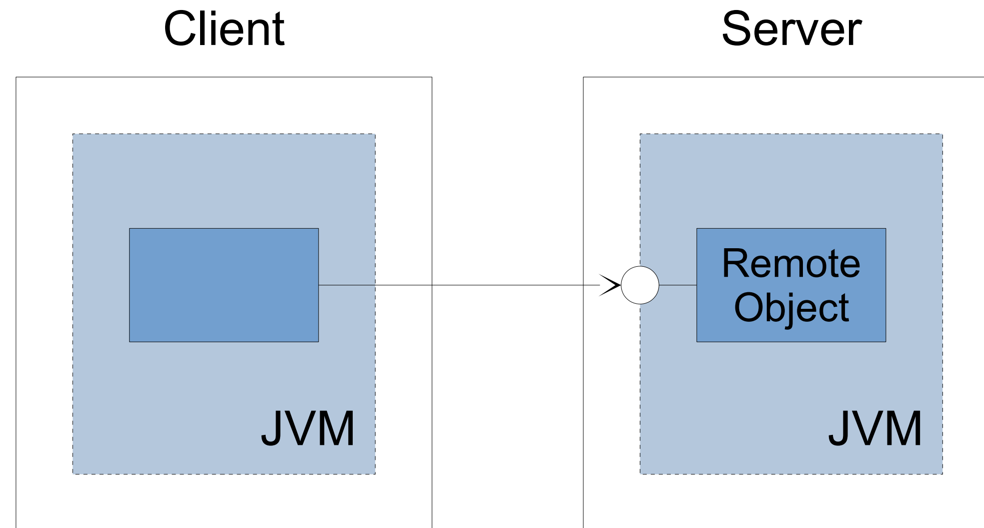
Remote Method Invocation (RMI)

Introduction

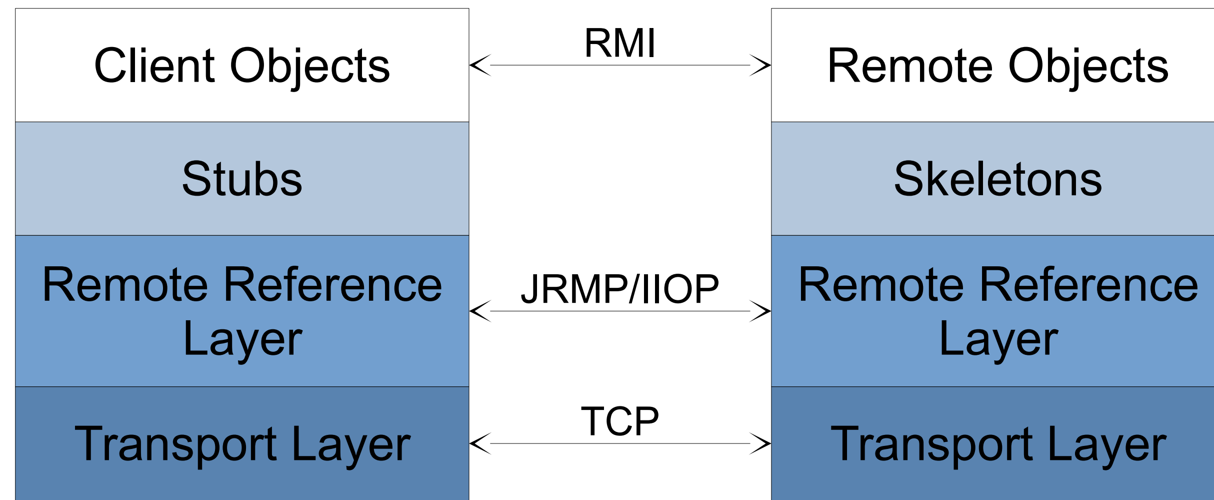
- RMI allows to invoke the methods of Java objects over the network (remote procedure call)
- Method invocations of remote objects look like those of local objects (locality transparency)
- Instead of defining a low-level message protocol, Java interfaces are used as application protocol
- RMI restricts to Java programs but provides interoperability with CORBA

Remote Objects



- A remote object is a Java object whose methods can be invoked from outside the virtual machine in which it lives
- A remote object has a remote interface which defines the methods that can remotely be invoked

RMI Architecture



- The client initiates a remote method invocation by calling the corresponding method on the stub
- The stub forwards the method invocation to the remote reference layer which sends it over the transport layer to the server
- The skeleton receives from the server-side remote reference layer the remote request and converts it into a call of the actual remote object
- If the method generates a return value or an exception, it is returned the same way back to the client

Stubs and Skeletons

A stub

- is a client-side proxy object which implements the same methods as the remote object
- maintains an internal reference to the remote object it represents
- forwards a method invocation to the remote reference layer
- is responsible for the marshalling of method arguments and the unmarshalling of return values and exceptions

