Distributed Computing

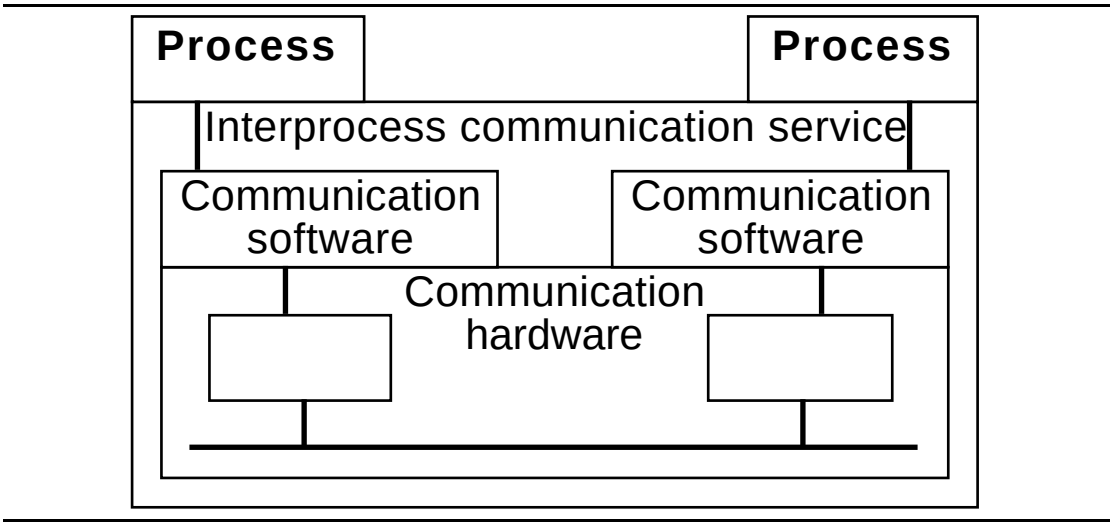
- Computing on many systems to solve one problem
- Why?
  - Combination of cheap processors often more cost-effective than one expensive fast system
  - Flexibility to add according to needs
  - Potential increase of reliability
  - Sharing of resources
- Why not?
  - Security
  - Physical distribution may not match logical distribution of program
  - Programming effort & overhead

Distributed system

- Supports distributed computing
- What should a distributed system provide?
  - Illusion of one system while running on multiple systems
  - Transparancy of resources
- Issues
  - Communication, failure handling, synchronization
  - Protection, security
  - Resource management (allocation of process, devices, memory, re-allocation)
  - Naming (of resources, locating)
  - Data management (files, sharing)
  - Deadlock

Java RMI

- Communicating entities

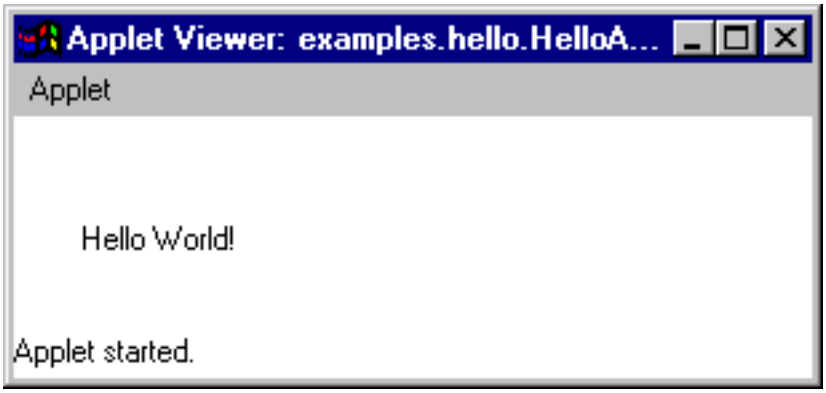


  - Implementing some application for user
  - Using support of distributed services
  - Layers of support
  - Client/server

- Embedded in language Java
  - Object variant of remote procedure call
  - Adds naming compared with RPC
  - Restricted to Java environments

Example

- Hello world (or: a difficult way of saying...)



