Distributed Objects

Remote Method Invokation

Distributed Systems

BUSINESS LAYER

Application Façade

Business Workflow | Business Components | Business Entities

DATA LAYER

Data Access Components | Data Helpers/ Utilities | Service Agents

PRESENTATION LAYER

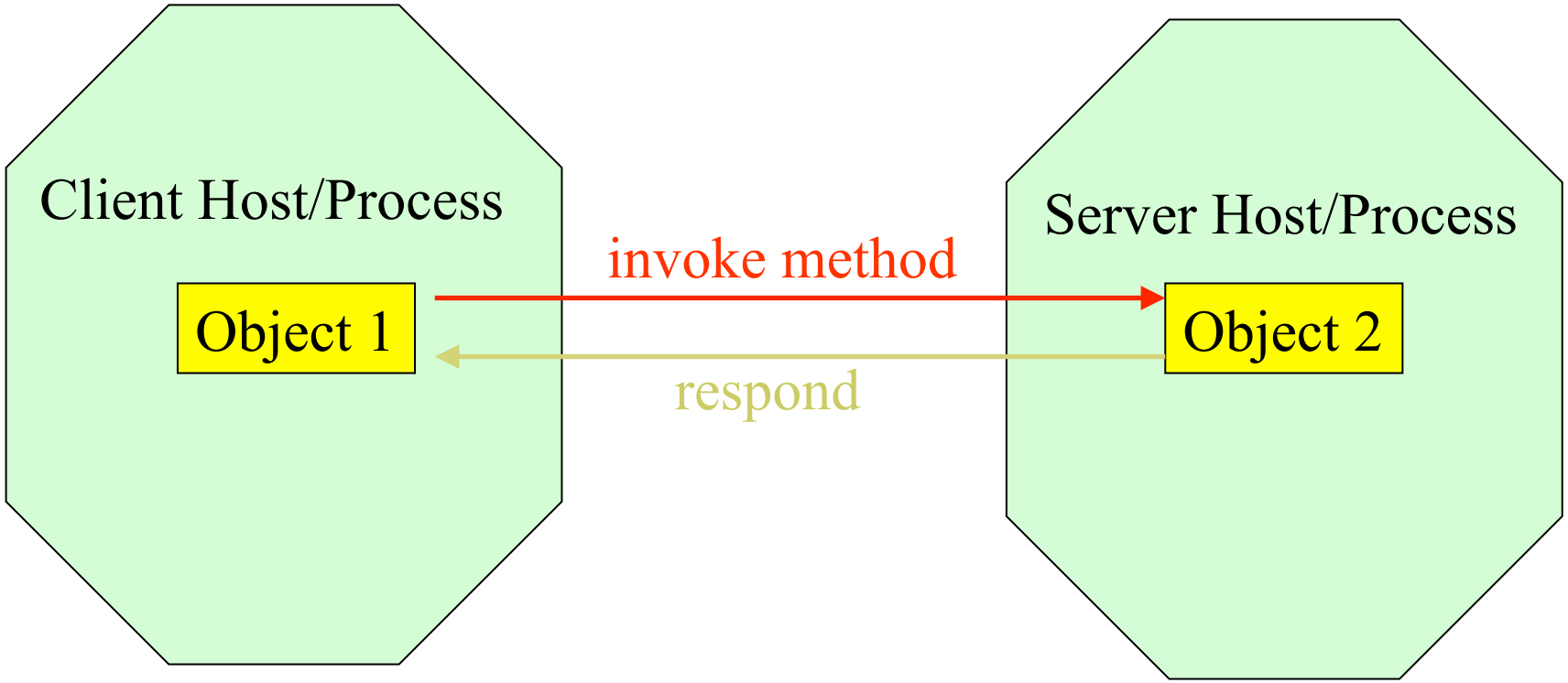
UI Components

UI Process Components

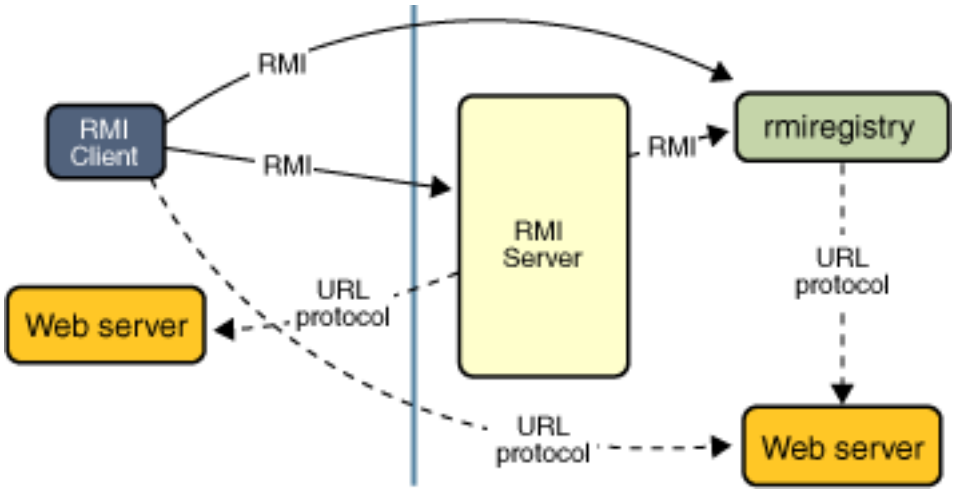
Data Sources

Object Oriented Paradigm

Distributed Object Oriented Paradigm



The RMI model



Distributed object applications need to:

- Locate remote objects. Applications can use various mechanisms to obtain references to remote objects. For example, an application can register its remote objects with RMI's simple naming facility, the RMI registry. Alternatively, an application can pass and return remote object references as part of other remote invocations.
- Communicate with remote objects. Details of communication between remote objects are handled by RMI. To the programmer, remote communication looks similar to regular Java method invocations.
- Load class definitions for objects that are passed around. Because RMI enables objects to be passed back and forth, it provides mechanisms for loading an object's class definitions as well as for transmitting an object's data.

