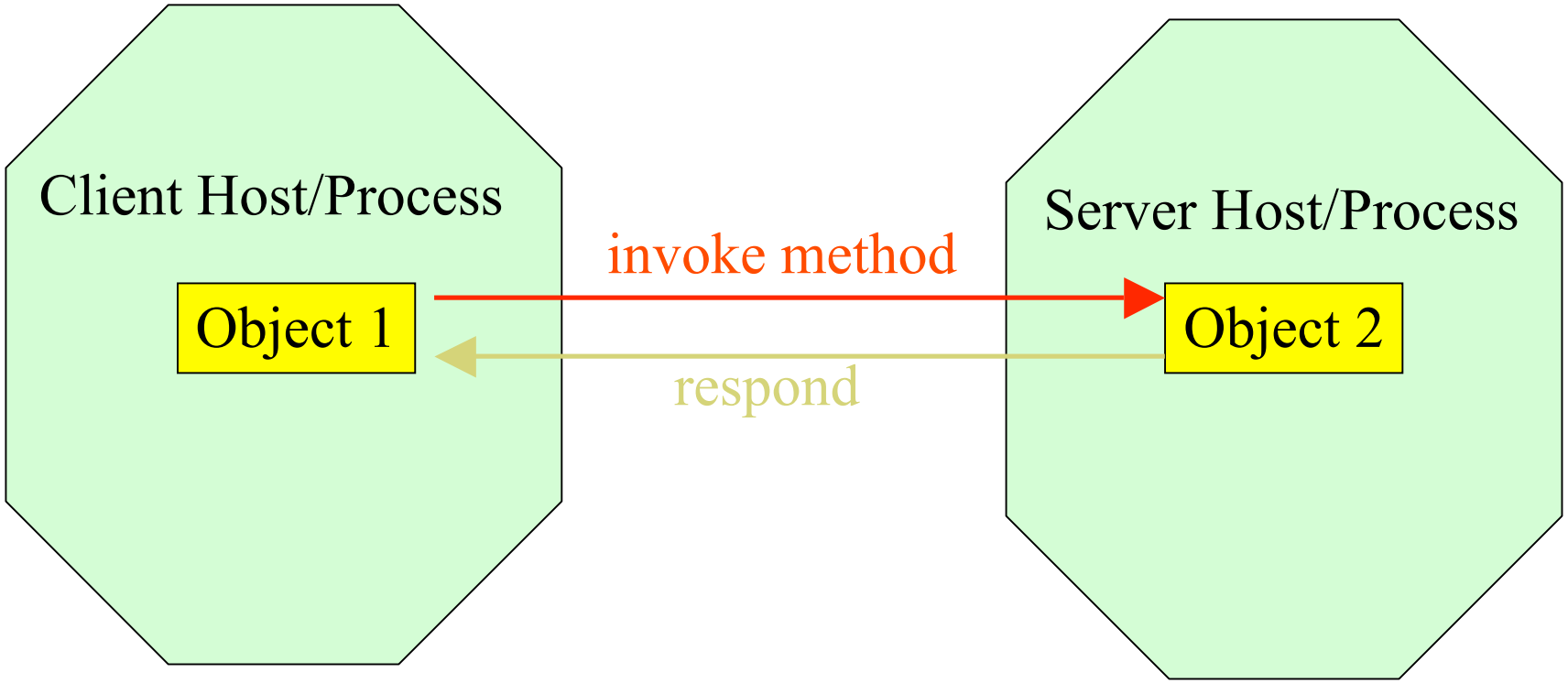
Distributed Objects

