A Design Pattern for Distributed Applications

Vander Alves* Paulo Borba
Centro de Informática
Universidade Federal de Pernambuco

Abstract

Distribution has become an essential non-functional requirement of most applications. However, the design and test of distributed applications are considerably more difficult than those of monolithic systems. This tutorial presents software architectures for distributed applications, introduces a design pattern for refining these architectures, and details the implementation of this pattern using RMI, Jini, and CORBA. The architectures and the pattern presented help to tame the complexity of distributed applications and enhance their quality with respect to reuse and extendibility.

1 Introduction

Over the last decade, with the rapid growth of the Internet and the World Wide Web, distribution has become an essential non-functional requirement of most applications, spanning areas such as Electronic Commerce, Distance Learning, and Artificial Intelligence. In fact, distributed applications have a number of advantages over monolithic systems: the former are scalable, fault tolerant, and promote resource sharing.

However, despite these advantages, the design and test of distributed applications are considerably more difficult than those of monolithic systems. Indeed, problems such as non-determinism, *livelock*, and *deadlock* are easily introduced even in distributed systems containing few components. The detection of these problems is also harder since local system testing techniques do not apply effectively in the distributed scenario. Moreover, distributed applications also have other non-functional requirements such as concurrency and persistence, which make their implementation a complex task.

To support implementation, we have seen the development of language and tools for distribution. For instance, Java [5] has a distributed object model provided by RMI [8], which supports the implementation of distributed object collaborations in the language itself without the need for low-level protocols. In addition, RMI also provides a compiler tool for automatic generation of components involved in development of distributed applications.

However, language and tool support are not sufficient to avoid the problems in developing distributed applications we mentioned above. Support at a higher level is also required. In this context, the design of the software architecture [7] of such applications

*Supported by CNPq. Electronic mail: `vra@cin.ufpe.br`. Av. Professor Luis Freire s/n Cidade Universitária 50740-540 Recife PE Brazil.

is critical. The architecture describes the system's gross organization, its components, and how they interact. It helps to tame the system's inherent complexity and provides a partial blueprint for its construction. Furthermore, it promotes reusability and extendibility of the software.

In order to implement a software architecture, its components and their relationships must be refined. Patterns [4, 3] play a central role in this. By using them, programmers build software systems with predictable non-functional properties. Therefore, a pattern supporting software extendibility leads to the construction of applications where a change in a non-functional requirement, such as persistence or distribution, does not entail the need to redevelop the whole system.

This tutorial presents software architectures for distributed applications, introduces a design pattern for refining these architectures, and details the implementation of this pattern using the following technologies: RMI [8], JINI [9], and CORBA [6]. The ideas in this tutorial are based on previous research [2, 10], where structuring guidelines for distributed applications were presented. However, the main contribution of this work is to represent these guidelines abstractly as a design pattern and show how to implement this pattern using different technologies.

2 Software Architectures for Distributed Applications

Here we present software architectures for distributed systems. The architectures have three types of components: Source, Target, and Distribution components. A Target component provides services to a Source component, which accesses these services remotely by means of a Distribution component. Notice that these types are not mutually exclusive. For instance, a component can be both a Target and a Source. Figure 1 shows some architectures.

Figure 1(a) shows a three-component architecture: a user interface component as a Source, a Distribution component, and a `Bank1` component as a Target. Figure 1(b) shows a five-component architecture, where the `Bank1` component is both a Target, providing services to the user interface component, and a Source, using services provided by the `Bank2` Target component. Notice that in this case there are two Distribution components.

A Source is usually a user interface component, that is, it consists of code related to the presentation of the application (a GUI, for example). A Distribution component has code intended to provide remote access to other application's services. This component can be implemented by several technologies, each one having specific API and protocols. However, the interface exported by the Distribution component, such as `IBank1` in Figure 1(a), is generic and does not depend on a particular technology. In fact, the Source, which relies on this generic interface, is not tied to any technology. In addition, as illustrated in Figure 1(b), there can be more than one Distribution component in the architectures we are presenting. Each of these components can use a particular technology.

A Target component is usually structured according to the Facade design pattern [4], which provides a unified interface for all services of a subsystem. This, however, is not a strong restriction since Facade is a simple and well-known pattern for structuring appli-

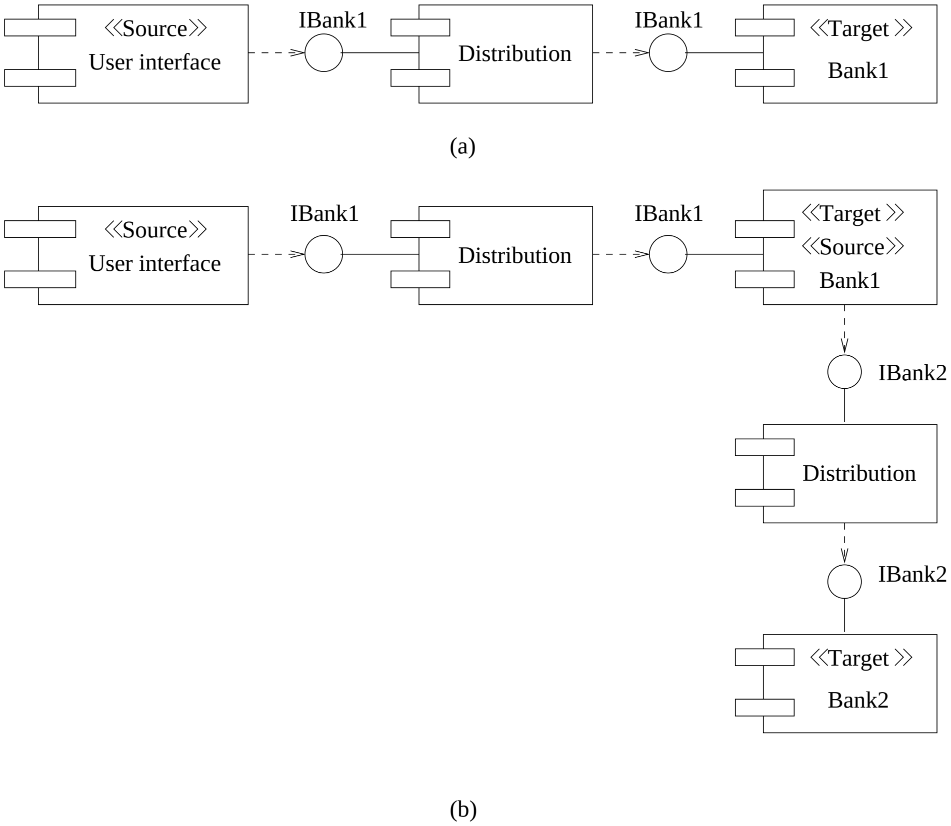

Figure 1: Software architectures for distributed applications

cations. A Facade component[1] is typically related to business aspects of an application. For instance, the Facade of a banking application keeps references to entities such as account and customer records, and has operations for manipulating these entities:

```
class Bank implements IBank {
   private AccountRecord accounts;
   Bank() throws BankInitializationException {
     ...
   }
   void addAccount(Account account)
             throws AccountAlreadyExistsException {
          accounts.add(account);
   }
   void deposit(String accountNumber, double value)
             throws UnknownAccountException {
         accounts.deposit(accountNumber,value);
   }
}
```

where `AccountRecord` provides services for manipulating a record of accounts (insertion, updating, querying, deletion, etc.) and also for depositing to or withdrawing from them. The exceptions `AccountAlreadyExistsException` and `UnknownAccountException` are

[1] Hereafter *Facade component* refers to a Facade structured Target component.

specific to the banking application. The `IBank` interface implemented by the banking facade is a *business facade interface*. It abstracts the behavior of the application:

```
interface IBank {
   void addAccount(Account account)
          throws CommunicationException,
                 AccountAlreadyExistsException;
   void deposit(String accountNumber, double value)
          throws CommunicationException,
                 UnknownAccountException;
}
```

where `CommunicationException` is a general exception representing failure in the Distribution component. It is not tied to any particular distribution technology and is defined since the application might be distributed. As we will discuss later, the facade methods do not raise this exception; that is done by methods in the Distribution components.

