Concurrent Programming (Part II)
Lecture 15: Distributed Object Systems and Java RMI (Remote Method Invocation)

Dr Kevin Bryson

K.Bryson@cs.ucl.ac.uk

Room 8.04

Course Web Site on Moodle

http://moodle.ucl.ac.uk/course/view.php?id=753

Enrolment Key: ATOMIC

Motivation for *not* using sockets !

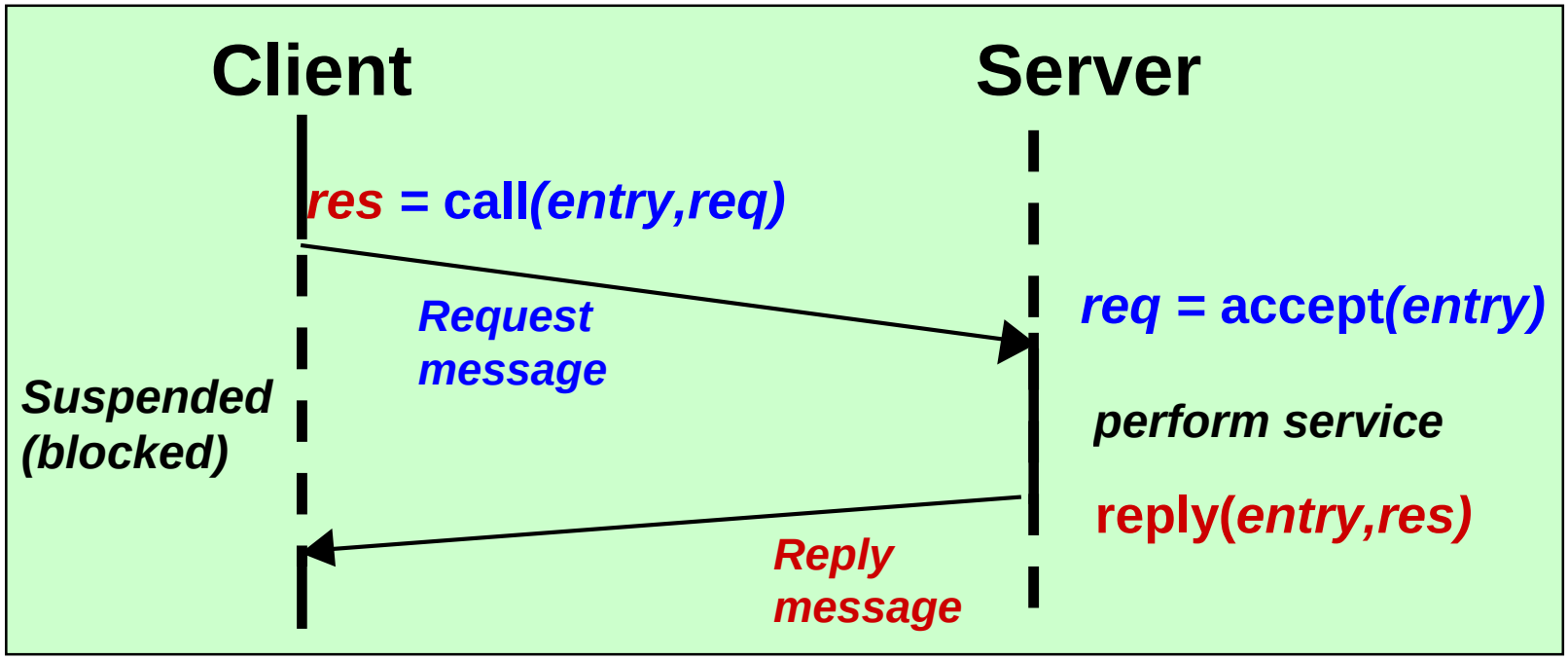
- The previous Remote File Reader Demo involved a very simple protocol where the client sent the filename to the server and the server returned the text of the required file.
- Unfortunately socket-based protocols become very complex as additional functionality is added.
- For example, the HyperText Transfer Protocol (HTTP/1.1) is a fairly simple protocol used to communicate between Web Browsers and Web Servers … but the standard protocol definition (called RFC 2616) is 176 pages of very concisely written text!
- This is good if we want to define an international standard that can be adopted using many different machine architectures, operating systems & languages … but …
- There must be a simpler way than sending individual ASCII messages if we want to build our own complex distributed systems …

Remember *Rendezvous ?*

• Rendezvous is a higher-level mechanism which structures the way individual messages are sent. Where a client sends a request and blocks waiting for a reply.

