Computer Network Programming

Remote Invocation

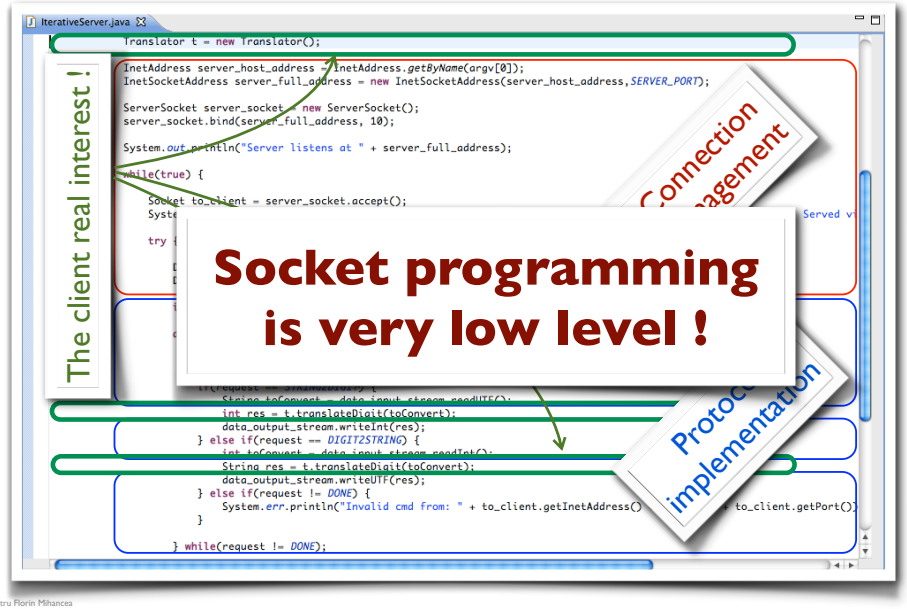
Dr. Petru Florin Mihancea

1

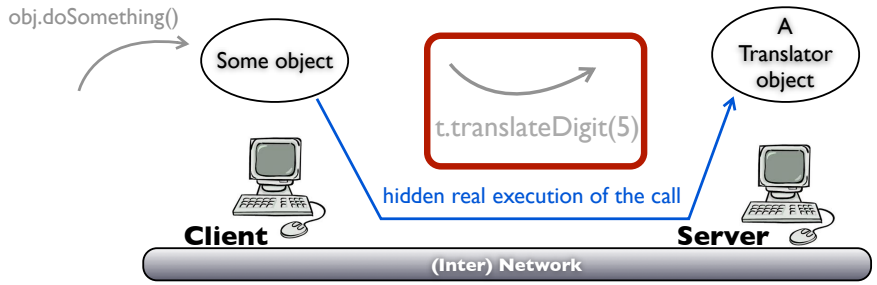
The Problem

and the Solution Principle

Why?



What We Want?



But using the same code...

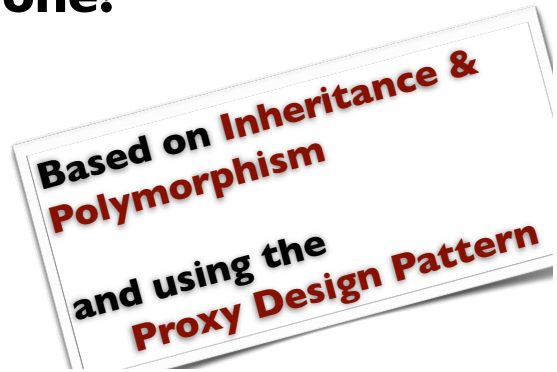
```
// in a non network program
public void doSomething() {
  ...
  t.translateDigit(5);
  ...
}
```

... even if the objects are in diffe

Done by Remote Procedure Call infrastructures and, in particular, by Java RMI

But ...

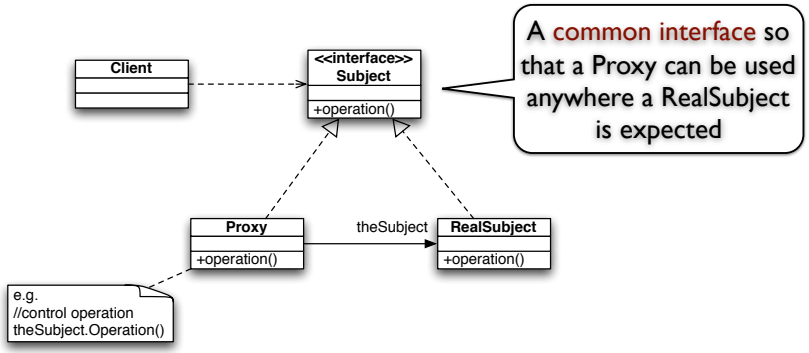
... how can one "fool" the Java client code to think that it invokes a local object and not a remote one?