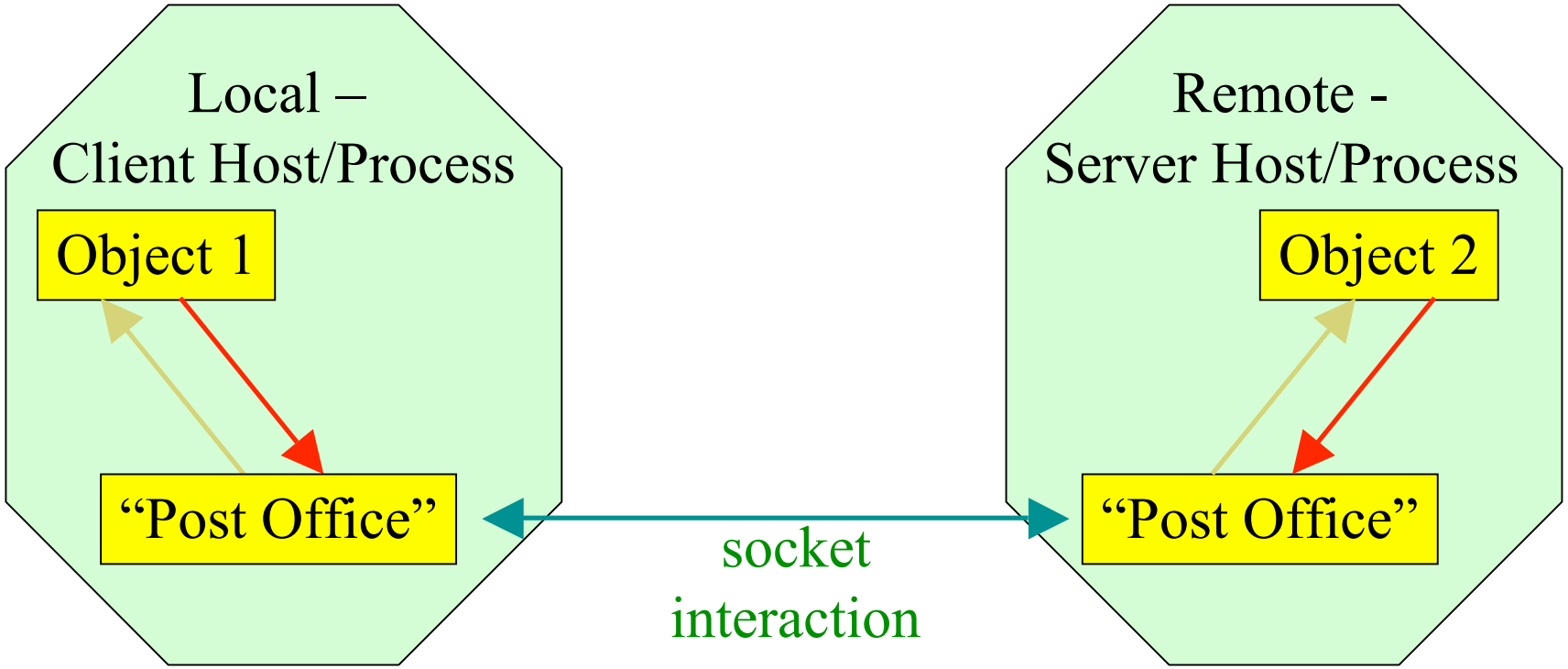
Remote Method Invokation:
Conceptual model

