Phases of RPC based distributed applications

We distinguish between 3 phases:
- a) design and implementation
- b) binding of components
- c) invocation: a client invoking a server operation.

Component binding

Mediation and brokering

Distributed applications based on RPC

How to implement distributed applications based on remote procedure calls?

Distributed application
In order to isolate the communication idiosyncrasies of RPCs and to make the network interfaces transparent to the application programmer, so-called stubs are introduced.

Stubs
Stub functionality
Implementing a distributed application

RPC language
Phases of RPC based distributed applications

Mediation and brokering

Possible terms for a mediation component are: registry, broker or trading; Corba uses the term object request broker.

Functionality of a broker

servers register their available service interfaces with the broker ("export interface")

the broker supplies the client with information in order to localize a suitable server and to determine the correct service interface ("import interface.

Client-to-server binding

Broker information

Handling client requests
Broker may either just provide the service interface to the client or act as a mediator between client and server.

direct communication between C and S.

indirect communication between C and S; communication between C and S is only possible via broker V (or several brokers).
A broker manages information about the available, exported interfaces:

- server names ("white pages")
- service types ("yellow pages")
- behavioral or functional attributes:
  - static attributes: functionality of the provided services, cost, required bandwidth.
  - dynamic attributes: current server state.

We distinguish between 3 phases:

a) design and implementation
b) binding of components
c) invocation: a client invoking a server operation.

Component binding
Mediation and brokering

**Definitions**

**Remote object** is an object whose method can be called by an object residing on another Java Virtual Machine (JVM), even on another computer.

**Remote interface** is a Java interface specifying the methods of a remote object.

**Remote method invocation (RMI)** allows object-to-object communication between different Java Virtual Machines (JVM), i.e., it is the action of invoking a method of a remote interface on a remote object.

The method calls for local and remote objects have the same syntax.

**Remote Method Invocation (RMI)**

RMI supports communication among objects residing on different Java virtual machines (JVM). RMI is an RPC of the object-oriented Java environment.

**Definitions**

**RMI characteristics**
**RMI architecture**
**Locating remote objects**
**Developing RMI applications**
**Parameter Passing in RMI**
**Distributed garbage collection**
How does RMI work

Java RMI uses a registry to provide naming services for remote objects, stub and skeleton to facilitate communications between client and server.

RMI works as follows:
1. A server object is registered with the RMI registry.
2. A client looks through the RMI registry for the remote object.
3. Once the remote object is located, its stub is returned to the client.
4. The remote object can be used in the same way as a local object.
   Communication between client and server is handled by stubs and skeletons.

RMI architecture

Stub/Skeleton layer:
- Layer intercepts method calls by the client and redirects these calls to the remote object.
- Object serialization/deserialization; hidden from the application.

Remote Reference layer:
- Connects client and remote objects exported by the server environment by a 1-to-1 connection link.
- The layer provides JRMP (Java Remote Method Protocol) via TCP/IP.
- Mapping of stub/skeleton operations to the transport protocol of the host; it interfaces the application code with the network communication.
- The layer supports the method invoke.

Remote Reference layer:
- Object invoke (Remote obj, java.lang.reflect.Method method, Object [] params, long timeout) throws Exception
Remote Method Invocation (RMI)

RMI supports communication among objects residing on different Java virtual machines (JVMs). RMI is an RPC of the object-oriented Java environment.

- Definitions
- RMI characteristics
- RMI architecture
- Locating remote objects
- Developing RMI applications
- Parameter Passing in RMI
- Distributed garbage collection

**Naming interface methods**

```java
public static void bind (String name, Remote obj)
        throws AlreadyBoundException, java.net.MalformedURLException, RemoteException,
        example for name: rmi://host/service-port/service-name
        if name is already bound to an object, then AlreadyBoundException is triggered.

public static void rebind (String name, Remote obj)
        throws java.net.MalformedURLException, RemoteException,
        associates always the remote object obj with name (in URL format).

public static Remote lookup (String name)
        throws NotBoundException, java.net.MalformedURLException, RemoteException,
        returns as a result a reference (a stub) to the remote object.
        if name is not bound to an object, then NotBoundException is triggered.

public static void unbind (String name)
        throws NotBoundException, RemoteException.

public static String [ ] list (string name)
        throws java.net.MalformedURLException, RemoteException.
        as a result, it returns all names entered in the registry.
```

