Architecture of Software Systems

RMI and Distributed Components

Martin Rehák

The point of RMI (Original)

I want to invoke methods on objects in other processes (on other hosts) as if they were local.

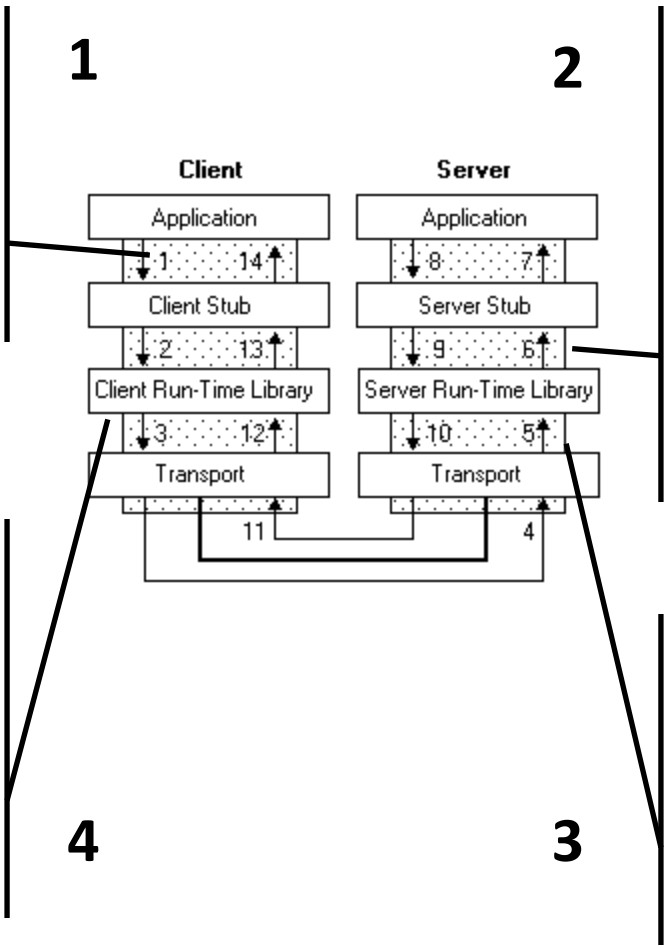
RMI Challenges

- Know which interface to call and how
- Identify or get the correct resource (object instance)
- Identify the correct method on the object instance
- Pass input and output values on the network
- Keep coherence between subsequent calls
- Handle failures

History:Remote Procedure Call

- Retrieves the required parameters from the client address space.
- Translates the parameters as needed into a transmission-friendly format.
- Calls functions in the RPC client run-time library to send the request

- The client RPC run-time library receives the return values and returns them to the client stub.
- The client stub converts the data from its NDR to the format used by the client computer. The stub writes data into the client memory and returns the result to the calling program on the client.
- The calling procedure continues .



- The server RPC run-time library functions accept the request and call the server stub procedure.
- The server stub retrieves the parameters from the network buffer and converts them from the network transmission format to the target format.
- The server stub calls the actual procedure on the server.

- The remote procedure returns its data to the server stub.
- The server stub converts output parameters to the network format and returns them to the RPC rtl.
- The server RPC run-time library functions transmit the data on the network to the client computer.

http://msdn.microsoft.com/en-us/library/windows/desktop/aa373935(v=vs.85).aspx

Patterns Used

- Proxy
  - Also called Client Stub
  - Combined with Façade
  - Hides the RPC-specific processing from the client
- Stub
  - Translates between the network formats and receiving server part and the actual servant "object"
- Servant
  - Does all the hard work, implements the functionality on the server side

Towards Remote Objects

- **RMI**: RPC for Java
- **RMI-IIOP**: CORBA Interoperable version, introduced in Java 1.3
- **EJB & J2EE**: Significant automation on the server integration part and additional patterns
  - Initial spec heavily based on IDL, CORBA and IIOP
  - Progressively made simpler, more lightweight

Word of Caution

- RPC, RMI and CORBA-like technologies are today rarely used directly in new systems
- … but are hidden inside other standards, technologies and products
- … and you need to know them…