Figure 1 reveals essential properties of the software architectures we are presenting:

- interfaces separate the components;
- the Distribution and the Target components export the same interface.

These properties have two advantages. First, we may have flexible architectures such as the one in Figure 1(b), where a Target component depends on another remote Target. Second, the Source component is unware of distribution, since it could, in principle, depend directly on the Target itself. This may happen during the early phases of system development before distribution is introduced to the application.

The main advantage of the architectures presented here is their modularity. As a consequence, their components can be easily reused and extended. Extending distribution functionality is restricted to the Distribution component, thereby isolating others from this change. In addition, more than one distribution technology may be used to access the same Target component at the same time. Source and Target components can be reused in applications using different distributed technologies; in essence, all that has to be rewritten is the Distribution component.

Modularity also improves debugging, making it more localized, since a communication error can only arise in the Distribution component, which is where efforts should turn to fix it. Similarly, the Distribution component can be ignored when debugging business code. As a consequence, applications based on the architectures presented have improved quality.

In the following section, we detail the Distribution component of the architectures, using a design pattern [4, 3], which, in later sections, is implemented in different technologies. The pattern's goal is to introduce distribution preserving the modularity of the architectures. We illustrate it, considering a simple banking application.

3 A Design Pattern for Distributed Applications

After presenting software architectures for distributed applications, we now introduce the Distributed Adapters Pattern as a design pattern for refining the distribution component of these architectures and its relationship with other components.

Context

The Distributed Adapters Pattern is considered in the context of remote communication between two components.

Problem

In order to accomplish their tasks, components in a distributed scenario communicate with each other by means of some inter-process communication mechanism. The components may either handle communication themselves or delegate it to other components.

Although the first alternative usually requires fewer components, it leads to applications where the core functionality of its components is interwoven with communication tasks. Therefore, the application depends on a particular communication mechanism, and its components are hard to reuse and extend.

The second alternative leads to modular applications with a set of decoupled and interoperating components. Such applications are easier to maintain and to extend; its components can also be easily resued in other applications.

The Distributed Adapters Pattern balances the following forces:

- a component should be able to access remote services provided by another component;
- the components should be decoupled from the communication mechanism;
- the code modification in components to support communication should be minimized;
- changing the communication mechanism should be a simple task.

Solution

Introduce a pair of object adapters [4] to achieve better decoupling of components in the architectures presented in Section 2. The adapters basically encapsulate the code that is necessary for allowing the distributed or remote access of Target objects. In this way, the Source component of an application becomes totally independent of the Distribution component, so that changes in the latter do not impact the former.

There are two kinds of adapters: *source adapters* and *target adapters*. Roughly, the latter wraps Target objects in the places where they are located, and the former represents those objects in remote locations. In a typical interaction, a user interface object (a GUI, for instance) in one machine would request the services of a source adapter located in this machine. The source adapter would then request the services of a corresponding target adapter residing in another machine. Finally, the target adapter would request the services of a Facade object also located in the latter machine. Figure 2 illustrates this example.

Structure

Figure 3 details the structure of the Distributed Adapters Pattern. The class stereotypes denote the elements of the pattern itself, whereas the class names denote a specific implementation of these elements in a simple banking application. The Source and

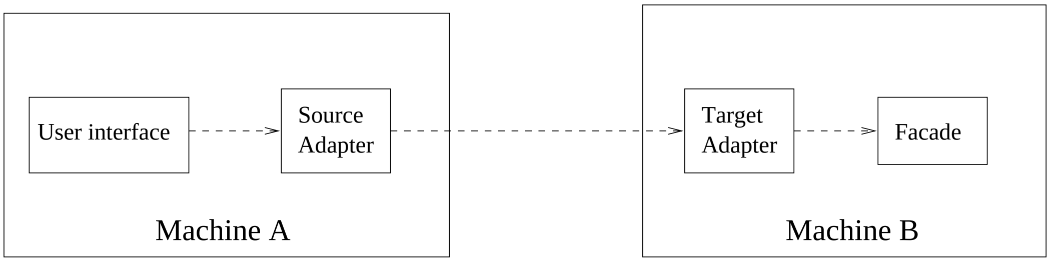

Figure 2: An example of the Distributed Adapters Pattern

Target classes abstract the Source and Target components of the software architectures in Section 2. The Target Interface is a business facade interface, abstracting the behavior of the Target class in a distributed scenario. However, this interface, the Source and Target classes have no communication code. These three elements constitute a distribution-independent layer in the pattern. The remaining elements of the pattern deal with distribution.

The core elements of the pattern handling distribution itself are the source and target adapters. These are tied to a specific distribution API and encapsulate the communication details. The source adapter is, of course, an adapter [4], isolating the Source class from distribution code. It resides on the same machine as the Source and works as proxy [4] to the target adapter. This latter may reside on another machine and is an adapter [4], isolating the Target class from distribution code. Since the target and source adapters usually reside in different machines, and thus do not interact directly, the target adapter implements a Remote Interface, on which the source adapter depends. The Name Service class provides a name service, which has operations for registering and looking up a remote object; this class is used by both adapters. The Initializer class also resides in the same machine as the target adapter and the Target class, and is responsible for creating Target and target adapter objects. Its importance lies in the fact that it allows the same Target object to be accessed at the same time by different target adapters, representing different distribution technologies.

Dynamics

Figure 4 shows the sequence diagram of a typical scenario for the Distributed Adapters Pattern applied to the simple banking application. The `BankInitializer` creates a `Bank` facade object and a target adapter, passing to the latter a reference to the former. The target adapter registers itself as a distributed object in the name service by invoking its `register` method. During initialization, the `User interface` object creates a source adapter, which performs a `lookup` operation on the name service to obtain a reference to the remote banking service, offered by the target adapter. The `User interface` object then invokes a local `deposit` operation on the source adapter, which in turn calls the remote `deposit` operation of the target adapter; this latter delegates the call locally to the `Bank` object.

Consequences

The use of the Distributed Adapters Pattern offers the following consequences:

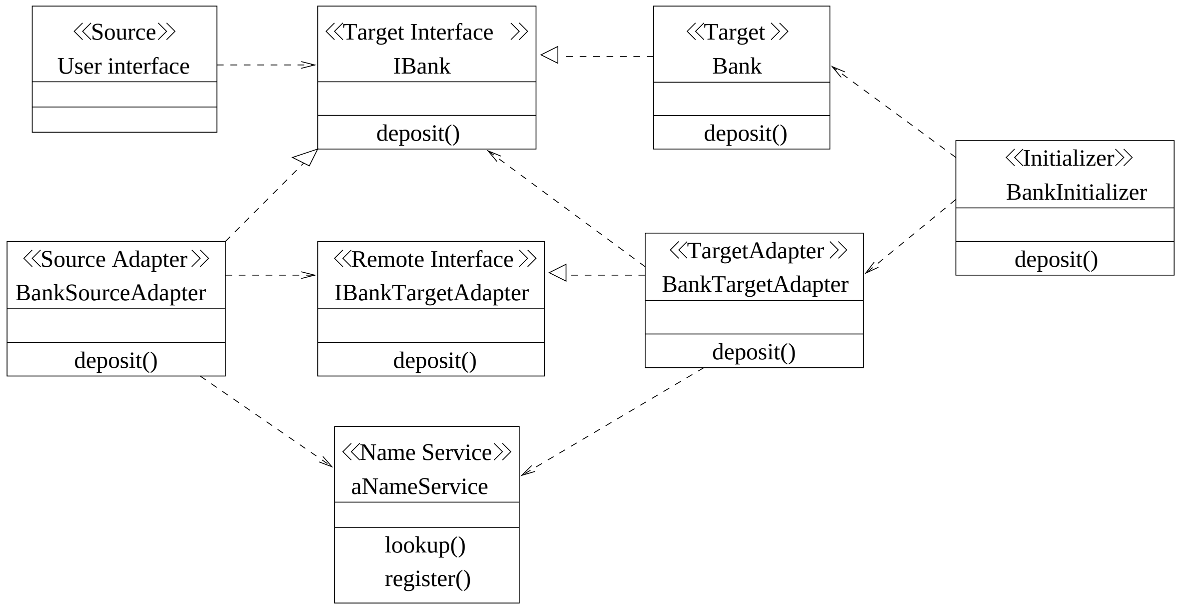

Figure 3: Structure of the Distributed Adapters Pattern

- *Modularity.* This pattern structures the distribution aspects modularly, promoting loose coupling between the Source, Target, and Distribution components of the architectures we presented in Section 2.