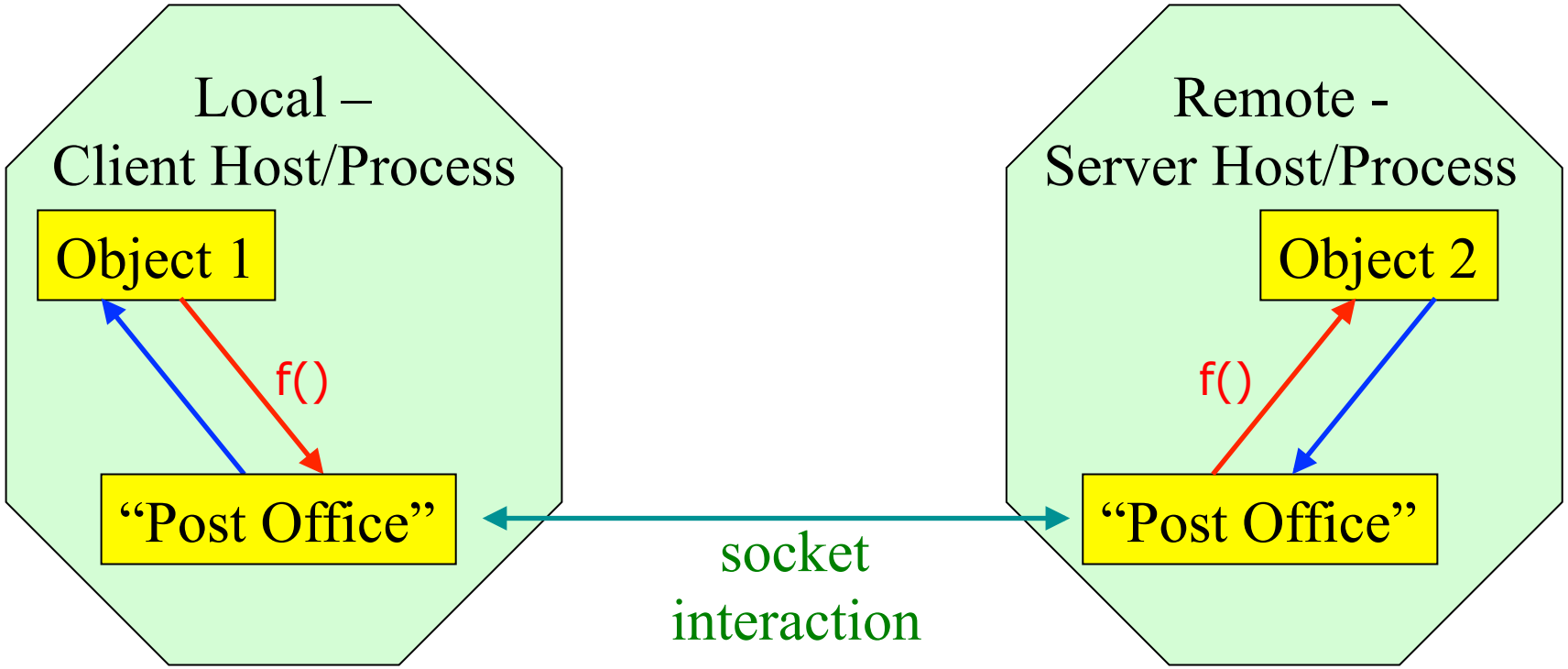
Distributed Objects

Remote Method