RMI Basic Concepts

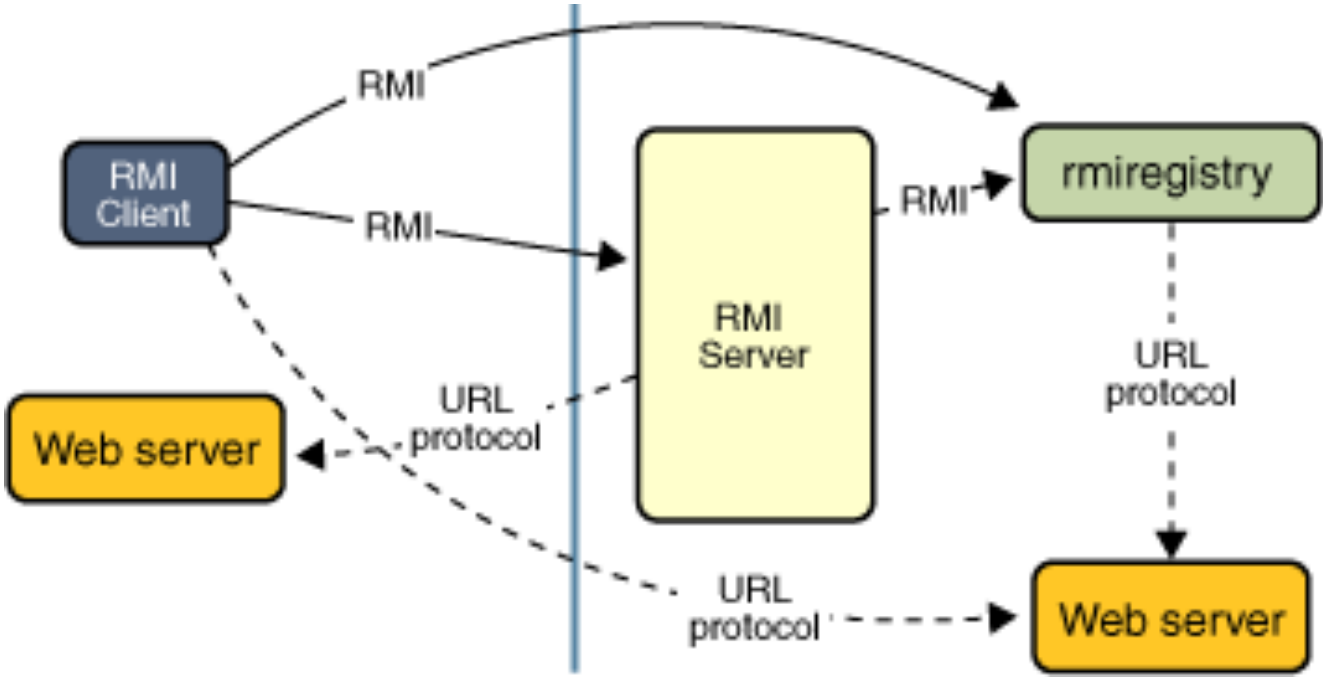
- **Client**: Java process on a client or server host, requesting the services /resources from another process either on the same or remote host
  - "Local" remote calls are frequent and wasteful
- **Server**: Java process providing a service or resource to the other processes
- **Object remoting**: Using remote objects as local objects

