```

We have highlighted the changes made. Shortly, every occurrence of Message has been replaced with the new class MessageForwarderProxy and a runtime support, DIEngine, has been added.

Now, to complete the remotizing activity, the programmer has to create a simple class that runs the DIEngine support for Message and MessageReceiverProxy.

```java
import di.*;
public class MainServer {
    public static void main(String[] args) {
        DIEngine.make("//localhost:9000");
        System.out.println("MainServer built");
    }
} } // MainServer
```

The MainServer program will run on the JVM at localhost:9000 where it will listen to incoming calls from other JVMs.

Once the class Simple has been modified and the source file of the MainServer class has been created, we are ready to test the client server version of our little application. Since we are in a single host scenario, we will create two directories where we will move the client part and the server part of our application. There will be a different JVM running correspondingly to each directory and, obviously, the CLASSPATH environment variable contains none of them.

The first directory, d:\projects\java\examples\client\, contains Simple and SimpleForwarderProxy.

| Date | Time | Size | File
|------|------|------|------|
| 5/10/99 | 9.29 | 3,036 | MessageForwarderProxy.class
| 5/10/99 | 10.23 | 833 | Simple.class

While the content of the second directory d:\projects\java\examples\server\ is:

| Date | Time | Size | File
|------|------|------|------|
| 5/10/99 | 9.54 | 640 | MainServer.class
| 5/10/99 | 9.19 | 717 | Message.class
| 5/10/99 | 9.29 | 2,956 | MessageReceiverProxy.class

That’s all, now the programmer can run the two parts of the application. First of all, he has to run the MainServer application, then he can run the modified version of Simple. Fig.5.3 shows the remotized Message at run-time.
The message “Hello World!” is shown on the server side while the return value is displayed on the client side. As you can see we have created a distributed application from a traditional one with little effort and, above all, without knowing anything on the topic of distributed architectures.

Now suppose a programmer chooses a traditional approach to develop the distributed version of our example. Whatever the choice—among Java IDL, Java RMI and Voyager—some changes would be required to the source code of only one class or both classes. For instance, Java IDL and Java RMI would require for the programmer to conceive a wrapper class that can receive and redirect all the invocations of *say* to the original method. Provided he has the required technical knowledge, the programmer has to write further code to create the wrapper and use it in *Simple*. This activity becomes even more cumbersome with Java IDL. Indeed, Java IDL would require to write an IDL file, compile it and so on, before creating this wrapper class. The least invasive is the Voyager approach since it can remote-enable a class without changing its code. Yet, some changes to the caller source code are still required in order to create a remote instance of Message and invoke *say*. Hence, even with Voyager, the programmer must learn new APIs. We believe that our *Dic* compiler achieves a step forward to simplify the development, even with respect to Voyager remote-enabling feature.

Nevertheless, someone might consider the stand-alone version of our *Simple* application and its remotized variant not to be equivalent from a user’s standpoint. In fact, the message “Hello World!” is printed far away from the caller, while in the original application they are printed together.

This evaluation may or may not be reasonable, depending on the service a remotized class offers. Sometimes it is better that a computation is done remotely. For instance, an application has to query a database only available on the same host. Sometimes it is better that the remote object runs locally to the caller. For instance, a GUI component should be where it is called. Thereafter, if we wanted to create a distributed version according to this second case, this would require that “Hello
5.1. **The "HELLO WORLD!" Example**

World! is printed where Simple runs.

Once again we can use our DIC compiler to solve this issue. This tool, when receiving a second Internet address in input, creates a slight different version of the two Proxies. This time, after an instance of Message is created on the JVM running at localhost:9000, it will be moved to the JVM running at localhost:7000.

Nevertheless, we have to make a change to Message since in Java an object can be transmitted only if it implements the Serializable interface. Thereafter, the class Message becomes:

```java
public class Message implements java.io.Serializable { // changed
    public String say(String msg) {
        System.out.println(msg);
        return "I have just said: " + msg;
    }
} // Message
```

Everything else works as before. We run DIC on Message passing the remote address and the local address.

```
[d:\...\examples]dic Message localhost:9000 localhost:7000
=================================================================
Creating 'MessageReceiverProxy.java'
=================================================================
Executing 'javac MessageReceiverProxy.java'.
=================================================================
Creating 'MessageForwarderProxy.java'
=================================================================
Executing 'javac MessageForwarderProxy.java'.
```

We move the files to two different directories, as we have done previously, named clientmove and servermove respectively. Now we can run our example.

<table>
<thead>
<tr>
<th>Server Side</th>
<th>Client Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>[d:...\servermove]java MainServer MainServer built</td>
<td>[d:...\clientmove]java Simple Hello World! I have just said: Hello World!</td>
</tr>
</tbody>
</table>

Notice that if the class Message had not implemented the Serializable interface, the attempt to move its instance would have thrown the following exception. For further details we refer you to the technical description in this chapter.

```
[d:\projects\java\examples\clientmove]java Simple
Exception in thread "main" java.io.NotSerializableException: MessageReceiverProxy
```
Before closing this section, the reader may wonder how a programmer can extend a remote class. The answer is straightforward: the descendant must inherit from, we say extend in Java parlance, the generated ForwarderProxy. To explain this last concept, let us see what happens if a programmer wants to extend the class Message to create a specialized version called PoliteMessage.

![PoliteMessage-Message hierarchy](image)

Figure 5.4: PoliteMessage-Message hierarchy

Beforehand we describe the stand-alone application version of the class PoliteMessage, see Fig. 5.4, subsequently we will show the distributed version.

The class MessagePolite overrides the method say to give a flavor of politeness to the string to be displayed. Moreover, the superclass’ method say is still used since it is invoked inside the overridden version.

```java
public class PoliteMessage extends Message {
    public String say(String msg) {
        System.out.print("I am delighted to say ");
        String s=super.say(msg);
        return "Sir, " + s;
    }
}
```

To use this class we need a console class such as SimplePolite.

```java
public class SimplePolite {
    public static void main(String[] args) {
        PoliteMessage message;
        String m;

        message=new PoliteMessage();
        m=message.say("Hello World!");
    }
}
```
5.1. THE "HELLO WORLD!" EXAMPLE

System.out.println(m);

} } // SimplePolite

When we run SimplePolite we obtain a courteous version of the "Hello World!" example.

[d:/Projects/Java/Reflection/examples/] java SimplePolite
I am delighted to say Hello World!
Sir, I have just said: Hello World!

Now suppose that we want to build a distributed variant of this application. In addition, we want to have the class Message far away from the class PoliteMessage. Thereafter, PoliteMessage should be modified in order to extend the local representative of Message i.e., MessageForwarderProxy, as shown in Fig. 5.5.

Figure 5.5: PoliteMessage - Message remote hierarchy

This is how the remote class inheritance works. Indeed, the class PoliteMessage inherits remotely from the class Message. In Java parlance, PoliteMessage extends remotely Message even though it actually extends MessageForwarderProxy.

```java
public class PoliteMessage extends MessageForwarderProxy {
    // changed

    public String say(String msg)
    {
        System.out.println("I am delighted to say ");
        String s = super.say(msg);
    }
```
return "Sir, "+s;
} } // PoliteMessage

At last, SimplePolite only has to use the run-time support provided by DIEngine.

```java
import di.*; // added
public class SimplePolite {

    public static void main(String[] args) {
        PoliteMessage message;
        String m;

        DIEngine ris=DIEngine.make(); // added
        message=new PoliteMessage();
        m=message.say("Hello World!");
        System.out.println(m);

        DIEngine.destroy(); // added
    } } // SimplePolite
```

Notice that we do not need to use the DIC compiler for this example because we use the previously generated proxies. Even the directories are the same. In this case we have only to put the modified PoliteMessage and SimplePolite in the directory client.

When we run the modified SimplePolite using the proxies generated for the client-server version the outcome is a bit funny, as shown below. The method say of the superclass continues to run on the server side, while its subclass runs the overridden method say on the client side. Thus, the courteous variant of the message “Hello World!” is displayed split into two parts.

<table>
<thead>
<tr>
<th>Server Side</th>
<th>Client Side</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Server Side Image" /></td>
<td><img src="image2.png" alt="Client Side Image" /></td>
</tr>
</tbody>
</table>

Instead, if we use the proxies that move the instance of Message locally to the instance of PoliteMessage, we obtain what we wanted.

<table>
<thead>
<tr>
<th>Server Side</th>
<th>Client Side</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image3.png" alt="Server Side Image" /></td>
<td><img src="image4.png" alt="Client Side Image" /></td>
</tr>
</tbody>
</table>
5.2. The Distributed Hierarchy Architecture

This example shows how easy it is, from the programmer’s point of view, to make a descendant of a remote class. He has to slightly modify the descendant class in order to extend the forwarder proxy instead of the actual superclass. In this way, a class hierarchy, like the PoliteMessage-Message hierarchy, seems to be distributed between two or more JVMs.

Though we have shown how to turn a stand-alone program into a distributed application, nothing forbids the design of an application from scratch as a distributed one. The classes designed to be remote, but implemented as they had to offer their services locally, can be remote-enabled using the DTC compiler. Few code changes made to the clients’ code, similar to the changes described before in the example of Simple, and the application is ready.

5.2 The distributed hierarchy architecture

From a computational point of view, a remotized object, running on a remote JVM, is seen as if it were local. In reality, it is the communication framework i.e., the forwarder and the receiver, that hide its whereabouts. In fact, the purpose of this framework is to offer a transparent view of an object, running on a JVM at a specified address, to a client running on a different JVM. This architectures relies on the Voyager ORB for all the communication issues such as remote object instantiation, remote method invocation and object mobility.

Let us explain more in detail the structure of this framework, shown in Figure 5.6, and how it works.

![Figure 5.6: Remotized object](image)

A forwarder class declares a class variable url, which contains the address of the remote object, and an instance variable pz of type Proxy. The instance variable pz contains an object, having as class a subclass of Proxy, that represents locally the receiver object created on a JVM at the address url. As far as the receiver is concerned, it declares a variable named superClass that contains an instance of the actual class created at the receiver instantiation time. These two classes work in this way: every time an instance of a forwarder is created, an instance of a receiver, through the Voyager class Factory class, and an instance of the remotized class are automatically created as well. Subsequently, if a method of a forwarder object is called, the invoked method relays this call, through the Voyager proxy held in pz, to a receiver object. The target receiver object runs an homonymous method that,
in turn, forwards the call to an instance of the original class held in the variable
superClass. At last, the actual object runs the requested method. At the end of the
method execution, the receiver collects the result and sends it back to the forwarder
object. Thus, the result is delivered to the caller that has started the whole process.

According to our approach, a forwarder must offer the same services of the original
class in order to allow a programmer to use it as he does with the original class.
This means that a forwarder must declare all the non-private method signatures of
the original class. In this way, any other class in the application can use directly an
instance of the forwarder or can even extend it. Obviously, for every method declared
in the forwarder there is a corresponding method declared in the receiver. Neverthe-
less, the visibility modifiers in the forwarder are preserved while in the receiver all
the methods created from the original class are public.

The rationale for this change in the visibility modifier is straightforward: receiver
and forwarder are installed on different JVMs. Therefore, an instance of a forwarder
can invoke a method of a remote receiver object only using a ORB, in our case
Objectspace Voyager. Well, an ORB allows to invoke exclusively a public method of
a remote object. Hence, all the methods in the receiver are public so that a forwarder
can invoke them using the Voyager class Sync.

There is another subtlety: the receiver proxy cannot use directly all the methods
of the actual object held in superClass. Only the public methods of an object are
available when a container class tries to invoke them. Alas, this is not what we
want. In fact, a descendant class can use all the public, protected and, if it is in the
same package, so-called friendly methods of its ancestor. Notice that all the private
methods are ignored since they are not visible outside a class and in its descendants.
This means that, in order to mimic the inheritance mechanism, the receiver must
be able to invoke all the methods of its aggregated class without considering their
visibility modifiers. Only the reflection API can overcome this problem since it allows
to invoke a method ignoring its visibility modifier.

Notice that the forwarder changes only slightly if a programmer specifies also
the Internet address of the JVM where he wants to move the receiver together with
the remote object. In this case, the DIC compiler generates a forwarder that moves
the receiver together with the actual object after its instantiation, as shown in the
second part of the class Simple example. Obviously, the actual object and all the
objects referenced must implement java.io.Serializable in order to be moved at run-
time. When the migration is done to JVM of the forwarder, in our opinion the most
interesting case, all the subsequent executions of the actual methods will affect the
local JVM.

5.3 The forwarder and receiver framework

Let us present in detail the structure of a forwarder and a receiver created from a
class developed for this purpose. Let us consider a class Dummy as shown in Fig. 5.7.
Let us suppose that Dummy.class is the class bytecode file produced by a standard
Java compiler from the Dummy.java source code.

In order to remotize Dummy, the Dummy.class bytecode is used to feed the
DIC compiler. The DIC compiler loads this class, creates two new source files called
DummyForwarderProxy.java and DummyReceiverProxy.java and eventually compiles them.

Figure 5.7: The class Dummy

Given the highlighted methods in Dummy, we will show how the Dic compiler will create the corresponding methods in DummyForwarderProxy.java and DummyReceiverProxy.java.