- *Reuse and extendibility.* Due to the modularity provided by the pattern, the Source and Target components can be easily reused in other applications based on other middleware technologies. In addition, changes to the middleware aspects are simpler, since these are restricted to the Distribution component.

- *Progressive implementation.* The pattern supports progressive implementation. During the early phases in development, a functionally complete prototype is constructed, where the Source component (a GUI, for example) depends directly on the Target component (a business Facade, for example). Later, the Distribution component is added in a seamless way, since this latter component implements the same interface as the Target.

- *Increased number of classes.* A pair of adapters is needed; however, their structure is simple and their generation could be mostly automated by tools.

- *Extra indirection.* Due to the pair of adapters, two additional calls need to be made for the same remote request. However, both of these additional calls are local, which are much less expensive than the remote one.

4 RMI implementation

In this section, we provide a description of RMI [8] and show how to implement the Distributed Adapters Pattern in this platform.

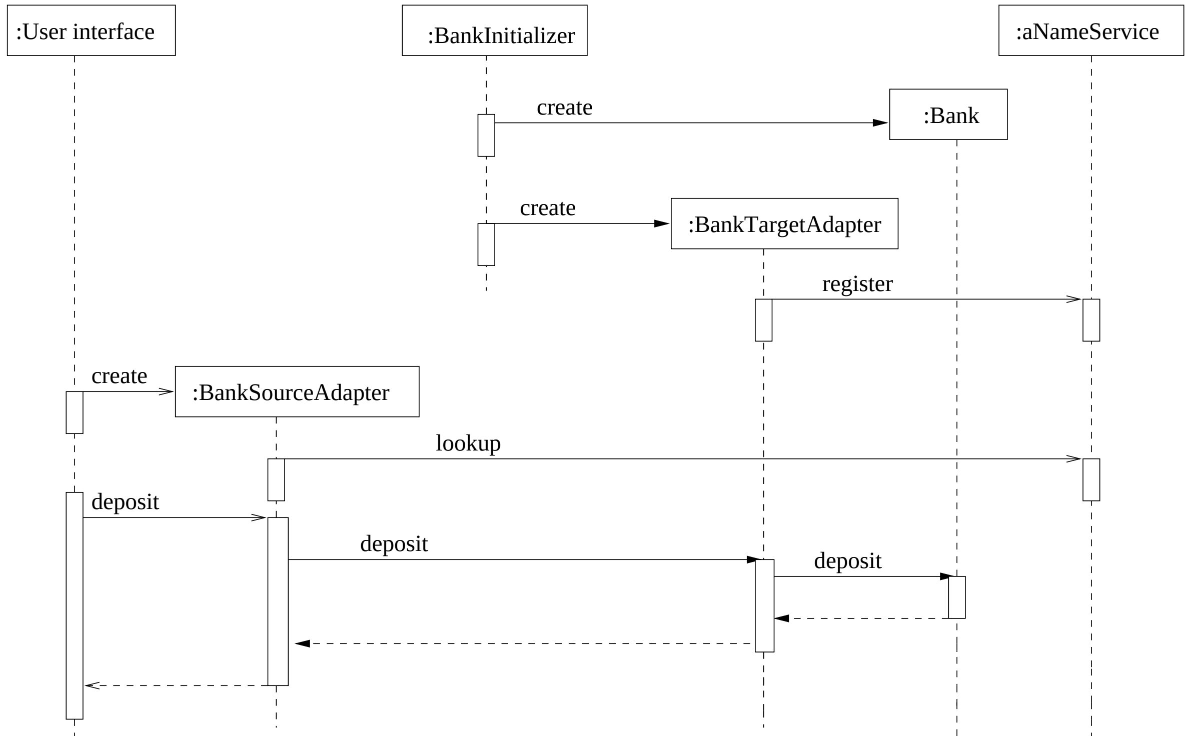

Figure 4: Dynamics of the Distributed Adapters Pattern

4.1 Platform

Remote Method Invocation (RMI) enhances Java's object model to a distributed scenario: an object in one Java Virtual Machine (JVM) can call a method on a object in another JVM, possibly located in a different host in a local or in a farther network. The syntax of invocation is the same as the local one, so that at the programming level interacting with a remote object is similar to interacting with a local object. Figure 5 shows a remote method invocation.

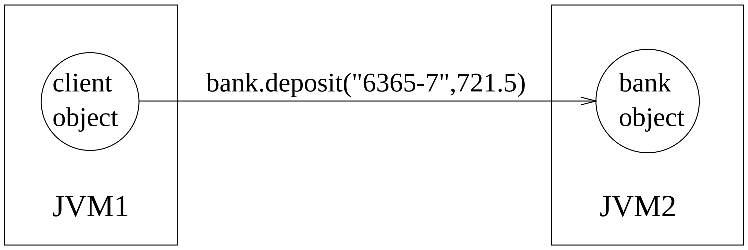

Figure 5: Remote method invocation

Distributed applications can also be developed using sockets. This, however, requires that the client and server engage in application-level protocols for coding and decoding messages, which is a low-level and an error-prone alternative. RPC systems provide the abstraction of calling a procedure remotely using the syntax of a local procedure call. Therefore, its level of abstraction is similar to RMI. RPC, however, does not provide the object abstraction that RMI does.

As mentioned, a remote method call is syntactically identical to a local method call. There are, however, semantical differences mostly due to the usual presence of a network between the client and the server. In the following, we explain RMI's architecture and programming model. To motivate the presentation, we contrast it with local method call.

We start by considering the following Java code fragment:

```
Bank bank = new Bank();
bank.deposit("6365-7",721.5);
```

Assume also that `Bank` is a local object. In these lines, a `Bank` object is created, a reference to it is assigned to the variable `bank`, and the `deposit` method is called on the newly created `Bank` object. All this happens in the same JVM.

Consider now a distributed version of this code. Object creation takes place at the server, and method invocation takes place at the client.

At the server:

```
Bank bank = new Bank();
```

At the client:

```
bank.deposit("6365-7",721.5);
```

Different from the local scenario, however, the client of the `Bank` object is located in another JVM and thereby it does not automatically receive a reference to this object. This reference must be published somewhere by the server and imported from there by the client. In addition, the semantics of object creation and remote method invocation are more elaborated.

In the following sections, details are given on how to create remote objects, register and lookup references for them, and call methods on these objects.

4.1.1 Object creation

A remote object must implement at least one *remote interface*. A remote interface specifies the methods to be called from another JVM. For the banking example, such interface may be defined as follows:

```
interface IBank extends java.rmi.Remote {
  void deposit(String accountNumber, double value) throws
         UnknownAccountException, RemoteException;
}
```

Notice that every method in a remote interface must declare `RemoteException` in its `throws` clause, in addition to any application-specific exceptions.

In addition, a remote object must also make itself available for remote calls. This includes allocating system ports where it waits for calls and be accomplished by inheriting from `UnicastRemoteObject` [8]. Finally, the constructor of a remote object must throw `RemoteException`. A remote banking object can be defined as follows:

```
class Bank extends java.rmi.server.UnicastRemoteObject
         implements IBank {
    public Bank() throws RemoteException {
```

```
        ...
    }
    void deposit(String accountNumber, double value)
          throws UnknownAccountException, RemoteException {
        ...
    }
}
```

4.1.2 References to remote objects

As mentioned earlier, a server process must explicitly export a reference to a newly created remote object so that a client in another JVM may import this reference. For this purpose, RMI provides a name service, known as *Registry*. Once created, a reference to a remote object is registered at the Registry with a name (a `String`) by the server process:

```
Bank bank = new Bank();
Naming.rebind ("BankServer", bank);
```

The `Naming` class provides access to the Registry. The client then searches the Registry for a reference named "BankServer", located in a machine named "`www.cin.ufpe.br`":

```
IBank bank = (IBank) Naming.lookup("//www.cin.ufpe.br/BankServer");
```

In the client, the type of such reference is the remote interface `IBank`: the client does not know the definition of the remote object, which is in the server. Notice that the Registry and the server process registering the remote object must run in the same machine.

