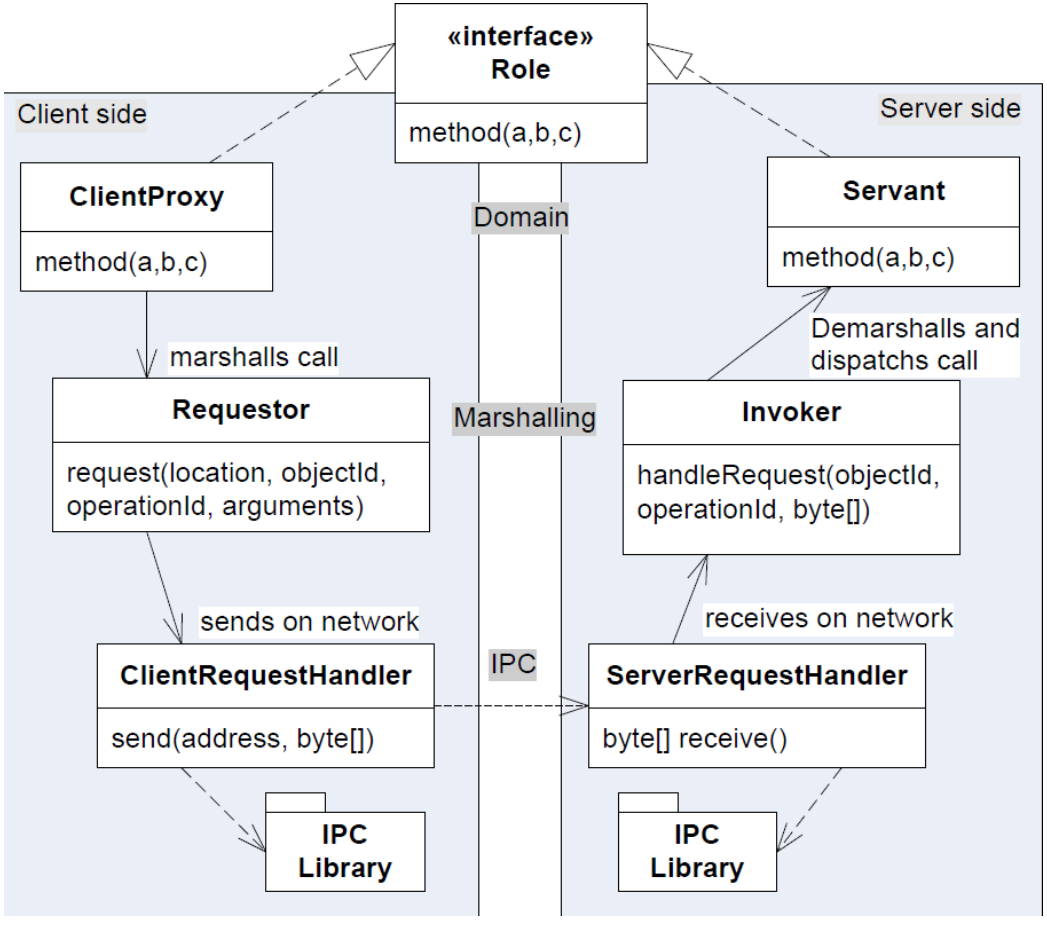
AARHUS UNIVERSITET

Software Engineering and Architecture

Distribution using Java RMI

Motivation

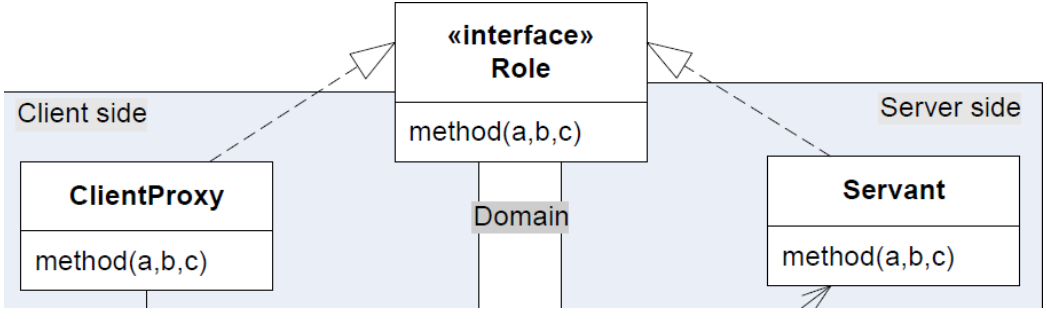
- Much of the implementation of the Broker delegates is repeating the same boilerplate code ☹
- Why not let a tool do it?