• Remote Procedure Call (RPC) is a type of rendezvous that treats the request/response messages as a single operation (the call).

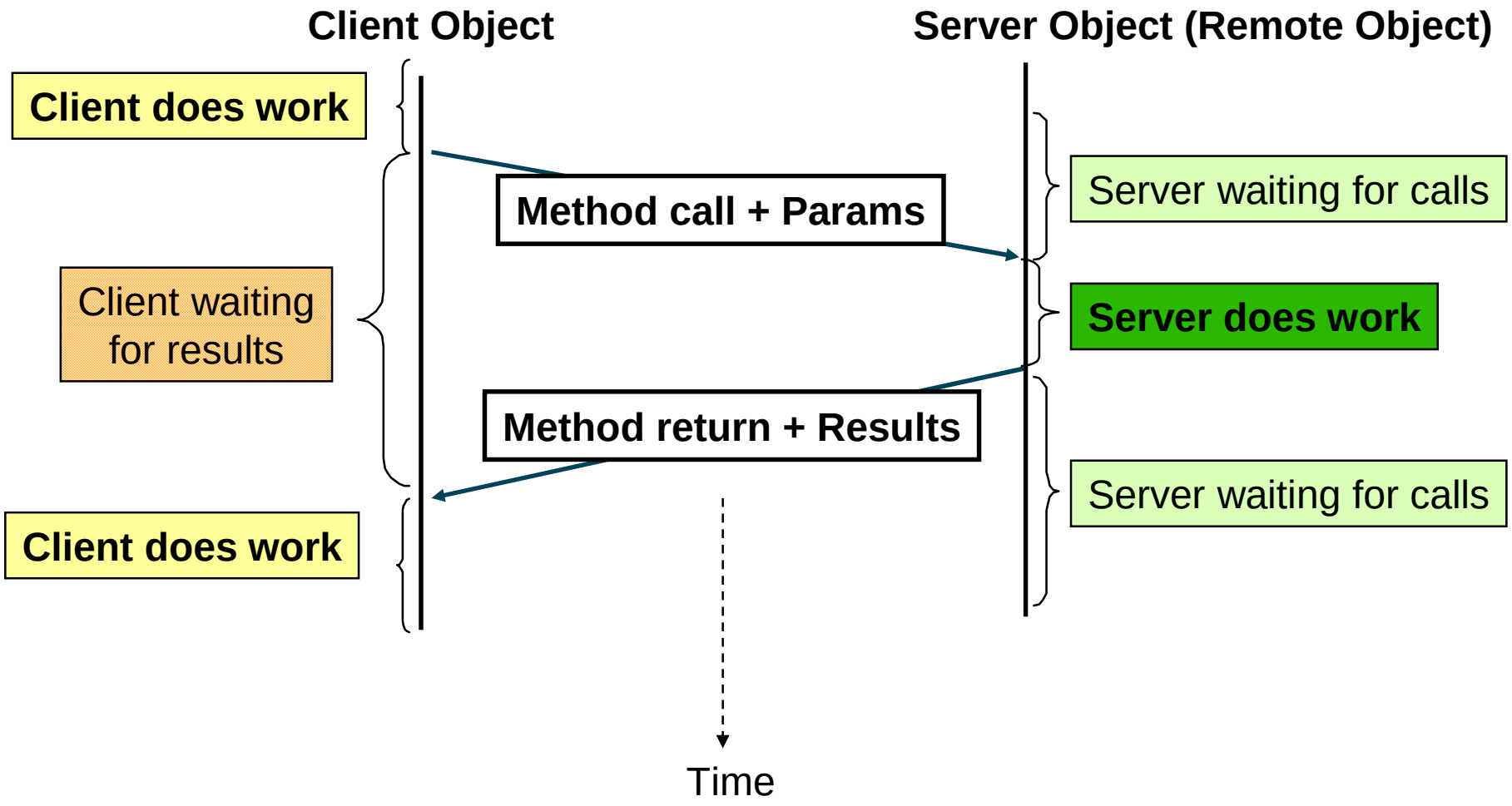
• Remote procedure calls allow one to think at a higher level than just messages being passed – they simplify the design process.