More basic concepts

RMI = Remote Method Invocation

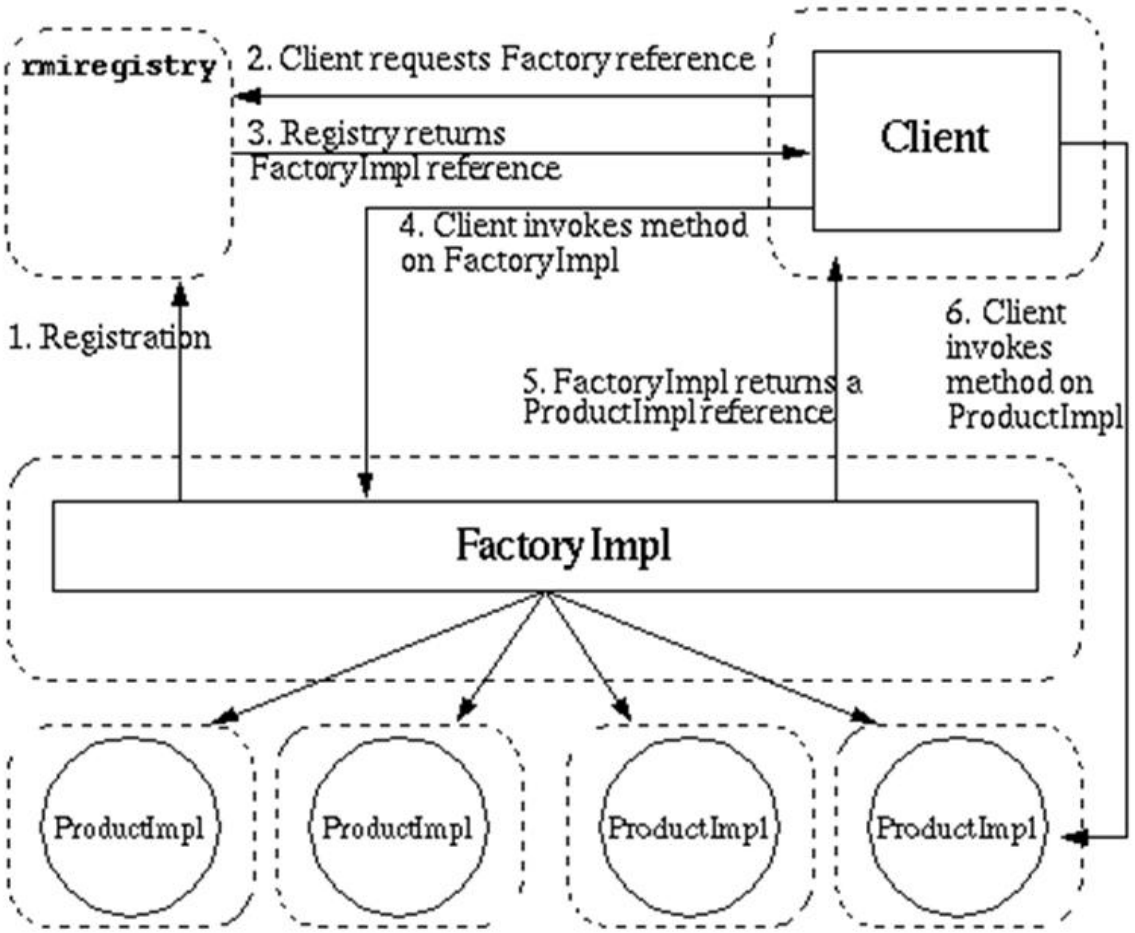
- **remote object** = object whose methods can be invoked from another JVM
  - called this even from the server's point of view, where the object is local!
  - perhaps a better name: remote-able object
- **remote interface** = interface exposed (for 'remote-ing') by a remote object
  - can have multiple interfaces so exposed
  - can have other interfaces and methods which are not exposed
- **remote reference** or **remote pointer** = a reference to a remote object (really a reference to the local stub)

Architecture



Oracle (http://docs.oracle.com/javase/tutorial/rmi/overview.html)

More Patterns: RMI and Factory



http://sourcecodemania.com/abstract-factory-design-pattern/

Interfaces

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
   // extends Remote is the key part
   String sayHello() throws RemoteException;
}

// Simple, Java-like definition
// RemoteException better be checked for…
```

Interface (2)