```java
import java.io.*;

public class Dummy implements Serializable {

    static int num=0;
    String name;
    int mynum;

    public Dummy(String name) { // method described
        this.name=name;
        mynum=num++;
    }

    public Dummy() {
        name="defaultName";
        mynum=num++;
    }

    static void mstatic(String s){
        System.out.println("method mstatic called, the parameter is:"+s);
    }

    public String toString() {
        return name:"+instance number "+Integer.toString(mynum);
    }

    public void info(String s) { // method described
```
System.out.println(toString()+"="+s);
}

protected void finalize() throws java.lang.Throwable {
    System.out.println(toString()+" DESTROYED!!");
    num--;
}

private int privateMethod() {
    return 0;
}

public final void noparam() {
    System.out.println("Hello from no param");
}

public int intNet(int i) throws java.io.IOException, java.lang.InterruptedException{
    if (i==0) throw new java.io.IOException();
    if (i==1) throw new java.lang.InterruptedException();
    if (i==2) i/=i; // create a RuntimeException that is forwarded
    return (i*1);
}

public char charm(char c) {
    return c;
}

public boolean booleanm(boolean b) {
    return (!b);
}

byte byteb(byte b) {
    return (byte)(b*1);
}

double doublem(double d) {
    return d*1;
}

float floatm(float f) {
    return f*1;
}

long longm(long l) {
    return l*1;
}

short shortm(short s) {
    return (short) (s*1);
}

int[] arraym(int i1, int i2) {
    int[] ia=new int[i2];
    if(i1>i2)
    ia[0]=i1;
    ia[1]=i2;
    return ia;
}

static public Dummy create(String s) { // method described
    return new Dummy(s);
}

Boolean compareTo(Dummy d) {
    return new Boolean(name.compareTo(d.name)==0));
}

Boolean compareTo(Dummy d1,DummySecond ds,Dummy d2) {
    return new Boolean((compareTo(d1).booleanValue())&&(!compareTo(d2).booleanValue()));
}
Before describing the forwarder and the receiver classes, since both these proxies use some classes in the package `di`, we will first introduce this package.

### 5.3.1 The package `di`

The package `di`, distributed hierarchy, consists of all the classes used at runtime by the forwarders and the receivers created by the DiC compiler. This package contains respectively the classes `Invoker`, `Builder`, `DiEngine` and `ArgumentHolder` as shown in Fig. 5.8.

![Figure 5.8: Package `di`](image)

The reflection package, as we have seen, requires an array of types to obtain a method from a class and their corresponding values to invoke it (see `Invoker.dynaCall()` below). This package contains the class `ArgumentHolder` to manage these arrays efficiently.

Precisely, `ArgumentHolder` manages an array of class types, needed to get a method from a class and an array of values, to be passed to a method during an invocation operation. `ArgumentHolder` allows to add elements, whatever their type, to these arrays in a straightforward way. In fact, this operation is achieved through the overloaded method `setArgument`.

Note that this method can receive in input both a primitive type and a reference type. There is only one difference between the two kinds of parameters: a primitive type must be converted into the corresponding Java wrappers before being added to `args` while the reference type is directly added to this array. Nevertheless, a primitive type preserves its identity since its class type is recorded as primitive in the class types array `cl`.

When needed, the methods `getArgumentClasses()` and `getArguments()` allow to return respectively the array containing the values and the array containing their class type.
public class ArgumentHolder {
    protected Class[]  cl;
    protected Object[] args;
    protected int argc;
    ...
    public Class[] getArgumentClasses() { return cl; }
    public Object[] getArguments() { return args; }
    public int setArgument(boolean b) {
        return this.setArgument(argc, new Boolean(b), Boolean.TYPE);
    }
    public int setArgument(byte b) {
        return this.setArgument(argc, new Byte(b), Byte.TYPE);
    }
    ...
    public int setArgument(Object obj) {
        return this.setArgument(argc, obj, obj.getClass());
    }
    public int setArgument(Object obj, Class c) {
        return this.setArgument(argc, obj, c);
    }
    ...
    public int setArgument(int argnum, Object obj, Class c) {
        if (argnum >= args.length) {
            ...
        }
        args[argnum] = obj;
        cl[argnum] = c;
        return argnum;
    }
} // ArgumentHolder

The class Builder declares a single method dynaBuild. When this method is invoked, it creates an instance of a given class using the constructor whose parameters match the elements of an array of types. The instance of ArgumentHolder received in input returns this array. Note that the instantiation is done ignoring the visibility modifier of the selected constructor.

public class Builder {
    public static Object dynaBuild(Class c, ArgumentHolder a) throws InvocationTargetException {
        try
5.3. THE FORWARDER AND RECEIVER FRAMEWORK

```java
{  
    Constructor cst = c.getDeclaredConstructor  
        ( a.getArgumentClasses() );  
    cst.setAccessible(true);  
    return cst.newInstance(a.getArguments());  
}
catch ( NoSuchMethodException e )  
{
    ....
}
} } // Builder
```

A receiver uses this helper class to create an instance of the actual object.

The class Invoker has a static method `dynaCall`. This method can run a method of an object whatever is the visibility modifier. The actual parameters used for this method invocation are obtained from an array of values passed to `dynaCall` by means of an `ArgumentHolder` object. Below is the code snippet taken from the method `dynaCall`.

```java
public class Invoker {  
    static public Object dynaCall(Class c, Object o, String m,  
        ArgumentHolder a) throws InvocationTargetException  
    {
        try  
        {
            Method meth = Reflector.getAccessibleMethod(c, m,  
                a.getArgumentClasses());  
            meth.setAccessible(true);  
            return (meth.invoke(o, a.getArguments()));  
        }  
    catch ( IllegalArgumentException e )  
    ...  
    catch ( IllegalAccessException e )  
    ...  
        } } // Invoker
```

A receiver uses `Invoker` to access the methods of the actual object.

Finally, the class `DIEngine` offers the runtime support to the distributed hierarchy framework. This class allows to start the voyager orb in order to accept incoming calls and to stop it when needed. As we have seen in the previous example, the usage of `DIEngine` is the only code added to a program that uses remotized class. Note that this class can be ignored if the programmer wants to use Voyager of its own.
5.3.2 Forwarder Structure

A forwarder class declares two instance variables: a variable `px` to contain a `Proxy` object to the receiver and a class variable `url` representing the Internet address of the corresponding receiver. The variable `px` will be used to invoke the methods on the receiver. A method `Init`, invoked inside an instance initializer, enables the class downloading from `url`. In this way, a receiver object and its class file can be moved locally to the forwarder object.

We remind you that an instance initializer, according to the Java language specification [GJS96], is called before any constructor invocation. Below is shown the basic structure of `DummyForwarderProxy`.

```java
public class DummyForwarderProxy {
    static // instance initializer
    {
        Init();
    }

    ... // Fields
    private static String url;
    private Proxy px;
    ...
    private static void Init(){
        try {
            url = "//localhost:9000";
            VoyagerClassLoader.addURLResource("http:" + url);
        }
        catch (java.net.MalformedURLException $e) {
            System.err.println($e);
            $e.printStackTrace(System.err);
        }
    }

} // DummyForwarderProxy
```

Constructors

For each constructor in the original class `Dummy` a corresponding constructor is generated. All the constructors can build an instance of the receiver. Before any further description, it is better to give an example of how a constructor of `Dummy` having as formal parameter a `String` is modified. Below is the skeleton of the constructor.

```java
public Dummy(String name) { ... }
```

This constructor corresponds to the forwarder constructor:
public DummyForwarderProxy(java.lang.String $p0)
{  
    ArgumentHolder ah = new ArgumentHolder();  
    if ($p0==null)  
        ah.setArgument(null, java.lang.String.class);  
    else  
        ah.setArgument($p0);  

    try  
    {  
        px=Factory.create("DummyReceiverProxy",  
            ah.getArguments(), url);  
    }  
    catch (RuntimeException $e)  
    {  
        throw (RuntimeException) $e;  
    }  
    catch (Exception $e)  
    {  
        System.err.println($e);  
        $e.printStackTrace(System.err);  
    } // DummyForwarderProxy

In this constructor an instance of ArgumentHolder manages all the constructor parameters. The forwarder instantiation is done using the create method of the Voyager class Factory. This method allows to create an instance of a given class, DummyReceiverProxy in our case, available on a remote host at the address url. The constructor used when creating this instance is chosen according to the type and number of parameters in the array given in input as if the instantiation were local.

An instance of a Proxy subclass, which represents locally the built receiver instance, is returned as the result of the Factory.create call. This object is assigned to the instance variable px.

As we have seen, a programmer has two choices when remotizing a class: use the object on the remote JVM or move it locally. In any case, we have a three-tier architecture that at runtime builds an instance of the original class inside a receiver instance. When we need to perform the computation locally, we have to move the receiver instance together with the actual object from the remote url location to the location specified in localURL. LocalURL is an instance variable added to the forwarder structure when the user specifies in input a further Internet address.

public class DummyForwarderProxy {
...
private static String localUrl; // migration address  
} // DummyForwarderProxy
The address received specified by the user is assigned to localURL in `Init`. See below the modified version of `Init`.