Invokation:

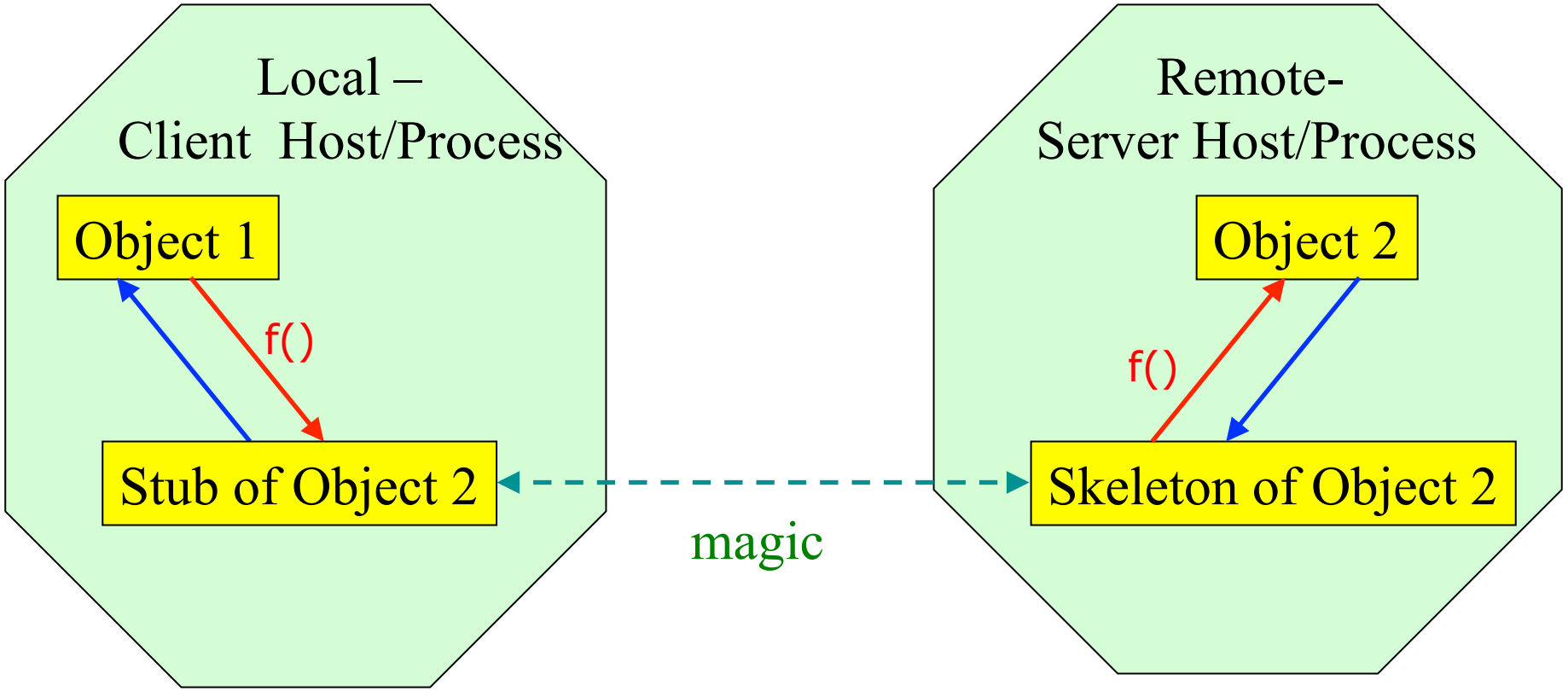
How does it work?

