Agent and **O**bject **T**echnology **Lab**
Dipartimento di Ingegneria dell’Informazione
Università degli Studi di Parma

Distributed and Agent Systems

RMI

Prof. Agostino Poggi

What is RMI?

- Supports communication between different Java virtual machines, potentially across the network

- Allows an object running in one Java virtual machine to invoke methods on an object running in another Java virtual machine

- Its acronym means Remote Method Invocation

RMI Features

- Locate remote objects
  - Applications either can obtain references to remote objects from a naming facility, the RMI registry, or can pass and return remote object references as part of other remote invocations
- Communicate with remote objects
  - Details of communication handled by RMI
  - To the programmer, remote communication looks similar to regular Java method invocations
- Load class definitions for remote objects
  - RMI provides mechanisms for loading an object's class definitions as well as for transmitting an object's data

Remote Interface

- RMI does not have a separate IDL
  - RMI is used between Java virtual machines
  - Java already includes the concept of interfaces
- An Interface to be a remote interfaces needs to extend the interface **Java.rmi.Remote**
- All methods in a remote interface must be declared to throw the **java.rmi.RemoteException** exception

Remote Class

- A class to be a remote class needs to implement a remote interface

- Most often a remote class also extends the library class **java.rmi.server.UnicastRemoteObjec**t
  - This class includes a constructor that exports the object to the RMI system when it is created, thus making the object visible to the outside world

- A common convention is to name the remote class appending *Impl* to the name of the corresponding remote interface

Remote Method

- A client can request the execution of all those methods that are declared inside a remote interface

- A client request can contain some parameters

- Parameters can be of three types:
    - Atomic types
    - Objects
    - Remote objects

Parameter Passing

	Atomic Type	**Local Object**	**Remote Object**
Local Call	Call by Value	Call by Reference	Call by Reference
Remote Call	Call by Value	Call by Value	Call by Reference

Serialization

- Take any object and turns it into a sequence of bytes that can be stored into a file and later can be fully restored into the original object
- A serializable object either must implement the java.io.Serializable interface or must extend a class that extends it
- Primitive data are serializable, but static and transient data are not serialized
  - Static data are not in the object state
  - Transient data are data of which serialization is not wanted

Serialization

```
public class  ProductDescription implements Serializable {
  public static int counter;
  private string name;
  private float price;
  private Geometry geo;
  private transient Image image;
}
```

Write an Object

- Create a file to store the object

```
OutputStream os = new FileOutputStream(“c:\myClass”);
```

- Create ObjectOutputStream

```
ObjectOutputStream oos = new ObjectOutputStream(os);
```

- Write object into the file

```
MyClass instance = new MyClass();
....
oos.writeObject(instance);
```