```
package compute;
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface Compute extends Remote {
        <T> T executeTask(Task<T> t) throws RemoteException;
}

package compute;
public interface Task<T> {
        T execute();
}
```

Oracle (http://docs.oracle.com/javase/tutorial/rmi/overview.html)
Including the whole example on the following slides

From Interface to Server (1)

- (Arguably) she simplest framework for remote invocation

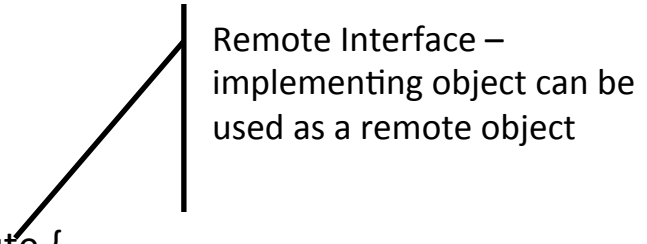
1. Declare the remote **interfaces** being implemented
2. Define the **constructor** for each remote object
3. Provide an **implementation** for each remote method in the remote interfaces
4. Handle the registration

From Interface to Server (1)

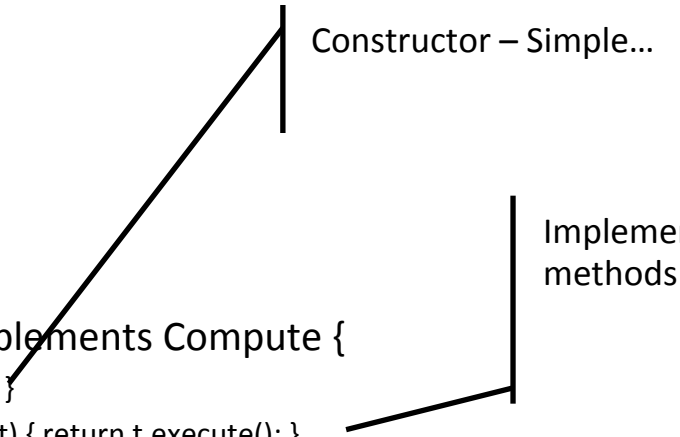
```
package engine;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import compute.Compute;
import compute.Task;

public class ComputeEngine implements Compute {
    public ComputeEngine() {  super(); }
    public <T> T executeTask(Task<T> t) { return t.execute(); }
    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager()); }
            try {
                String name = "Compute";
                Compute engine = new ComputeEngine();
                Compute stub = (Compute) UnicastRemoteObject.exportObject(engine, 0);
                Registry registry = LocateRegistry.getRegistry();
                registry.rebind(name, stub);
                System.out.println("ComputeEngine bound");
            } catch (Exception e) { System.err.println("ComputeEngine exception:"); e.printStackTrace(); } } }
```

From Interface to Server (2)

```
package engine;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import compute.Compute;
import compute.Task;

public class ComputeEngine implements Compute {
    public ComputeEngine() {  super(); }
    public <T> T executeTask(Task<T> t) { return t.execute(); }
    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager()); }
            try {
                String name = "Compute";
                Compute engine = new ComputeEngine();
                Compute stub = (Compute) UnicastRemoteObject.exportObject(engine, 0);
                Registry registry = LocateRegistry.getRegistry();
                registry.rebind(name, stub);
                System.out.println("ComputeEngine bound");
            } catch (Exception e) { System.err.println("ComputeEngine exception:"); e.printStackTrace(); } } }
```



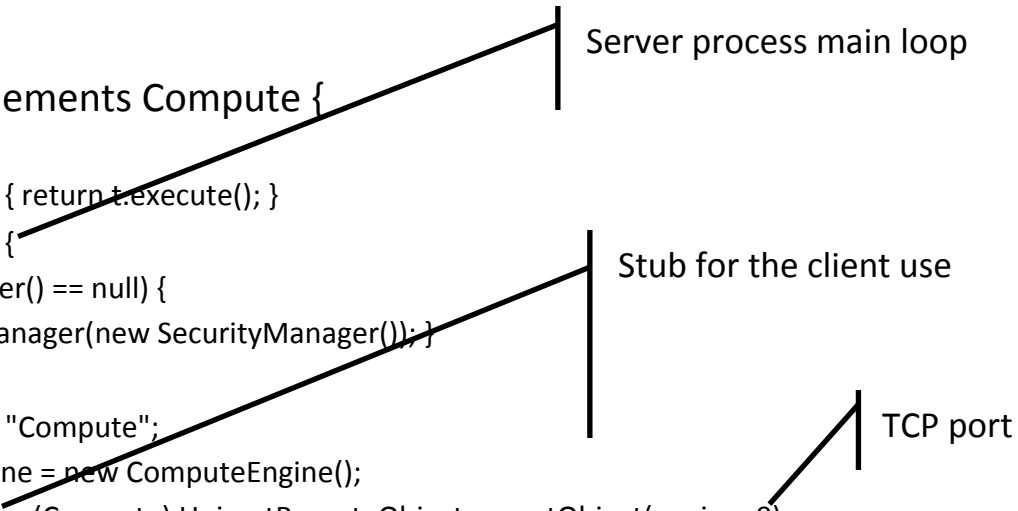
From Interface to Server (3)