Object Oriented Paradigm

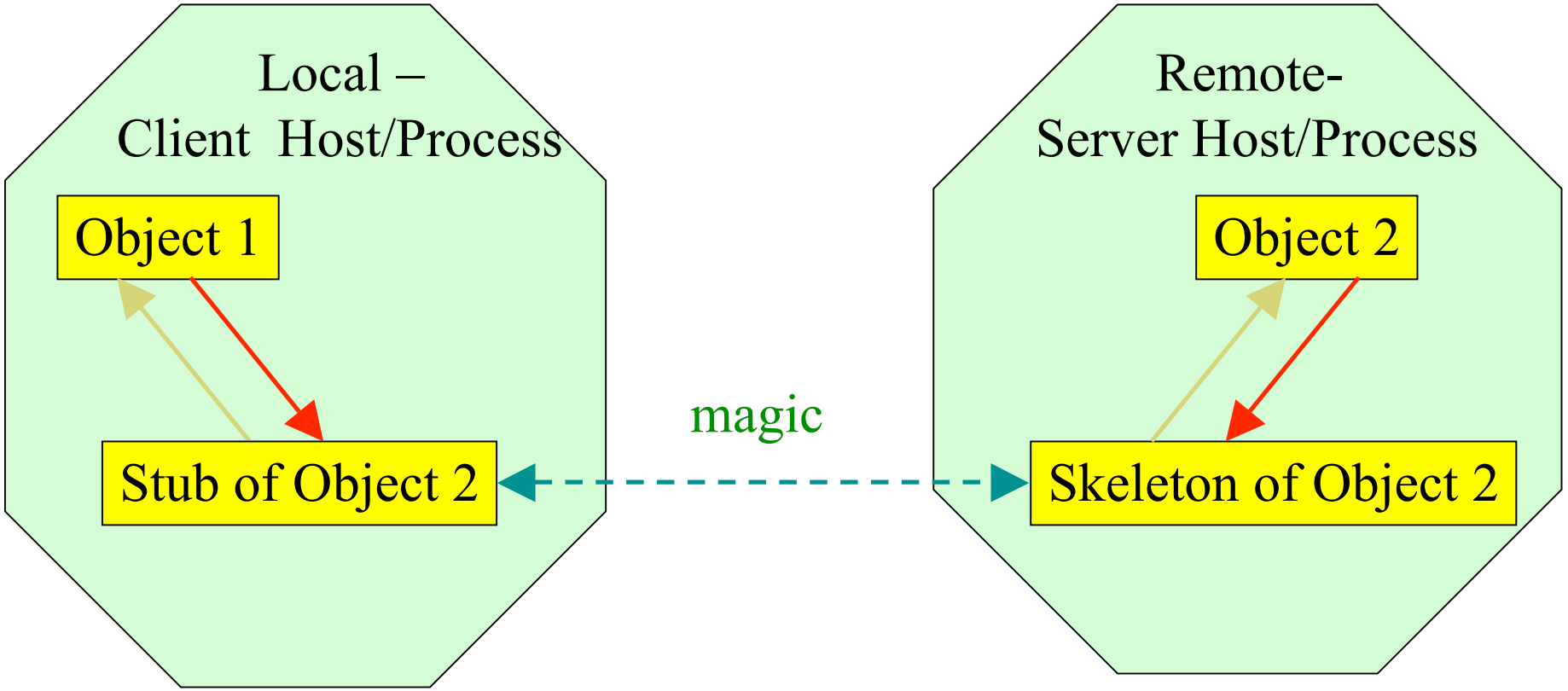
Distributed Object Oriented Paradigm



Distributed Object Oriented: implementation



Distributed Object Oriented: RMI paradigm



Distributed Objects

A “do it yourself” implementation

Un oggetto distribuito “fai da te”

1. Person: l’interfaccia

```
package distributedobjectdemo;

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

Un oggetto distribuito “fai da te”

2. Person: la classe

```
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");
  }
}
```

Un oggetto distribuito “fai da te”

3. Person: lo 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...
```

Un oggetto distribuito “fai da te”

3. Person: lo 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...
```

Un oggetto distribuito “fai da te”

3. Person: lo 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...
```

Un oggetto distribuito “fai da te”

3. Person: lo 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
```

Un oggetto distribuito “fai da te”

4. Person: lo 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...
```

Un oggetto distribuito “fai da te”