4.1.3 Remote method invocation

Once a server process has created a remote object and registered it at the Registry, the client can search the Registry for a reference this object. With this reference, the client can call methods on this remote object as in the banking example:

```
bank.deposit("6365-7",721.5);
```

When a remote method is called, its parameters are transformed into a stream of bytes (a process called *marshalling*) so that they can be sent over the network to the server process. There this stream is transformed back into objects (a process called *unmarshalling*), the operation in the remote server is performed, and the result value or exception is marshalled back to the client. The elements of RMI's architecture involved in this process are the stub, the skeleton, and the transport layer [8]. The Java Serialization mechanism is also used in the marshalling process [8].

The parameters of and the result from a remote method may be any *serializable* Java object. This includes Java primitive types as in the example above, remote Java objects, and non-remote Java objects implementing the `java.io.Serializable` interface. Non-remote objects are passed by value, and remote objects are passed by reference.

In addition, the call is synchronous: the client waits until the operation gets executed at the server, and the result value or exception is sent back. Moreover, the remote object may receive concurrent calls from different clients. For this reason, the remote methods must be thread-safe.

Furthermore, since the network or the server process may be down, there is a new failure mode, which is indicated by the exception `RemoteException` being thrown by the RMI system, and which must be caught by the client:

```
try {
     bank.deposit("6365-7",721.5);
} catch (RemoteException e) {
      MessageWindow.show("Communication error. Please try later.");
  }
```

4.2 Pattern implementation

Here we consider how to implement the Distributed Adapters Pattern using RMI [8] as the distribution technology. Consider the following implementation issues:

- *Java platform.* The pattern components must be implemented in Java [5] since RMI is part of the Java platform.
- *Serialization of business objects.* As RMI supports a value parameter passing mechanism for local objects, the classes of these objects must implement the `java.io.Serializable` interface [8]. There are no methods in this interface and it simply indicates to the RMI system that an object may be transformed into a stream of bytes in order to be transmitted over a network. However, this is not a maleficent dependence between the business and the distribution layers since the former calls no method on the latter; in fact, no change on the latter will affect the former.

We now provide sample code for the elements in the pattern, using the simple banking application as an example. The `Bank` class and the `IBank` interface were sketched in Section 2. The `User interface` class would simply create a `BankRMISourceAdapter` and forward client requests to it. The RMI source adapter implements `IBank` so that the `User interface` class is unaware of the specific middleware technology. The constructor obtains a reference to the target adapter, by invoking the `connect()` method:

```
public class BankRMISourceAdapter implements IBank {
   private IBankRMITargetAdapter  bank;
   public BankRMISourceAdapter() throws CommunicationException {
      connect();
   }
   public void connect() throws CommunicationException {
      try {
           bank = (IBankRMITargetAdapter)
                     Naming.lookup("//www.cin.ufpe.br/BankServer");
      } catch (Exception e) {
          throw new CommunicationException (...);
        }
   }
```

The `connect()` method can be called later in case the connection with the target adapter fails. The `deposit` method forwards `User interface` deposit requests to the target adapter:

```
   public void deposit (String accountNumber, double value)
                throws CommunicationException,
                       UnknownAccountException {
      try {
          bank.deposit(accountNumber,value);
      } catch (RemoteException e) {
           throw new CommunicationException (...);
        }
   }
} //end of BankRMISourceAdapter
```

Note that, both in the constructor and in the deposit methods, the source adapter replaces the RMI specific RemoteException with the general CommunicationException.

The IBankRMITargetAdapter interface is the type of the reference to the target adapter and its methods must raise RemoteException:

```
public interface IBankRMITargetAdapter extends Remote {
   void deposit(String accountNumber, double value)
          throws CommunicationException, UnknownAccountException,
                 RemoteException;
}
```

where Remote is an RMI interface used to identify remote object types [8].

The target adapter becomes an RMI remote object by inheriting from UnicastRemoteObject [8]. It implements the remote interface IBankRMITargetAdapter, so that the source adapter can call its methods remotely. The constructor of the target adapter receives a facade object as a parameter and registers the adapter itself in the name service:

```
public class BankRMITargetAdapter extends UnicastRemoteObject
          implements IBankRMITargetAdapter {
   private IBank bank;
   public BankRMITargetAdapter(IBank bank)
              throws RemoteException, MalformedURLException {
       this.bank = bank;
       Naming.rebind("BankServer", this);
   }
```

The deposit method is invoked remotely by the source adapter, and this operation forwards the call to the corresponding method in the facade object:

```
   public void deposit(String accountNumber, double value)
                throws CommunicationException, RemoteException,
                       UnknownAccountException {
      bank.deposit(accountNumber, value);
   }
} // end of BankRMITargetAdapter
```

The BankInitializer performs some platform initialization procedures, creates a Bank facade object and a target adapter, passing to the latter a reference to the former:

```
public class BankInitializer {
   public static void main(String[] args) {
      ...
      IBank bank = new Bank();
      BankRMITargetAdapter targetAdapter =
                  new BankRMITargetAdapter(bank);
   }
}
```

5 Jini Implementation

We now describe Jini [9] and the implementation of the Distributed Adapters Pattern in this plataform.

5.1 Platform

As mentioned earlier, distribution has become a usual requirement in most applications developed nowadays, being present in diverse domains such as Electronic Commerce, Distance Learning, Multimedia, and Artificial Intelligence. In the enterprise level, there are distributed information systems. There is, however, a considerable diversity of electronic devices such as Personal Digital Assistant, cellular phones, home appliances and others with the potential of constituting distributed systems. The idea of connecting these devices and enterprise systems, forming a network of services, is subject of intense current research and is the context from which the Jini distribution platform emerged.

In this context, Jini's design requirements are as follows:

- *Simplicity.* Not only in programming language design, but also in distribution platform design is simplicity a desirable property. In Jini, the fundamental abstraction is that of a service. Services have interfaces describing them and connect to form communities.

- *Reliability.* Jini's communities are often dynamic: services enter and leave these communities frequently. To keep these communities in a consistent state, the platform must be reliable. In addition, there should also be a minimum of configuration necessary to achieve this.

- *Scalability.* The platform must accommodate both small and large number of services. This is related to the concept of federation and requires support for incremental maintenance of services.

- *Device agnosticism.* Different kinds of components connect to a Jini community. These may be software, hardware, or a combination of both. To use a service, its interface is the only information a client needs.

The requirements above are not all met by other platforms. RMI, for instance, is not scalable since it requires that the name server (Registry) and the remote object be in the same machine; in addition, a service is searched for by a `String` name of a specific object implementation rather than by an interface describing its functionality. Therefore, if there are multiple objects implementing the same service, they must be registered with different names at the Registry, and the client will have to know these names, though it may only need to know the interface of a service. Another drawback of RMI is that there is an implicit order for starting components: first the server, then the client, which is another required item of configuration. Finally, RMI is also not reliable, since a service remains registered indefinitely at the Registry, even though the service itself may crash after registration.

Jini is a Java-based distribution platform aiming at simplicity, reliability, scalability, and device agnosticism. It is suitable for building dynamic networks of services where a minimum of configuration is necessary. In the following, we examine its architecture and programming model.

5.1.1 Architecture

According to its design goals, Jini's architecture is simple. Indeed, there is only one fundamental abstraction, that of a service. In this section, we describe this abstraction and a specific kind of service, the name service.