The conceptual model

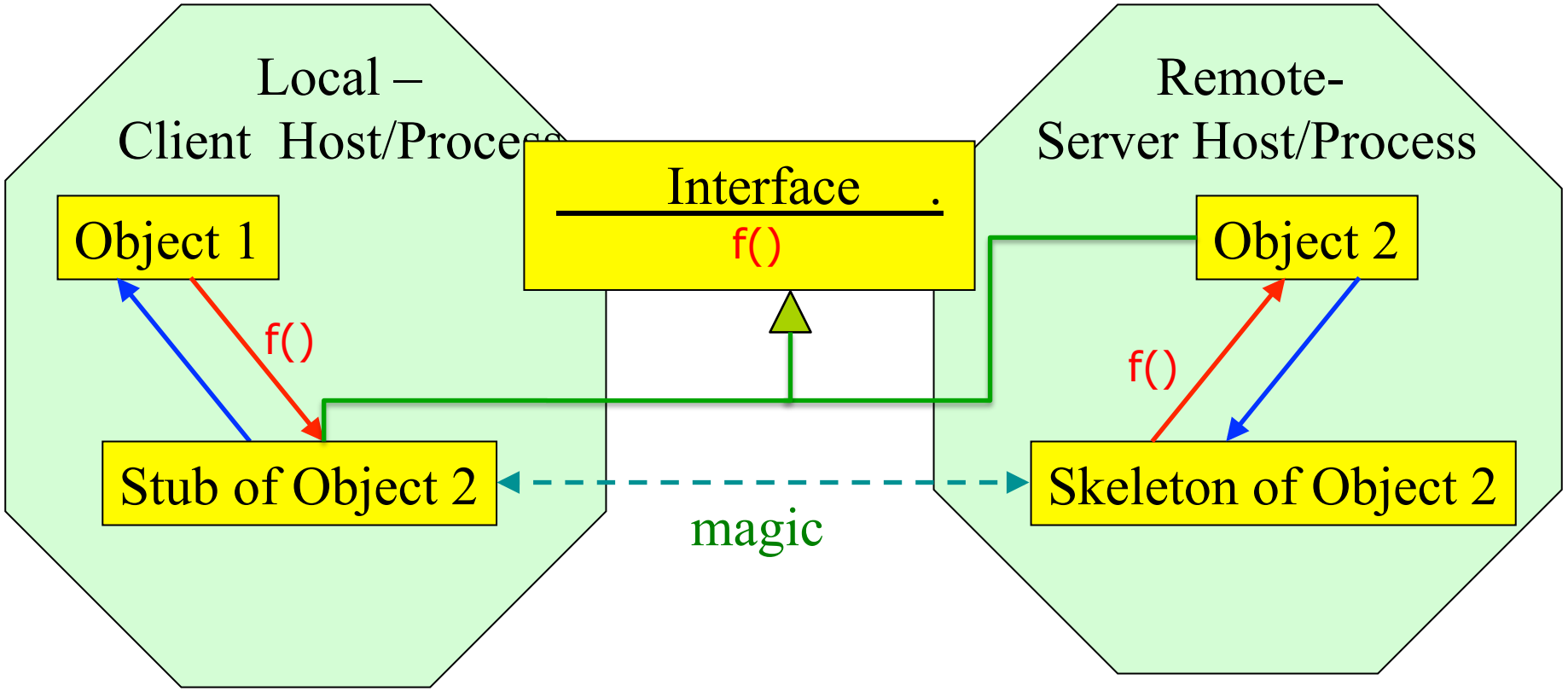
Distributed Object Oriented: implementation