Proxy Pattern in a Nutshell

Intent & Structure

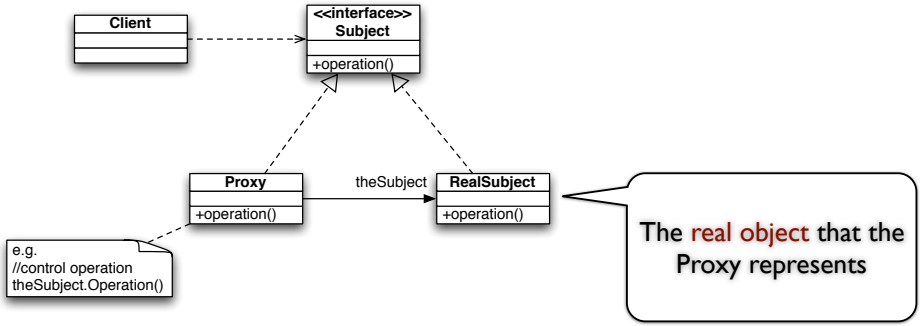
Provides a surrogate or placeholder for another object to control access to it



Proxy Pattern in a Nutshell

Intent & Structure

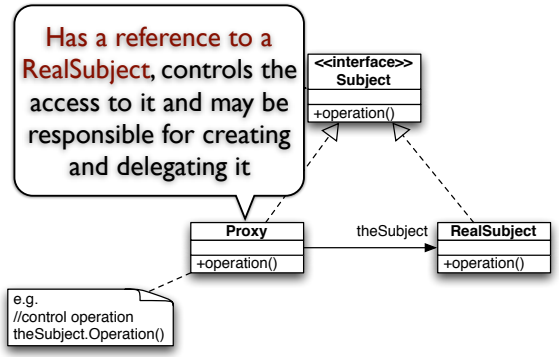
Provides a surrogate or placeholder for another object to control access to it



Proxy Pattern in a Nutshell

Intent & Structure

Provides a surrogate or placeholder for another object to control access to it

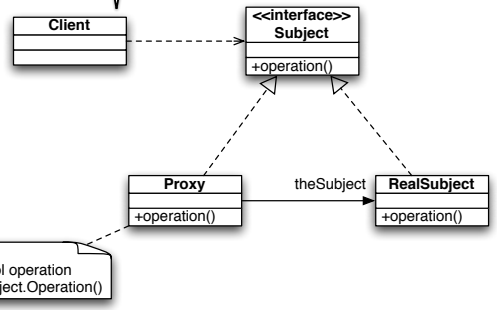


Proxy Pattern in a Nutshell

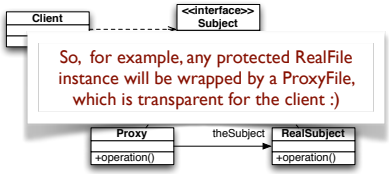
& Structure

les a surrogate or placeholder for ther object to control access to it

Client uses the Subject and so it can actually work with a Proxy and not with a RealSubject



Proxy Pattern in a Nutshell (2)



So, for example, any protected RealFile instance will be wrapped by a ProxyFile, which is transparent for the client :)

```
interface File { //the Subject
    ...
    boolean write(String data);
    String read();
}
```

```
class RealFile implements File { //the RealSubject
    ...
    public boolean write(String data) {
        //writes data on disk; returns true on success
    }
    public String read() {
        //reads data from disk and returns it
    }
}
```

```
class ClientOne { //a Client and we have
                  //many other Client classes
    ...
    public void display(File x) {
        String data = x.read();
        ...
    }
    public void addCommentIn(File x) {
        ...
        boolean res = x.write(data);
        ...
    }
}
```

Write is password protected for some clients, in some contexts, for some files etc.

```
class ProxyFile implements File {
    private RealFile theSubject;
    public ProxyFile(RealFile f) { theSubject = f;}
    pubic String read() {
        return theSubject.read();
    }
    public boolean write(String data) {
        //check the password if not done yet
        if(ok) return theSubject.write(filteredData);
        else return false;
    }
    ...
}
```