A service is Jini's central concept. A service may be software, hardware, or a combination of both (according to the design goal of device agnosticism). It implements an interface describing its behavior. This interface is required of every service by the platform, since it is the point of interaction between the service and its clients. The implementation of a service, however, is only known to itself. More specifically, a service is described by three elements: an identifier, a proxy, and attributes. The identifier is assigned to the service when it starts; the proxy is a mobile entity which represents the service at the client, implements the interface describing the service, and isolates the communication protocol with the backend server from the client. The attributes provide additional description of the service (location, status, GUI and others). These three elements are known as a `ServiceItem`.

Services associate themselves to form *communities* (also known as *groups*). A community is a logical entity represented by a `String` and reflects either the physical or the organizational structure of its services. For example, at the physical level, services in a local network may form a community named "siteA"; at the organizational level, services in the marketing and management departments may be grouped in communities named "marketing" and "management". Moreover, communities may be connected to form a *federation* (according to the design goal of scalability).

Within a community, services interact with one another either as clients or servers. To support this interaction, they must get references to themselves, which is accomplished with support from a special service, the name service, known in Jini as the Lookup Service. This basic service defines the available services in a Jini community, providing operations for service search and service registration. Before invoking these operations, however, a new service joining an existing community must get a reference to a Lookup Service in that community. This process is defined by the *Discovery* protocol [9]. In addition, the Lookup Service also provides an administrative interface and plays a key role in federation of Jini communities.

Before using the Lookup Service, a new service must get a reference to it by means of the *Discovery* protocol. In this protocol, this new service does not need to know the location of the Lookup Service, which means that the client needs no prior configuration to find the name service. The protocol is asynchronous and its implementation uses UDP multicast to search a reference to a Lookup Service in the local network. When it is found, a remote notification is sent back to the new service. The following code fragment illustrates the use of this protocol:

```
class Service {
   ...
   class Listener implements DiscoveryListener {
      public void discovered(DiscoveryEvent ev) {
         ServiceRegistrar[] newregs = ev.getRegistrars();
         ...
      }
     ...
   }
...
LookupDiscovery disco = new LookupDiscovery(new String[] {"public"});
disco.addDiscoveryListener(new Listener());
...
}
```

The protocol is implemented by the class `LookupDiscovery`. When the `disco` instance of this class is created, an asynchronous search for a Lookup Service of the community named "public" begins in the local network. Then a `Listener` object is registered with `disco` so that when a Lookup Service is found, the `Listener`'s `discovered` method is invoked to get a reference to this service (Jini's Lookup Service implements the `ServiceRegistrar` interface).

Once a new service has a reference to the Lookup Service, it can register itself by invoking the Lookup Service's `register` method:

```
ServiceItem item = new ServiceItem(null, createProxy(), null);
ServiceRegistration sr = lookupService.register(item, LEASE_TIME);
```

The `item` argument represents the service, providing its identifier, proxy and attributes (only the second is not null in the example). The second argument is a constant defining for how long the registration is valid. The `sr` object is a record of the registration and may be used for renewing it later. In general, when registering in a Lookup Service, a service also follows a set of conventions, known as the *Join* protocol [9]. These conventions state that the service must renew its registration regularly and keep its attribute and identifier description consistent with every Lookup Service it registers itself with.

A service in a community may also look for another service. This can be done by invoking the Lookup Service's `lookup` method:

```
ServiceTemplate template = new ServiceTemplate(null, types, null);
Object serviceProxy = lookupService.lookup(template);
```

The `template` object specifies the service's identifier, the interfaces implemented by it, and its attributes (in the example only the interfaces argument is not null). The result of the search is the service proxy. Figure 6 illustrates interaction with the Lookup Service.

5.1.2 Programming model

There are three basic elements in Jini's programming model: leasing, distributed events, and distributed transactions.

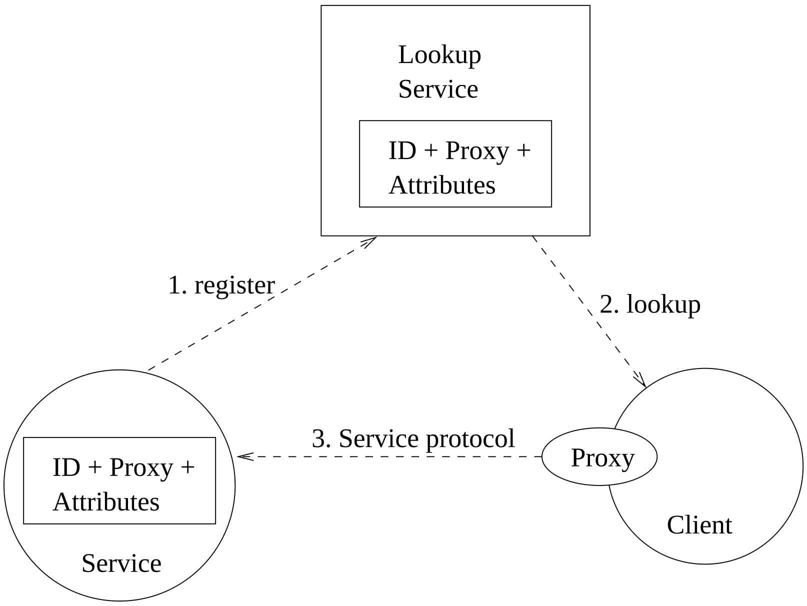

Figure 6: Interaction with the Lookup Service

Leasing

The concept of Leasing is that access to a resource is valid only for a period of time. This period is negotiable between the entities involved and, after it expires, may be renewed. According to Jini's design goal of reliability, leasing is present in this platform's programming model to acknowledge failure in distributed systems and to attempt automatic recovery of consistency after failure. Therefore, if the client of a resource crashes, this resource will not remain allocated indefinitely to this client (an inconsistency), but only until the lease to this client expires.

The participants in the leasing model are a resource, the lease grantor, and the lease holder. A resource is an abstract concept and may be memory, computation, event notification record among others; a lease grantor is the entity serving the resource, and the lease holder is the client of the resource. For example, the lease holder could be an incoming service in a community, the lease grantor could be the Lookup Service, and the resource could be memory for service registration at the Lookup Service. In the following code, the incoming service registers itself in the Lookup Service for a period of time and later requests renewal of its Lease:

```
ServiceItem item = new ServiceItem(null, createProxy(), null);
ServiceRegistration sr = lookupService.register(item, LEASE_TIME);
...

Lease lease = sr.getLease();
try {
     lease.renew(LEASE_TIME);
} catch (LeaseDeniedException e) {...}
  catch (UnknownLeaseException e) {...}
  catch (RemoteException e ) {...}
```

As seen earlier, in the first two lines of the code, the service registers itself in the Lookup

Service. The `sr` object is a record of this registration and holds a `Lease` object representing the memory lease. Later, when the lease is to expire, the service attempts to renew it by calling its `renew` method. The Lookup Service may deny the renewal, in which case a `LeaseDeniedException` is thrown; in the case the lease has already expired, `UnknownLeaseException` is thrown; if there is a communication error with the Lookup Service, `RemoteException` is thrown.

In the example above, if the service did not renew its lease with the Lookup Service, this latter would remove the former from the registered services, and, as a result, no other service in the community would notice the existence of the former service. The service could deliberately do this in the case it would undergo maintenance, showing how Leasing also supports the important issue of Software Evolution in distributed systems.

Distributed events

An event is an asynchronous notification of an external state change. A usual pattern for using events is to register interest for an event passing a listener as a parameter. When the event happens, the listener handles it. Event programming is flexible, modular, and efficient.