AARHUS UNIVERSITET

Long Tradition

- CORBA
  - Common Object Request Broker Architecture, started 1991
  - Specify interfaces in IDL = interface definition language
  - Compile to (almost) any concrete language
    - Java, C#, C++, C, Fortran, …
  - Generates codebases in the specified language
    - Skeleton code = Servant roles
    - Stub code = Proxy roles



Long Tradition

- Microsoft DCom
- .Net Remoting

- And …

- Java RMI

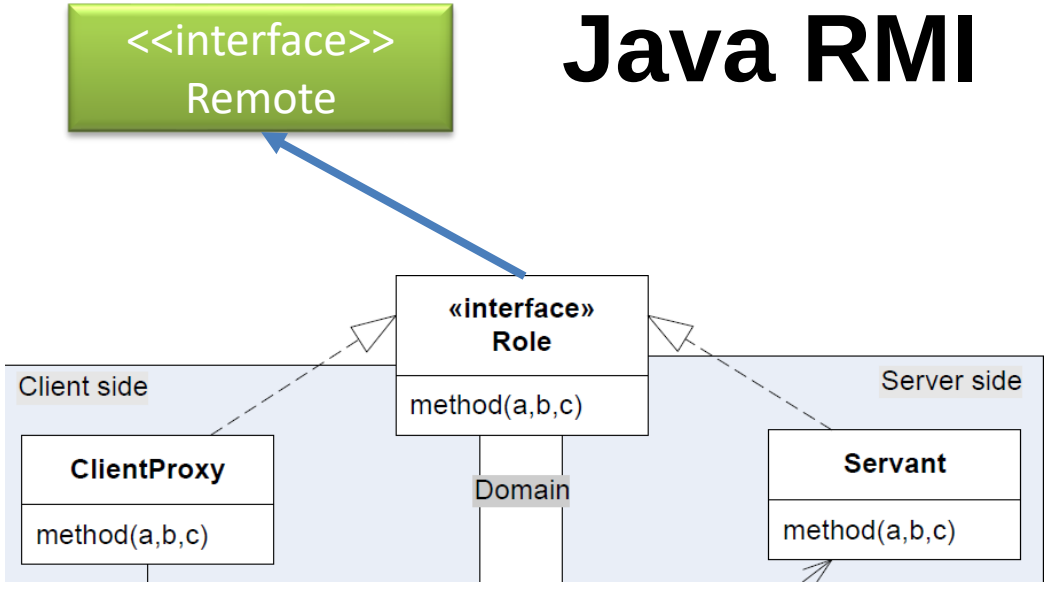
AARHUS UNIVERSITET

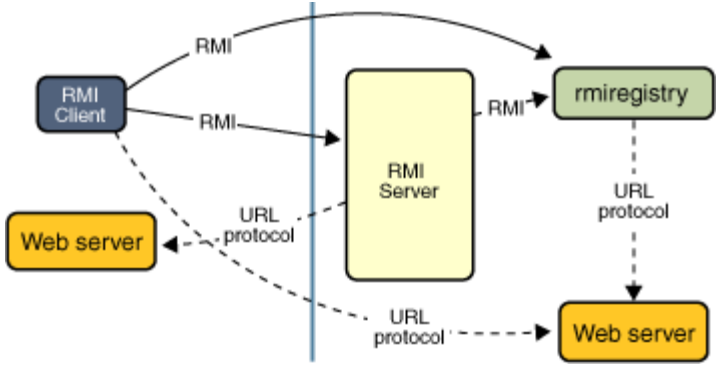
Overview

- Java RMI is full blown Broker and OO paradigm
  - I.e. it can handle *any* architectural style
    - You can make peer-2-peer, observer pattern, big ball of mud, …
- Pass by
  - Value — Objects implement Serializable
  - Reference — Objects implements Remote
- Name Service
  - Rmiregistry — Bind object reference to a name for lookup
- Security
  - Policy files — Govern what programs can and can not