Proxy Pattern in a Nutshell (3)

Consequences

Transparent indirection has many uses:

1. **protection proxy can check for access permissions**
2. **virtual proxy can perform optimizations (e.g. lazy creation)**
3. **remote proxy can hide the fact that an object is in a different address space**

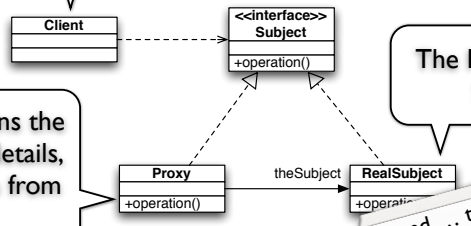
Proxy Pattern in a Nutshell (3)

Consequences

Tr has many uses:

1. **heck for access permissions**
2. **rm optimizations (e.g. lazy creation)**
3. **e the fact that an object is in a**

The Client is defined in terms of Subject and will actually work with a Proxy



The Proxy contains the communication details, which are hidden from the Client

The RealSubject is on Server side

and … the proxy uses features provided by an underlying communication infrastructure and can be generated automatically :)

2

Basics of Java RMI Middleware

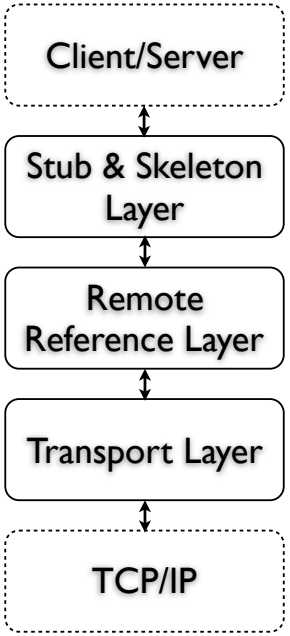
Some Terminology

Remote object
an object whose methods can be invoked from another process

Remote interface
an interface that declares the methods that can be invoked remotely

Remote reference
a reference to a remote object

RMI Infrastructure Architecture

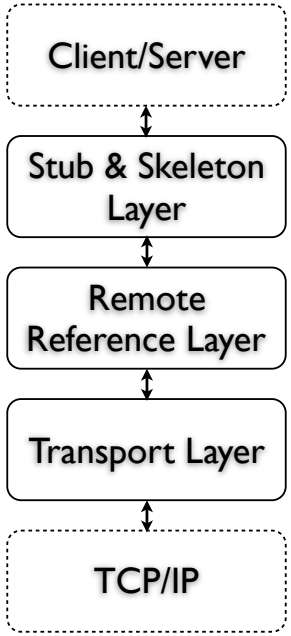


Hides the low-level details from the actual client & server

Stub (or Proxy)
A representative for a remote object (from server) on the client-side that:
- is referred by client objects with local references
- implements the same remote interface as the remote object
- knows how to forward the calls + data

Skeleton
A representative for a client on the server-side that:
- has a local reference to the remote object
- prepares the data and calls the method on the real remote object
- forwards back the result to the client side

RMI Infrastructure Architecture

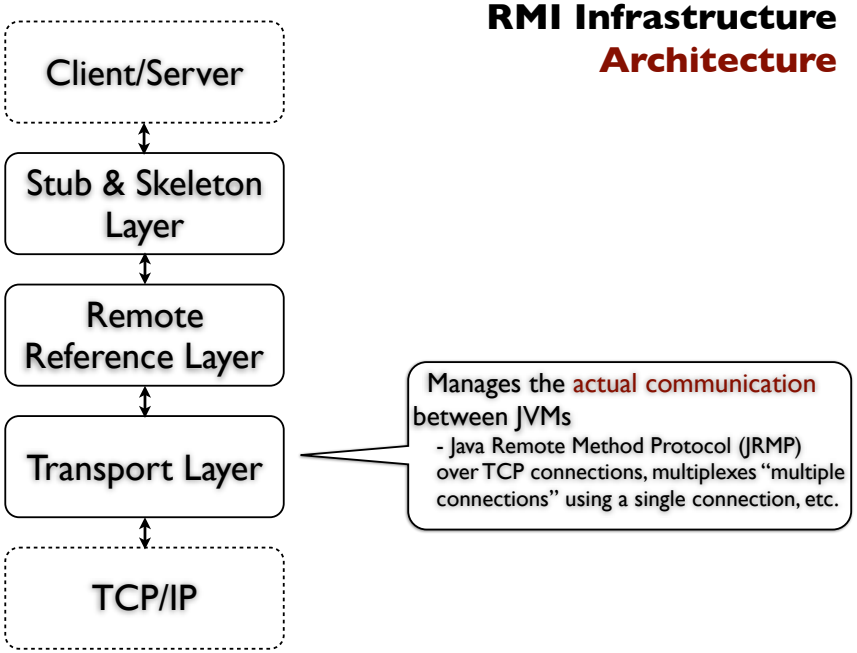


Manages remote references and thus, also supports distributed garbage collection activities