**Registry-Lookup**

The client invokes a lookup for a particular URL, the name of the service (rmi://host:port/service). The following describes the steps:

1) a socket connection is opened with the host on the specified port.
2) a stub to the remote registry is returned.
3) the method Registry.lookup() on this stub is performed. The method returns a stub for the remote object.
4) the client interacts with the remote object through its stub.
Remote Method Invocation (RMI)

RMI supports communication among objects residing on different Java virtual machines (JVM). RMI is an RPC of the object-oriented Java environment.

Definitions
RMI characteristics
RMI architecture
Locating remote objects
Developing RMI applications
Parameter Passing in RMI!
Distributed garbage collection

1 Defining a remote interface

A remote interface is the set of methods that can be invoked remotely by a client.

- The remote interface must be declared public.
- The remote interface must extend the java.rmi.Remote interface.
- Each method must throw the java.rmi.RemoteException exception.
- If the remote methods have any remote objects as parameters or return types, they must be interfaces rather than implementation classes.

Example: Remote interface definition

```java
public interface HelloInterface extends java.rmi.Remote {
    /* this method is called by remote clients and it is implemented by the remote object */
    public String sayHello() throws java.rmi.RemoteException
}
```

2 Implementing the remote interface

Definition of an implementation class that defines the methods of the remote interface;

- the abstract class java.rmi.server.RemoteServer provides the basic semantics to support remote references.
- java.rmi.server.RemoteServer has subclasses:
  - java.rmi.server.UnicastRemoteObject: defines a non-replicated remote object whose references are valid only while the server process is alive.
  - java.rmi.activation.Activatable: defines a remote object which can be instantiated on demand (if it has not been started already).

Example: Remote interface implementation

```java
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
import java.util.Date.*;

public class HelloServer extends UnicastRemoteObject implements HelloInterface{
    public HelloServer() throws RemoteException {
        super();
        /* call superclass constructor to export this object */
    }

    public String sayHello() throws RemoteException {
        return "Hello World, the current system time is " + new Date();
    }
}
```

3 Generating stubs and skeletons

The tool rmic generates stub and skeleton from the implemented class (up to Java version 5).
Every remotely accessible object must be registered in a registry in order to make it available; stubs are needed for registration.
the registry is started at the host of the remote object.

**Example for object registration**

```java
import java.rmi.*;

public class RegisterIt {
    public static void main (String args [])
    try { // Instantiate the object
        HelloServer obj = new HelloServer();
        System.out.println("Object instantiated: " + obj);
        Naming.rebind("/HelloServer", obj);
        System.out.println("HelloServer bound in registry");
    } catch (Exception e) {
        System.out.println(e)
    }
}
```

**Example: Client implementation**

```java
import java.rmi.*;

public class HelloClient {
    public static void main (String args [])
    try { // Instantiate the object
        HelloServer obj = new HelloServer();
        System.out.println("Object instantiated: " + obj);
        Naming.rebind("/HelloServer", obj);
        System.out.println("HelloServer bound in registry");
    } catch (Exception e) {
        System.out.println("HelloClient exception: " + e);
    }
}
```

Missing access rights results in the exception:

1. **Defining a remote interface**
2. **Implementing the remote interface**
3. **Generating stubs and skeletons**
4. **Remote object registration**
5. **Client implementation**

At the end the client must be started.

Parameters with primitive data types are passed with their values between JVMs; for object parameters, a distinction is made between local and remote:

1. **local object parameter**
   - RMI passes the object itself, rather than the object reference,
   - The transmitted object must implement the interface java.io.Serializable or java.io.Externalizable.

2. **remote object parameter**
   - RMI transmits the stub of the remote object; the stub is a reference to the remote object.
Utilization of life references for each JVM; reference counter represents the number of life references.

The first client access creates a referenced message sent to the server.
If there is no valid client reference, then an unreferenced message is sent to the server.
Time limit of references ("lease time", e.g. 10 minutes); the connection to the server must be renewed by the client, otherwise the reference becomes invalid.