```java
private static void Init(){
try {
    url="//localhost:9000";
    localUrl="//localhost:7000";
    VoyagerClassLoader.addURLResource("http:"+url);
} catch (java.net.MalformedURLException $e) {
    ...  
}
}
```

Therefore, the constructor of `DummyForwarderProxy` mentioned above is modified in this way:

```java
public DummyForwarderProxy(java.lang.String $p0){
  ArgumentHolder ah = new ArgumentHolder();
  if ($p0==null)
    ah.setArgument(null, java.lang.String.class);
  else
    ah.setArgument($p0);
  try {
    px=Factory.create("DummyReceiverProxy",
                      ah.getArguments(), url);
    IMobility mobility=Mobility.of(px);
    mobility.moveTo(localUrl);
  } catch( MobilityException $e) {
    ...  
}
}
```

The migration of the remote receiver and the actual object to the local JVM implies that each method call will affect the local JVM and not the remote JVM where the object was built.

Besides the constructors created from the corresponding constructors declared in `Dummy`, the DIC compiler adds to the receiver class a further private constructor that gets in input a `DummyForwarderProxy` parameter. In brief, this constructor is used when a method in the original class has `Dummy` itself as return type. In this case, every time the actual method is called, a new receiver object containing this
object and a new forwarder object will be created. For further details, see the parts corresponding to the compareTo method in the next sections.

```java
private DummyForwarderProxy(Proxy $p0) {
    px=$p0;
}
```

**Methods**

Analogously, a method can be invoked using another Voyager class called Sync. This class allows to invoke a method given a Proxy instance representing the target remote object, the method name and the actual parameters.

Let us see an example. Given the method info from the class Dummy, this method prints on the standard output a string parameter, as shown in the following fragment of code:

```java
public class Dummy {
    ...
    public void info(String s)
    {
        System.out.println(s);
    }
    ...
} // Dummy
```

The DIC compiler creates in DummyForwarderProxy an analogous method info, having the same signature, that forwards the call to the original method info. This is achieved by means of Sync.invoke. This method allows to redirect each method call to an instance of the receiver DummyReceiverProxy that px represents locally.

As shown below, all the parameters are managed by an instance of ArgumentHolder. Using its method getArguments, we obtain an array of parameters for the method info.

```java
public class DummyForwarderProxy {
    ...
    public void info(java.lang.String $p0){
        ArgumentHolder ah = new ArgumentHolder();
        if ($p0==null)
            ah.setArgument(null,java.lang.String.class);
        else
            ah.setArgument($p0);
        try
        {
            Result $r= Sync.invoke(px,"info(java.lang.String)"
```
,ah.getArguments();

} catch (RuntimeException $e) {
    throw (RuntimeException) $e;
} catch (Exception $e) {
    System.err.println($e);
    $e.printStackTrace(System.err);
}

...} // DummyForwarderProxy

Note that the RuntimeExceptions generated by the invocation of the original method are re-thrown.

Since the method info does not have a return value, this code snippet does not indicate how a possible return value is managed. If we had this situation we would need to modify the last part of the above code. The change required is straightforward. In fact, Sync.invoke returns an object instance of a Voyager class called Result that contains the value returned by the original method. Since this value should be transmitted, it is encapsulated within this class that implements the Serializable interface. To get back the original value after the method invocation we can use its method getObject that returns an instance of Object. This object will be assigned either to a primitive type wrapper variable or to a more specialized reference variable according to the actual return type of the original method.

If we need to use the wrapper, the corresponding primitive value can be obtained through a method getXXX where XXX is a primitive type name, as described in Chapter two. For instance, if we had to return an int value, we would have the following code:

```
Result $r = Sync.invoke(px, ...);
Object $v0 = $r.readObject();
Integer $v1;
$v1 = (Integer)$v0;
return $v1.intValue(); ...
```

Let us consider the method intMet in Dummy. What happens if the actual method throws an exception? In order for the forwarder to have locally the same behavior as the actual intMet method these exceptions should be re-thrown. This can be done with little effort since Sync.invoke receives back the exceptions generated by the original method.

This method, as described below, throws two checked exceptions (exceptions that require either to be caught or declared in the throws clause of each method).
5.3. THE FORWADER AND RECEIVER FRAMEWORK

```java
public int intMet(int i) throws java.io.IOException, java.lang.InterruptedException{
    if (i==0) throw new java.io.IOException();
    if (i==1) throw new java.lang.InterruptedException();
    return (i+1);
}
```

Well, the receiver version of the `intMet` method delivers each exception to the `Sync.invoke` statement. In turn, `Sync.invoke` re-throws the exception as a generic one. This is caught and re-thrown again according to its actual type. In this way, the same exceptions thrown by the actual method are thrown by the forwarder version of `intMet`. How this method is represented in the forwarder is described in the following code excerpt:

```java
public int intMet(int $p0) throws java.io.IOException, java.lang.InterruptedException{
    ArgumentHolder ah = new ArgumentHolder();
    ah.setArgument($p0);
    try{
        Result $r = Sync.invoke(px,"intMet(int)",ah.getArguments());
        Object $v0= $r.readObject();
        Integer $v1;
        $v1 = (Integer)$v0;
        return $v1.intValue();
    }
    catch (java.io.IOException $e){
        throw (java.io.IOException)$e;
    }
    catch (java.lang.InterruptedException $e){
        throw (java.lang.InterruptedException)$e;
    }
    catch (RuntimeException $e){
        throw (RuntimeException) $e;
    }
    catch (Exception $e){
        System.err.println($e);
        $e.printStackTrace(System.err);
        return 0;
    }
}
```

Besides the exceptions in the throws clause, also the `RuntimeExceptions` generated by the invocation of the original method are re-thrown. We remind you that the
*RuntimeExceptions* are those kinds of exceptions that Java does not require to catch or put in the throws clause of a method. Nevertheless, to maintain a coherent behavior, even those exceptions must be re-thrown by the forwarder.

Another interesting feature is the management of parameters having the same class type of the input class, i.e. `Dummy`, as mentioned earlier. Since this case represents an anomaly with respect to a usual method, it must be dealt with precaution. When a programmer uses a `DummyForwaderProxy` in his class, he does not have available the original class. Hence, everywhere the original class is used, it is replaced with its local representative, that is, the forwarder class.

For instance, if we had a method such as `compareTo` that receives in input another instance of `Dummy` and returns a `Boolean` as in the fragment of code shown:

```java
Boolean compareTo(Dummy d) {
    ...
    return ...;
}
```

The `DPC` compiler will create the forwarder version of `compareTo` in this way:

```java
java.lang.Boolean compareTo(DummyForwarderProxy $p0){
    ArgumentHolder ah = new ArgumentHolder();
    if ($p0==null)
        ah.setArgument(null,Proxy.class);
    else
        ah.setArgument($p0.px); // proxy added to the argument list
    try
    {
        Result $r=Sync.invoke(px
            ,"compareTo(com.objectspace.voyager.Proxy)"
            , ah.getArguments());
        Object $v0= $r.readObject();
        java.lang.Boolean $v1;
        $v1= (java.lang.Boolean)$v0;
        return $v1;
    }
    catch (RuntimeException $e)
    {
        throw (RuntimeException) $e;
    }
    catch (Exception $e)
    {
        System.err.println($e);
        $e.printStackTrace(System.err);
        return null;
    }
} }
```
As you can see, the parameter type becomes `DummyForwarderProxy` instead of `Dummy`. Its content is used to add a `Proxy` instance to the `ArgumentHolder` object. In turn, the resulting array obtained from `ArgumentHolder` will be used to invoke the `compareTo` method of the receiver. Notice that this method gets as parameter a `Proxy`. The remainder of the method has the same structure described earlier for a conventional method.

Let us now see the approach to manage a return value whose type is the class itself, i.e. `Dummy`. This circumstance is typical of a factory method. For instance, in our class `Dummy` we have a static method `create` that returns a new instance of `Dummy` itself.

```java
static public Dummy create(String s) {
    return new Dummy(s);
}
```

The `create` method is implemented in `DummyForwarderProxy` so that it invokes a method `create$Proxy` belonging to the receiver. `Create$Proxy` returns an instance of `Proxy` representing the result of the original create invocation. In turn, a new instance of `DummyForwarderProxy` is created using a private constructor that takes as parameter a `Proxy` instance. As usual, the `Proxy` instance returned by `create$Proxy` is stored in the attribute variable `px` of `DummyForwarderProxy`. This instance is actually returned from the `create` method of the forwarder.

Note that this excerpt shows also how to invoke remotely a static method using `Sync.invoke`.

```java
public static DummyForwarderProxy create(java.lang.String $p0){
    ArgumentHolder ah = new ArgumentHolder();
    if ($p0==null)
        ah.setArgument(null,java.lang.String.class);
    else
        ah.setArgument($p0);
    try {
        Result $r= Sync.invoke("DummyReceiverProxy",
                "create$Proxy(java.lang.String)", ah.getArguments(), url);
        Object $v0= $r.readObject();
        if ($v0!=null)
            { Proxy $v1;
              $v1= (Proxy)$v0;
              return new DummyForwarderProxy($v1); // create a new instance
            }
        else
            return null;
    } catch (RuntimeException $e)
```
{  
throw (RuntimeException) $e;
}
catch (Exception $e)  
{  
System.err.println($e);  
$e.printStackTrace(System.err);
return null;
}

With this last example we have almost completed our tour. In fact, we have shown how the Dic compiler creates the methods of a forwarder from the methods declared in a class received in input. From the examples shown, the Dic compiler seems to be able to remotize any kind of method. Nevertheless, there are some method parameters under particular circumstances that cannot be managed properly in our framework. We mean all those methods that declare as parameter an interface that is also implemented by the class itself. Since it is not possible to know in advance the kind of object that will be passed at run-time, it could happen that an instance of the forwarder class is passed. In this case the "remotized" version of the method does not work as expected since it should invoke the corresponding receiver method as if it had received a forwarder object, therefore using the content of px, and not an object that can be used directly. At the moment this issue has not been solved since it raises several problems on the approach to follow.

We adopt a specific approach for the methods inherited from Object such as clone, equals and finalize. We do not create a corresponding method hashcode in the forwarder since it does not make sense to obtain the hashcode of an object in another JVM. The method toString is like the others seen previously, therefore it does not require further attention.

Let us examine clone. This method is implemented to clone the forwarder and at the same time invokes the clone method on the receiver. The receiver returns a proxy to be assigned to the instance variable px belonging to the cloned forwarder.