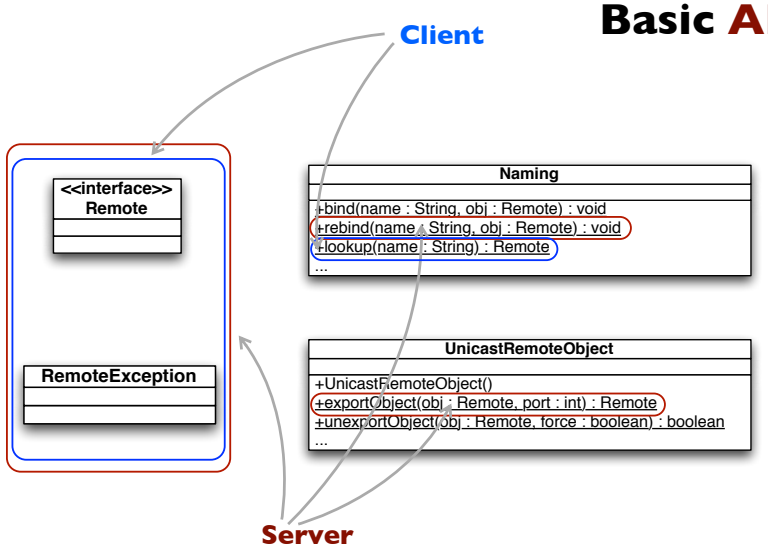
Invocation semantics: unicast point-to-point (i.e., for a remote call, invoke a single remote object)

Support for activation (i.e., a remote object might not be in execution and if so, it is instantiated and its state is restored form disk)

RMI Infrastructure Architecture



Basic API

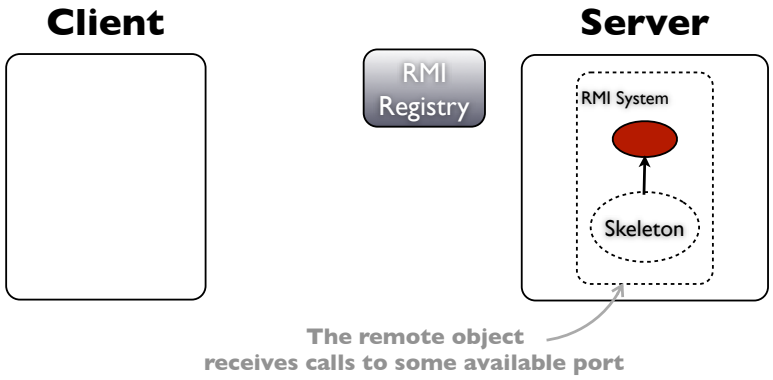


How Does It Work?

```
...
Remote obj =  ... //instantiate the remote object
```

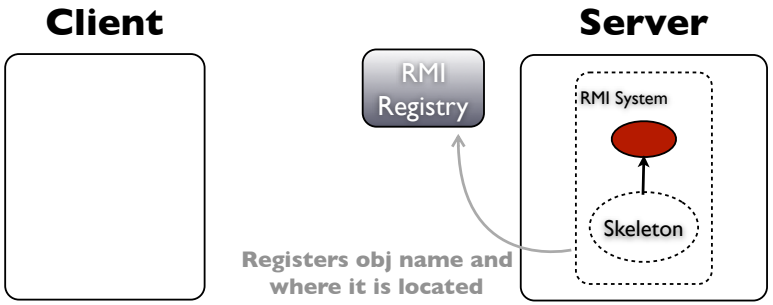
How Does It Work?

```
...
Remote obj =  ... //instantiate the remote object
obj = UnicastRemoteObject.exportObject(obj,0);
```



How Does It Work?