```
package engine;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import compute.Compute;
import compute.Task;

public class ComputeEngine implements Compute {
    public ComputeEngine() {  super(); }
    public <T> T executeTask(Task<T> t) { return t.execute(); }
    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager()); }
            try {
                String name = "Compute";
                Compute engine = new ComputeEngine();
                Compute stub = (Compute) UnicastRemoteObject.exportObject(engine, 0);
                Registry registry = LocateRegistry.getRegistry();
                registry.rebind(name, stub);
                System.out.println("ComputeEngine bound");
            } catch (Exception e) { System.err.println("ComputeEngine exception:"); e.printStackTrace(); } } }
```

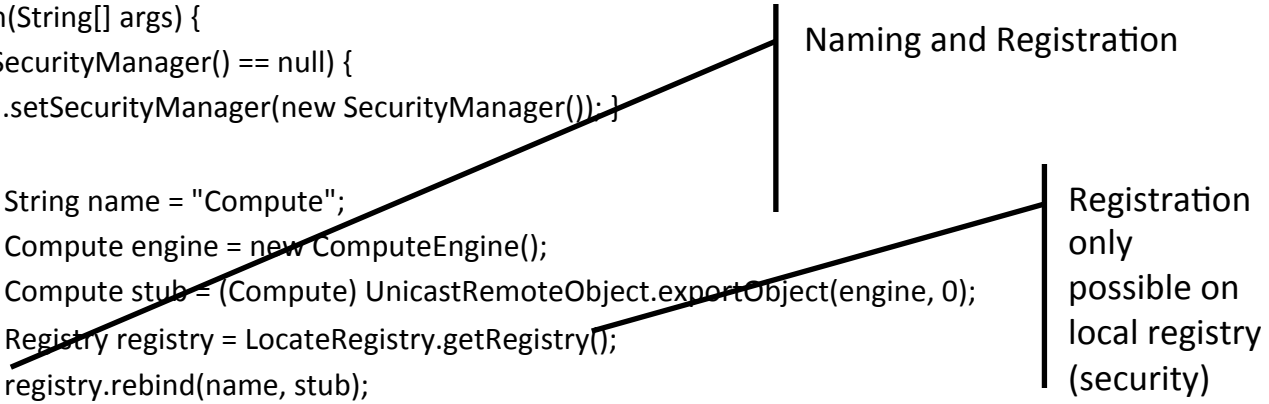
From Interface to Server (4)

```
package engine;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import compute.Compute;
import compute.Task;

public class ComputeEngine implements Compute {
    public ComputeEngine() {  super(); }
    public <T> T executeTask(Task<T> t) { return t.execute(); }
    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager()); }
            try {
                String name = "Compute";
                Compute engine = new ComputeEngine();
                Compute stub = (Compute) UnicastRemoteObject.exportObject(engine, 0);
                Registry registry = LocateRegistry.getRegistry();
                registry.rebind(name, stub);
                System.out.println("ComputeEngine bound");
            } catch (Exception e) { System.err.println("ComputeEngine exception:"); e.printStackTrace(); } } }
```



From Interface to Server (5)