```java
protected java.lang.Object clone() throws java.lang.CloneNotSupportedException{
    java.lang.CloneNotSupportedException $c0=null;
    try  
    {  
        $c0=(DummyForwarderProxy)super.clone();  
    }
    catch (java.lang.CloneNotSupportedException $e)  
    {  
        ...  
    }
    ArgumentHolder ah = new ArgumentHolder();
```
5.3. THE FORWARDER AND RECEIVER FRAMEWORK

try {
    Result $r= Sync.invoke(px,"clone", ah.getArguments());
    Object $v0= $r.readObject();
    $c0.px= (Proxy)$v0;
    return $c0;
} catch (java.lang.CloneNotSupportedException $e) {
    throw (java.lang.CloneNotSupportedException)$e;
} catch (RuntimeException $e) {
    throw (RuntimeException)$e;
} catch (Exception $e) {
    System.err.println($e);
    $e.printStackTrace(System.err);
    return null;
} }

What happens on the receiver side when clone is invoked will be described later in the corresponding section.

The method equals in DummyForwarderProxy is always created since Dummy has equals in any case. In the original class this method can be either directly inherited from Object or overridden. In our case, Dummy overrides equals. The forwarder version is show below:

```java
public boolean equals(java.lang.Object $p0){
    ArgumentHolder ah = new ArgumentHolder();
    if ($p0==null)
        ah.setArgument(null,Proxy.class);
    else
        ah.setArgument(((DummyForwarderProxy)$p0).px);
    try {
        Result $r= Sync.invoke(px
            ,"equals(com.objectspace.voyager.Proxy)"
            ,ah.getArguments());
        Object $v0= $r.readObject();
        Boolean $v1;
        $v1= (Boolean)$v0;
        return $v1.booleanValue();
    } catch (RuntimeException $e) {
```
{ 
    throw (RuntimeException) $e;
}
catch (Exception $e) 
{
    System.err.println($e);
    $e.printStackTrace(System.err);
    return false;
}

The method equals in the forwarder returns true if and only if the method equals applied to the objects wrapped inside the corresponding receivers returns true. As shown above, a method equals, that gets a parameter of type Proxy, is invoked on the instance of DummyReceiverProxy represented locally by px.

As far as the finalize method is concerned, it is simply rewritten to manage the local instance of DummyForwarderProxy. This method will be invoked by the local garbage collector when the object is destroyed. Note that it does not call the finalize method of the corresponding DummyReceiverProxy object. This method will be called according to the policy of the garbage collector running on the JVM where this object was created.

protected void finalize() throws java.lang.Throwable {
    px=null;
    super.finalize();
}

5.3.3 Receiver structure

Now we can see how the receiver is created and how it works. The receiver has the same methods of the original class and only one instance variable named superClass. This variable contains the actual instance of the original class. For instance, given the usual class Dummy, the receiver created has the following structure:

public class DummyReceiverProxy {
    ...
    // Fields
    Dummy superClass=null;
    ...
} // DummyReceiverProxy

The instance variable superClass is initialized inside the constructors.

Constructors

A sample constructor that satisfies the request on the other side to create an instance is shown below. This constructor receives as parameter a String.
public DummyReceiverProxy(java.lang.String $p0) {
    try {
        ArgumentHolder ah = new ArgumentHolder();
        if ($p0==null)
            ah.setArgument(null, java.lang.String.class);
        else
            ah.setArgument($p0, java.lang.String.class);
        superClass=(Dummy) Builder.dynaBuild(Dummy.class, ah);
    } catch (java.lang.reflect.InvocationTargetException $ite) {
        Throwable $re=$ite.getTargetException();
        if ($re instanceof RuntimeException)
            throw (RuntimeException) $re;
    }
}

As in the forwarder where we use the Voyager class Factory to create a remote instance of a class, here we use a helper class Builder to create a local instance of a class. The static method dynaBuild creates a class instance using the constructor whose formal parameters match the actual parameters class type. Note that the method dynaBuild does not consider the constructor visibility modifier.

This kind of operation could raise an exception that is re-thrown catching InvocationTargetException. InvocationTargetException is a checked exception that wraps an exception thrown by an invoked method or constructor. In this case the wrapped RuntimeExceptions are re-thrown since they are the only ones that could be generated by the method.

Methods

Let us see how the DIC compiler creates the method info in the receiver.

public void info(java.lang.String $p0) {
    try {
        ArgumentHolder ah = new ArgumentHolder();
        if ($p0==null)
            ah.setArgument(null, java.lang.String.class);
        else
            ah.setArgument($p0, java.lang.String.class);
        Object $v0= Invoker.dynaCall(Dummy.class, superClass
            , "info", ah);
    } catch (java.lang.reflect.InvocationTargetException $ite) {
        throw (java.lang.reflect.InvocationTargetException) $ite;
    }
}
```java
public int intMet(int $p0) throws java.io.IOException, java.lang.InterruptedException
{
    try
    {
        ArgumentHolder ah = new ArgumentHolder();
        ah.setArgument($p0);
        Object $v0 = Invoker.dynaCall(Dummy.class, superClass, "intMet", ah);
        Integer $v1 = (Integer)$v0;
        return $v1.intValue();
    }
    catch (java.lang.reflect.InvocationTargetException $ite)
    {
        Throwable $re = $ite.getTargetException();
        if ($re instanceof java.io.IOException)
            throw (java.io.IOException) $re;
        else if ($re instanceof java.lang.InterruptedException)
            throw (java.lang.InterruptedException) $re;
        else if ($re instanceof RuntimeException)
            throw (RuntimeException) $re;
        else
            return 0;
    }
}
```

Suppose one or more parameters of a method are the class itself, then the receiver declares a method that turns each parameter type into Proxy. In this way, the method receives Proxy objects that represent other receiver objects. For instance, given the compareTo method of Dummy, we obtain the following version of compareTo in the receiver:

```java
public java.lang.Boolean compareTo
(com.objectspace.voyager.Proxy $p0) {
```
try {
    ArgumentHolder ah = new ArgumentHolder();
    if ($p0==null)
        ah.setArgument(null,Dummy.class);
    else
        ah.setArgument(
            ((DummyReceiverProxy)$p0.getLocal()).superClass , Dummy.class);
    Object $v0= Invoker.dynaCall(Dummy.class, superClass
            , "compareTo", ah);
    java.lang.Boolean $v1;
    $v1= (java.lang.Boolean)$v0;
    return $v1;
} catch (java.lang.reflect.InvocationTargetException $ite)
{
    Throwable $re=$ite.getTargetException();
    if ($re instanceof RuntimeException)
        throw (RuntimeException) $re;
    else
        return null;
}

In compareTo the actual object is obtained using the $p0 parameter. Since it is an instance of Proxy, it is possible to obtain the receiver represented by means of the getLocal method. Once obtained the receiver object, we can invoke the actual compareTo method passing the object contained in superClass together with the actual object.

Now we can see how the DummyReceiverProxy rewrites those kinds of methods that return an instance of the class itself such as the create method in Dummy. Given this method, we obtain in DummyReceiverProxy a method named create$Proxy as show below:

```java
public static com.objectspace.voyager.Proxy create$Proxy
    (java.lang.String $p0){
    try {
        ArgumentHolder ah = new ArgumentHolder();
        if ($p0==null)
            ah.setArgument(null,java.lang.String.class);
        else
            ah.setArgument($p0,java.lang.String.class);
        Object $v0= Invoker.dynaCall(Dummy.class, null
            , "create", ah);
        Proxy $v1;
        $v1= Proxy.of(new DummyReceiverProxy((Dummy) $v0));
    }
```
return $v1;
}  
catch (java.lang.reflect.InvocationTargetException $ite)
{
    Throwable $re=$ite.getTargetException();
    if ($re instanceof RuntimeException)
        throw (RuntimeException) $re;
    else
        return null;
}  

This method must return a proxy to a receiver containing the Dummy object returned by the actual create. This is achieved by means of the Voyager class Proxy. This class has a static method named of that creates an instance of Proxy from an object. This is the Proxy object sent back to the DummyForwarderProxy object.

Now the remainder of the DummyReceiverProxy description concerns the methods inherited from Objects such as equals, finalize and clone. The method hashcode is not remotized as explained in the forwarder part. The method toString does not have a particular behavior that requires any specific precautions. It is like the methods seen so far. Yet, the others imply some attention since their semantics must be adapted to the distributed context. Since a partial description on the forwarder side has been previously done, some details will be taken for granted.

The method clone is rewritten in the class DummyReceiverProxy in order to create a new instance of DummyReceiverProxy and assign its variable superClass the result of the clone invocation on the object contained in the receiver.

```
  public java.lang.Object clone() throws
        java.lang.CloneNotSupportedException{
        DummyReceiverProxy $c0=null;
        try
        {
            $c0=(DummyReceiverProxy)super.clone();
        }
        catch (java.lang.CloneNotSupportedException $e)
        {
            ...
        }
        try
        {
            ArgumentHolder ah = new ArgumentHolder();
            Object $v0= Invoker.dynaCall(Dummy.class, superClass
            , "clone", ah);
            $c0.superClass=(Dummy)$v0;
            return Proxy.of($c0);
        }
```
5.3. THE FORWARDER AND RECEIVER FRAMEWORK

```java
    catch (java.lang.reflect.InvocationTargetException $ite)
    {
        Throwable $re=$ite.getTargetException();
        if ($re instanceof java.lang.CloneNotSupportedException)
            throw (java.lang.CloneNotSupportedException) $re;
        else
            if ($re instanceof RuntimeException)
                throw (RuntimeException) $re;
            else
                return null;
    }
```

As far as the `equals` method is concerned, it applies the original `equals` method to the current content of `superClass` and to the content of `superClass` obtained from the `Proxy` instance received in input.

```java
    public boolean equals(com.objectspace.voyager.Proxy $p0){
        try {
            ArgumentHolder ah = new ArgumentHolder();
            if ($p0==null)
                ah.setArgument(null,java.lang.Object.class);
            else
                ah.setArgument(  
                    ((DummyReceiverProxy)$p0.getLocal()).superClass   
                    .java.lang.Object.class);

                Object $v0= Invoker.dynaCall(Dummy.class, superClass  
                    , "equals", ah);

                Boolean $v1;
                $v1= (Boolean)$v0;
                return $v1.booleanValue();
        }
        catch (java.lang.reflect.InvocationTargetException $ite)
        {
            Throwable $re=$ite.getTargetException();
            if ($re instanceof RuntimeException)
                throw (RuntimeException) $re;
            else
                return false;
        }
    }
```

Finally, the `finalize` method, as in the forwarder, simply destroys the content of the variable `superClass`.

```java
    protected void finalize() throws java.lang.Throwable {
        superClass=null;
        super.finalize();
    }
```
5.3.4 Forwarders and receivers at work

Now we must assess the changes required in the source code that uses the "remotized" class together with the generated framework. According to the description given above, it is clear that only a few changes are required. In fact, given a source file that uses our class *Dummy*, each occurrence of *Dummy* must be replaced with its forwarder counterpart *DummyForwarderProxy* and before using it the runtime support must be launched adding *DIEngine.make()* to the source code. Nothing else changes or has a different behavior. Everything continues to work as before, the only difference is where the class "remotized" now resides. It is on a different host and communicates using the couple receiver-forwarder.

For instance, given a program that uses our class *Dummy* such as:

```java
import di.*;
public class MainClient {
    public static void main(String[] args) {
        Dummy dp=new Dummy();
        dp.info("Hallo!!!");
        int j=0;
        do {
            try {
                int i=dp.intMet(j);
                System.out.println(new Integer (i));
            } catch (java.io.IOException ex){
                System.out.println("IOException");
            } catch (java.lang.InterruptedException ex){
                System.out.println("InterruptedException");
            }
            j++;
        } while (j<5);
    } } // MainClient
```

We must modify it in order to use the forwarder. Below is a simple class that makes these changes in order to use *DummyForwarder* and start the engine. Here only the forwarder class must be available locally.

```java
import di.*;
public class MainClient {
    public static void main(String[] args) {
```
5.3. THE FORWARDER AND RECEIVER FRAMEWORK

```java
DIEngine ris=DIEngine.make(); // added
DummyForwarderProxy dp=new DummyForwarderProxy();
    // changed

dp.info("Hallo!!!");
int j=0;

do
{
  try {
    int i=dp.intMet(j);
    System.out.println(new Integer(i));
  }
  catch (java.io.IOException ex){
    System.out.println("IOException");
  }
  catch(java.lang.InterruptedException ex){
    System.out.println("InterruptedException");
  }
  j++;
}
while (j<5);
} } // MainClient
```

On the server side we have the program that starts the engine and waits for the incoming requests. The JVM that runs this program must have available the receiver class, that is DummyReceiverProxy, and the original class Dummy.