- Application (applet) code, client

```
public class HelloApplet extends java.applet.Applet
{
  String message = "";

  public void init()
  {
    try {
      Hello obj = (Hello)
        Naming.lookup("//" + getCodeBase().getHost() +
                "/HelloServer");
      message = obj.sayHello();
    } catch (Exception e) {
      System.out.println("HelloApplet: an exception
                occurred:");
      e.printStackTrace();
    }
  }

  public void paint(Graphics g) {
    g.drawString(message, 25, 50);
  }
}
```

Example (cont'd)

- Server code

```
public class HelloImpl
     extends UnicastRemoteObject
     implements Hello
{
   private String name;

   public HelloImpl(String s) throws
                java.rmi.RemoteException {
     super();
     name = s;
   }

   public String sayHello() throws RemoteException {
     return "Hello World!";
   }

   public static void main(String args[])
   {
     // Create and install the security manager
     System.setSecurityManager(new RMISecurityManager());

     try {
       HelloImpl obj = new HelloImpl("HelloServer");
       Naming.rebind("HelloServer", obj);
       System.out.println("HelloImpl created and bound in
                the registry to the name HelloServer");

     } catch (Exception e) {
       System.out.println("HelloImpl.main: an exception
                occurred:");
       e.printStackTrace();
     }
   }
}
```

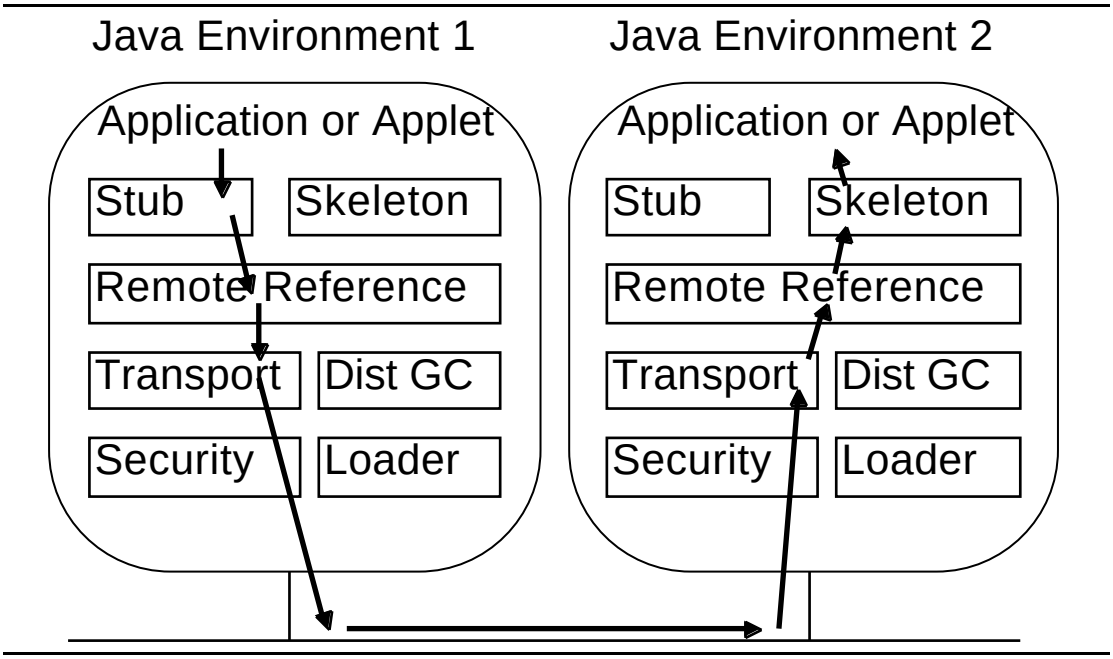
Example (cont'd)

- Server side
  - Creates an object which does what it should do (i.e. implements Hello interface)
  - Tells the registry to register the object under some name (registry should be running)
  - Stops
  - Runtime environment directs requests to object, invoking sayHello
- Client side
  - Asks registry for object corresponding to name on some machine
  - Invokes methods on object

RMI features

- Distributed object model
  - Objects: normal and remote
- Idea
  - Remote object exists on other host
  - Remote object can be used as normal object
  - Behavior described by interface
  - Environment takes care of remote invocation
- Differences normal and remote objects
  - Remote references can be distributed freely
  - Clients only know/use interface, not actual implementation
  - Passing remote objects by reference, normal objects by copying
  - Failure handling more complicated since invocation itself can also fail