```
package engine;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import compute.Compute;
import compute.Task;

public class ComputeEngine implements Compute {
    public ComputeEngine() {  super(); }
    public <T> T executeTask(Task<T> t) { return t.execute(); }
    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager()); }
            try {
                String name = "Compute";
                Compute engine = new ComputeEngine();
                Compute stub = (Compute) UnicastRemoteObject.exportObject(engine, 0);
                Registry registry = LocateRegistry.getRegistry();
                registry.rebind(name, stub);
                System.out.println("ComputeEngine bound");
            } catch (Exception e) { System.err.println("ComputeEngine exception:"); e.printStackTrace(); } } }
```



Parameter passing - general

Value

- Two independent instances of the state on both sides of the computational process
- State coherence guaranteed in the moment of call (parameters) or return (results) by the library
- Objects need to be serializable
- Deep copy, references to other objects, local-specific state management problems
- Problems with mapping to the OO model

Reference

- Remote objects are essentially passed by reference. A *remote object reference* is a stub, which is a client-side proxy that implements the complete set of remote interfaces that the remote object implements.
- ORB needs to create local stub for the remote object
- Every method call/property access becomes a remote call
- Clean Architecture, Engineer's nightmare

Parameter passing in RMI

- Three cases:
  - Primitive type: passed by value
  - Serializable object
  - Remote object – Remote interfaces are available to subscribers, object instance as such is not
- Security and policy implications

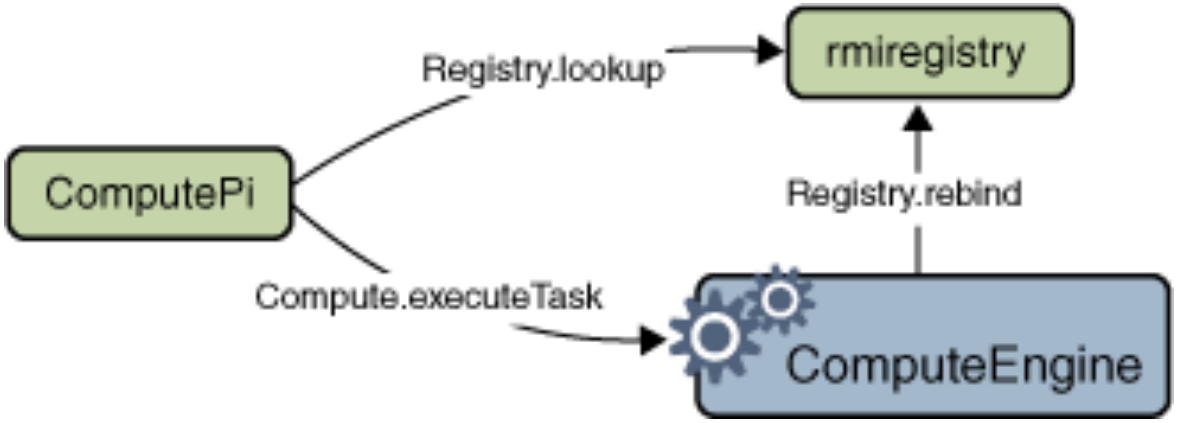
Recommended Design

- Remote object references shall only be used as addresses to relatively persistent services or computational units
- Most business data shall be passed by value (copy)
- Client code shall explicitly manage state coherence (or ather the lack of thereof)
- Copy and network communication is cheap under most circumstances

Recommended Design (2)

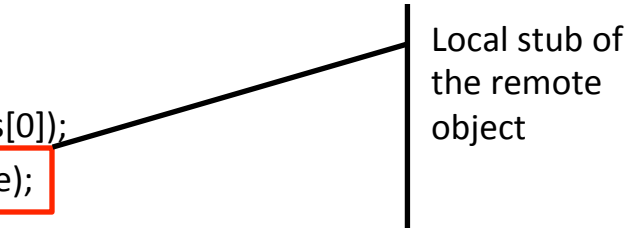
- Remote object references shall only be used as addresses to relatively persistent services or computational units
- Most business data shall be passed by value (copy)
- Client code shall explicitly manage state coherence (or ather the lack of thereof)
- Copy and network communication is cheap under most circumstances
- **Reliability** implications
  - When each node in a distributed system tightly relies on other nodes for essential data, failure of any single node affects the whole system

Implementing the client



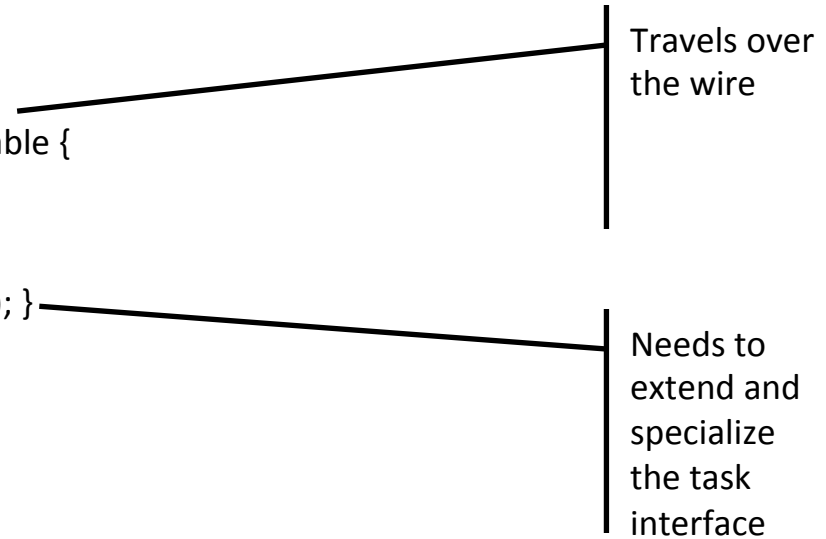
Implementing the Client