Distributed Object Oriented: RMI paradigm



Distributed Object Oriented: RMI paradigm



Distributed Objects

A “do it yourself” implementation

A “do it yourself” implementation

1. Person: the interface

```
package distributedobjectdemo;

public interface Person {
  public int getAge() throws Throwable;
  public String getName() throws Throwable;
}
```

A “do it yourself” implementation

2. Person: The class

```
package distributedobjectdemo;

public class PersonServer implements Person{
  int age;
  String name;
  public PersonServer(String name,int age){
    this.age=age;
    this.name=name;
  }
  public int getAge(){
    return age;
  }
  public String getName(){
    return name;
  }
 public static void main(String a[]) {
    PersonServer person = new PersonServer("Marko", 45);
    Person_Skeleton skel = new Person_Skeleton(person);
    skel.start();
    System.out.println("server started");
  }
}
```

A “do it yourself” implementation

3. Person: the skeleton

```
package distributedobjectdemo;
import java.net.Socket;
import java.net.ServerSocket;
import java.io.*;

public class Person_Skeleton extends Thread {
  PersonServer myServer;
  int port=9000;

  public Person_Skeleton(PersonServer server) {
    this.myServer=server;
  }
// la classe continua...
```

A “do it yourself” implementation

3. Person: the skeleton

```
public void run(){
   Socket socket = null;
   ServerSocket serverSocket=null;
   try {
     serverSocket=new ServerSocket(port);
   }
   catch (IOException ex) {
    System.err.println("error while creating serverSocket");
    ex.printStackTrace(System.err);  System.exit(1);
   }

   while (true) {
    try {
     socket=serverSocket.accept();
     System.out.println("Client opened connection");
    }
    catch (IOException ex) {
     System.err.println("error accepting on serverSocket");
     ex.printStackTrace(System.err);  System.exit(1);
    }
    // il metodo continua...
```

A “do it yourself” implementation

3. Person: the skeleton

```
try {
  while (socket!=null){
    ObjectInputStream instream=
              new ObjectInputStream(socket.getInputStream());
    String method=(String)instream.readObject();
    if (method.equals("age")) {
      int age=myServer.getAge();
      ObjectOutputStream outstream=
              new ObjectOutputStream(socket.getOutputStream());
      outstream.writeInt(age);
      outstream.flush();
    } else if (method.equals("name")) {
      String name=myServer.getName();
      ObjectOutputStream outstream=
              new ObjectOutputStream(socket.getOutputStream());
      outstream.writeObject(name);
      outstream.flush();
    }
  }
//prosegue con il catch...
```

A “do it yourself” implementation

3. Person: the skeleton

```
    } catch (IOException ex) {
       if (ex.getMessage().equals("Connection reset")) {
         System.out.println("Client closed connection");
       } else {
         System.err.println("error on the network");
         ex.printStackTrace(System.err);   System.exit(2);
       }
     } catch (ClassNotFoundException ex) {
       System.err.println("error while reading object from the net");
       ex.printStackTrace(System.err);     System.exit(3);
     }
    }//fine del ciclo while(true)
  }  //fine del metodo run
}    //fine della classe
```