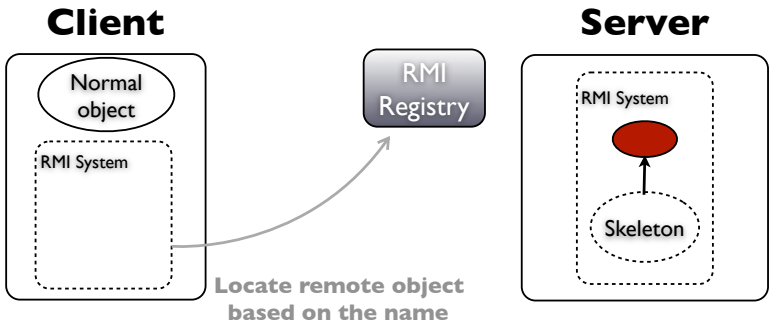
```
...
Remote obj =  ... //instantiate the remote object
obj = UnicastRemoteObject.exportObject(obj,0);
Naming.rebind("rmi://localhost:1099/TheName", obj);
```



How Does It Work?

```
...
Remote obj =
    Naming.lookup("rmi://address:port/TheName");
```

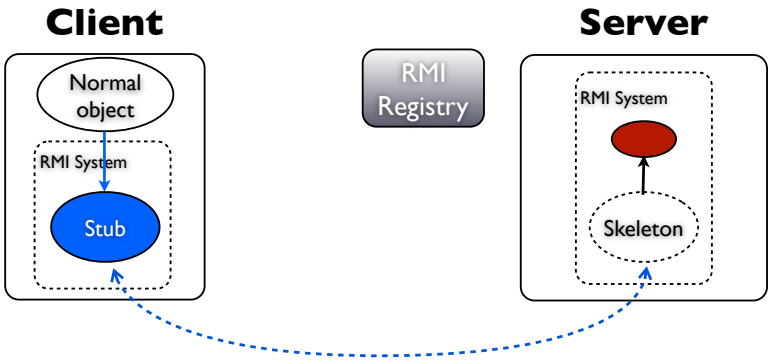
```
...
Remote obj =  ... //instantiate the remote object
obj = UnicastRemoteObject.exportObject(obj,0);
Naming.rebind("rmi://localhost:1099/TheName", obj);
...
```



How Does It Work?

```
...
Remote obj =
    Naming.lookup("rmi://address:port/TheName");
//Usually a downcast to the specific remote interface
//and calls to its methods (exec. by the remote object)
```

```
...
Remote obj =  ... //instantiate the remote object
obj = UnicastRemoteObject.exportObject(obj,0);
Naming.rebind("rmi://localhost:1099/TheName", obj);
...
```



Development Steps

1. Define the remote interface
2. Implement the remote object
3. Generate stub and skeleton classes
4. Implement the server
5. Implement the client
6. Deploy & run the application

Example

Server with a remote object that:

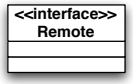
1. Convert an int representing a digit into the corresponding string
   e.g. 1 - "one", 2 - "two", 11 - "Not a digit!"

2. The reverse
   e.g. "one" - 1, "zer" - -1

Client

Uses these operations

1. Define the Remote Interface



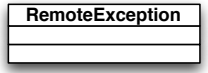
Remote interfaces must extend the java.rmi.Remote interface

```
interface TranslatorRMIInterface extends Remote {

    public String translateDigit(int x) throws RemoteException;

    public int translateDigit(String x) throws RemoteException;

}
```

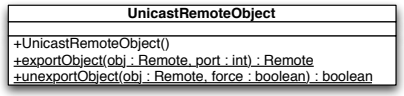


Each method must declare that it might throw java.rmi.RemoteException

2. Implement the Remote Object

A. Must implement the previously declared remote interface

B. May additionally extend java.rmi.server.UnicastRemoteObject




```
class TranslatorRMIImplementation implements TranslatorRMIInterface {
    ...
    public String translateDigit(int x) throws RemoteException {
        ...
    }
    ...
    public int translateDigit(String x) throws RemoteException {
        ...
    }
    ...
}
// in this case (A) remote object export into the RMI system must
//be explicit using UnicastRemoteObject.exportObject(Remote, int)
```


3. Generate Stub and Skeleton

Possibly optional step

- **Skeletons generation required only in Java 1.1**
- **Stub classes can be generated at runtime starting with Java 1.5**
- **Optionally or for backward compatibility they must be generated**
  - **rmic compiler from JDK**

rmic -d folder -classpath paths full_class_name

folder - where to generate the sub/proxy classes

paths - are the root folders where the class files are located and/or paths to the jar files (so the previous code must be compiled)

full_class_name - name of the class (not file) implementing the remote object

By default only stub classes are generated (i.e. > Java 1.1)