A skeleton

- is the server-side counterpart of a stub
- converts requests from the remote reference layer into appropriate calls on the associated remote object
- is responsible for the unmarshalling of method arguments and the marshalling of return values and exceptions

Remote Reference Layer

The remote reference layer

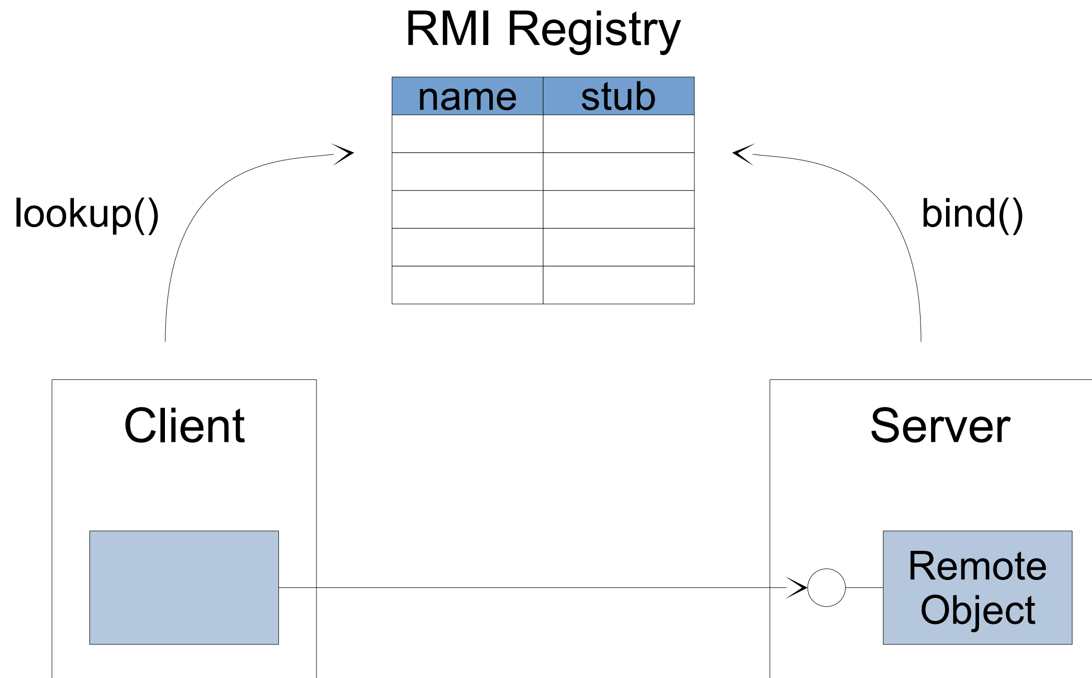
- handles the creation and management of remote object references
- knows the communication style for a given remote object (point-to-point, replicated, multicast)
- generates the corresponding transport-level requests

Marshalling and Unmarshalling

Marshalling and unmarshalling is the process of converting arguments, return values and exceptions into a serialized form, and vice versa:

- Primitive values are marshalled to their internal byte representation
- Local objects are marshalled using Java object serialization, i.e. the receiver obtains a copy of the object (pass by value)
- With remote objects, the stubs are used as marshalled data, i.e. the receiver obtains a stub over which it can invoke the object (pass by reference)