Remote Method Invocation (RMI)

- Java doesn't call *procedures*, Java *invokes methods*.
- Thus the equivalent to the Remote Procedure Call in Java is named **Remote Method Invocation** (**RMI**) – this is what we will study in this lecture.
- RMI is a framework which allows objects which live in different JVMs to invoke each other's methods, potentially across a network.
- RMI is built into the standard Java distribution and is one of the easiest frameworks for creating distributed object systems in Java. It also highlights some of the concurrency concerns that exist within many distributed object frameworks.

RMI Protocol – rendezvous but objects are invoking methods on other objects



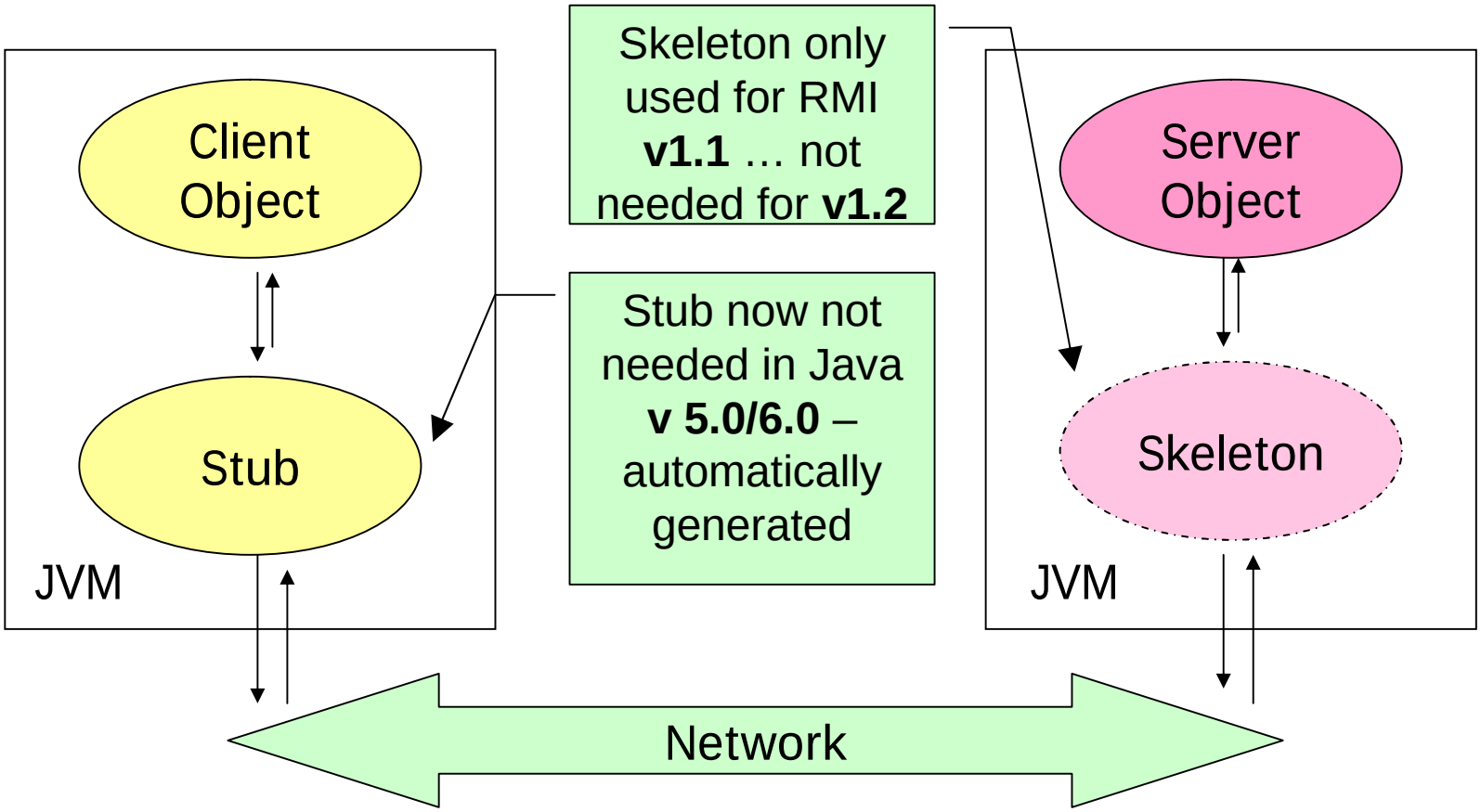
Benefits of Remote Method Invocation (RMI)