```
package client;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.math.BigDecimal;
import compute.Compute;

public class ComputePi {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {System.setSecurityManager(new SecurityManager()); }
        try {
            String name = "Compute";
            Registry registry = LocateRegistry.getRegistry(args[0]);
            Compute comp = (Compute) registry.lookup(name);
            Pi task = new Pi(Integer.parseInt(args[1]));
            BigDecimal pi = comp.executeTask(task); System.out.println(pi); }
        catch (Exception e) { System.err.println("ComputePi exception:"); e.printStackTrace(); } } }
```



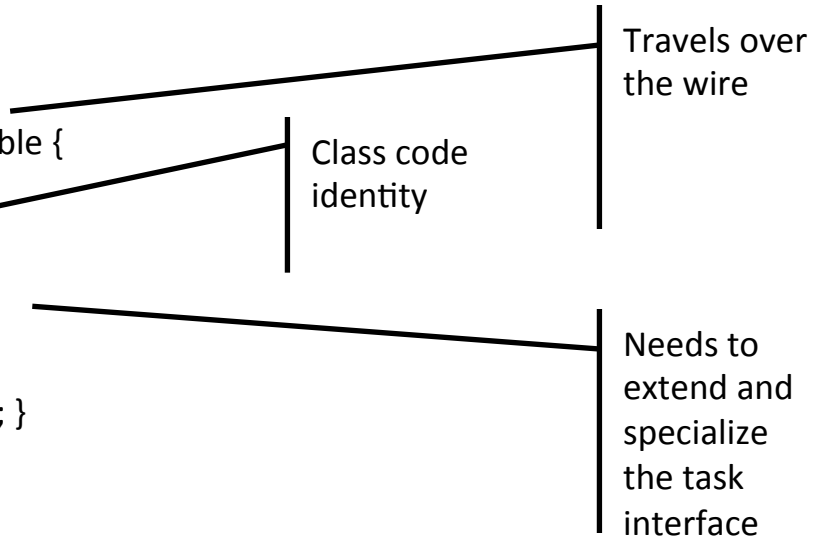
Implementing the client – the Task