A “do it yourself” implementation

4. Person: the stub

```
package distributedobjectdemo;
import java.net.Socket;
import java.io.*;

public class Person_Stub implements Person {
  Socket socket;
  String machine="localhost";
  int port=9000;

  public Person_Stub() throws Throwable {
    socket=new Socket(machine,port);
  }
  protected void finalize(){
    System.err.println("closing");
    try { socket.close(); }
    catch (IOException ex) {ex.printStackTrace(System.err); }
  }
  // la classe continua...
```

A “do it yourself” implementation

4. Person: the stub

```
public int getAge() throws Throwable {
   ObjectOutputStream outstream=
         new ObjectOutputStream(socket.getOutputStream());
   outstream.writeObject("age");
   outstream.flush();
   ObjectInputStream instream=
         new ObjectInputStream(socket.getInputStream());
   return instream.readInt();
  }

  public String getName() throws Throwable {
   ObjectOutputStream outstream=new
ObjectOutputStream(socket.getOutputStream());
   outstream.writeObject("name");
   outstream.flush();
   ObjectInputStream instream=
         new ObjectInputStream(socket.getInputStream());
   return (String)instream.readObject();
  }
} // fine della classe
```

A “do it yourself” implementation

5. Person: the client

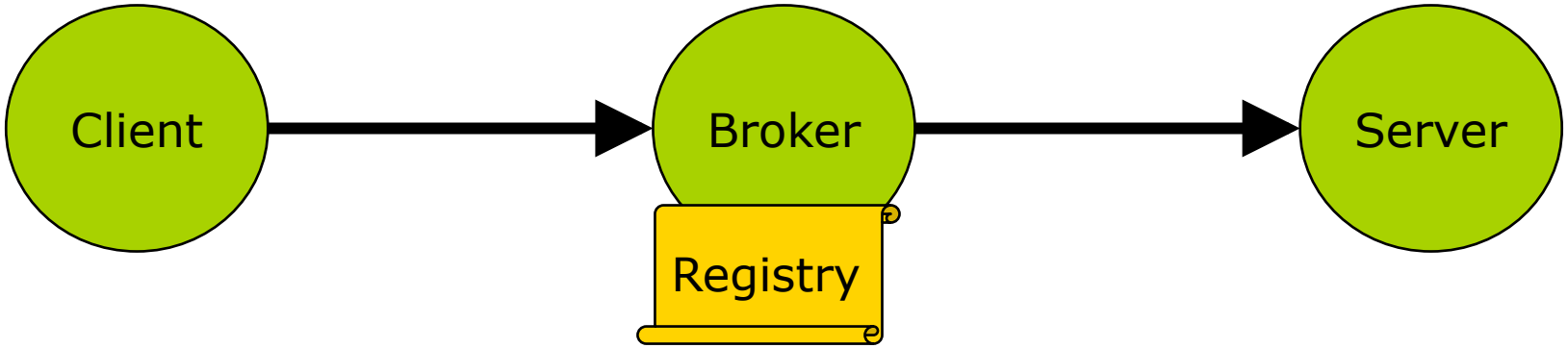
```
package distributedobjectdemo;

public class Client {

  public Client() {
    try {
      Person person=new Person_Stub();
      int age=person.getAge();
      String name=person.getName();
      System.out.println(name+" is "+age+" years old");
    }
    catch (Throwable ex) {
      ex.printStackTrace(System.err);
    }
  }
  public static void main(String[] args) {
    Client client1 = new Client();
  }
}
```

Open issues

- multiple instances
- Automatic stub and skeleton generation
- on demand server dentification
- on demand remote class activation



Distributed Objects

An RMI basic implementation

CLIENT & SERVER: iCalendar (interface)

1. Define the common interface

```
import java.rmi.*;
public interface iCalendar extends Remote {
  java.util.Date getDate () throws RemoteException;
}
```

SERVER: CalendarImpl

```
import java.util.Date;
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;
public class CalendarImpl
       extends UnicastRemoteObject
       implements iCalendar {
  public CalendarImpl() throws RemoteException {}
  public Date getDate () throws RemoteException {
    return new Date();
  }
```