- RMI is tightly integrated with Java providing consistency in approach and also ***strong data typing***.
- Remote method calls take parameters that are not only primitive data types – but also complete objects. These need to be transferred across the network to the server.
- Similarly a method can return a primitive data type or a complete object that is again passed across the network back to the client.
- Think how difficult this would be with socket communication !

- Since Java can work across different machine architecture – heterogeneous distributed systems can be built.

- But one must always be aware that we are working across a network and there are still problems associated with distribution such as those outlined in a previous lecture (Latency, Bandwidth, Security, etc).
- Although RMI guarantees *at-most-once* delivery semantics …

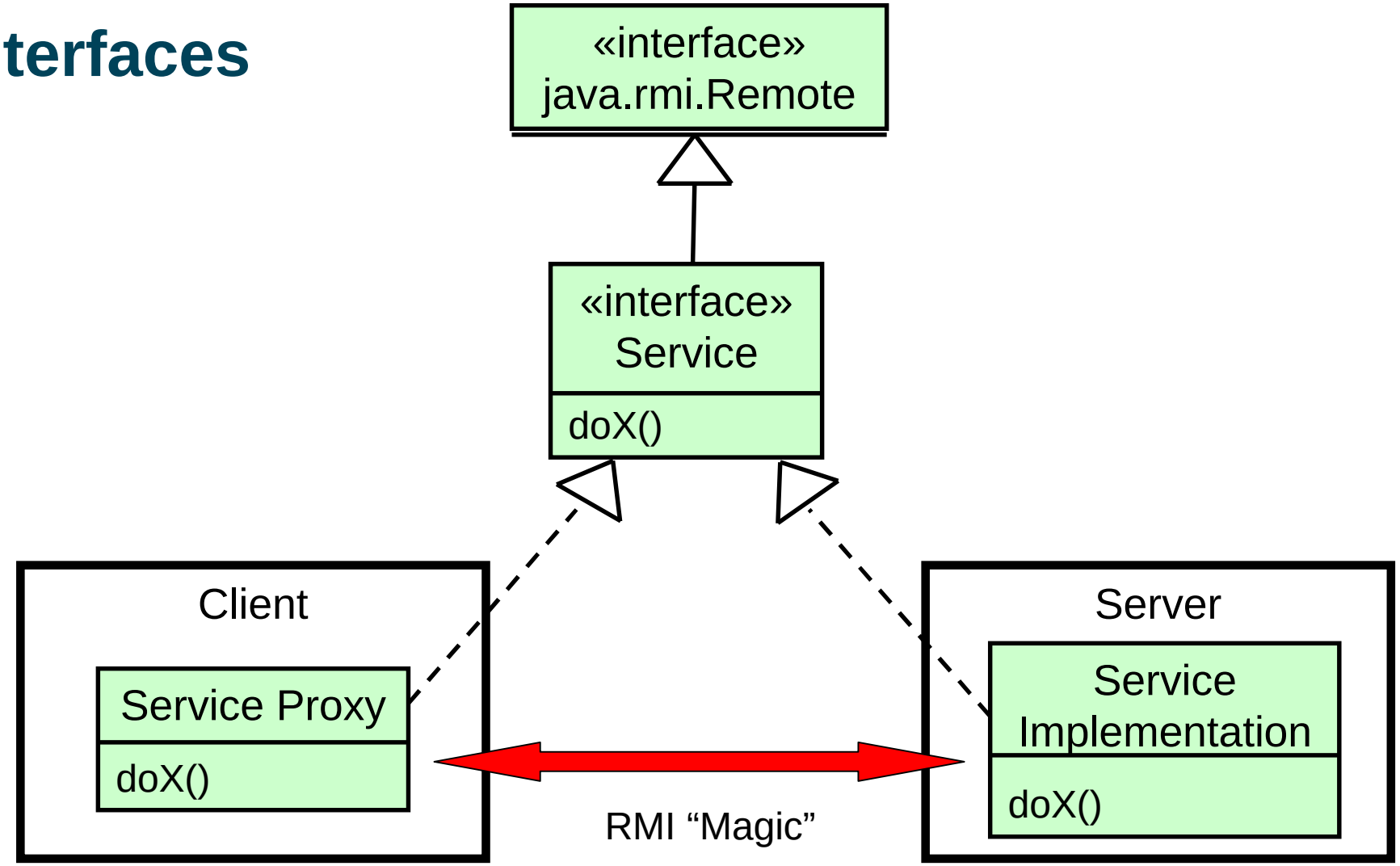
How does RMI work ? Stubs and Skeletons



Remote Interfaces

- How does the client know which method to call and the signature of the method?
- There is a common **remote interface** defined by the server so that:
  - The client knows what remote methods are available.
  - The server knows what remote methods need to be implemented.
- The *implementation of the remote interface* is known only on the server side

Remote Interfaces



Java Interfaces and Remote Objects

- The existing interface `java.rmi.Remote` indicates to the Java RMI framework that this object can be remote.
- Remote interfaces extend `java.rmi.Remote`
- Remote classes *implement* remote interfaces
- Remote objects are *instances* of remote classes

The RMI “magic” consists of:

- **Stubs** derived from their remote interface that **marshal** calls from client objects by converting the calls into streams of data which can be sent across a network.
- **Skeletons** derived from the remote interface that **un-marshals** these streams of data into calls which are carried out on the remote objects.