```
package client;
import compute.Task;
import java.io.Serializable;
import java.math.BigDecimal;
public class Pi implements Task<BigDecimal>, Serializable {
…

public BigDecimal execute() { return computePi(digits); }
…

}
```



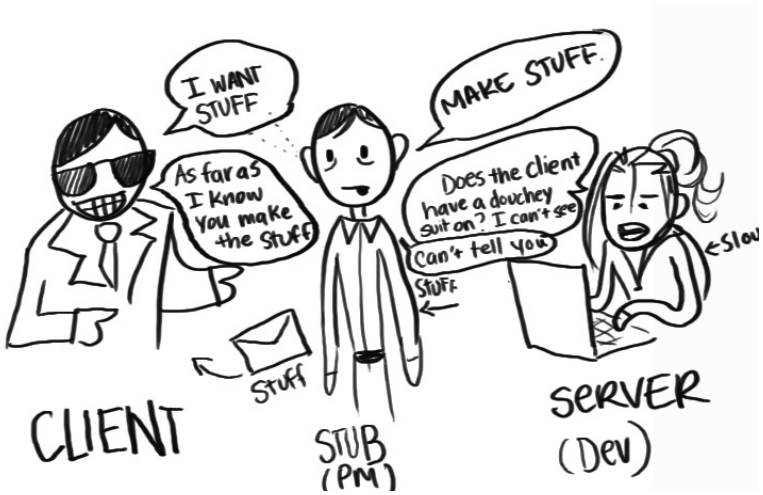
Implementing the client – the Task (2)

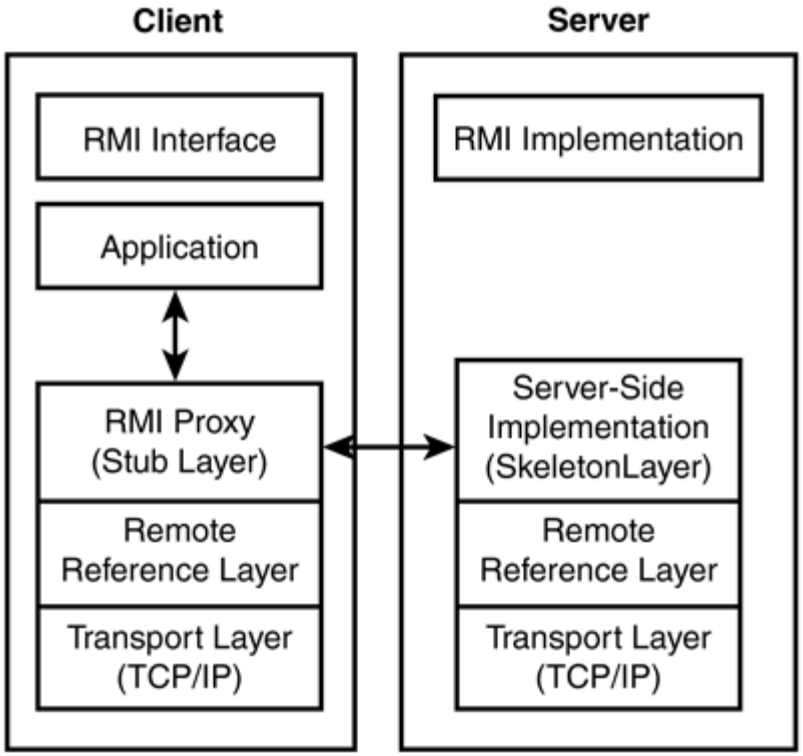
```
package client;
import compute.Task;
import java.io.Serializable;
import java.math.BigDecimal;
public class Pi implements Task<BigDecimal>, Serializable {
…
private static final long serialVersionUID = 227L;
…

public BigDecimal execute() { return computePi(digits); }
…

}
```



- This class is transmitted over the network/between the processes
- It includes the transfer of the class bytecode
- Version management, classloaders, serialVersionUID,…

Proxy/Stub Concept





Not-so good features

One of the most significant capabilities of the Java™ platform is the ability to dynamically download Java software from any Uniform Resource Locator (URL) to a virtual machine (VM) running in a separate process, usually on a different physical system. The result is that a remote system can run a program, for example an applet, which has never been installed on its disk. For the first few sections of this document, codebase with regard to applets will be discussed in order to help describe codebase with regard to Java Remote Method Invocation (Java RMI).

http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/codebase.html