2. Implement the service

```
public static void main(String args[]) {
  CalendarImpl cal;
   try {
     LocateRegistry.createRegistry(1099);
     cal = new CalendarImpl();
     Naming.bind("rmi:///CalendarImpl", cal);
      System.out.println("Ready for RMI's");
    } catch (Exception e) {e.printStackTrace();}
  }
}
```

SERVER: CalendarImpl

```
import java.util.Date;
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;
public class CalendarImpl
       extends UnicastRemoteObject
       implements iCalendar {
  public CalendarImpl() throws RemoteException {}
  public Date getDate () throws RemoteException {
    return new Date();
  }
```

3. Create Registry

```
public static void main(String args[]) {
  CalendarImpl cal;
   try {
     LocateRegistry.createRegistry(1099);
     cal = new CalendarImpl();
     Naming.bind("rmi:///CalendarImpl", cal);
      System.out.println("Ready for RMI's");
    } catch (Exception e) {e.printStackTrace()}
  }
}
```

SERVER: CalendarImpl

```
import java.util.Date;
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;
public class CalendarImpl
       extends UnicastRemoteObject
       implements iCalendar {
  public CalendarImpl() throws RemoteException {}
  public Date getDate () throws RemoteException {
    return new Date();
  }
```

4. Register yourself

```
public static void main(String args[]) {
  CalendarImpl cal;
   try {
     LocateRegistry.createRegistry(1099);
     cal = new CalendarImpl();
     Naming.bind("rmi:///CalendarImpl", cal);
      System.out.println("Ready for RMI's");
    } catch (Exception e) {e.printStackTrace()}
  }
}
```

Server

It is not necessary to have a thread wait to keep the server alive. As long as there is a reference to the CalendarImpl object in another virtual machine, the CalendarImpl object will not be shut down or garbage collected.

Because the program binds a reference to the CalendarImpl in the registry, it is reachable from a remote client, the registry itself! The CalendarImpl is available to accept calls and won't be reclaimed until its binding is removed from the registry, *and* no remote clients hold a remote reference to the CalendarImpl object.

CLIENT: CalendarUser

```
import java.util.Date;
import java.rmi.*;
public class CalendarUser {
  public static void main(String args[]) {
    long t1=0,t2=0;
    Date date;
    iCalendar  remoteCal;
    try {
      remoteCal = (iCalendar)
                  Naming.lookup("rmi://HOST/CalendarImpl");
      t1 = remoteCal.getDate().getTime();
      t2 = remoteCal.getDate().getTime();
    } catch (Exception e) {e.printStackTrace();}
    System.out.println("This RMI call took " + (t2-t1) +
      " milliseconds" );
  }
}
```

6. Use Service

Preparing and executing

SERVER

```
C:dir
CalendarImpl.java
iCalendar.java

C:javac CalendarImpl.java
C:rmic CalendarImpl

C:dir
CalendarImpl.java
iCalendar.java
CalendarImpl.class
iCalendar.class
CalendarImpl_Stub.class
CalendarImpl_Skel.class

C:java CalendarImpl
```

copy

CLIENT

```
C:dir
CalendarUser.java
iCalendar.java

C:javac CalendarUser.java

C:dir
CalendarUser.java
iCalendar.java
CalendarImpl_Stub.class

C:java CalendarUser
```

Preparing and executing (version in package rmidemo)

SERVER

```
C:dir rmidemo
CalendarImpl.java
iCalendar.java

C:javac rmidemo/CalendarImpl.java
C:rmic rmidemo.CalendarImpl

C:dir rmidemo
CalendarImpl.java
iCalendar.java
CalendarImpl.class
iCalendar.class
CalendarImpl_Stub.class
CalendarImpl_Skel.class

C:java rmidemo.CalendarImpl
```

CLIENT

```
C:dir rmidemo
CalendarUser.java
iCalendar.java

C:javac rmidemo/CalendarUser.java

C:dir rmidemo
CalendarUser.java
iCalendar.java
CalendarImpl_Stub.class

C:java rmidemo.CalendarUser
```

Distributed Objects

An RMI implementation

- VERY IMPORTANT NOTES-

VERY IMPORTANT: Parameter passing

Java Standard:

```
void f(int x) :
Parameter x is passed by copy
void g(Object k) :
```

Parameter k and return value are passed by reference

Java RMI:

```
void h(Object k) :
```

Parameter k is passed by copy!