- Close file and ObjectOutputStream

```
oos.close();
os.close();
```

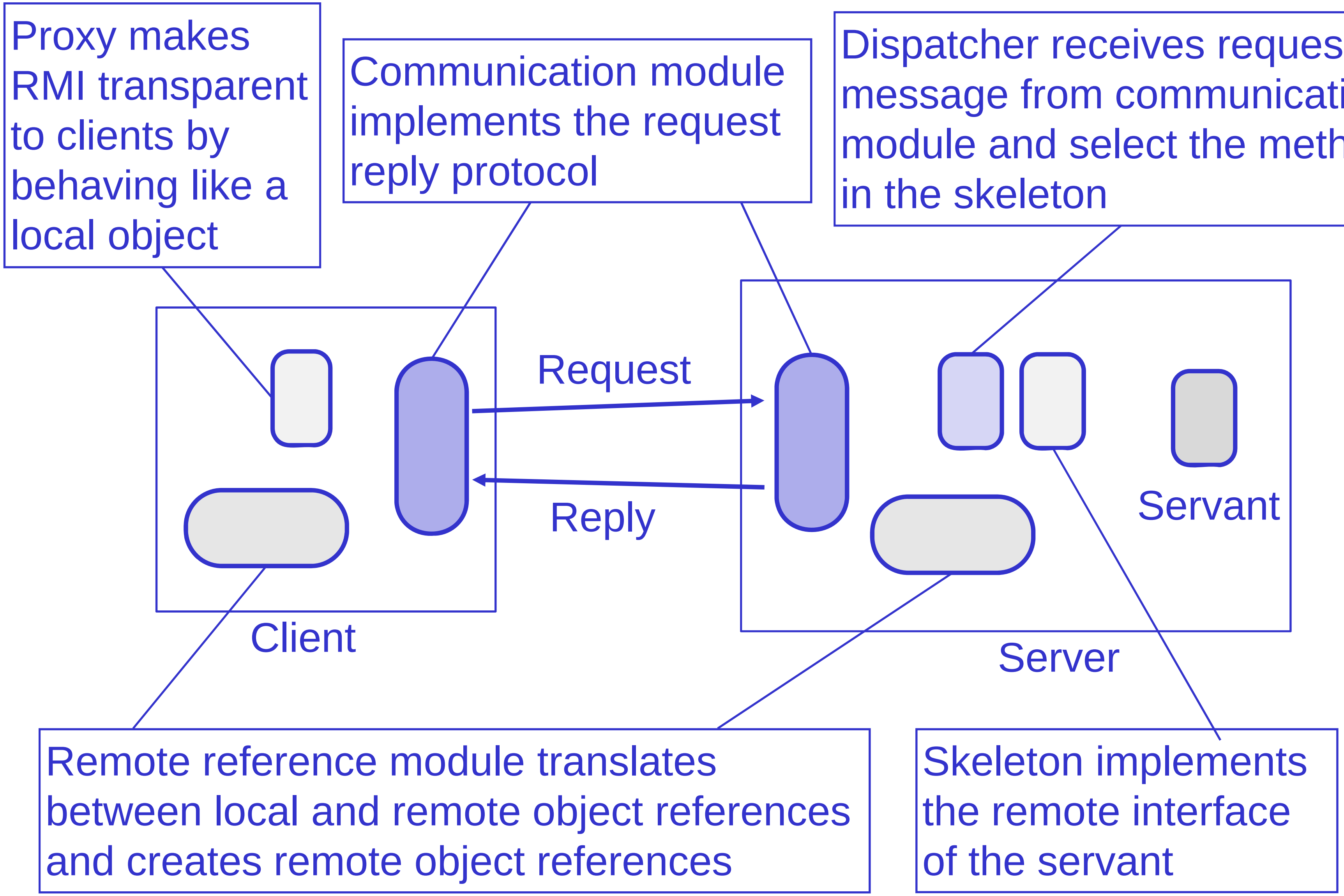
Read an Object

- Open the file that stores the object
  - InputStream is = new FileInputStream(“c:\myClass”);
- Create ObjectInputStream
  - ObjectInputStrem ois = new ObjectInputStream(is);
- Read the object into the file
  - MyClass instance = (MyClass) ois.readObject();
- Close file and ObjectOutputStream
  - ois.close();
  - is.close();

RMIC

- It is the compiler that generates stubs used to connect remote objects
  - Its input is a remote implementation class
  - Its output is a new class that implements the same remote interfaces as the input class
- Such new class supports the interaction with a remote object whose Internet address is stored in the stub instance through a set of methods for
  - Sending arguments to the remote object
  - Receiving results from the remote object

Application Components



Proxy makes RMI transparent to clients by behaving like a local object

Communication module implements the request reply protocol

Dispatcher receives request message from communication module and select the method in the skeleton

Remote reference module translates between local and remote object references and creates remote object references

Skeleton implements the remote interface of the servant

Remote Object Binding

- Explicit by using static methods in the java.rmi.Naming class
    - Server binds/rebinds name with the rmiregistry
    - Client looks up name with the rmiregistry
- Implicit by receiving a remote object reference
    - Client can invoke methods on it as if it were a local interface