```java
import di.*;
public class MainServer {

  public static void main(String[] args) {
    DIEngine.make("//localhost:9000");
    System.out.println("MainServer built");
  }
} } // MainServer
```

The tool usage is simple, as we have seen in the example at the beginning of this chapter. The programmer, once compiled the original class Dummy, to create the corresponding framework must type the following command:

dic Dummy localhost:9000

This command creates DummyReceiverProxy.java and DummyForwarderProxy.java. In a real context, the receiver, the class Dummy and a server program that at least starts the engine should be moved on a different directory or host. The forwarder together with all the classes using Dummy, updated properly in order to use the forwarder, can be used directly or must be moved to a production directory where they can be instantiated.
Afterwards, if we try everything on a single host, we open two different shells. Beforehand we have to start the MainServer program typing `java MainServer`, subsequently we start the MainClient typing `java MainClient`.

<table>
<thead>
<tr>
<th>Server Side</th>
<th>Client Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>[d:../serverDmy/] Java MainServer MainServer built</td>
<td>[d:../clientDmy/] Java MainClient</td>
</tr>
<tr>
<td>Hallo!!</td>
<td>IOException</td>
</tr>
<tr>
<td></td>
<td>InterruptedException</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

As you can see all the methods invocations return their value or throw the exceptions where MainClient runs, while their actual execution happens on the JVM running the MainServer program. Actually, the method info prints Hello on the MainServer side, that is on the JVM running at the localhost:9000 address, while the results of all the invocations of intMet, included the exceptions thrown, are displayed on the other side.

If we had chosen to generate the forwarder so that it moves locally the instance of the remote receiver together with the instance of the original class, we would have obtained the following result:

<table>
<thead>
<tr>
<th>Server Side</th>
<th>Client Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>[d:../serverDmy/] Java MainServer MainServer built</td>
<td>[d:../clientDmy/] Java MainClient</td>
</tr>
<tr>
<td></td>
<td>IOException</td>
</tr>
<tr>
<td></td>
<td>InterruptedException</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

Let us go back to what we said earlier. We mentioned this alternative approach as useful when we have some classes whose behavior must be available locally to their client. It does not matter whether the client inherits, encapsulates or simply uses this class. In any case, it requires that the execution of the methods effects the local JVM.

As in the above example the string “Hallo!!!” is printed on the forwarder side. Despite that, a receiver instance and a Dummy are created on the other JVM.

This kind of forwarder can be generated typing:

dic Dummy localhost:9000 localhost:7000

As we have seen in other examples, the first address represents the location where each receiver instance and the encapsulated instance of Dummy must be created, the second one is the address where they have to be moved after their instantiation. In
our scenario one or more instances of the forwarder run on the JVM at localhost:7000, while the receiver instances are built at localhost:9000 and afterwards moved to localhost:7000.

The widespread usage of the address localhost is due to the fact that the development has been done mainly on a single pc. Thus, we had to distinguish between the forwarder and the receiver location using the port number. In our example the receiver listens to incoming calls either on the port 9000 or, if it has been moved, on the port 7000. Instead, the forwarder does not need a specific location, a default one is assigned by the engine, unless it has to request a move action to the receiver. In this case the address where it runs, that is localhost:7000 in our example, must be specified.

Nevertheless, although in this example the receiver and the forwarder are on the same host, the framework has also been tested with the forwarder and the receiver on distinct hosts. In addition, some interesting trials have been done with several predefined Java classes such as ArrayList, Vector and so on. Our Dic compiler has remotized them smoothly, allowing us to use a remote instance of an ArrayList or a remote instance of Vector without problems. Notice that the trial with these classes has been possible even without the availability of their source code. Only their compiled version is required as we will see in the next section.

5.4 Remotizing a system class

Let us suppose we have only the bytecode file of a class that we want to remotize. This is not an issue for the Dic compiler since it does not need the source code to create the forwarder class and the receiver class. It needs only the compiled file, and everything will work as described in the previous examples.

For instance, if we want to create a remote version of java.util.ArrayList, a system class file shown in Fig. 5.9, then we have only to run our Dic compiler as shown below:

```
dic java.util.ArrayList localhost:9000
```

As expected, the Dic compiler will create the usual two files, ArrayListForwarderProxy and ArrayListReceiverProxy. As shown in Fig. 5.10, the class ArrayListForwarderProxy implements the same interfaces of ArrayList. This means that ArrayListForwarderProxy and its instances can be used everywhere the original ArrayList and its instances are expected. Analogously, ArrayListReceiverProxy, see Fig. 5.11, implements the same interfaces of ArrayList. Notice that the two proxies do not belong to the java.util package. It is not possible to add new classes to a system package.

ArrayListReceiverProxy together with the ubiquitous Mainserver program, see the previous examples for further details, represent the server side of the application. While the client side of the application consists of ArrayListForwarderProxy together with the program ShuffleMy shown below.

```
import java.util.*;
import di.*;
```
Figure 5.9: java.util.ArrayList
Figure 5.10: `ArrayListForwarderProxy`
Figure 5.11: ArrayListReceiverProxy
public class ShuffleMy {
    public static void main(String args[]) {
        DIEngine ris=DIEngine.make();  //added

        ArrayListForwarderProxy l =
            new ArrayListForwarderProxy();  //changed
        for (int i=0; i<args.length; i++)
            l.add(args[i]);
        Collections.shuffle(l, new Random());
        System.out.println(l);
        // print all the elements in ArrayList
        //through ArrayList.toString

        DIEngine.destroy();  //added
    } }  // ShuffleMy

This application adds to an instance of the remotized ArrayList all the elements that it receives in input, shuffles them and prints the resulting list of elements. Notice how the ArrayListForwarderProxy is used as if it were a standard ArrayList when Collections.shuffle is called.

Typing java Mainserver we run the server program. When we run java ShuffleMy 1 10 6 7, the main class ShuffleMy creates a remote instance of the class ArrayList, it fills this instance with all the elements received in input and it prints these elements shuffled, that is 6 10 1 7.

<table>
<thead>
<tr>
<th>Server Side</th>
<th>Client Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>[d:/.../server Array/]java Mainserver built</td>
<td>[d:/.../client.Array/]java ShuffleMy 1 10 6 7 [6, 10, 1, 7]</td>
</tr>
</tbody>
</table>

Notice that the actual instance of the ArrayList is not running on the JVM where ShuffleMy is launched.

### 5.5 Remotizing a linked list

Let us suppose we want to remotize a more complex class. In this example we show how to remotize a linked list. Given a class Node such as:

```java
import java.io.*;
public class Node implements Serializable {
    int info;
    Node next;
}
```
public Node() {
    info=0;
    next=null;
}

public Node(int info, Node next){
    this.info=info;
    this.next=next;
}

public Node getNext(){
    return next;
}

public void setNext(Node next){
    this.next=next;
}

public int getInfo(){
    return info;
}

public void setInfo(int info){
    this.info=info;
} } // Node

In this case an instance variable of the class Node is a link to the next Node of the list. To access this instance variable there are the methods getNext and setNext. Below is shown a simple program that creates a local list made of three elements and traverses this list to print the content of the instance variable info for each Node.

public class MainList {
    public static void main(String[] args) {

        Node head, head_aux;

        head = new Node();
        head = new Node(1,head);
        head = new Node(2,head);
        for(head_aux=head;head_aux!=null;head_aux=head_aux.getNext())
            System.out.print(head_aux.getInfo());
    } } // MainList

Now suppose that we remotize the class Node. As usual we use the DIC compiler.

dic Node localhost:9000
In this way we create the usual couple of class files named in this case NodeForwarderProxy and NodeReceiverProxy. We modify the class MainList in order to use the forwarder instead of the actual class Node as shown below.

```java
public class MainList {
    public static void main(String[] args) {
        NodeForwarderProxy head, head_aux;
        head = new NodeForwarderProxy();
        head = new NodeForwarderProxy(1, head);
        head = new NodeForwarderProxy(2, head);

        for (head_aux = head; head_aux != null; head_aux = head_aux.getNext())
            System.out.println(head_aux.getInfo());
    }
} // MainList
```

Subsequently, we move the forwarder and the receiver to two distinct directories in order to mimic a client-server application on a single host. We put the forwarder together with the modified version of MainList in the directory d:\projects\java\examples\clientList\. While the directory d:\projects\java\examples\serverList\ contains the class NodeReceiverProxy, the original class Node and the class MainServer that represents the run-time support.

First of all, we start the run-time support MainServer. After that, we start the MainList program in order to create the linked list and print its content.

<table>
<thead>
<tr>
<th>Server side</th>
<th>Client side</th>
</tr>
</thead>
<tbody>
<tr>
<td>[d:\serverList] Java MainServer MainServer built</td>
<td>[d:\clientList] java MainList</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

This example shows how to create, using the DICE compiler, a linked list that seems to be local although it is actually remote. It is apparent that the main difference with respect to the previous example, where the whole collection ArrayList is remotized, relies on the remote nature of each Node.

5.6 Summary

Our approach makes the difference on the development stage because it simplifies the implementation task. An application can be developed without using any known
distributed architecture such as CORBA, Java RMI and so on. Thereafter, a programmer having little or no knowledge of distributed architectures can develop a distributed application in a shorter time and with less effort.

There is only one disadvantage when a class is remotized: its public instance variables and class variables are no longer accessible from its remote clients. At the moment we are assessing the validity of this choice. Another viable approach could be an automatic generation of a couple of accessor methods in the forwarder and the receiver for each instance or class variable. Whatever the choice, the source code where an instance of the original class is used needs to be overhauled.
Chapter 6

The dic compiler

Using the Java reflection mechanism, like the RMIC compiler that creates stub and skeleton from a compiled class, the dic compiler creates a framework from a class file and an Internet address. If a second address is passed in input, this is used to create a framework where the actual object is moved after its instantiation.

As we have seen, this framework consists of two Java classes, a receiver proxy and a forwarder proxy to the original class, to be installed on the client and server side respectively. The forwarder class name is obtained concatenating the original class name with the suffix "ForwarderProxy", while the receiver class name is obtained concatenating the original class name with the suffix "ReceiverProxy".

It would be possible to avoid the suffix “ForwarderProxy” if the dic compiler did not create this class in the same directory of the original class. In this way, with some precautions, the forwarder would have the same name of the original class. Therefore, we would avoid the changes in all the clients that use the remotized class. This feature, probably, would be supported in the next release of the dic compiler.

To create the two proxies, the dic compiler gathers the signatures of the non private methods, including the inherited, defined in the loaded class and writes to file both the proxies using this information.

All the classes that are part of the dic compiler project are bundled in the package named ditools, distributed hierarchy tools. This package, also referred as the proxy generator suite, contains the following classes: CreateProxy, CreateForwarderProxy, CreateReceiverProxy, Reflector, ResultHolder, dic and some other utility classes. See Fig. 6.1 and Fig. 6.2.

The key class in ditools is the class Reflector since it has an important role during the forwarder and receiver creation. Reflector has a couple of methods, getAllInterfaces and getAccessibleMethods, that receive in input a Class object and return respectively all the declared non-private methods and all the implemented interfaces. The first one, getAllInterfaces, collects in an array a list of all the interfaces implemented by a class and its superclasses. The second one, getAccessibleMethods, returns an array of all the methods, except the private ones, that a given class declares or inherits from its ancestors. To gather this data, both these methods traverse the hierarchy tree corresponding to the given class. During the traversal they collect, in an instance of the class Vector, all the interfaces or all the methods corresponding
to each ancestor met. For instance, `getAccessibleMethods` collects every non-private method available in each ancestor as long as there is no previous occurrence in the `Vector` object. In fact, it might happen to meet a method more than once if it is an overridden one.

Notice that the services offered by `Reflector` are not directly available in the reflection API\(^1\).

The method `equalStructure` compares two methods according to the following criteria: they are equal if they have the same name and the same parameter types. Since in Java the covariant redefinition of the return type is forbidden, see [BoC97] for an extension proposal, the methods do not compare with each other the return types.