UNLESS k is a REMOTE OBJECT (in which case it is passed as a REMOTE REFERENCE, i.e. its stub is copied if needed)

IMPORTANT: Parameter passing

Passing By-Value

When invoking a method using RMI,all parameters to the remote method are passed *by-value* .This means that when a client calls a server,all parameters are copied from one machine to the other.

Passing by remote-reference

If you want to pass an object over the network by-reference,it must be a remote object, and it must implement java.rmi.Remote.A stub for the remote object is serialized and passed to the remote host. The remote host can then use that stub to invoke callbacks on your remote object. There is only one copy of the object at any time,which means that all hosts are calling the same object.

Serialization

- Any basic primitive type (int,char,and so on) is automatically serialized with the object and is available when deserialized.

- Java objects can be included with the serialized or not:

- Objects marked with the ***transient*** keyword are not serialized with the object and are not available when deserialized.

- Any object that is not marked with the transient keyword must implement ***java.lang.Serializable*** .These objects are converted to bit-blob format along with the original object. If your Java objects are neither transient nor implement ***java.lang.Serializable*** ,a NotSerializable Exception is thrown when ***writeObject()*** is called.

When not to Serialize

- The object is large.Large objects may not be suitable for serialization because operations you do with the serialized blob may be very intensive. (one could save the blob to disk or transporting the blob across the network)

- The object represents a resource that cannot be reconstructed on the target machine.Some examples of such resources are database connections and sockets.

- The object represents sensitive information that you do not want to pass in a serialized stream..

Distributed Objects

An RMI implementation

- Addendum -

RMI-IIOP

- **RMI-IIOP is a special version of RMI that is compliant with CORBA.**

  RMI has some interesting features not available in RMI-IIOP,such as distributed garbage collection, object activation and downloadable class files.

  EJB and J2EE mandate that you use RMI-IIOP, not RMI.

  rmic –iiop generates IIOP stub and tie (instead of stub and skeleton)
  rmic –idl generates OMG IDL

  See docs.oracle.com/javase/7/docs/technotes/tools/#rmi

Preparing and executing

NOTES:

- **starting from Java 2 the skeleton may not exist (its functionality is absorbed by the class file).**

Starting from Java 5 the rmic functionality has been absorbed by javac, so the whole process becomes transparent (but even more misterious…)

See docs.oracle.com/javase/tutorial/rmi/ for an example of current usage of rmi

Alternatives – starting the register

- Instead of writing in the server code:

```
LocateRegistry.createRegistry(1099);
```

You can start the registry from the shell:

```
C: rmiregistry 1099
```
(port number is optional)

Note: in Java 2 you need an additional parameter:

```
C: rmiregistry -J-Djava.security.policy=registerit.policy
```

where registerit.policy is a file containing:

```
grant {permission java.security.AllPermission}
```

Or some permission restriction. Typically the file is kept in %USER_HOME%/.java.policy

Preparing and executing - security

The JDK security model requires code to be granted specific permissions to be allowed to perform certain operations.

You need to specify a policy file when you run your server and client.

```
grant { permission java.net.SocketPermission "*:1024-65535",
"connect,accept";
permission java.io.FilePermission "c:\\…path…\\", "read"; };
```

```
java -Djava.security.policy=java.policy executableClass
```

Access to system properties

- **Nota: instead of specifìying a property at runtime (-D switch of java command), You can hardwire the property into the code:**

```
-Djava.security.policy=java.policy

System.getProperties().put(
                "java.security.policy",
                "java.policy");
```