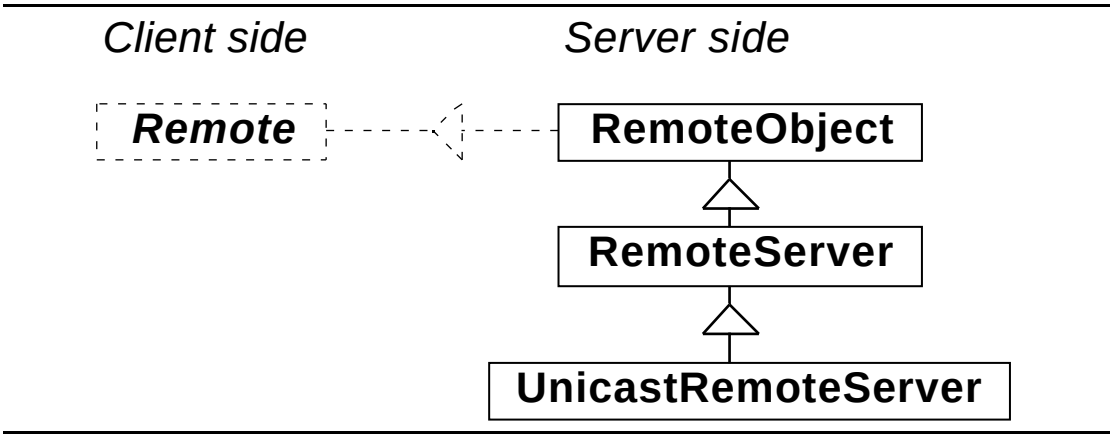
Implementation architecture



- Client interacts with stub (e.g. in env 1), invokes stub method (proxy pattern)
- Stub uses administration of remote reference to locate server (e.g. env 2)
- Invocation packaged & passed via transport & network to server
- Handled by skeleton which invokes locally appropriate method

Interfaces and classes

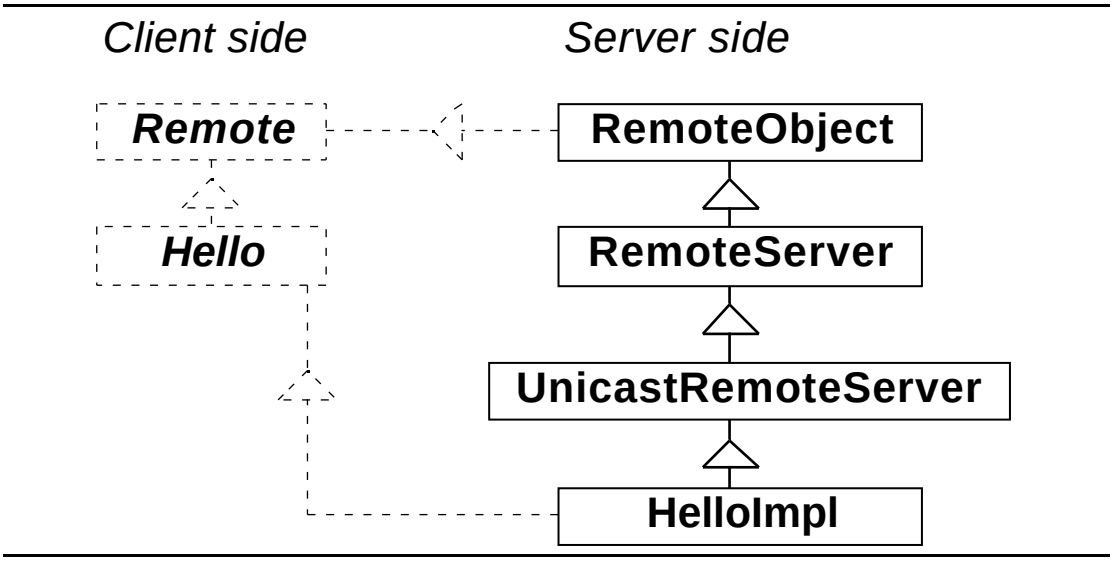
- For client and server



  - Client uses interface `Remote`
  - Server implements interface `Remote` via extending `RemoteObject`