3. Generate Stub and Skeleton

TranslatorRMIImplementation_Stub.java

```java
// Stub class generated by rmic, do not edit.
// Contents subject to change without notice.

public final class TranslatorRMIImplementation_Stub
    extends java.rmi.server.RemoteStub
    implements TranslatorRMIInterface
{
    private static final long serialVersionUID = 2;

    private static java.lang.reflect.Method $method_translateDigit_0;
    private static java.lang.reflect.Method $method_translateDigit_1;

    static {
    try {
        $method_translateDigit_0 = TranslatorRMIInterface.class.getMethod("translateDigit", new java.lang.Class[] {
        $method_translateDigit_1 = TranslatorRMIInterface.class.getMethod("translateDigit", new java.lang.Class[] {
    } catch (java.lang.NoSuchMethodException e) {
        throw new java.lang.NoSuchMethodError(
        "stub class initialization failed");
    }
    }

    // constructors
    public TranslatorRMIImplementation_Stub(java.rmi.server.RemoteRef ref) {
    super(ref);
    }
```

Line: 1 Column: 1 Java Tab Size: 4

4. Implement the Server

A. Export the object if its class does not extend UnicastRemoteObject

UnicastRemoteObject
+UnicastRemoteObject()
+exportObject(obj : Remote, port : int) : Remote
+unexportObject(obj : Remote, force : boolean) : boolean
...

B. Register the remote object in the registry service

Naming
+bind(name : String, obj : Remote) : void
+rebind(name : String, obj : Remote) : void
+lookup(name : String) : Remote
...

The Name

rmi://<name_service_host> [:<name_service_port>]/<obj_name>

```java
public class RMIServer {
  public static void main(String argv[]) {
    ...
    TranslatorRMIInterface engine = new TranslatorRMIImplementation();
    TranslatorRMIInterface obj = (TranslatorRMIInterface)
        UnicastRemoteObject.exportObject(engine, 0);
    Naming.rebind("rmi://localhost:1099/Translator", obj);
    ...
  }
}
```


5. Implement the Client

Lookup for the remote object based on its name

Naming
+bind(name : String, obj : Remote) : void
+rebind(name : String, obj : Remote) : void
+lookup(name : String) : Remote
...

```java
public class RMIClient {
  ...
  public static void main(String argv[]) {
    ...
    TranslatorRMIInterface trans =
        (TranslatorRMIInterface) Naming.lookup("rmi://localhost:1099/Translator");
    ...
    int result_int = trans.translateDigit("five");
    String result_string = trans.translateDigit(0);
    ...
  }
}
```


6. Deploy the Application

Client, Server & RMIRegisty on different machines
but which classes (their bytecode) go where?

Client JVM

1. Classes on which the code depends statically including the remote interface & other classes used in the interface declaration and the other local classes
2. Classes seen only at runtime including the stub and other subtypes of objects passed-in by server via a remote call

Server JVM

1. Classes on which the code depends statically including the remote interface & other classes used in the interface declaration and the other local classes including the remote object implementation
2. Classes seen only at runtime including the stub, the skeleton and other subtypes of objects passed-in by client via a

Solution 1 (Closed configuration)
Copying each class file where required

Solution 2 (Dynamic configuration)
With some code downloading at runtime

Closed configuration

6. Run the Application (2)

I. Start the rmiregistry application (found in JDK)

rmiregistry -J-Djava.class.path=paths_to_remote_interface_related_classes_and_stub

II. Start the server

java -cp paths_to_remote_interface_related_classes : paths_to_local_classes :
paths_to_client_classes_seen_only_at_runtime_by_server_to_stub_and_to_skel
-Djava.rmi.server.hostname=host_where_the_remote_object_run
Server

III. Start the client

java -cp paths_to_remote_interface_related_classes : paths_to_local_classes :
paths_to_sever_classes_seen_only_at_runtime_by_client_and_to_stub
Client

Independently on configuration, it is also possible for a server to create a name service removing step I - see java.rmi.registry.LocateRegistry

Dynamic configuration
assuming here only clients need to download code but, in general, the arguments are used symmetrically

6. Run the Application (3)

I. Start the rmiregistry application (found in JDK)

rmiregistry -J-Djava.rmi.server.useCodebaseOnly=false (to enable code download)