RMI Registry

- Runs on each machine, which hosts remote objects and accepts requests for services

- Clients use the registry to find the remote objects

```
MyClass instance =
   (Myclass) Naming.lookup(“rmi://localhost/myclass”);
MyClass instance =
   (Myclass)
   Naming.lookup(“rmi://localhost:2001/myclass”);
```

- Its default TCP/IP port is 1099

RMI Registry

- *void rebind (String name, Remote obj)*
  - Registers the identifier of a remote object by name
- *void bind (String name, Remote obj)*
  - Registers a remote object by name, but if the name is already in the registry an exception is thrown
- *void unbind (String name, Remote obj)*
  - Removes a binding
- *Remote lookup(String name)*
  - Looks up a remote object by name
- *String [] list()*
  - Returns the names bound in the registry

Reflection

- Reflection enables Java code to:
  - Discover information about the fields, methods and constructors of loaded classes
  - Use reflected fields, methods, and constructors to operate on the corresponding class instances
- RMI uses reflection for the execution of a remote call

Reflection

- Get a class

```
Class c = Class.forName("com.example.MyClass")
```

- Get fields and methods

```
Field[] fs      = c.getFields();
Method[] ms = c.getMethods();
```

- Get parameters and return type

```
Class<?>[] pts = ms[0].getParameterTypes();
Class<?> rt     = ms[0].getReturnType();
```

Remote Call Execution

- Proxy has to marshal information about a method and its arguments into a request message
  - An object of class **Method**
  - An array of objects for the method’s arguments
- The dispatcher
  - Obtains the remote object reference
  - Unmarshals the **Method** object and its arguments
  - Calls the **invoke** method on the object reference and array of arguments values
  - Marshals the result or any exceptions into the reply message

Remote Object Code

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface  MessageWriter  extends Remote {
void writeMessage(String s) throws RemoteException;
}
```

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class MessageWriterImpl
    extends UnicastRemoteObject
    implements MessageWriter {
  public MessageWriterImpl() throws RemoteException { … }
  public void writeMessage(String s) throws RemoteException {
    System.out.println(s);
      }
  }
```

Server Program Code

creates a remote object with local name server

```
import java.rmi.Naming;
public class HelloServer {
  public static void main(String [] args) throws Exception {
    MessageWriter server = new MessageWriterImpl();
    Naming.rebind(“MessageWriter”, server);
   }
}
```

Publishes a remote reference to that object with external name *MessageWriter*

Client Program Code

Looks up a reference to a remote object with external name *MessageWriter*, and stores the returned reference with local name server

```
import java.rmi.Naming;

public class HelloClient {
  public static void main(String [] args) throws Exception {
    MessageWriter server =
     (MessageWriter) Naming.lookup(“rmi://localhost/MessageWriter”);
    server.writeMessage(“Hello, other world”) ;
   }
}
```

Invokes the remote method, **writeMessage()**, on server

Compiling and Running

```
javac *.java
rmic MessageWriterImpl
```

```
start rmiregistry
java HelloServer
java HelloClient
```

Distributed Garbage Collection

- The aim of a distributed garbage collector are:
  - Retain the local and remote objects when it is still be referenced
  - Collect the objects when none holds reference to them
- RMI garbage collection algorithm is based on reference counting
  - Server maintain processes set that hold remote object references to it
  - Client notify server to modify the process set
  - When the process set becomes empty, server local garbage collector reclaims the space

Explicit Notification

```
import java.rmi.server.UnicastRemoteObject;
Import java.rmi.server.Unreferenced;
public class RemoteServerImpl
    extends UnicastRemoteObject
    implements MyRemoteInterface, Unreferenced {
  public RemoteServerImpl() {
    super();
    . . .
    //allocate resources
  }
  . . .
  public void unreferenced() {
    //deallocate resources here
  }
}
```