Java RMI

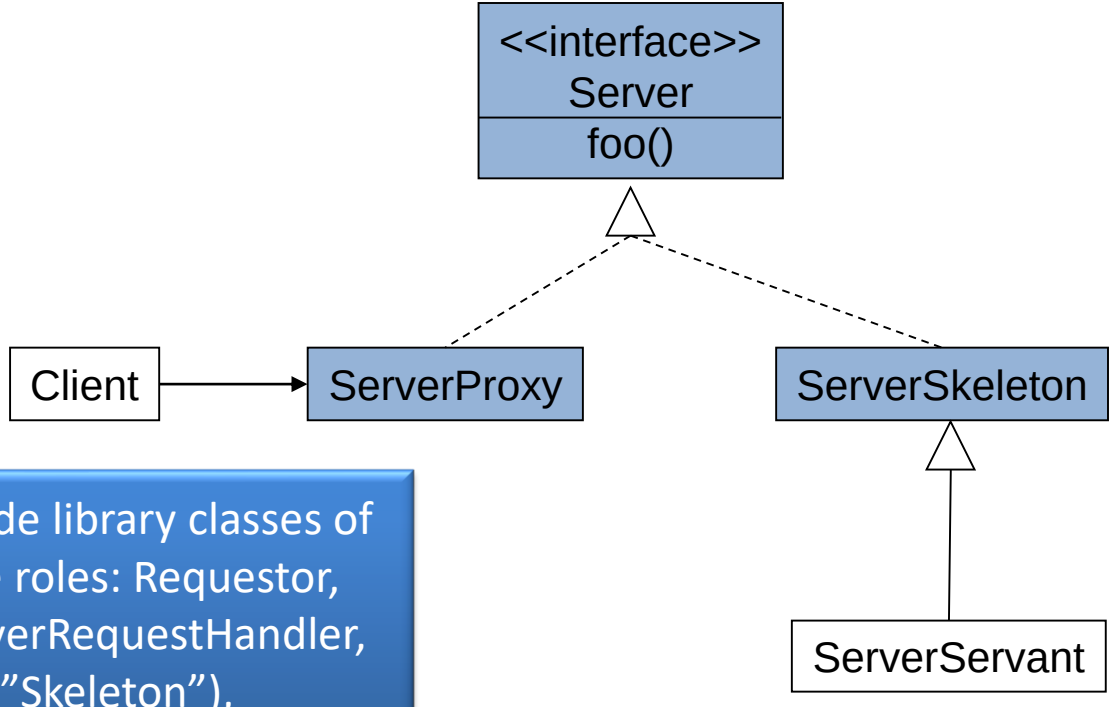
- Let your Role interface extend ”Remote”
  - I.e. you have already high coupling to RMI ☹
- Normal Java compile
  - Will call ‘rmic’ tool which will generate (see next slide…)





That is…

- The ‘rmic’ compiler will produce not one class file but two:



As well as provide library classes of the rest of the roles: Requestor, Client- and ServerRequestHandler, Invoker (”Skeleton”),

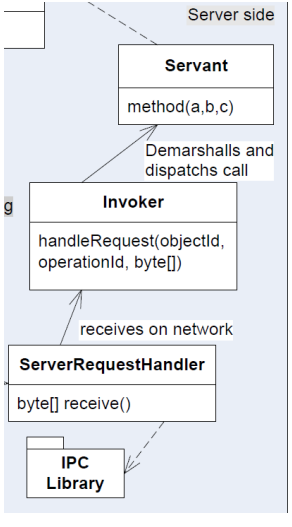
Example/Server side

- Define interface. All methods *throws RemoteException*

```
public interface TemperatureSensor extends Remote {
  /** register a listener to receive temperature data.
      @param tl the temperature listener instance to broadcast to.
  */
  public void addTemperatureListener( TemperatureListener tl )
    throws RemoteException;
}
```

- Implement Servant (serverside implementation)
  - Ups – inheritance based coupling to the Invoker ☹

```
public class TemperatureSensorImpl extends UnicastRemoteObject
  implements TemperatureSensor, Runnable {
```



AARHUS UNIVERSITET

Name Service

- Remote objects must be registered!
  - Here we ‘bind’ the object reference to the name ”section1” in the registry (from the server side)

```
// create a temperature sensor object representing the TS-05
TemperatureSensorImpl
  tss = new TemperatureSensorImpl();

String name = registry_host+"section1";
Naming.rebind(name, tss);
System.out.println( "Sensor object has been bound to name: "+name );

// make tss run in its own thead...
Thread t = new Thread( tss );
t.start();
```

Example/Client

- Clients look-up the name and ”get” the remote reference
  - Or rather, downloads a ClientProxy object from the registry