```java
public class Reflector {
    static boolean debug=false;

    static boolean equalStructure(Method m1, Method m2) {
        boolean equal=false;
```}

\(^1\)The reflection package allows only to collect either the methods declared directly inside a class or all its public methods, it does not return all the methods that a class offers. Similarly, it is not possible to obtain all the interfaces that are implemented by a class and its ancestors using the basic services offered by the reflection package.
if (m1.getName().equals(m2.getName()))
{
    Class[] pt1=m1.getParameterTypes();
    Class[] pt2=m2.getParameterTypes();
    if (pt1.length==pt2.length)
    {
        equal=true;
        for (int i=0;(i<pt1.length)&&(equal);i++)
        {
            if (!pt1[i].getName().equals(pt2[i].getName()))
                equal=false;
        }
        return equal;
    }
}

// Reflector

Reflector is widely used by the proxy builders CreateReceiverProxy and CreateForwarderProxy to create respectively a receiver class and a forwarder class. These two classes work in a similar way: they introspect a class to create a new class that has almost the same method signatures. This basic behavior is implemented in a common ancestor named CreateProxy. Hence, an early insight in the structure of this class is required to fully understand its derived classes.

CreateProxy provides the basic services to create a receiver class file and a forwarder class file. CreateProxy is built out of the following methods: print_package to write the package part, print_methods to write on an output stream all the method signatures obtained using the Reflector method getAccessibleMethods, and print_constructors to print all the constructors signatures. In turn, both print_methods and print_constructors invoke a method named print_method_or_constructor which writes in the proper way the method signature of a given method or constructor. Notice that in createProxy this method is empty since it is overridden in its subclasses.

The actual process of creating respectively the forwarder and the receiver is done in CreateReceiverProxy and CreateForwarderProxy. These two classes have a static method make that introspects a compiled class received in input and writes on file, through the mentioned methods print_package, print_methods and print_constructors, a proxy class. Obviously, the behavior of these methods is specialized according to the target proxy. In fact, CreateForwarderProxy overrides print_method_orConstructor and some other methods in order to create the forwarder source code. So does CreateReceiverProxy to create the receiver source code.

Notice that the DCC compiler puts the two proxies obtained when remotizes a class in the same package of the original class, except for all the classes that belong to the core packages, i.e. the packages in the java.* domain. In this case the default package is used.

A convenience class named DCC combines the two Proxy creators in a single program to give the user a single point of access to both classes. Briefly, DCC invokes the
Figure 6.2: Package ditools: CreateProxy, CreateForwarderProxy, CreateReceiverProxy
method *make* of *CreateForwarderProxy* and *CreateReceiverProxy* to create respectively a forwarder class and a receiver class source code, compiles them by means of the class *Exec* that invokes two external processes that run the standard Java compiler installed on the system.

Note that the DIC compiler used thoroughly during our examples is a Windows batch file, as shown below, that invokes the class DIC in the package *ditools*.

```java
ditools.dic %1 %2 %3
```

Where %1, %2 and %3 are placeholders for the parameters passed to the script.
Chapter 7

Conclusion

After an analysis of all the available distributed architectures, we realized that the class inheritance mechanism is an issue completely neglected and nothing gives the impression that there are plans to support it in the near future. Moreover, based on our experience, we can affirm that almost all the current architectures are quite difficult to learn; at least for someone who has never developed a distributed application.

In effect, a novice ready to make his first steps in the arena of distributed computing has to learn both the gist of this fascinating world and the technicalities of an actual architecture. Yet, if he moves, for some reasons, from one architecture to another, he must retrain himself. In fact, some architectures have such huge differences that it is required to learn almost everything from scratch. For instance, think about the effort required to learn Microsoft DCOM even though you are a skilled CORBA developer.

Therefore, our proposal aims to simplify as much as possible this scenario. At the same time, it tries to overcome the lack of support of class inheritance among distributed classes. We believe that this deficiency is not acceptable since the object-oriented paradigm is not completely fulfilled. Notwithstanding, we have noticed almost a complete lack of interest in this topic by the scientific community. In fact, after an extensive research of publications on the topic, we have only found the DS\(^1\) and GOS\(^2\) proposals described in Chapter four.

As we have seen, the DS project has completely failed on the inheritance issue although it was an interesting experiment aimed at introducing in Smalltalk all that features now common in programming languages or architectures for distributed computing. While, GOS, with its division of state and behavior and the attachment on-demand feature is the first viable solution available until now.

Our work arises from the ideas of GOS, although we propose a new object model. In GOS the state of an object, the chassis, exists in a single entity while its behavioral part is made of one or more remote entities called engines, see Fig. 7.1. The whole behavioral part is obtained by the composition of all the engines.

In our model the objects are managed in a different way. The simplest scenario occurs when a class is remotized and directly used. In this case an instance of the

---

\(^1\)Distributed Smalltalk.

\(^2\)Global Object Space.
class is created through its corresponding forwarder. Actually, the forwarder object is local while the actual object is remotely created on the JVM where the class is available. In this case, state and behavior are both remote and co-resident from the client's point of view.

Nevertheless, it is also possible to create an instance of a class having as ancestor a forwarder. This case implies that an object has state and behavior scattered among two or more JVMs as shown in Fig. 7.2. The instance variables and the methods declared in the descendant class are local, while the methods and the instance variables of the ancestor class are remote. Similarly, the ancestor might have a remote superclass that, in turn, might be the descendant of a remote class and so on. This chain might continue until the root class is reached. It is like having an object sliced horizontally, as shown in Fig. 7.3, and each slice, containing both state and behavior, is on a different JVM. Every layer corresponds to a level in a distributed inheritance hierarchy. Obviously, the more layers we have the less efficient is a method invocation. However, it is possible to cancel the communication overhead by moving locally the remote class and its instance dynamically as we have seen in section 5.1.

It is clear that the GOS approach and our own approach are different. In GOS an object is vertically partitioned into state and behavior, while, in our distributed hierarchy, an object is horizontally partitioned into multiple layers containing both state and behavior. Apart from the object model, the GOS environment, in our opinion, has a considerable communication overhead due to the separation of state and behavior. Moreover, GOS still requires to use a specific CORBA implementation that is not transparent to the programmer. In fact, a programmer has to write an IDL file, compile it, and use the generated classes to develop his application.

On the other hand, though one of our objectives was to introduce class inheritance
in the distributed context, the other was to simplify the development effort of remote objects. We believe that we have fulfilled our objective. Our DIC compiler enables an object to offer its services to remote clients smoothly and, in addition, allows a class to extend a remote class. In fact, in the examples proposed we have seen how to develop a distributed object application without forcing a programmer to use any specific architecture. Our simple tool can give the power of distributed object oriented computing to the reach of the average Java programmer.

This is possible since the receiver and the forwarder are standard Java classes that can be seamlessly integrated with any program. A programmer needs only to use the DIC compiler to generate them. Except for this chore, an application can be developed using all the object oriented facilities available in Java, including class inheritance. Actually, a remotized class is like a local one during the development stage. In fact, an instance can be created using a simple new, it can be extended using the usual extends clause, its methods can be overridden and overloaded by a descendant class. In other words, from the programmer’s point of view, it is like developing a common Java class.

It is apparent that a skilled programmer, according to its specific needs, could create a framework similar to the receiver-forwarder pair automatically generated by the DIC compiler. Nevertheless, this would require a great development effort and it will not allow to use the class inheritance mechanism in a systematic way. Any handcrafted framework would be bound to a specific application.

7.1 Further developments

The first scheduled improvement of the DIC compiler is aimed to further simplify the development of a distributed application. In fact, we will allow a programmer to choose if he wants to create a forwarder class with or without the suffix “Forwarder-Proxy”. In case this suffix is dropped, the forwarder should be created on a different directory with respect to the directory of the original class. Besides this subtlety, this small change will avoid to modify the source code of all the classes that use the original class. They will be ready to use the remotized version in a moment. This is feasible since Java is a dynamic language that loads all the classes of an application on demand. Therefore, if in the CLASSPATH of the client application is available only the forwarder, a JVM will load the bytecode of the forwarder instead of the byte-
code of the actual class; everything will continue to work properly provided that the DIC compiler creates a forwarder class that has the same name and the same method signatures of the original class.

In order to simplify the use of our tool, we will also modify the DIC compiler in such a way that it creates a framework that does not use Voyager. In fact, we will create a set of Java RMI classes similar to all the Voyager classes used in the forwarder and the receiver. In this way, the proxies created by the DIC compiler will work on every Java Virtual Machine without having Voyager installed on the client and server side of the application.

An issue completely neglected during this thesis is security. Therefore, we will also consider how the Java security manager can be used to enforce specific security requirements when an instance is moved locally.

Finally, besides fixing these issues, we will improve our framework with automatic class updating. We intend to extend both the forwarder and the receiver to allow at run-time the updating of the remote object’s class. We aim at creating a receiver that can reload the class of the actual object when a new version of that class is available. There is a short example in [LIB98] that is suited to our needs.

In this way, we will update the services offered by the remote object without stopping the application. The system will save the object using object serialization, load the modified class and restore the saved state. Obviously, this must not happen during a method invocation from the corresponding receiver unless we use a mechanism that allows to freeze the execution thread, save and restore it as shown in a prototype developed in [Fun98]. Nevertheless, at the moment we have not completely figured out when this updating should happen. For instance, it could happen at specified time intervals, only when the class file time stamp is changed, or on a specific request from the forwarder.

This issue will require further research in order to develop an improved version of the DIC compiler and the generated framework.

In order to reach people interested in using the DIC compiler, we will soon release this development tool in the public domain. We are confident that we will receive all the feedback required to improve it.
Appendix A

Class Diagrams in UML

In order to clarify the class diagrams shown throughout this thesis, let us see a simple example that will illustrate the Unified Modeling Language (UML) [BRJ97] notation adopted for classes and interfaces. We will show how the public, protected, private, and friendly members of a class or an interface are represented. In addition, we will see how the relationships between classes or interfaces i.e., inheritance, delegation and interface implementation, are depicted.

All the diagrams were created using JAVISION\(^1\), a Win32 program able to draw class diagrams from compiled Java classes.

\[\text{Figure A.1: A sample class diagram}\]

In Fig. A.1 are depicted three boxes representing respectively a class Example, a subclass ExampleSubClass and an interface IExample. Each box is divided into three

\(^1\)www.object-insight.com
parts. The first contains the name of the entity and its own package. The second part shows all the variables, while the third part shows the declared methods.

A far as the members are concerned, the following symbols are used to show their modifiers: “-” for private, “#” for protected, “+” for public, nothing for friendly members. In addition, all the underlined members are static.

The arrow from ExampleSubClass to SubClass represents the inheritance relationship between these two classes, the dotted arrow from Example to IExample indicates that the class Example implements the interface IExample. Finally, the line from ExampleSubClass to AssociatedClass means that ExampleSubClass declares an instance variable, named secondInstanceVariable, whose type is AssociatedClass.
Appendix B

“Hello World!” in Java IDL, RMI and Voyager.

In this appendix we will see how to remote-enable the class Message, see the application Simple in section 5.1 for the source code of this class, using all the architectures described in this thesis. In this way, the reader will be able to compare the remotized version of Simple created by the Dic compiler with the application Simple developed according to Java IDL, Java RMI and Objectspace Voyager.

B.1 Java IDL

Suppose we want to develop the application Simple, described in Chapter 5, without modifying the class Message. Since Java IDL is a CORBA implementation, it is required to write an IDL interface that describes the methods of our system.