Interfaces and classes

- Hello example



  - Hello is a `Remote`

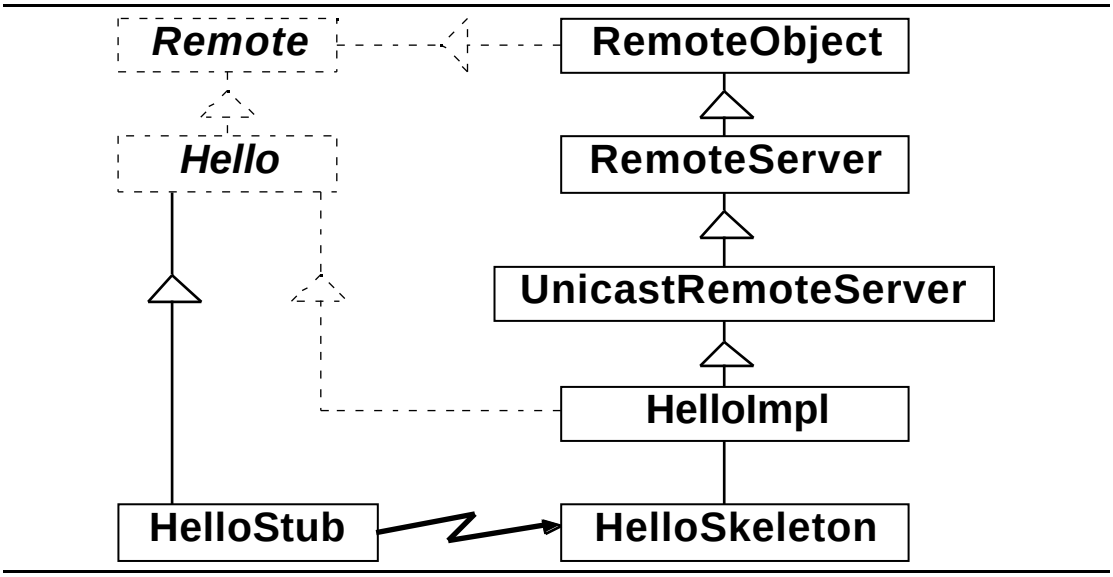
```
public interface Hello extends
                    java.rmi.Remote
```

  - Server implements `Hello`

```
public class HelloImpl
   extends UnicastRemoteServer
   implements Hello
```

Interfaces and classes

- Hello invocation



  - Separate compiler (rmic) makes stub and skeleton
  - Server HelloImpl makes itself known to naming service

```
HelloImpl obj =
   new HelloImpl("HelloServer");
Naming.rebind("HelloServer", obj);
```

  - Client asks Naming service for the stub

```
Naming.lookup
   ("//" + getCodeBase().getHost() +
                  "/HelloServer");
```

  - Client invokes stub for Hello

```
message = obj.sayHello();
```

Exceptions

- Extra things may go wrong
  - Connection may break
  - Server may be down (or go down halfway invocation)
  - Server may refuse (e.g. too busy)
  - Server does not trust data from client (security problem)
  - etc.

Exceptions

- Extra category of exceptions: `RemoteException`
  - May be thrown by all remote methods

```
public interface Hello
   extends java.rmi.Remote
{
   String sayHello()
      throws java.rmi.RemoteException;
}

public class HelloImpl
      extends UnicastRemoteObject
      implements Hello
{
   private String name;

   public String sayHello()
      throws RemoteException
   {
      ...
   ...
```

  - Failure must ultimately be handled by client

Parameter passing

- Non-remote objects
  - Passed by copy
  - Graph of related objects is packaged (marshalled) at client side
  - Packaged data is unmarshalled at server side and reconstructed into local object graph
- Remote objects
  - Stub of remote object is passed as non-remote object
  - Stub contains information on location of server object

Locating remote objects

- How to get a remote reference
  - As a result value of another (remote) method invocation
- Via a naming service
  - Given an URL (Uniform Resource Locator) and server made known to naming service

```
String url = "rmi://java.sun.com/echo"
                    ;
EchoImpl echo = new EchoImpl() ;
java.rmi.Naming.bind( url, echo ) ;
```

  - Looked up by client

```
echo = (Echo)java.rmi.Naming( url ) ;
```

  - Naming service itself may also be server
  - Naming service is bootstrap service, so needs fixed way of finding it

Naming

- Registry maps names to objects
- Default registry of objects uses URL's as names

```
rmi://java.sun.com:2001/root
```

- Registry maintains flat name space
- Binding name to object

```
void bind( String url, Remote obj )
```