Jini's events are remote. Due to distribution, there may be delay in their arrival, they may arrive out of order, or they may not arrive at all. In fact, according to Jini's design goal of simplicity, there is only one kind of distributed event, represented by the `RemoteEvent` class. The protocol for event registration is defined by each service, whereas event notification is uniform: the listener's `notify` method is called when an event happens. In addition, this notification is accomplished by using RMI. The code below shows a service registering for event notification with a Lookup Service:

```
class client {
   class EventListener extends UnicastRemoteObject
            implements RemoteEventListener {
      public EventListener() throws RemoteException {
      }
      // invoked when the event is received
      public void notify(RemoteEvent ev)
            throws RemoteException, UnknownEventException {
           if (ev instanceof ServiceEvent) {
              ServiceEvent sev = (ServiceEvent) ev;
              ServiceItem item = sev.getServiceItem();
               ...
           }
      }
   }
   ...
   EventRegistration er = lookupService.notify(template,
      ServiceRegistrar.TRANSITION_NOMATCH_MATCH, new EventListener(),
      null, LEASE_TIME);
   ...
}
```

The service registers interest in a Lookup Service's event by calling its `notify` method. The event is to be sent to an `EventListener` when a service, specified by `template`, appears in the Lookup Service (condition defined by the `TRANSITION_NOMATCH_MATCH` argument). The service also requests that the event registration be valid for a `LEASE_TIME` period of time. When the `RemoteEvent` is received by the `EventListener` object, its `notify` method is invoked, and the description of the searched service is obtained.

Transactions

A transaction is a group of operations that are executed atomically. In Jini, transactions are distributed, which means these operations are called on different services, probably in different machines. The players in Jini's transactions are services known as client, participant, and manager. The client creates a transaction, call participant's operations under this transaction, and then requests the execution of the transaction. This execution is orchestrated by the manager, which implements a two-phase commit protocol (2PC) [9].

5.2 Pattern implementation

After describing the platform itself, we now consider how to implement the Distributed Adapters Pattern using Jini [9] as the distribution technology. The following are implementation issues to consider:

- *Java platform.* The pattern components must be implemented in Java [5] since Jini is built on the Java platform.
- *Target Adapter refinement.* The Target Adapter element in the pattern is implemented as two classes, JiniTargetAdapter and JiniTargetAdapterProxy. The latter is a mobile entity, moving from the server side to the client side, and uses a specific communication protocol with the former, isolating this protocol from the Source Adapter. This is different from the RMI implementation, since its Source Adapter is tied to a specific communication protocol when interacting with the RMI Target Adapter. In Jini, the source adapter interacts with the proxy and this latter interacts with the target adapter.
- *Backend communication protocol.* We use RMI as the communication protocol between JiniTargetAdapterProxy and JiniTargetAdapter. Although this is a common choice, others protocols could be used such as CORBA's IIOP [6]. If another protocol were chosen, only the JiniTargetAdapterProxy and the JiniTargetAdapter would need to change, since they encapsulate the protocol; the Jini Source Adapter would remain unchanged.
- *The exception* `JiniCommunicationException` is the general communication exeception within Jini. Such general exception exists since JiniTargetAdapterProxy and JiniTargetAdapter may use whatever communication protocol they wish, whereas the Jini Source Adapter only deals with `JiniCommunicationException`.
- *Proxy and business objects serialization.* The JiniTargetAdapterProxy and other business objects must implement `java.io.Serializable` since these components are usually transmitted over a network.

In the following, we provide sample code for the elements in the pattern. We consider only the source and target adapters, the Initializer, and the Name Service since these are the ones involved with distribution itself. Again, we use the simple banking application as an example.

The Jini source adapter implements `IBank` in order to isolate the `User interface` class from the communication details. The constructor defines a template specifying the type of the target adapter proxy with which the source adapter will interact, and calls the `connect` method to obtain a reference to Jini's name service:

```
public class BankJiniSourceAdapter implements IBank {
   private LookupDiscovery disco;
   private ServiceTemplate template;
   private IBankJiniTargetAdapterProxy targetAdapterProxy;
   public BankJiniSourceAdapter() throws CommunicationException {
           Class[] types = {IBankJiniTargetAdapterProxy.class};
           template = new ServiceTemplate(null, types, null);
           connect();
   }
```

The `connect` method starts the Discovery protocol [9], which we also described in Subsection 5.1, to obtain a reference to Jini's name service, the Lookup Service. Since the search for this service is asynchronous, the `connect` method also registers a `Listener` object to handle the event triggered when the service is found:

```
public void connect() throws CommunicationException {
   try {
        disco = new LookupDiscovery(new String[]{ "" });
        disco.addDiscoveryListener(new Listener());
   } catch (Exception e) {
       throw new CommunicationException (...);
     }
}
```

The `discovered` method of the `Listener` inner class is called when a Lookup Service is found; in this method, the `lookup` operation is invoked on a lookup service, which implements the `ServiceRegistrar` interface, to search it for the target adapter proxy:

```
class Listener implements DiscoveryListener {
   public void discovered(DiscoveryEvent ev) {
      ServiceRegistrar[] lookupServices = ev.getRegistrars();
      targetAdapterProxy = (IBankJiniTargetAdapterProxy)
                            lookupServices[0].lookup(template);
   }
   ...
}
```

The `deposit` operation forwards calls to the Target Adapter proxy, replacing `JiniCommunicationException` with `CommunicationException`, since the source adapter implements `IBank`:

```
public void deposit(String accountNumber, double value)
      throws CommunicationException, UnknownAccountException {
   if (targetAdapterProxy == null)
```

```
            throw new CommunicationException(...);
         try {
              targetAdapterProxy.deposit(accountNumber,value);
         } catch (JiniCommunicationException  e) {
            throw new CommunicationException(...);
           }
     }
  } // end of BankJiniSourceAdapter
```

Notice that the `if` command is necessary, as the Discovery protocol may not have found a Lookup Service. This is not necessary in the RMI source adapter since the name service, the Registry, is already present in the source adapter. Note also that in the RMI implementation the source adapter forwards calls directly to the target adapter.

As we mentioned before, the target adapter in jini is implemented by two classes, JiniTargetAdapter and JiniTargetAdapterProxy. In this example, `BankJiniTargetAdapterProxy` interacts with `BankJiniTargetAdapter` using RMI as the communication protocol[2]. Therefore, the proxy keeps a reference to the target adapter; this reference is typed by the remote interface `IBankJiniTargetAdapter`:

```
  public class BankJiniTargetAdapterProxy implements Serializable,
                IBankJiniTargetAdapterProxy {
     private IBankJiniTargetAdapter targetAdapter;
     public BankJiniTargetAdapterProxy(IBankJiniTargetAdapter target) {
         targetAdapter = target;
     }
```

The source adapter invokes the proxy's `deposit` method, and this latter redirects the call to the target adapter; since the communication between the proxy and the target adapter uses RMI, this redirection is simply a remote method call:

```
  public void deposit(String accountNumber, double value)
           throws JiniCommunicationException, CommunicationException,
                  UnknownAccountException {
        try {
            targetAdapter.deposit(accountNumber,value);
        } catch (RemoteException e) {
             throw new JiniCommunicationException(...);
          }
  }
} // end of BankJiniTargetAdapterProxy
```

Notice that the proxy replaces `RemoteException` with `JiniCommunicationException`, as it hides the specific communication protocol from the source adapter.

Since the proxy uses RMI to communicate with the Jini target adapter, this latter is implemented as a remote object by extending `UnicastRemoteObject`. The constructor receives a `Bank` facade object as a parameter, starts the search for the Lookup Service by means of the Discovery protocol, and registers a `Listener` object to be invoked when this service is found:

[2] As mentioned before, other protocols may be used.

```
public class BankJiniTargetAdapter extends UnicastRemoteObject
         implements IBankJiniTargetAdapter {
   public static final int LEASE_TIME = 10000;
   private IBank bank;
   private ServiceItem     proxy;
   private LookupDiscovery disco;
   public BankJiniTargetAdapter(IBank bank)
            throws IOException, RemoteException {
      this.bank = bank;
      disco = new LookupDiscovery(new String[]{ "" });
      disco.addDiscoveryListener(new Listener());
   }
```

The Discovery protocol calls the `discovered` method of the `Listener` inner class when it finds a Lookup Service. This method creates a proxy and registers it in the Lookup Service:

```
   class Listener implements DiscoveryListener {
      public void discovered(DiscoveryEvent ev) {
         ServiceRegistrar[] lookupServices = ev.getRegistrars();
         proxy = new ServiceItem(null,
                       new BankJiniTargetAdapterProxy(this), null);
         lookupServices[0].register(proxy, LEASE_TIME);
      }
      ...
   }
```

The method above registers the proxy at the Lookup Service for a `LEASE_TIME` period. We do not renew the registration here. However, that could be done by subclassing the adapter.