```idl
interface RemoteMessage {
    string say(in string msg);
};
```

This interface is given in input to an idl compiler as shown below.

```
[d:\projects\javaidl]idltojava -fno-cpp RemoteMessage.idl
```

The IDL compiler will create the following Java files, as shown in the listing of the directory d:\projects|JavaIDL|.

```
  30/10/99  17.38  69  RemoteMessage.idl
  30/10/99  18.03  261 RemoteMessage.java
  30/10/99  18.03  1.822 RemoteMessageHelper.java
  30/10/99  18.03  820 RemoteMessageHolder.java
  30/10/99  18.03  1.790 _RemoteMessageImplBase.java
  30/10/99  18.03  1.081 _RemoteMessageStub.java
```
Since `RemoteMessageHolder` is not needed in this application, we will no longer consider it. The programmer has to extend the `_RemoteMessageImplBase` class in order to implement the required services.

```java
class RemoteMessageImpl extends _RemoteMessageImplBase {
    Message actualMessage;

    public String say(String msg)
    {
        return actualMessage.say(msg);
    }

    public RemoteMessageImpl()
    {
        actualMessage=new Message();
    }
}
```  // RemoteMessageImpl

The server program `MainServerIDL` that creates and registers an instance of `RemoteMessageImpl` with the naming service is shown below.

```java
// HelloServer will use the naming service.
import org.omg.CORBA.Naming.*;

// The package containing special exceptions thrown by
// the name service.
import org.omg.CORBA.NamingContextPackage.*;

// All CORBA applications need these classes.
import org.omg.CORBA.*;

public class MainServerIDL
{
    public static void main(String args[])
    {
        try{

            // Create and initialize the ORB
            ORB orb = ORB.init(args, null);

            // Create the servant and register it with the ORB
            RemoteMessageImpl remoteEnabledMessage =
                new RemoteMessageImpl();
```
orb.connect(remoteEnabledMessage);

// Get the root naming context
org.omg.CORBA.Object objRef =
    orb.resolve_initial_references("NameService");
NamingContext ncRef = NamingContextHelper.narrow(objRef);

// Bind the object reference in naming
NameComponent nc = new NameComponent("message", ",");
NameComponent path[] = {nc};
ncRef.rebind(path, remoteEnabledMessage);

// Wait for invocations from clients
java.lang.Object sync = new java.lang.Object();
synchronized (sync){
    sync.wait();
} }
} catch(Exception e) {
    System.err.println("ERROR: " + e);
    e.printStackTrace(System.out);
}
} } // MainServerIDL

Once we have compiled both RemoteMessageImpl and MainServerIDL using the standard javac compiler, we will move their bytecode class files to the directory d:\projects\JavaIDL\server. The directory will also contain the ImplBase class, our original Message class, the RemoteMessage interface generated by the IDLTOJAVA compiler.

| 30/10/99 | 17.39 | 1.538 | MainServerIDL.class |
| 30/10/99 | 17.39 | 604   | Message.class      |
| 30/10/99 | 17.39 | 215   | RemoteMessage.class|
| 30/10/99 | 17.39 | 401   | RemoteMessageImpl.class |
| 30/10/99 | 17.39 | 2.279 | _RemoteMessageImplBase.class |

Now we can develop the client application. As shown below SimpleIDL uses the naming service to obtain the reference of the remote object, narrows this reference through the Helper class, assigns the returned object to a RemoteMessage variable and calls say. Roughly, a narrow corresponds to a cast done according to the IDL neutral object model.

| import org.omg.CosNaming.*; // HelloClient will use
| // the naming service. |
| import org.omg.CORBA.*; // All CORBA applications |
| // need these classes. |
public class SimpleIDL
{
  public static void main(String args[])
  {
    try{
      RemoteException message;
      String m;

      // Create and initialize the ORB
      ORB orb = ORB.init(args, null);

      // Get the root naming context
      org.omg.CORBA.Object objRef =
          orb.resolve_initial_references("NameService");
      NamingContext ncRef =
          NamingContextHelper.narrow(objRef);

      // Resolve the object reference in naming
      NameComponent nc = new NameComponent("message", ");
      NameComponent path[] = {nc};
      message = RemoteMessageHelper.narrow(ncRef.resolve(path));

      // Call the server object and print results
      m = message.say("Hello World!");
      System.out.println(m);
    } catch(Exception e) {
      System.out.println("ERROR : "+ e);
      e.printStackTrace(System.out);
    }
  } } // SimpleIDL

The directory d:|projects|JavaIDL|client| contains the original class Message together with the Stub and the Helper classes.

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Size</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>30/10/99 17.39</td>
<td>215</td>
<td>RemoteMessage.class</td>
<td></td>
</tr>
<tr>
<td>30/10/99 17.39</td>
<td>2.072</td>
<td>RemoteMessageHelper.class</td>
<td></td>
</tr>
<tr>
<td>30/10/99 17.39</td>
<td>1.553</td>
<td>SimpleIDL.class</td>
<td></td>
</tr>
<tr>
<td>30/10/99 17.39</td>
<td>1.387</td>
<td>_RemoteMessageStub.class</td>
<td></td>
</tr>
</tbody>
</table>

Now we are ready to run the application. First of all, we have to start the TNAMESERV naming service. This service listens for naming service requests on a specific port and provides access to the named object directory it manages. Subsequently we will run the server MainServerIDL. When we run the client application SimpleIDL, the "Hello
World!” message is displayed on the server side and the other message “I have just said: Hello World!” on the client side.

<table>
<thead>
<tr>
<th>Server Side</th>
<th>Client</th>
</tr>
</thead>
</table>
| [d:\]\nameserv
Initial Naming Context:
IOR:0000000000000002849444c3a686d72e6f726
72436f7346d6966e7724e616d696e6743686c
74657874312c300000000000000000000000000000
2c00010000000000066c61786e7460047e000000
18afabcde00000002ea92e190000000800000000
0000000
TransientNameServer: setting port for initial object references to: 900
[d:\server]java MainServerIDL
Hello World! | [d:\client]java SimpleIDL
I have just said: Hello World! |

This concludes the development and execution of our “Hello World!” application as far as the Java IDL architecture is concerned.

## B.2 Java RMI

In this section we will implement the same example using Java RMI, that is, we will show how to remote-enable the class `Message.java` according to the Java RMI architecture. At the end of this section it will be clear that it is simpler to develop this application using Java RMI than Java IDL. In fact, we do not need to use an IDL interface since this architecture works exclusively with Java.

As we have done previously, the first step consists of writing the Java interface `IRemoteMessage`, shown below, that describes the services offered by the remote-enabled class.

```java
interface IRemoteMessage extends java.rmi.Remote {
    public String say(String msg) throws
        java.rmi.RemoteException ;

    } // IRemoteMessage
```

This interface must implement the interface `java.rmi.Remote`, a predefined Java interface, and has to declare all the methods that must be invokable from remote clients. Notice that each remote method must explicitly declare in its throws clause `java.rmi.RemoteException`.

Now we can write the class `RemoteMessage` that will allow to call the method `say` in the actual class `Message`. This class must extend the system class `UnicastRemoteObject` that offers the basic behavior required to export a class. It is not compelling to extend the class `UnicastRemoteObject`, it is also possible to export an object using
UnicastRemoteObject.exportObject(). Nevertheless, whatever the approach chosen, any remote object must implement at least an interface that extends IRemoteMessage. Notice that all the methods must declare RemoteException in their throws clause. This exception is thrown if there is a problem accessing a method of Message.

```java
import java.rmi.*;
import java.rmi.server.*;

public class RemoteMessage extends UnicastRemoteObject
    implements IRemoteMessage {

    Message actualMessage;

    public String say(String msg) throws RemoteException
    {
        return actualMessage.say(msg);
    }

    public RemoteMessage() throws RemoteException
    {
        actualMessage=new Message();
    }

} // RemoteMessage
```

As usual this class has an instance variable, named actualMessage, that refers to a Message object. RemoteMessage creates an instance of the class Message inside its constructor. This instance will be the actual target of all the method invocations that RemoteMessage receives from other objects running on different JVMs. In other words, when a client invokes the method say of a RemoteMessage object, in turn this method will invoke say of a Message object held in the variable actualMessage. Once we have developed IRemoteMessage and RemoteMessage, we can compile both these classes as shown below:

```
javac RemoteMessage.java IRemoteMessage.java
```

As a result we will obtain, in the current directory, the following files:

```
30/10/99   13.07    265    IRemoteMessage.class
30/10/99   12.52    491    RemoteMessage.class
```

After having compiled both RemoteMessage and IRemoteMessage, using the standard javac compiler, we must create the corresponding stub and skeleton classes required to remote-enable the class RemoteMessage. As we have described in Section 3.2, the stub class is used by the client code to communicate with the server skeleton code. The rmic compiler will create stub and skeleton using the bytecode class file RemoteMessage.class obtained from the compilation of RemoteMessage.java.

```
rmic RemoteMessage
```
The RMI compiler will create the following classes:

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Size</th>
<th>Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>30/10/99</td>
<td>12.52</td>
<td>1.746</td>
<td>RemoteMessage_Skel.class</td>
</tr>
<tr>
<td>30/10/99</td>
<td>12.52</td>
<td>3.277</td>
<td>RemoteMessage_Stub.class</td>
</tr>
</tbody>
</table>

Note that it is also possible to inspect their source code if the option -keep is specified\(^2\). Now we need a server program that creates an instance of this class and binds the object to a name stored in a lookup application, the RMIREGISTRY.