II. Start the server

java -cp paths_to_remote_interface_related_classes : paths_to_local_classes :
paths_to_skel
-Djava.rmi.server.hostname=host_where_the_remote_object_run
-Djava.rmi.server.codebase="URLs_to_sever_classes_seen_only_at_runtime_by_client_and_to_stub URLs_to_remote_interface_related_classes"
Server

URLs
e.g., http://... , file://...
Do not forget the final / for codebase folders

III. Start the client

java -cp path_to_remote_interface_related_classes : path_to_
-Djava.rmi.server.useCodebaseOnly=false (to enable code download)
-Djava.security.manager (required when downloaded code is executed)
-Djava.security.policy=policy_file
Client

Policy file example; setting fine grained permission is recommended in production

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


3

Beyond RMI Basics

A

RMI & Threads

there are NO guarantees with respect to mapping remote object invocations to threads

remote object implementation must ensure the thread safety property

Do you remember synchronized?

B

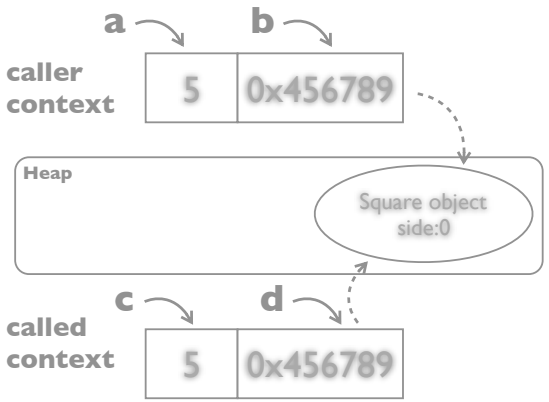
Parameter Passing

Non-remote invocations

```
void caller() {
    int a = 5;
    Square b = new Square();
    b.setSide(5);
    called(a, b);
```

```
void called(int c, Square d) {
```

passed by value



B

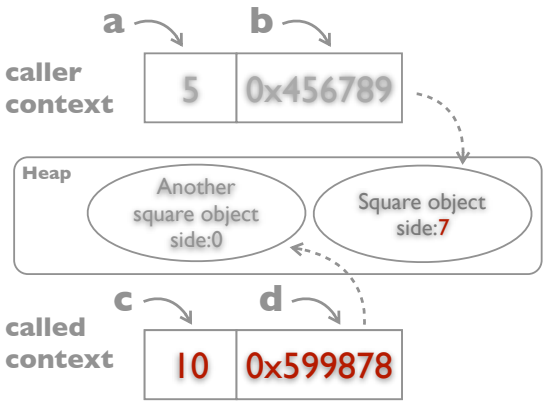
Parameter Passing

Non-remote invocations

```
void caller() {
    int a = 5;
    Square b = new Square();
    b.setSide(5);
    called(a, b);
```

```
void called(int c, Square d) {
    c = 10;
    d.setSide(7);
    d = new Square();
}
```

passed by value



B

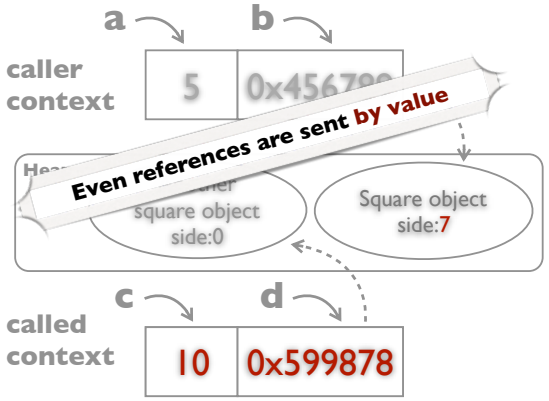
Parameter Passing

Non-remote invocations

```
void caller() {
    int a = 5;
    Square b = new Square();
    b.setSide(5);
    called(a, b);
    //a is still 5
    //b is the same square object
    //side of square referred by b is 7!
}
```

```
void called(int c, Square d) {
    c = 10;
    d.setSide(7);
    d = new Square();
}
```

passed by value



B

Parameter Passing

Remote invocations

"a different sort of passed by value"