The proxy invokes the `deposit` method remotely, and this operation delegates the call to the local `Bank` facade object:

```
   public void deposit(String accountNumber, double value)
            throws CommunicationException, RemoteException,
                   UnknownAccountException {
       bank.deposit(accountNumber,value);
   }
} // end of BankJiniTargetAdapter
```

As in the RMI implementation, the Jini `BankInitializer` also performs some platform initialization procedures; it then creates a facade object and a target adapter; this latter obtains a reference to the former:

```
public class BankInitializer {
   public static void main(String[] args) {
      ...
      IBank bank = new Bank();
      BankJiniTargetAdapter targetAdapter =
                                new BankJiniTargetAdapter(bank);
   }
}
```

6 CORBA implementation

In this section, we give an overview of CORBA [6] and illustrate how the Distributed Adapters Pattern can be implemented in this plataform.

6.1 Platform

Common Object Request Broker Architecture (CORBA) is a standard architecture of object-based distributed systems. It is a specification rather than an implementation and is defined by the Object Management Group (OMG), which is a consortium of 800 companies in the distributed objects field, including application and tool developers. The OMG is also responsible for standardizing the UML [1]. One advantage of being a specification defined by the OMG is that CORBA is vendor independent; this is in contrast with others platforms such as DCOM, which is defined exclusively by Microsoft.

Different CORBA implementations exist. Some examples: JavaIDL (comes with JDK 1.2 and later), VisiBroker, ORBacus. Different from RMI and Jini, which are Java based and thus interoperable with respect to operating system, computer architecture, and network, CORBA implementations are also interoperable with programming languages. For instance, a client of CORBA service may be written in C++, and the service implementation in Java.

In the following sections, we outline CORBA's architecture and programming model.

Architecture

The basic CORBA paradigm is that of service request to a distributed object. To support this paradigm, CORBA's architecture has these design goals: abstraction, location and access transparency, and interoperability.

The idea of abstraction is that a service in CORBA is specified in a neutral language, the Interface Definition Language (IDL), whereas service implementation is specified in a language such as Java or C++, for example. Therefore, the architecture abstracts implementation.

Location and access transparency means that a service is invoked without knowledge of its location and that this invocation is syntactically identical to that of a local service. This is implemented by the ORB, which locates the remote object, communicates the request to this object, waits for results, and sends this result back to the client. The ORB's structure is shown in Figure 7.

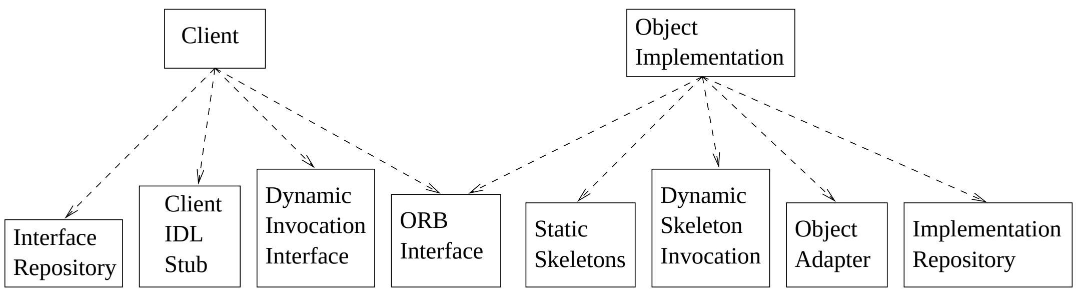

Figure 7: Structure of the ORB

On the client side of the ORB, there are the following components:

- ORB interface. It is an API to local services. For instance, conversion of object references to string and vice-versa.
- Client IDL stub. A Proxy to the remote object, supporting static invocation of its methods.
- Dynamic Invocation Interface (DII). Allows method calls to be constructed at run-time.
- Interface repository. It is a run-time database containing the interface specifications of each object the ORB recognizes.

On the server side of the ORB, there are the following components:

- ORB interface. It has the same function as on the client side.
- Static Skeleton. Static interface to the service exported by the server. Corresponds to the stub on the client.
- Dynamic Skeleton Invocation. Corresponds to the DII on the client, being used in the absence of a static skeleton.
- Object Adapter. Gives to the client the illusion that server objects are active all the time.
- Implementation repository. Additional information about the implementation of the ORB.

As mentioned, interoperability in CORBA happens at the levels of operating system, network, computer architecture, and programming language. To provide this, there are in CORBA a network protocol (IIOP), message formats, and common data representations for communication between ORBs (GIOP). With the standard CORBA2.0, implementations must support GIOP over IIOP.

The macro architecture of CORBA applications, known as Object Management Architecture, is shown in Figure 8.

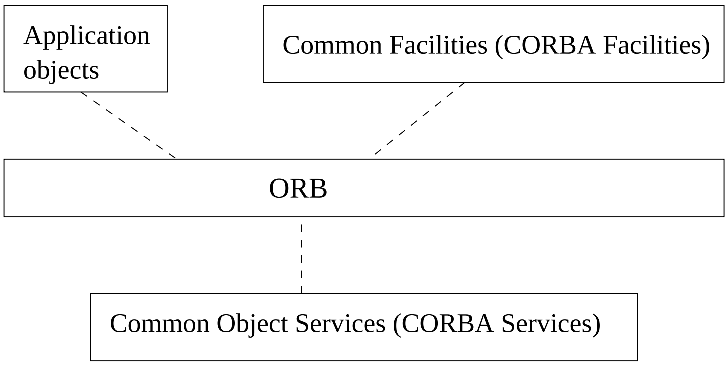

Figure 8: Object Management Architecture

Its elements are as follows:

- ORB. This was described earlier.

- Application objects. Services and clients introduced by the application developer.
- CORBA services. System level services that complement the ORB's functionality. Examples: name, event, and trader services. Many CORBA services are specified but not implemented yet.
- CORBA facilities. High level services useful in many or in specific domains. There are facilities for mobile agents, firewalls, and internationalization.

Programming model

The elements of CORBA's programming model are the IDL, the client of a service, the remote object implementing the service, and the server process.

The Interface Definition Language (IDL) is a neutral language for specification of services and thus is fundamental for language interoperability in CORBA. There are bindings from IDL to C, C++, Java, Ada, COBOL, Smalltalk, and Lisp. Since IDL is a specification language, it does not provide procedural structures; instead, it offers constructs for describing modules, interfaces, methods, exceptions, and data types. The following is an IDL specification of a simple banking application:

```
module BankCorbaApp {
   exception CorbaCommunicationException { };
   exception CorbaUnknownAccountException {
      string account;
   };
   valueType Account {
      void deposit(in double value);
      state {
              string number;
              double credit;
             };
   };
   interface BankCorba {
      void deposit(in string accountNumber, in double value)
              raises (CorbaCommunicationException,
                      CorbaUnknownAccountException);
      Account getAccount( in string number)
                raises (CorbaCommunicationException,
                        CorbaUnknownAccountException);
   };
};
```

The client of a service may invoke it either dynamically with DII or statically with IDL stubs. In dynamic invocation, the client constructs service calls at run time and does not need a stub. This programming style is flexible, but more complex, and usually an order of magnitude slower than static invocation. The latter needs a precompiled stub (obtained by compilation of the service IDL specification), is less complex and is the most commonly used style.