E.g., network and database connections can be released

Remote Class

- RMI can exchange objects through serialization, but serialized objects do not contain the code of the class they implement
- In fact, de-serialization process can be completed only if the client has available the code of the class to be de-serialized
- Class code can be provided by manually copy implementation class files to the client
- A more general approach is to publish implementation class files on a Web Server

Dynamic Class Loading

- Remote object's codebase is specified by setting the **java.rmi.server.codebase** property

```
java  –Djava.rmi.server.codebase=http://www/  HelloServer
java  –Djava.rmi.server.codebase=http://www/myStub.jar  HelloServer
```

- Client requests a reference to a remote object
- Registry returns a reference (the stub instance) to the requested class
- If the class definition for the stub instance is found locally in the client's CLASSPATH
    - Then it loads the class definition
    - Else it retrieves the class definition from the remote object's codebase

Security Manager and Policies

- Security is a serious concern since executable code is being downloaded from a remote location
- RMI normally allows the loading of code only from the local CLASSPATH
- RMI allows the loading of code from a remote location only if a suitable security manager is set and an appropriate security policy is defined
- RMI clients usually need to install security manager because they need to download stub files
- RMI servers usually do not need it, but it is still good practice to install it anyway

Security Manager and Policies

- A security manager can be set as follows:

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

- A security policy can be defined in a plain text file:

```
grant { permission  java.security.AllPermission  “”, “” ; } ;
```

- And assigned to the client as follows:

```
java  –Djava.security.policy=rmi.policy  HelloClient
```

Activatable Objects

- Activatable objects are remote objects that are created and execute "on demand", rather than running all the time

- Activatable objects are important
  - Remote objects could fail or could be shut down inadvertently or intentionally
  - Remote objects use a set of resources for all their life even if they are used few times

Remote Object Code

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface  MessageWriter  extends Remote {
  void writeMessage(String s) throws RemoteException;
}
```

```
import java.rmi.Remote;
import java.rmi.activation.Activatable;
public class MessageWriterImpl
    extends Activatable implements MessageWriter {
  public MessageWriterImpl(ActivationID id, MarshalledObject data)
    throws RemoteException { super(id, 0); … }
  public void writeMessage(String s) throws RemoteException {
              System.out.println(s);
       }
  }
```

Server Main Method Duties

- Install security manager for the ActivationGroup of the Java virtual machine.
- Set a security policy
- Create an ActivationGroupDesc instance
- Register it and get an ActivationGroupID
- Create an ActivationDesc instance
- Register it with rmid
- Bind or rebind the remote object instance with its name
- Exit the system

Server Main Method Code

```
public static void main(String[] args) throws Exception {
  System.setSecurityManager(new RMISecurityManager());
  Properties props = new Properties();
  props.put("java.security.policy", "/myrmi/rmi.policy");
  ActivationGroupDesc.CommandEnvironment ace = null;
  ActivationGroupDesc eg = new ActivationGroupDesc(props, ace);
  ActivationGroupID agi =  ActivationGroup.getSystem().registerGroup(eg);
  String location = "file:/myrmi/";
  MarshalledObject data = null;
  ActivationDesc desc =
    new ActivationDesc(agi, “MessageWriter“, location, data);
  MessageWriter server = (MessageWrite) Activatable.register(desc);
  Naming.rebind(“messageservice”, server);
  System.exit(0);
}
```

Compiling and Running

```
javac *.java
rmic MessageWriterImpl
```

```
start rmiregistry
start rmid –J-Djava.security.policy=rmi.policy
java -Djava.security.policy=rmi.policy HelloServer
java -Djava.security.policy=rmi.policy HelloClient
```

Callbacks

- A callback server's action of notifying clients about an event implementation
  - Improve performance by avoid constant polling
  - Delivery information in a timely manner
- In a RMI based system callbacks are implements as follows:
  - Client create a remote object
  - Client pass the remote object reference to the server
  - Whenever an event occurs, the server calls the client via the remote object