- And vice-versa for any data which is returned by the method.

An example – a calculator server object

- STEP 1 – Define the remote methods you want.
- Remote interface class must extend `java.rmi.Remote`
- All methods must throw `java.rmi.RemoteException`

```
import java.math.BigDecimal;

public interface Calculator extends java.rmi.Remote
{
   public BigDecimal calculate_pi(int digits) throws
                                 java.rmi.RemoteException;
}
```

Calculator Implementation Class

- STEP 2 – Create a class to implement this interface.
- Note that methods must throw RemoteException.

```
public class CalculatorImpl implements Calculator {

    /**
     * Remote method to calculate pi.
     */
    public BigDecimal calculate_pi(int digits) throws
                                java.rmi.RemoteException {
       return computePi(digits);
    }

    ... etc.
    /** Complex implementation of computePi() .. **/

}
```

RMI Registry - Naming & Discovery Service

- How does the client find the service offered?
- It has the remote interface that the remote object supports … but does not know the actual location of the remote object!

- This requires a **Naming & Discovery Service** for the distributed objects.
- The RMI registry is such a service …

RMI Registry

- Started on server machine using `rmiregistry`
- The RMI registry needs to be started so that it can access the remote interface class files. *Thus it is easiest to start it in the same directory as your server classes.*
  (The Java Sun tutorial given at the end of this lecture explains how to make these classes available via a web server …)

- The RMI registry runs on each machine that hosts remote service objects and accepts queries for services, by default on port 1099.

- Remote objects are given names that can be looked up in the rmiregistry, such as: `CalculatorService`

Server & Client interaction with the RMI Registry.