4. Person: lo 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
```

Un oggetto distribuito “fai da te”

5. Person: il client

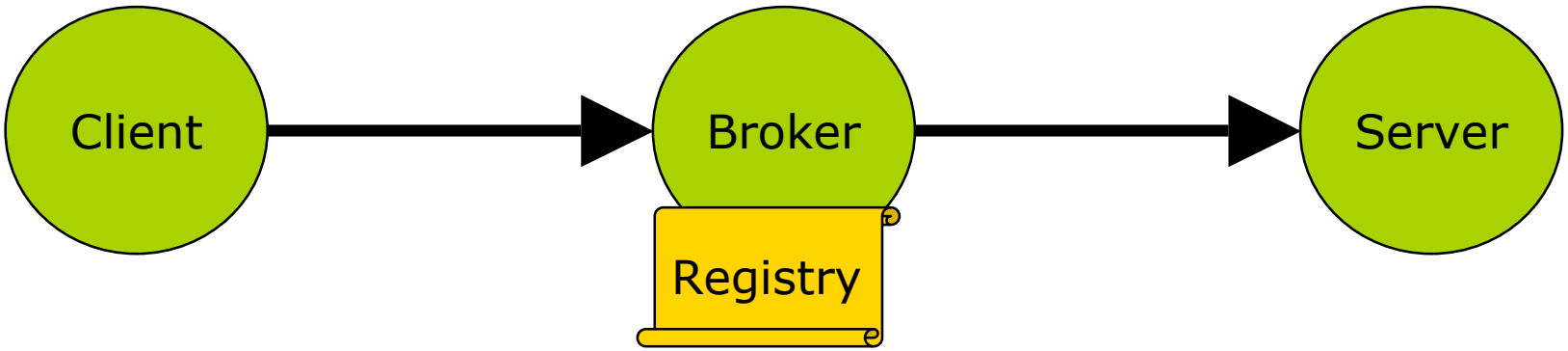
```
package distributedobjectdemo;

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
```

copy

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

- Addendum -

Preparing and executing - security

The JDK 1.2 security model is more sophisticated than the model used for JDK 1.1. JDK 1.2 contains enhancements for finer-grained security and requires code to be granted specific permissions to be allowed to perform certain operations.

Since JDK 1.2, 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
```

Accesso alle proprietà di sistema

- **Nota: instead of specifiying 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");
```

Preparing and executing

NOTE: in Java 2 the skeleton may not exist (its functionality is absorbed by the class file).

In order to use the Java 2 solution, one must specify the switch –v1.2

```
C:rmic –v1.2 CalendarImpl
```

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..

Alternatives – starting the register

- **Instead of writing in the server code:**

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

You can satrt 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

RMI-IIOP

- **RMI-IIOP is a special version of RMI that is compliant with CORBA and uses both *java.rmi* and *javax.rmi* .**

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

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

Distributed Objects

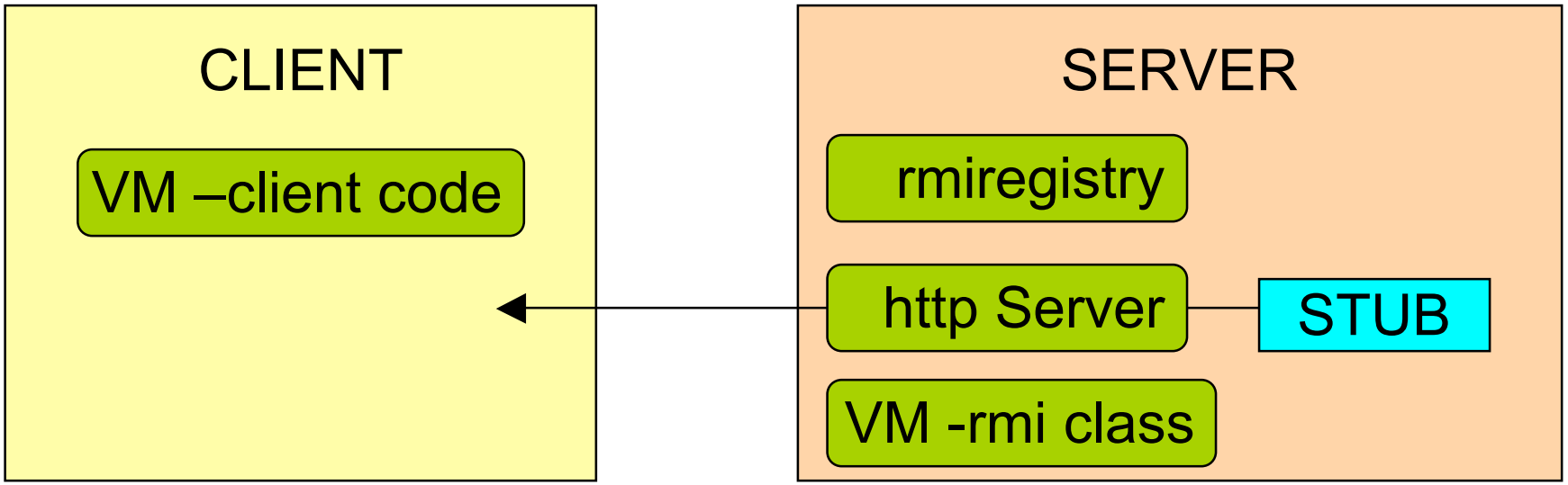
dynamic stub loading

Alternative 2 – stub dynamic loading

- **Instead of manually copying the stub from the Server to client, can we automatically load the stub at runtime?**

  "RMI can download the *bytecodes* of an object's class if the class is not defined in the receiver's virtual machine. The types and the behavior of an object, previously available only in a single virtual machine, can be transmitted to another, possibly remote, virtual machine. RMI passes objects by their true type, so the behavior of those objects is not changed when they are sent to another virtual machine. This allows new types to be introduced into a remote virtual machine, thus extending the behavior of an application dynamically."

Alternativa 2 – caricamento dinamico dello stub



CLIENT: CalendarUser-Caric.dinamico