RMI Registry

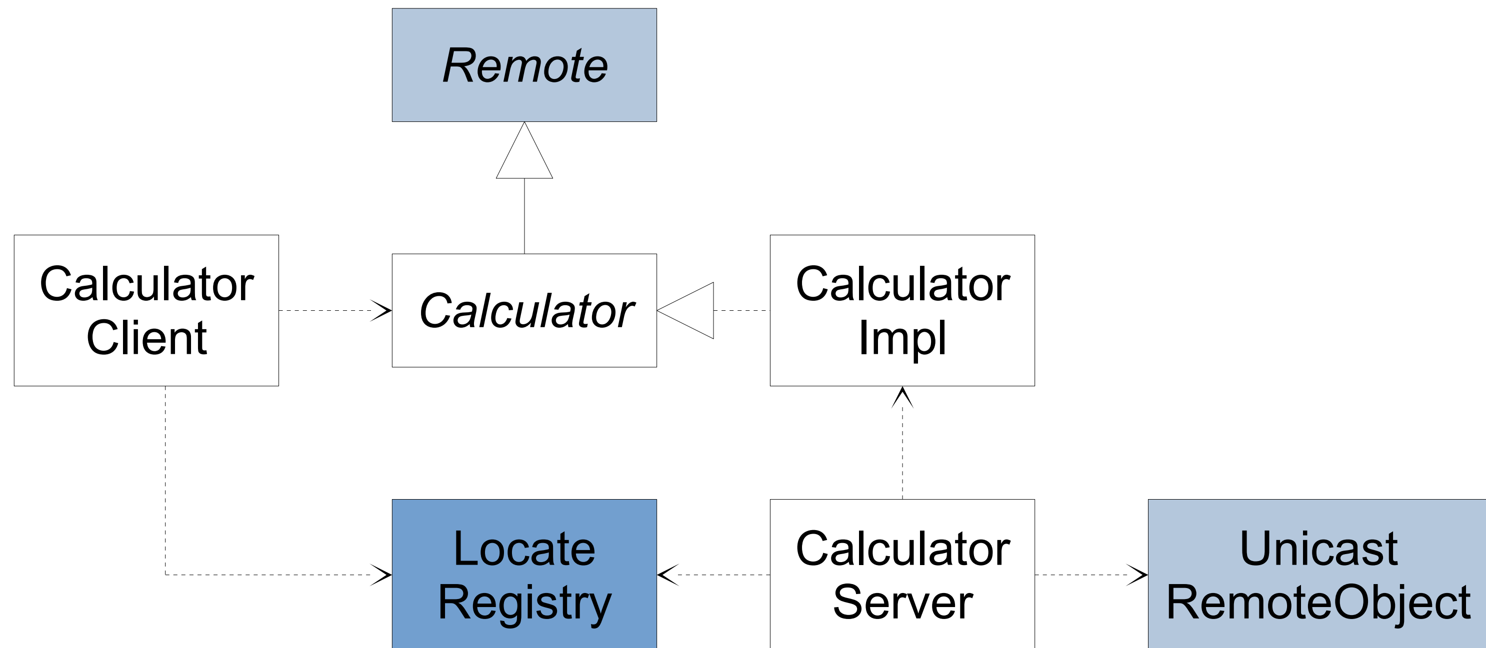


- The RMI registry is a naming service for remote objects
- The server program registers its remote objects with a local registry binding the corresponding stubs to names
- A client can look up the remote objects in the server's registry providing their names

RMI Interfaces and Classes

- java.rmi.Remote
  is an interface that marks remote interfaces
- java.rmi.RemoteException
  is the base exception class used by the RMI runtime to indicate client-side, server-side or network errors
- java.rmi.RemoteObject
  is the base class of remote objects and client stubs, it contains a remote reference and re-implements the Object behavior
- java.rmi.server.RemoteServer
  is an abstract class with static methods useful for implementing remote servers
- java.rmi.server.UnicastRemoteObject
  is a concrete subclass of RemoteServer that implements point-to-point non-persistent remote references
- java.rmi.registry.Registry
  is an interface which defines the access methods of the RMI registry
- java.rmi.registry.LocateRegistry
  is a class with static methods to get references of RMI registries

Example Application



The example application exports a remote calculator to provide clients with the calculation capacity of a server machine

Development Steps

1. Define the remote interface
2. Write a class that implements the remote interface
3. Implement a remote server program
4. Create a client program
5. Compile and run the programs

Defining a Remote Interface

- A remote interface has to extend the java.rmi.Remote interface
- Each method has to declare that it throws a java.rmi.RemoteException
- Method arguments and return values must be of primitive types, serializable objects or remote objects

```
public interface Calculator extends Remote {
   public double add(double x, double y) throws RemoteException;
   public double subtract(double x, double y) throws RemoteException;
   public double multiply(double x, double y) throws RemoteException;
   public double divide(double x, double y)
      throws ZeroDivisionException, RemoteException;
}
```

Implementing a Remote Interface

- The implementation class has to implement all the methods of the remote interface
- It may have additional methods which are not remotely callable

```
public class CalculatorImpl implements Calculator {
    public double add(double x, double y) {
        return x + y;
    }
    ...
    public double divide(double x, double y) throws ZeroDivisionException {
        if (y == 0) throw new ZeroDivisionException();
        return x / y;
    }
}
```

Implementing a Remote Server

A remote server

- creates one or more remote objects
- exports the remote objects to the RMI runtime such that they are remotely accessible
- registers at least one remote object with a local RMI registry

```
public class CalculatorServer {
    public static void main(String[] args) throws Exception {
        Calculator calculator = new CalculatorImpl();
        Calculator stub =
            (Calculator) UnicastRemoteObject.exportObject(calculator, 0);
        Registry registry = LocateRegistry.createRegistry(5495);
        registry.bind("Abacus", stub);
    }
    ...
}
```

Creating a Client Program

A client program

- gets a reference to the server's RMI registry
- looks up a remote object in the registry
- invokes the methods of the remote object

```
public class CalculatorClient {
    public static void main(String args[]) throws Exception {
        Registry registry = LocateRegistry.getRegistry(args[0], 5495);
        Calculator calculator = (Calculator) registry.lookup("Abacus");
        double x = calculator.add(1, 1);
        ...
    }
}
```

Compiling and Running the Programs

Server:

```
> javac -d build src/rmi/examples/*.java
> java -cp build rmi.examples.CalculatorServer
```

Client:

```
> javac -d build src/rmi/examples/*.java
> java -cp build rmi.examples.CalculatorClient serverhost
```