- Unbinding

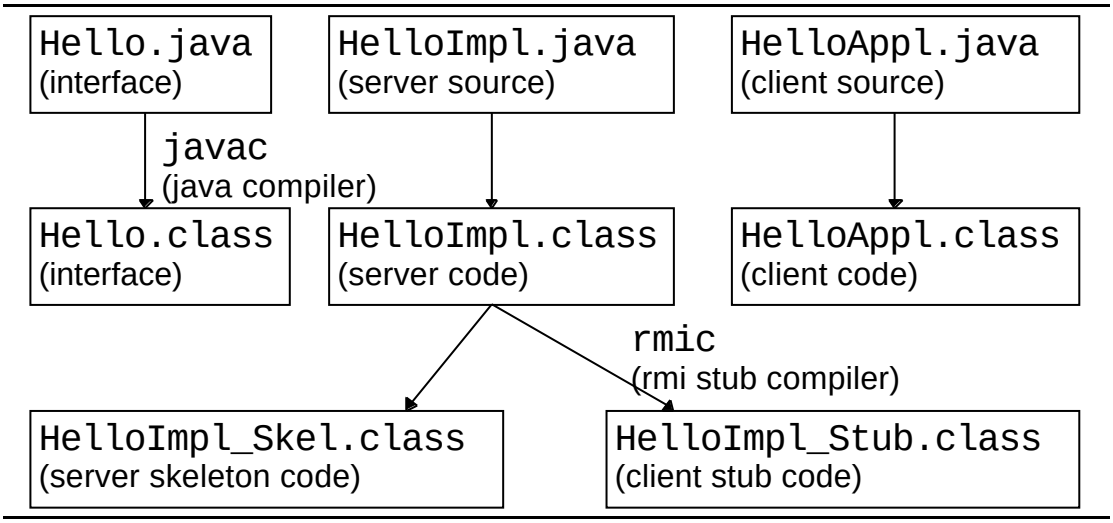
```
void unbind( String url )
```

- Listing

```
String[]  list( String url )
```

From source to client and server

- From source to code: Java compiler



- From code to stubs: RMI compiler
- From name to object (at runtime): rmiregistry as separate name server on well-known location (tcp/ip port)
- Server and client are started as separate programs

Security

- Loading of classes
  - When stub (i.e. remote object) is received (in server or client), its class is loaded
  - Only if not already loaded
  - Loaded by appropriate ClassLoader, same as class from which new class is needed
- Stubs may misbehave, e.g. trojan horse
  - Classes only loaded when SecurityManager is running
  - Stub contains location from which classes may be loaded
  - Class location may or may not be used
  - Policy implemented in SecurityManager
- Other security issues
  - Firewall inhibits direct communication (via sockets) needed by RMI
  - Invocation need to be wrapped in to HTTP requests (which are passed)

Garbage collection

- When is a remote object not needed anymore?
  - Normal garbage collection looks at *all* references to determine which are still in use
  - Not feasible to do 'normal' garbage collection, too large a task
- RMI runtime counts references
  - Incremented when reference enters a Java environment
  - When stub object itself is garbage collected, reference count is decremented
  - First time reference sends server a reference message, last reference gone sends unreference message
  - Tricky, servers may go down, clients too, networks may be separated
  - Remote objects may not longer exist even when client thinks so

Object references

- Passing data between client and server: objects are used everywhere
  - Parameters and results
  - Either 'normal' or remote object
- Normally
  - Objects are represented as pointer to data (which can contain pointers itself)
  - Object have address, a pointer
- Problem: remote access to objects
  - Objects have address which is only valid for machine where object lives (is created)
- Solution (in Java RMI): forbid it
  - Normal objects are copied, made into a clone on other machine
- Remote objects
  - Can only be used by invoking methods
  - Have a globally unique identification (address), containing host name, port, etc etc
  - Mapped in server to normal object

Passing objects

- Between hosts
- Object
  - Is not only sequence of data (bytes)
  - But is graph of objects referring to eachother
  - Possibly very large
- Hosts
  - May represent values in different ways (e.g. byte order)
- Communication
  - Passes only sequence of bytes, not a graph
- Java RMI value transfer: serialization
  - Object + objects referred to (recusrively) are marshalled into sequence of bytes
  - Except remote objects (reference is passed)
- Other issues
  - Customization of serialization (e.g. files by filename instead of content)

- Class evolution (formats change, different versions)

What can go wrong

- While publishing an object as remote
  - Class and/or stub not found or wrong name
  - Same for skeleton
  - Communication port already in use
- While invoking a method on a remote object
  - Unknown host, connection refused, ...
  - Remote object no longer available, or not exported
- While returning invocation result
  - Corrupt communication
  - Return value class unknown
  - Error in server (regular or due to RMI)
- While naming a remote object
  - Name already in use (during bind)
  - Name not used (during lookup)