```
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;
    System.setSecurityManager(new RMISecurityManager());
    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" );
  }
}
```

*This client expects a URL in the **marshalling stream** for the remote object. It will load the stub class for the remote object from the URL in the marshalling stream. Before you can load classes from a non-local source, you need to set a security manager.*

Note, as an alternative to using the RMISecurityManager, you can create your ownsecurity manager.

SERVER: CalendarImpl – Caricamento dinamico

```
import java.util.Date;
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;
public c
        e
        i
  public
  public
    retu
  }
```

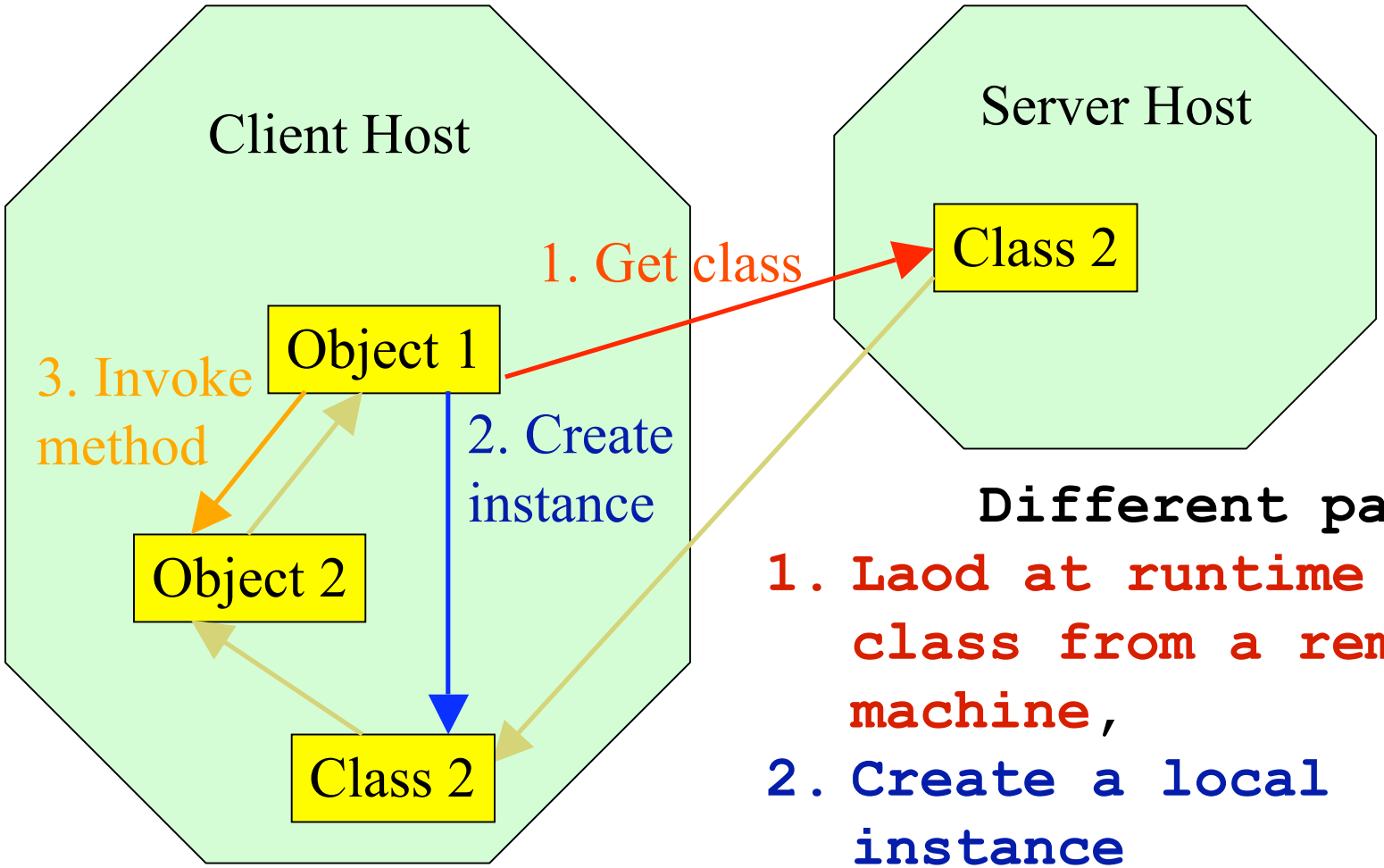
The first part remains untouched

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

Distributed Objects

A different paradigm:
dynamic loading of a remote class

Dynamic loading of a remote class



Different paradigm:

1. **Laod at runtime a class from a remote machine,**
2. **Create a local instance**
3. **Execute it.**

An utility class: a quitter Window