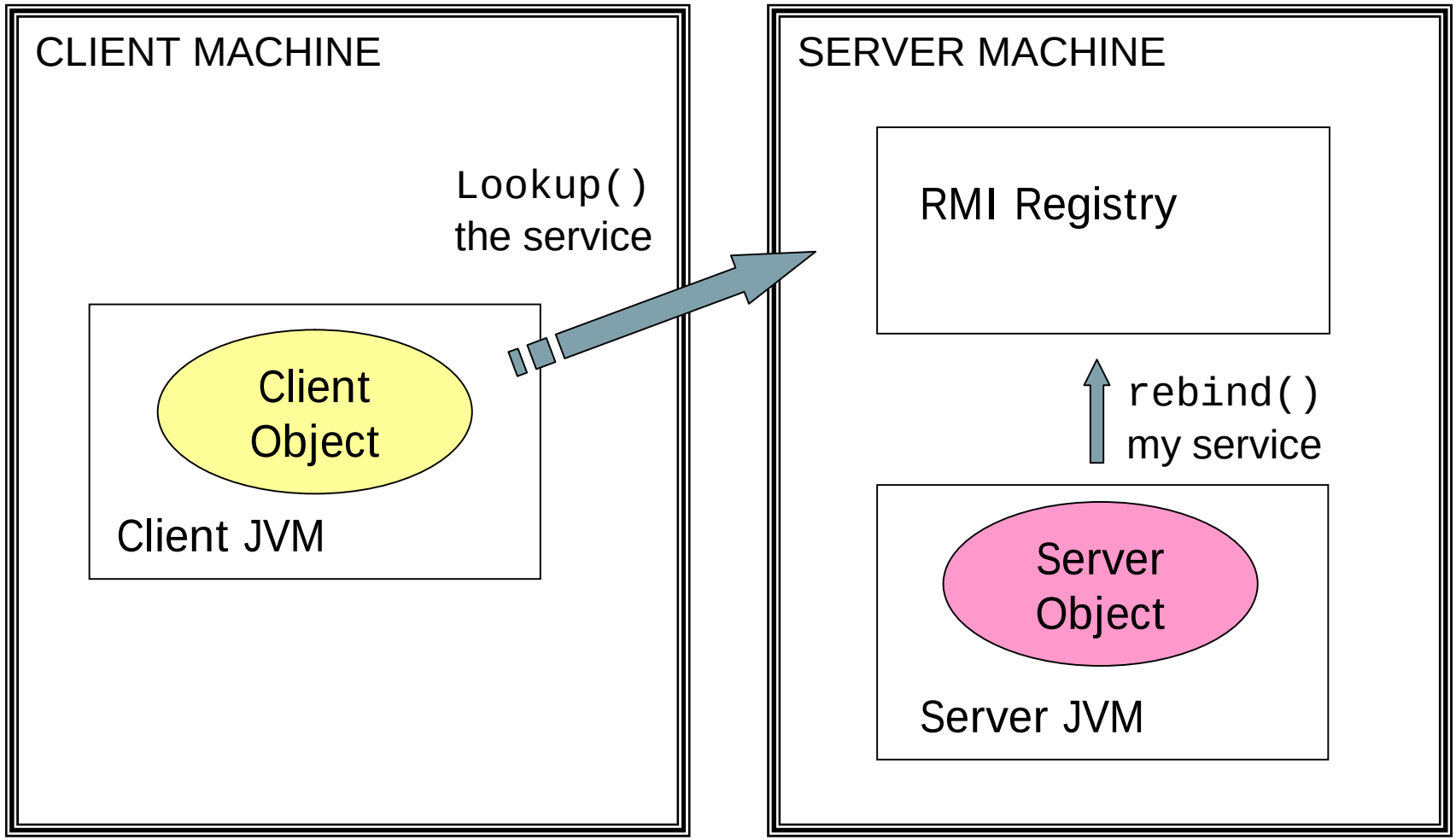
SERVER PROGRAM

- First a server program creates a remote service by creating a local object that implements the service.
- It then uses the class **UnicastRemoteObject** to export the object, this returns a stub for the object and allows connections to the object.
- It then gets a reference to the RMI Registry using the method LocateRegistry.getRegistry().
- Next the server **binds** or **rebinds** the service stub with the RMI registry.
- The remote object is then available for clients to use.