```java
import java.rmi.server.*;
import java.rmi.*;
import java.rmi.registry.*;

public class MainServerRmi {

    public static void main(String[] args) throws Exception{

        RemoteMessage remoteEnabledMessage;
        remoteEnabledMessage=new RemoteMessage();

        System.setSecurityManager(new RMISecurityManager());
        Naming.rebind("/localhost/message",
                      remoteEnabledMessage);

        System.out.println
            ("Remote Message built and bound");
    }
} // MainServer
```

As shown above, the main method creates an instance of RemoteMessage and installs an RMISecurityManager. Afterwards, a call to Naming.rebind binds the RemoteMessage object to the name “message”. Once compiled MainServerRmi.java, we can move all the files required on the server side of this application to the directory d:\projects\JavaRMI\server.

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Size</th>
<th>Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>30/10/99</td>
<td>13.07</td>
<td>265</td>
<td>IRemoteMessage.class</td>
</tr>
<tr>
<td>30/10/99</td>
<td>13.07</td>
<td>762</td>
<td>MainServerRmi.class</td>
</tr>
<tr>
<td>30/10/99</td>
<td>12.52</td>
<td>604</td>
<td>Message.class</td>
</tr>
<tr>
<td>30/10/99</td>
<td>12.52</td>
<td>491</td>
<td>RemoteMessage.class</td>
</tr>
<tr>
<td>30/10/99</td>
<td>12.52</td>
<td>1.746</td>
<td>RemoteMessage_Skel.class</td>
</tr>
<tr>
<td>30/10/99</td>
<td>12.52</td>
<td>3.277</td>
<td>RemoteMessage_Stub.class</td>
</tr>
</tbody>
</table>

The client side of this example is represented by the SimpleRmi class.

```java
import java.rmi.*;
```

\(^2\)The rmic compiler works analogously, since it creates and compiles the forwarder and receiver proxy class files from a compiled class.
public class SimpleRmi {

    public static void main(String[] args) {
        IRemoteMessage message;
        String m;
        try {
            message = (IRemoteMessage) Naming.lookup
                ("//localhost/message");
            m = message.say("Hello World!");
            System.out.println(m);
        } catch (Exception e) {
            ...
        }
    }
} // Simple

The method main of SimpleRmi attempts to locate, at the address localhost, an object registered as "message" by doing a lookup within the registry. The object that is returned is cast to IRemoteMessage and assigned to a variable. This variable will be used to invoke the method say on the RemoteMessage object.

Now we are ready to create a directory d:\projects |JavaRMI|client| containing all the files required on the client side of this application.

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>30/10/99</td>
<td>13.05</td>
<td>265</td>
</tr>
<tr>
<td>30/10/99</td>
<td>12.52</td>
<td>3.277</td>
</tr>
<tr>
<td>30/10/99</td>
<td>13.35</td>
<td>844</td>
</tr>
<tr>
<td>30/10/99</td>
<td>13.05</td>
<td>840</td>
</tr>
</tbody>
</table>

IRemoteMessage.class  RemoteMessageStub.class  SimpleRmi.class  SimpleRmi.old.class

Before starting the Mainserver, we need to start a simple naming lookup service named rmiRegistry\(^3\). The naming service must be active before Mainserver attempts to bind. Subsequently, we can run the Mainserver program and the client application.

<table>
<thead>
<tr>
<th>Server side</th>
<th>Client side</th>
</tr>
</thead>
<tbody>
<tr>
<td>[d:]start rmiregistry</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[d:...\server]java MainServerRmi</td>
</tr>
<tr>
<td></td>
<td>Remote Message built and bound</td>
</tr>
<tr>
<td></td>
<td>Hello World!</td>
</tr>
<tr>
<td></td>
<td>[d:...\client]java SimpleRmi</td>
</tr>
<tr>
<td></td>
<td>I have just said: Hello World!</td>
</tr>
</tbody>
</table>

Note that in order to simplify as much as possible the comparison, this example

\(^3\)In Java IDL the naming service is \texttt{tnameservice}.\n
does not show how to use a web-server to allow a client to dynamically download the stub. In our case the stub is still located in the client directory.

### B.3 Voyager

This section concludes our small tour of the development of the “HelloWorld!” application. This time we use Voyager to show how to create the application *Simple*. We will follow an approach that resembles what we have done previously with Java RMI and Java IDL. Nevertheless, Voyager offers, through its messaging features, another viable approach thoroughly used in the forwarder-receiver pair generated by the DIC compiler. Alas, since the messaging classes allow to create an object and invoke its methods dynamically, the JAVAC compiler cannot do all the checks at compile-time. Only at run-time will it be possible to verify whether an invoked method exists or not.

In this section we will show exclusively the more common approach. As usual we write an interface, `IRemoteMessage`, that describes all the services a remote object must offer. In this case `IRemoteMessage` does not have to extend a marker interface such as `java.rmi.Remote`.

```java
public interface IRemoteMessage{
    public String say(String msg);
} // IRemoteMessage
```

The server class `RemoteMessage` must only implement this interface. `RemoteMessage` has the usual variable that contains an instance of `Message`.

```java
public class RemoteMessage implements IRemoteMessage{
    Message actualMessage;
    
    public String say(String msg)
    {
        return actualMessage.say(msg);
    }
    
    public RemoteMessage()
    {
        actualMessage=new Message();
    }
} // RemoteMessage
```

Once compiled both classes, we can show the server program `MainServerVoyager`.

```java
import com.objectspace.voyager.*;
```
public class MainServerVoyager
{
    public static void main( String[] args ) throws Exception
    {
        Voyager.startup( "8000" ); // start up on port 8000
        ClassManager.enableResourceServer();
        // start the http server

        RemoteMessage remoteEnabledMessage;
        remoteEnabledMessage=new RemoteMessage();

        Namespace.bind( "message", remoteEnabledMessage );
        // bind to local naming service

        System.out.println( "Remote Message built and bound" );
    }
} // MainServerVoyager

This program runs the Voyager run-time support at localhost:8000, enables the built-in http server, creates an instance of RemoteMessage and binds it to a namespace (a sort of naming service) using the name “message”.

The http server is used by the clients to download the stub, a proxy according to the Voyager terminology, of a remote object. Notice that a programmer has only to compile the source code of a class when he wants to remote-enable it. An external compiler, such as RMIC in Java RMI, is not needed to create the stub and the skeleton. According to the Voyager architecture, a proxy to each remote-object is created and downloaded dynamically.

Let us create the directories containing the server and the client files of this application. As usual, the directory d:\projects\voyager\server contains all the files needed on the server side.

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Size</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>30/10/99</td>
<td>16.34</td>
<td>189</td>
<td>IRemoteMessage.class</td>
</tr>
<tr>
<td>30/10/99</td>
<td>16.34</td>
<td>821</td>
<td>MainServerVoyager.class</td>
</tr>
<tr>
<td>30/10/99</td>
<td>16.34</td>
<td>604</td>
<td>Message.class</td>
</tr>
<tr>
<td>30/10/99</td>
<td>16.34</td>
<td>409</td>
<td>RemoteMessage.class</td>
</tr>
</tbody>
</table>

On the client side only a small application called SimpleVoyager is needed. SimpleVoyager starts the Voyager run-time support and enables the class downloading from localhost:8000. In this way, a proxy representing the remote object can be downloaded from another JVM provided we have found the name “message” in the namespace.

import com.objectspace.voyager.*;
import com.objectspace.voyager.loader.*;

public class SimpleVoyager
{

public static void main( String[] args )
{
    IRemoteMessage message;
    String m;
    try
    {
        Voyager.startUp(); // start up on random port
        VoyagerClassLoader.addURLResource
        ( "http://localhost:8000/" );
        message = (IRemoteMessage)
        Namespace.lookup( "//localhost:8000/message" );
        m=message.say( "Hello World!" );
        System.out.println(m);
    }
    catch( Exception exception )
    {
        exception.printStackTrace();
    }
} } // SimpleVoyager

The directory d:\projects\voyager|client| contains exclusively the following files:

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Size</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>30/10/99</td>
<td>16.34</td>
<td>189</td>
<td>IRemoteMessage.class</td>
</tr>
<tr>
<td>30/10/99</td>
<td>16.34</td>
<td>969</td>
<td>SimpleVoyager.class</td>
</tr>
</tbody>
</table>

Now the application is ready to be run. We have only to start MainServerVoyager and SimpleVoyager. No external naming service, such as rmiregistry or tnameserv, are needed since Voyager's naming service is bundled in the run-time support.

<table>
<thead>
<tr>
<th>Server Side</th>
<th>Client Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>[d:...\server]java MainServerVoyager</td>
<td>[d:...\client]java SimpleVoyager</td>
</tr>
<tr>
<td>Remote Message built and bound</td>
<td></td>
</tr>
<tr>
<td>Hello World!</td>
<td>I have just said: Hello World!</td>
</tr>
</tbody>
</table>

B.4 Summary

In conclusion, although the development process with Java RMI is easier than the process required to develop an application with Java IDL, there are still a lot of technicalities in the Java RMI approach that do not simplify this task as expected. Fortunately, Voyager takes a step forward with respect to Java RMI and Java IDL. In fact, it simplifies enormously the development of a distributed application because there are no neutral object models such as the one described using IDL; no marker interfaces such as java.rmi.Remote that a class must implement in order to be remote-enabled; no stub (called proxy in Voyager) to create using an external compiler before running the application; no external naming service; and so on. Nevertheless, even
with Voyager a remote class is still used and instantiated differently than local classes and, obviously, this situation is even worse in all the other architectures mentioned. In fact, as we have seen in this appendix, all the flavors of the application Simple require at least to use a further package to create a remote-enabled object. Therefore, a programmer can develop a distributed application as long as he can use the classes in this package.

What's more, the applications described, whatever the architecture used, are not completely equivalent to the "HelloWorld!" example created by the Dic compiler. Instead, they implement a simpler system. In fact, the class Message remotized by the Dic compiler creates an instance of the client class (the forwarder) and one of the server class (the receiver) together with an instance of the actual class. Instead, all the described versions create only one instance of the server class whatever the number of clients. Moreover, this instance is created asynchronously by the server. We have made this choice in order to simplify, as much as possible, the comprehension of all the examples. Nevertheless, even with this precondition, it is clear that they are more difficult to implement than using the Dic compiler.

In order to have an analogous framework, a factory class would be needed on the server side, that creates and returns a new instance of RemoteMessage upon request. Since this instance is remote-enabled, the client would receive a remote reference from the remote factory. Thereafter, this reference will be used as usual to invoke the method say of a RemoteMessage object. Only in this case we will have a one to one correspondence between each client application and an instance of RemoteMessage like the pair forwarder-receiver.

Nevertheless, there are still some other differences to consider if we want to use a remote-object as if it were local. For instance, all the methods in the Message class must be declared public or friendly so that RemoteMessage can invoke them and the class RemoteMessage must declare public all the homonymous methods. Therefore, it is not possible to use the protected methods in case the programmer wants to build a sort of framework similar to the distributed hierarchy\(^4\) proposed in this thesis.

Even if a programmer were able to create a framework like the one built by the Dic compiler, this framework would be specific to the application. It is clear that the Dic compiler allows to remotize in a systematic way a class while any other hand-crafted framework would be application specific. In fact, once remotized, a class can be instantiated or extended as if it were local. On the contrary, none of the examples seen could be modified in order to support class inheritance between two JVMs.

---

\(^4\)Note that a class hierarchy where the RemoteMessage extended the Message class would not make almost any difference. Even in this case a remote client could access only the public methods.
Bibliography


Università La Sapienza
Dottorato di Ricerca in Ingegneria Informatica
Collana delle tesi
Collection of Theses


VIII-96-1 Luca Cabibbo. *Querying and Updating Complex-Object Databases*. May 1996.


XII-00-1 Marco Daniele. *Advances in Planning as Model Checking.* April 2000.


XII-00-3 Leonardo Tininini. *Querying Aggregate Data.* April 2000.