```
package rmiDynamicLoadingDemo;

import java.awt.event.*;
import javax.swing.*;

public class QuitterJFrame extends JFrame {
  //Overridden so we can exit when window is closed
  protected void processWindowEvent(WindowEvent e) {
    super.processWindowEvent(e);
    if (e.getID() == WindowEvent.WINDOW_CLOSING) {
      System.exit(0);
    }
  }
}
```

A remote class

```
package rmiDynamicLoadingDemo;
public interface Executable {
  public void exec();
}
```

```
package rmiDynamicLoadingDemo;
import javax.swing.*;
import java.awt.*;
public class NetworkApp implements Executable {
  JFrame f;
  public NetworkApp(QuitterJFrame f) {
    this.f = f;
  };
  public void exec() {
    f.setBackground(Color.DARK_GRAY);
    f.setForeground(Color.white);
    JLabel l = new JLabel("Latest version of your application.",
                          JLabel.CENTER);
    f.getContentPane().add("Center",l);
    f.pack();
    f.repaint();
  }
}
```

Caricatore dinamico-1

```
package rmiDynamicLoadingDemo;
import javax.swing.*;
import java.net.URL;
import java.rmi.RMISecurityManager;
import java.rmi.server.RMIClassLoader;
import java.lang.reflect.*;
import java.security.Permission;

public class ExecutableLoader {
  public static void main(String args[]) {
    System.setSecurityManager(
      new RMISecurityManager() {
        public void checkPermission(Permission p){}
      });;
```

Caricatore dinamico-2

```
    JFrame cf = new QuitterJFrame();
    cf.setTitle("NetworkApp");

    // download a class from the net, and create an instance of it
    try {
      URL url = new URL("http://latemar.science.unitn.it/java/");
      Class cl =
          RMIClassLoader.loadClass(
                  url,"rmiDynamicLoadingDemo.NetworkApp");

      Class argTypes[] = {cf.getClass()};
      Object argArray[] = {cf};

      // create an instance of cl using constructor cntr
      Constructor cntr = cl.getConstructor(argTypes);
      Executable client = (Executable)cntr.newInstance(argArray);

      client.exec();
      cf.show();
    } catch (Exception e) {e.printStackTrace();}
  }
}
```