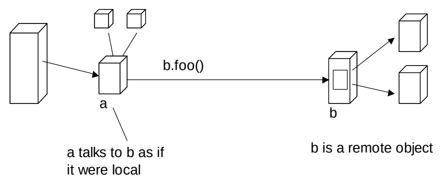
Chapter 3
RMI

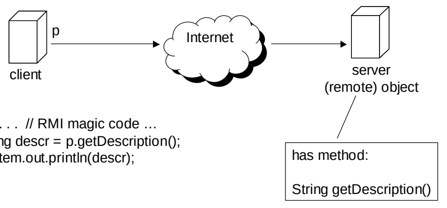
Remote Method Invocation

Java Remote Method Invocation

- Based on the RPC model of cross-platform communication
- GOAL: distributed applications are as easy to program as non-distributed

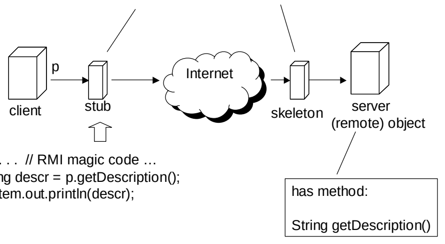


RMI Scenario



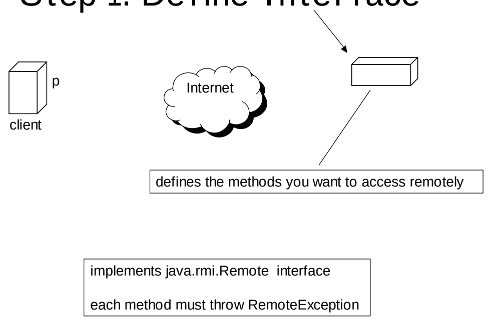
```
p = . . .  // RMI magic code ...
String descr = p.getDescription();
System.out.println(descr);
```

Proxy Objects

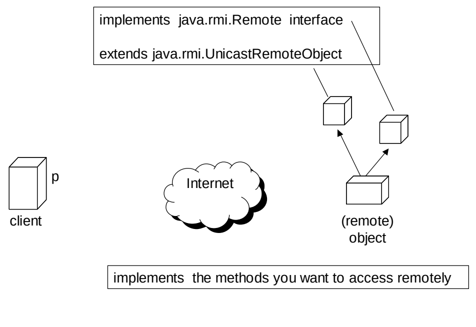


```
p = . . .  // RMI magic code ...
String descr = p.getDescription();
System.out.println(descr);
```

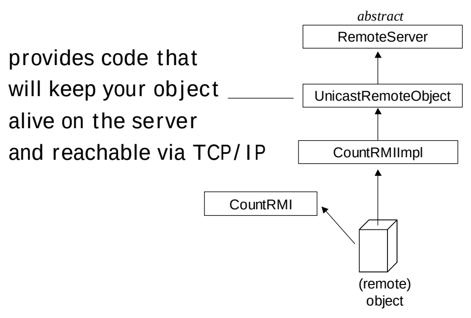
Step 1. Define Interface


p
client
Internet
defines the methods you want to access remotely
implements java.rmi.Remote interface
each method must throw RemoteException

Step 2. Implement the Interface

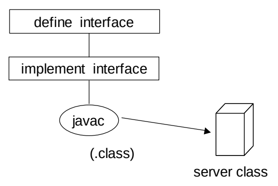

implements java.rmi.Remote interface
extends java.rmi.UnicastRemoteObject
p
client
Internet
(remote) object
implements the methods you want to access remotely

UnicastRemoteObject

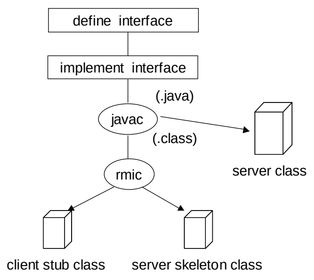


provides code that will keep your object alive on the server and reachable via TCP/IP
abstract
RemoteServer
UnicastRemoteObject
CountRMIImpl
CountRMI
(remote) object

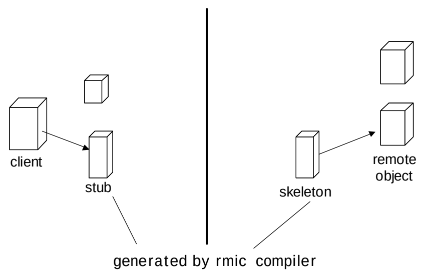
Step 3.Compile the server class



Step 3.Compile the server class



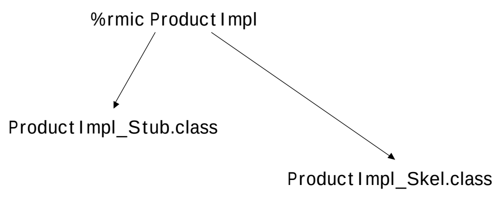
rmic in Action



prompt> rmic classfile

- The server classfile must have been compiled with javac
- rmic uses the CLASSPATH or command line –classpath argument
- Compiled classes can be put in another directory using the –d argument

rmic



Step 5.Start the RMI Registry on your server

- RMI supports a non-persistent naming service
- Allows you to retrieve and register server objects
- prompt>start rmiregistry
  - WIN95: start rmiregistry
  - Unix: rmiregistry &

Step 6.Start Server Objects

- Load server class and create instances of your remote objects

```
public class CountRMIServer {
  public static void main(String[] args) {
     System.setSecurityManager(new RMISecurityManager() );
     try {
        CountRMIImpl myCount =new CountRMIImpl( “myCountRMI ”);
        System.out.println( “RMIServer ready ”);
     }
   }
}
```