CLIENT PROGRAM

- On the client side, the RMI registry is obtained with the method **LocateRegistry.*getRegistry*(hostname).**
- This provides the method **lookup**() that a client uses to query a registry.
- The method **lookup**() accepts a service name and returns a reference (stub) to the remote object.

RMI Registry



Step 3 – Create the Remote Server

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class CalculatorServer {
  public CalculatorServer() {
    try {
         CalculatorImpl c = new CalculatorImpl();
         Calculator stub = (Calculator)
                          UnicastRemoteObject.exportObject(c, 0);

         // Bind the calculator server's stub in the RMI Registry
         // with the name ‘CalculatorService' that clients can lookup.
         Registry registry = LocateRegistry.getRegistry();
         registry.rebind("CalculatorService", stub);

         System.out.println("Calculator Server Running");

    } catch (Exception e) {
        System.out.println("Trouble: " + e);
    }
  }          ... etc.
```

And finally Step 4 – create the Client …

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class CalculatorClient {
   public static void main(String[] args) {
     try {

        // Get remote object reference (stub).
        Registry registry = LocateRegistry.getRegistry("localhost");
        Calculator c = (Calculator)
                        registry.lookup("CalculatorService");

    System.out.println(c.calculate_pi(Integer.parseInt(args[0])));
     }
                      … CONTINUED …
```

And all the exceptions it can throw …

```
                ... CONTINUED ...
  catch (java.rmi.RemoteException re) {
        System.out.println( "RemoteException");
        System.out.println(re);
     }
     catch (java.rmi.NotBoundException nbe) {
        System.out.println( "NotBoundException");
        System.out.println(nbe);
     }
     catch (java.lang.ArithmeticException ae) {
        System.out.println("java.lang.ArithmeticException");
        System.out.println(ae);
     }
   }
}
```

Doing calculations on a remote machine …

- We get the expected output …

```
[RMI_DEMO]$ java remotecalc.client.CalculatorClient 1000
    3.14159265358979323846264338327950288419716939937510582097
    49445923078164062862089986280348253421170679821480865132
    82306647093844609550582231725359408128481117450284102701938
    38521105559644622948954930381964428810975665933446128475648
    48233786783165271201909145648566923460348610454326648213393
    93607260249141273724587006606315588174881520920962829254091
    71715364367892590360011330530548820466521384146951941511609
    09433057270365759591953092186117381932611793105118548074462
    37379962749567351885752724891227938183011949129833673362440
    65664308602139494639522473719070217986094370277053921714362
    92931767523846748184676694051320005681271452635608277857713
    42757789609173637178721468440901224953430146549585371050792
    27968925892354201995611212902196086403441815981362977477130
    99605187072113499999983729780499510597317328160963185950244
    59455346908302642522308253344685035261931188171010003137838
    75288658753320838142061717766914730359825349042872554687311
    59562863882353787593751957781857780532171226806613001927876
    61119590921642019893
[RMI_DEMO]$
```

Doing calculations on a remote machine …

- This was a very simple example to demonstrate the key concepts, but this system could easily be extended to have much more complicated behaviour (RMI is more scalable than Socket-based systems).
- For example:
  - we could easily distribute this remote method to a large server machine and make it available programmatically to lots of Java client applications (which is the basis of J2EE systems).
  - we simply pass back a BigDecimal object from the server to the client … this could have easily been a much more complex data structure consisting of linked lists of customized objects …
  - unique to Java RMI … a complete Java object with particular behaviour can be passed across the network to be executed in another JVM (this is how Applets work). See the RMI tutorial given at the end to see an analogous system that does this.

Summary

- As distributed systems get larger, socket-based message passing protocols become much more complex.
- Distributed object frameworks facilitate the design and implementation of large systems.
- Remote Method Invocation (RMI) is such a framework and is Java-centric.
- Java RMI relies on stubs and skeletons to marshal and un-marshal remote method calls.
- It also relies on an RMI registry for remote object naming and discovery.
- We looked at a simple remote Calculator example … in the next lecture we will look at a much more complicated example and examine concurrency issues with distributed object systems.

For more details on RMI, see:
http://java.sun.com/docs/books/tutorial/rmi/