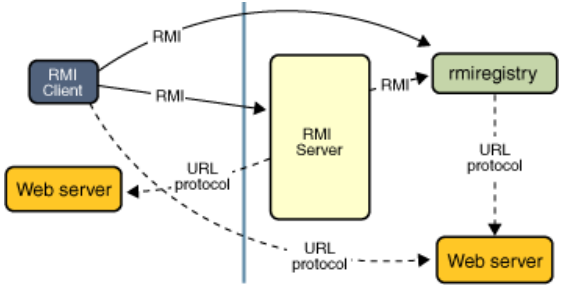
```
try {
  // get the temperature sensor object reference from the
  // RMI object request broker...
    String name = registry_host+"section1";
    System.out.println( "Looking up object reference: "+name );
    TemperatureSensor ts = (TemperatureSensor) Naming.lookup(name);

    System.out.println( "Located sensor object..." );
```

And …

- The setup is pretty heavy weight as
  - The client and the server must have access to the codebase in order to download the generated ClientProxy and Skeleton
  - There have to be a *security policy* telling the RequestHandlers what permissions are, in order to connect, read files, etc.

```
grant codeBase "file:/home/jones/src/" {
    permission java.security.AllPermission;
};
```

```
java -cp /home/ann/src:/home/ann/public_html/classes/compute.jar
     -Djava.rmi.server.codebase=http://mycomputer/~ann/classes/compute.jar
     -Djava.rmi.server.hostname=mycomputer.example.com
     -Djava.security.policy=server.policy
        engine.ComputeEngine
```



AARHUS UNIVERSITET

Value versus Reference Passing

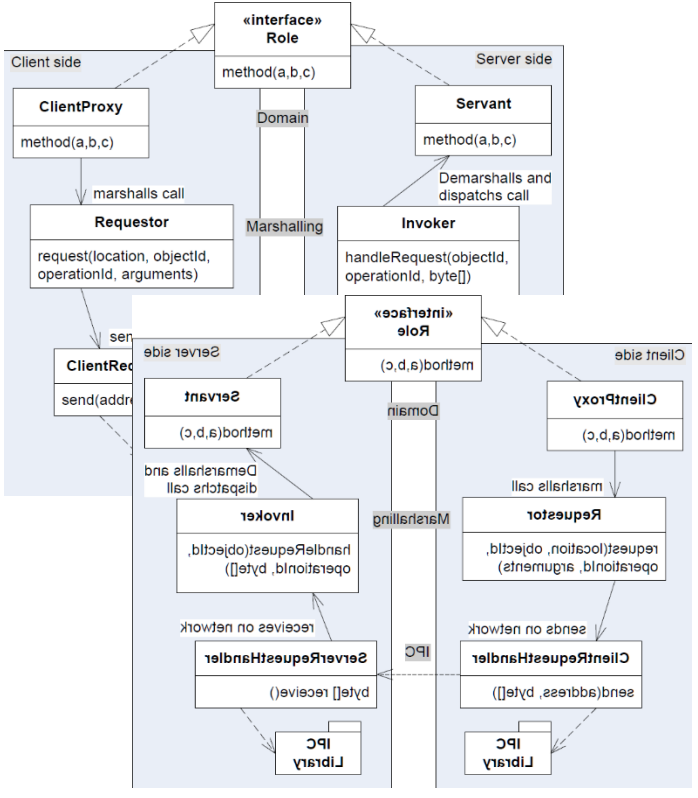
- RMI only works if *all* types that are part of return values and paramters lists in all remote interfaces are either
  - Remote implementations (X implements Remote)
  - Serializable implementatios (X implements Serializable)

- Which means you for each and every type must define them as either
  - Pass by reference (Remote)
  - Pass by Value (Serializable)

AARHUS UNIVERSITET

Mirrored Design

- Any Java RMI program is *both a client and a server!!!*
  - *A client can pass a Remote object, x, to server*
    - *Server call x.foo(a,b)*
      - *It is a proxy call from server to the client*
        That has the Servant implementation
- *RMI supports Observer pattern*
  - Client register (remote) observer
  - Server call o.update()
  - Client side update is called…



Demo ☹

- Nope ☹
  - Even though I have an example that I have used for years, I did not manage to get it running after a timebox of 4 hours !
    - Due to security policy issues, codebase issues, …
  - Searching for solutions showed very slim support…
    - Java 7 reworked the security policies in RMI with *breaking all existing behavior* changes…
    - Java RMI tutorials are *very* old on the internet
- Conclusion
  - Hmm hmm
  - Java RMI is technology of the past…