Step 7.Register Remote Objects with the Registry

```
public CountRMIImpl(String name) throws RemoteException {
      super();
      try {
         Naming.rebind(name, this);
         sum = 0;
      }
      . . .
}
```

Step 8.Write Client Code

```
CountRMI myCount = (CountRMI)Naming.lookup(:rmi:// ”+args[0]
                                         + “/ ” + “myCountRMI ”);
//set sum to initial value
System.out.println( “setting sum to zero ”);
myCount.sum(0);
```

Step 9.Compile Client Code

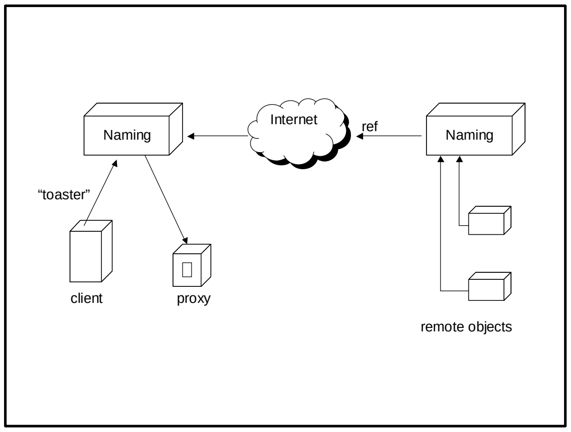
- javac CountRMIClient.java

Step 10.Start the Client

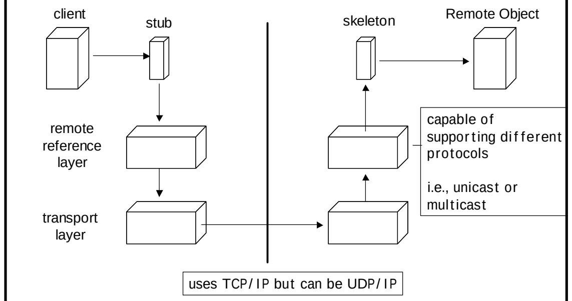
- On client (or in separate DOS window if using local host)
- java CountRMIClient

class: Naming

- An RMI class
- Must live on both client and server machines
- Serves as Lookup service for remote objects
- Remote objects must register with Naming service
- Clients use client-side Naming object to get the appropriate stub reference.



RMI Architecture



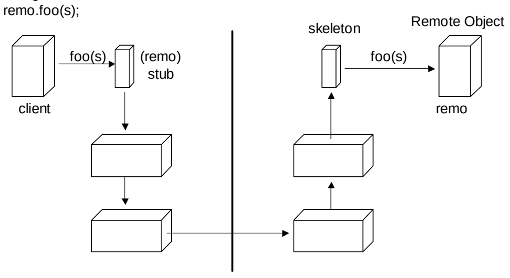
capable of supporting different protocols

i.e., unicast or multicast

uses TCP/IP but can be UDP/IP

Parameter passing in Remote Methods

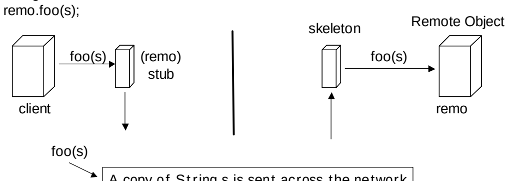
String s="hello"l
remo.foo(s);



socket and stream connection via TCP/IP

Parameter passing in Remote Methods

String s="hello"l
remo.foo(s);



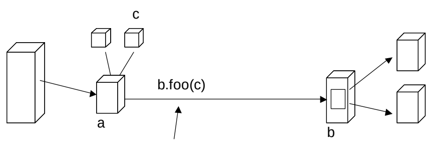
A copy of String s is sent across the network

When an object is passed as a parameter, a COPY is passed, not a reference

(But the object must be serializable)

RMI Difference 1

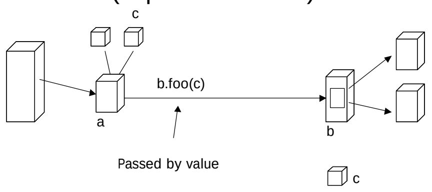
- Objects passed as parameters must be serializable (or be a Remote Object)



must be serializable

RMI Difference 2

- Objects passed as parameters or returned as values are PASSED BY VALUE (copies are made)



Passed by value

RMI Difference 3

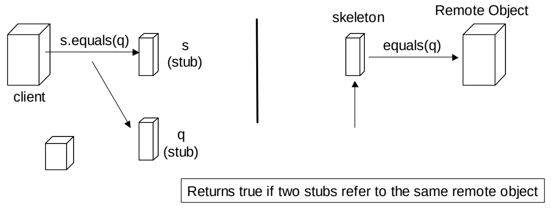
- Remote Objects override
  - equals()
  - hashCode()
  - toString()

equals()

- Default behavior is inherited from Object
- s.equals(q)
  - are s and q pointing to the SAME object
- Many classes override this in order to use Hashtables where
  - hashcode() is used to select a hash bucket
  - equals() is used to match a given object against other objects in the hash table that have the same hashcode

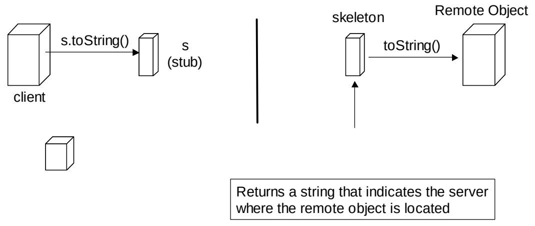
RemoteObject overrides equals()

- s.equals(q)
  - what if s and q are remote objects?



RemoteObject overrides toString()

- s.toString(q)
  - what if s is a remote object?


client
s.toString()
s
(stub)
skeleton
Remote Object
toString()
Returns a string that indicates the server where the remote object is located

END