In static invocation, the client initializes the ORB, uses the name service to obtain a reference to a remote object, and interacts with this object by means of this interface. The following code shows these steps:

```
class Client {
   BankCorba bank;
   public Client() {
      ORB orb = ORB.init(args, null);
      org.omg.CORBA.Object objRef =
                  orb.resolve_initial_references("NameService");
      NamingContext nameService = NamingContextHelper.narrow(objRef);
      NameComponent nc = new NameComponent("BankServer", "");
      NameComponent [] path = {nc};
      bank  = BankCorbaHelper.narrow(nameService.resolve(path));
   }
   ...
   bank.deposit(accountNumber,value);
   ...
}
```

The remote object may also be implemented to support either static or dynamic invocation. As mentioned earlier, the former is easier and more efficient, thus the most used style. With static invocation, the remote object implementing the service must extend the skeleton, wich is also generated when compiling the service IDL specification. The code fragment below is an example of a remote object implementation:

```
class  BankCorbaImpl extends _BankCorbaImplBase {
   public void deposit(String accountNumber, double value)
                 throws CorbaCommunicationException,
                        CorbaUnknownAccountException {
   }
}
```

Lastly, the server process initializes the ORB, creates a remote object, connects this object to the ORB, registers the remote object at the name service, and waits for invocation:

```
class Server {
   public static void main (String args[]) {
      ORB orb = ORB.init(args, null);
      BankCorbaImpl bank = new BankCorbaImpl();
      orb.connect(bank);
      org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
      NamingContext nameService = NamingContextHelper.narrow(objRef);
      NameComponent nc = new NameComponent("BankServer","");
      NameComponent [] path = {nc};
      nameService.rebind(path, bank);
      Thread.currentThread().join();
   }
}
```

6.2 Pattern implementation

Here are some issues to consider when implementing the Distributed Adapters Pattern using CORBA as the distribution technology:

- *Interoperability.* As CORBA provides not only operating system and network independence but also programming language independence, client and server side components may be implemented in different languages.

- *CORBA user exceptions.* Every business and communication exception must have a corresponding CORBA user exception, since these exceptions must be sent over CORBA's IIOP transport protocol [6]. These exceptions are defined in the service IDL specification and are created by the IDL compiler [6].

- *Value types.* Objects passed by value must be defined in the service IDL specification as value types. In addition, since code is not downloaded, it is assumed that there is a semantic agreement between client and server about the behavior of objects passed by value [6].

Using the banking application, we now provide sample code to illustrate the implementation of the pattern in this distribution platform. The essential elements of the pattern to consider are the source and target adapters, the Initializer, and the Name Service.

Like the source adapters in RMI and Jini, the CORBA source adapter implements `IBank` so that the `User interface` class is unware of communication details. The constructor initializes the `ORB` and calls the `connect` method, which obtains a reference to the name service, and searchs it for the banking service:

```
public class BankCorbaSourceAdapter implements IBank {
   private IBankCorbaTargetAdapter bank;
   private ORB orb;
   BankCorbaSourceAdapter(String[] args)
            throws CommunicationException {
        orb = ORB.init(args, null);
        connect();
   }
   public void connect() throws CommunicationException {
      try {
           NamingContext nameService = NamingContextHelper.narrow (
                   orb.resolve_initial_references("NameService"));
           NameComponent nc = new NameComponent("BankServer", "");
           NameComponent [] path = {nc};
           bank  = IBankCorbaTargetAdapterHelper.narrow (
                          nameService.resolve(path));
      } catch (Exception e) {
          throw new CommunicationException (...);
        }
   }

}
```

The deposit method forwards calls from the User interface class to the remote Bank object:

```
   public void deposit(String accountNumber, double value)
                throws CommunicationException, UnknownAccountException {
      try {
          bank.deposit(accountNumber,value);
      } catch (CorbaCommunicationException e) {
           throw new CommunicationException();
        }
      catch (CorbaUnknownAccountException e) {
        throw new UnknownAccountException(e.getMessage());
      }
   }
} // end of BankCorbaSourceAdapter
```

Note that the adapter replaces CorbaCommunicationException and CorbaUnknownAccountException with CommunicationException and UnknownAccountException, since this adapter implements IBank. Notice that this is different from the RMI and Jini source adapters; these do not need to handle business exception such as UnknownAccountException because all components are implemented in Java and this exception has the same implementation in all of them; however, with CORBA, there must be a language independent representation of the exception: CorbaUnknownAccountException.

The CORBA target adapter becomes a remote object by extending the _BankCorbaTargetAdapterImplBase skeleton generated by the IDL compiler. The constructor receives a Bank facade object as a parameter, initializes the ORB, and registers this object at the ORB and in the name service:

```
   public class BankCorbaTargetAdapter
                    extends _BankCorbaTargetAdapterImplBase {
      private IBank bank;
      BankCorbaTargetAdapter(IBank bank) {
         this.bank = bank;
         ORB orb = ORB.init(args, null);
         orb.connect(this);
         org.omg.CORBA.Object objRef =
              orb.resolve_initial_references("NameService");
         NamingContext nameService =
               NamingContextHelper.narrow(objRef);
         NameComponent nc = new NameComponent("BankServer", "");
         NameComponent path [] = {nc};
         nameService.rebind(path, bankCorbaRef);
      }
```

The deposit operation calls Bank's corresponding method, but replaces CommunicationException and UnknownAccountException with the CORBA user exceptions CorbaCommunicationException and CorbaUnknownAccountException:

```
   public void deposit(String accountNumber, double value)
            throws CorbaCommunicationException,
                   CorbaUnknownAccountException {
      try {
           bank.deposit(accountNumber,value);
       } catch (CommunicationException e) {
            throw new CorbaCommunicationException();
         }
       catch (UnknownAccountException e) {
           throw new CorbaUnknownAccountException(e.getMessage());
       }
   }
} // end of BankCorbaTargetAdapter
```

Notice that the exception handling above is different from the one in the RMI and Jini target adapters: the business exception `UnknownAccountException` and the general communication exception `CommunicationException` must be explicitly replaced by the CORBA representations of these exceptions.

The CORBA `BankInitializer` has the same structure as the RMI and the Jini initializers:

```
   public class BankInitializer {
      public static void main(String[] args) {
         ...
         IBank bank = new Bank();
         BankCorbaTargetAdapter targetAdapter =
                                    new BankCorbaTargetAdapter(bank);
      }
   }
```

7 Conclusion

In order to support the task of developing distributed applications, we have presented software architectures for structuring such applications, shown how to refine these architectures with a design pattern, and illustrated how to implement this pattern using different distribution platforms.

The architectures are modular, flexible, reusable, and support progressive implementation. Indeed, the software architectures presented here are closely related to Pim (Progressive Implementation Method [2]). In Pim, the initial version of the application is a functionally complete and local prototype; in such prototype, the GUI refers directly to the business layer. The prototype is useful for validating user requirements, since the local system is much simpler than the distributed one and, therefore, the impact of changing such requirements is smaller on this prototype. Later, when user requirements stabilize, distribution is introduced and then the GUI will depend on the distribution layer. Since the interfaces exported by the business and distribution layers are the same, the GUI is unaware of such change and thus does not need to be changed. This seamless process maintains the modularity of the software architectures, and the incremental nature of this method helps to tame the inherent complexity of distributed applications.

The design pattern presented refine these architectures, structuring distribution aspects modularly. Although it involves more classes than an *ad hoc* solution, it promotes software reuse and extendibility.

Acknowledgements

We thank our colleagues at Universidade Federal de Pernambuco for making important suggestions for improving this tutorial: Tiago Massoni, from the Languages and Software Engineering Group, Carlos Ferraz, Cidcley Souza, and Fernando Trinta, from the Distributed Systems Group.

References

[1] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modelling Language User Guide*. Addison-Wesley, 1999.

[2] Paulo Borba, Saulo Araújo, Hednilson Bezerra, Marconi Lima, and Sérgio Soares. Progressive implementation of distributed Java applications. In *Engineering Distributed Objects Workshop, ACM International Conference on Software Engineering*, pages 40–47, Los Angeles, USA, 17th–18th May 1999.

[3] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[4] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[5] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[6] Object Management Group. *CORBA/IIOP Specification*, 2.3.1 edition, October 1999.

[7] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[8] Sun Microsystems. *Java Remote Method Invocation Specification*, 1.50 edition, October 1998.

[9] Sun Microsystems. *Jini Architecture Specification*, 1.0 edition, January 1999.

[10] Susan Urban, Ling Fu, Jami Shah, Ed Harter, Tom Bluhm, and Brett Hartman. The implementation and evaluation of the use of CORBA in an engineering design application. In *Engineering Distributed Objects Workshop, ACM International Conference on Software Engineering*, pages 106–140, Los Angeles, USA, 17th–18th May 1999.