Primitive type parameters

similarly, their value is transmitted

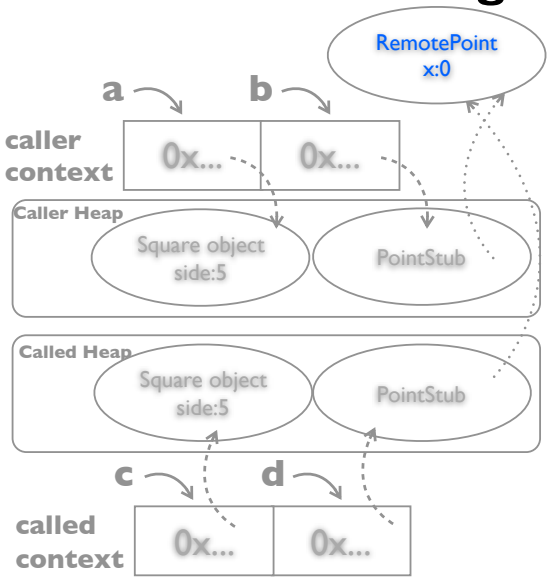
Reference type parameters

passing references has a different semantics

B Remote Parameter Passing

```
void caller() {
    Square a = new Square();
    a.setSide(5);
    RemotePoint b = Naming...
    b.setX(0);
    someRemoteObj.called(a, b);
```

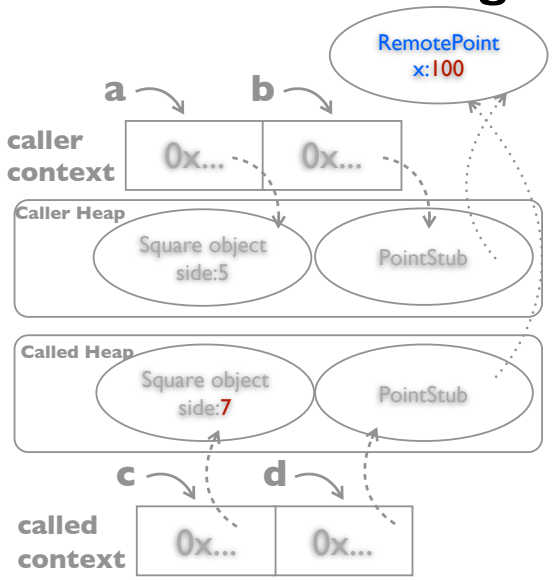
```
//This is the called remote object
void called(Square c, RemotePoint d) {
```



B Remote Parameter Passing

```
void caller() {
    Square a = new Square();
    a.setSide(5);
    RemotePoint b = Naming...
    b.setX(0);
    someRemoteObj.called(a, b);
```

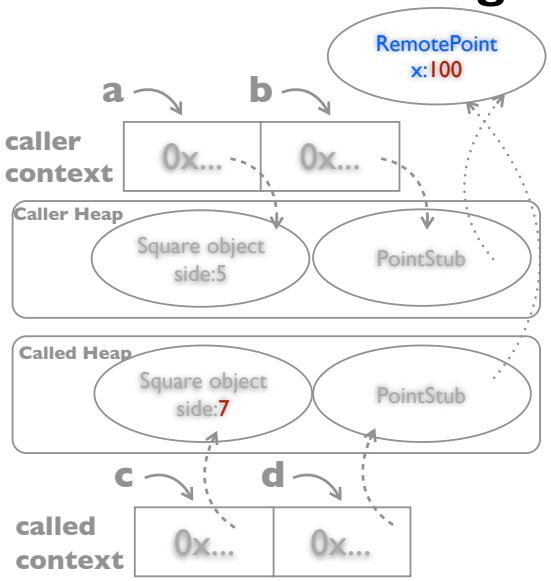
```
//This is the called remote object
void called(Square c, RemotePoint d) {
    c.setSide(7);
    d.setX(100);
}
```



B Remote Parameter Passing

```
void caller() {
    Square a = new Square();
    a.setSide(5);
    RemotePoint b = Naming...
    b.setX(0);
    someRemoteObj.called(a, b);
    //a, b refers the same objects
    //side of a is 5 !
    //x of b is 100
}
```

```
//This is the called remote object
void called(Square c, RemotePoint d) {
    c.setSide(7);
    d.setX(100);
}
```



B Parameter Passing

Remote invocations

"a different sort of passed by value"

Primitive types parameters
- values are transferred

Remote reference parameters
- the stub objects are copied

Other reference parameters
- the objects are copied

NOT every object can be copied! Do you remember object serialization?

B

Parameter Passing

RMI referential integrity

"If two references to an object are passed from one JVM to another JVM in parameters (or in the return value) in a single remote method call and those references refer to the same object in the sending JVM, those references will refer to a single copy of the object in the receiving JVM"

4

Demo Application

Same application but having specific operations for the required transformations in a remote object