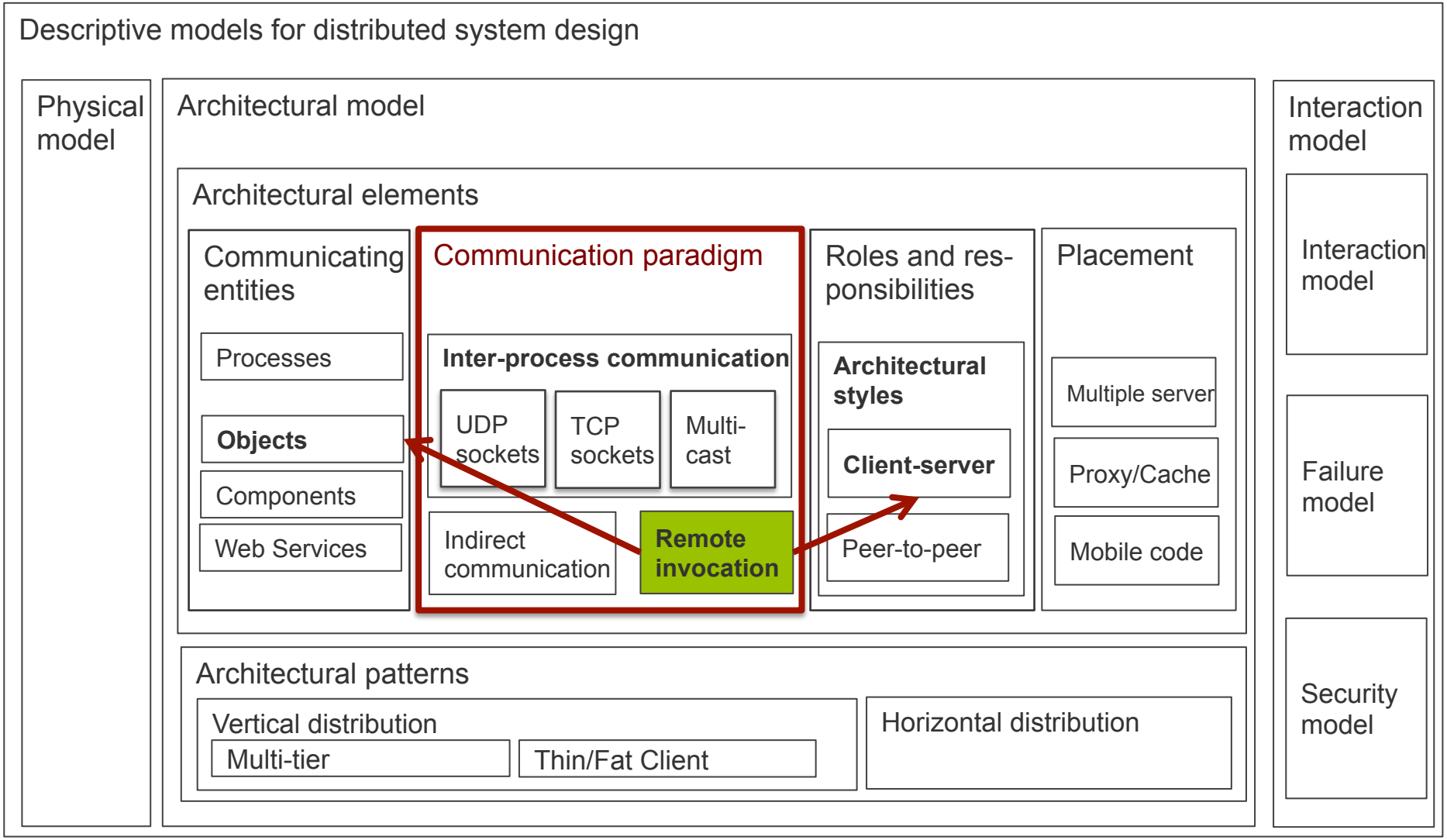
Prof. Dr. Claudia Müller-Birn
Institute for Computer Science, Networked Information Systems

Freie Universität Berlin

Remote invocation, part II (case study: Java RMI)

Netzprogrammierung
(Algorithmen und Programmierung V)

Our topics last week



Our topics today

Review: The process of remote method invocation

Java RMI architecture and its layers

Stub and Skeleton layer

- Proxy design pattern
- Reflections

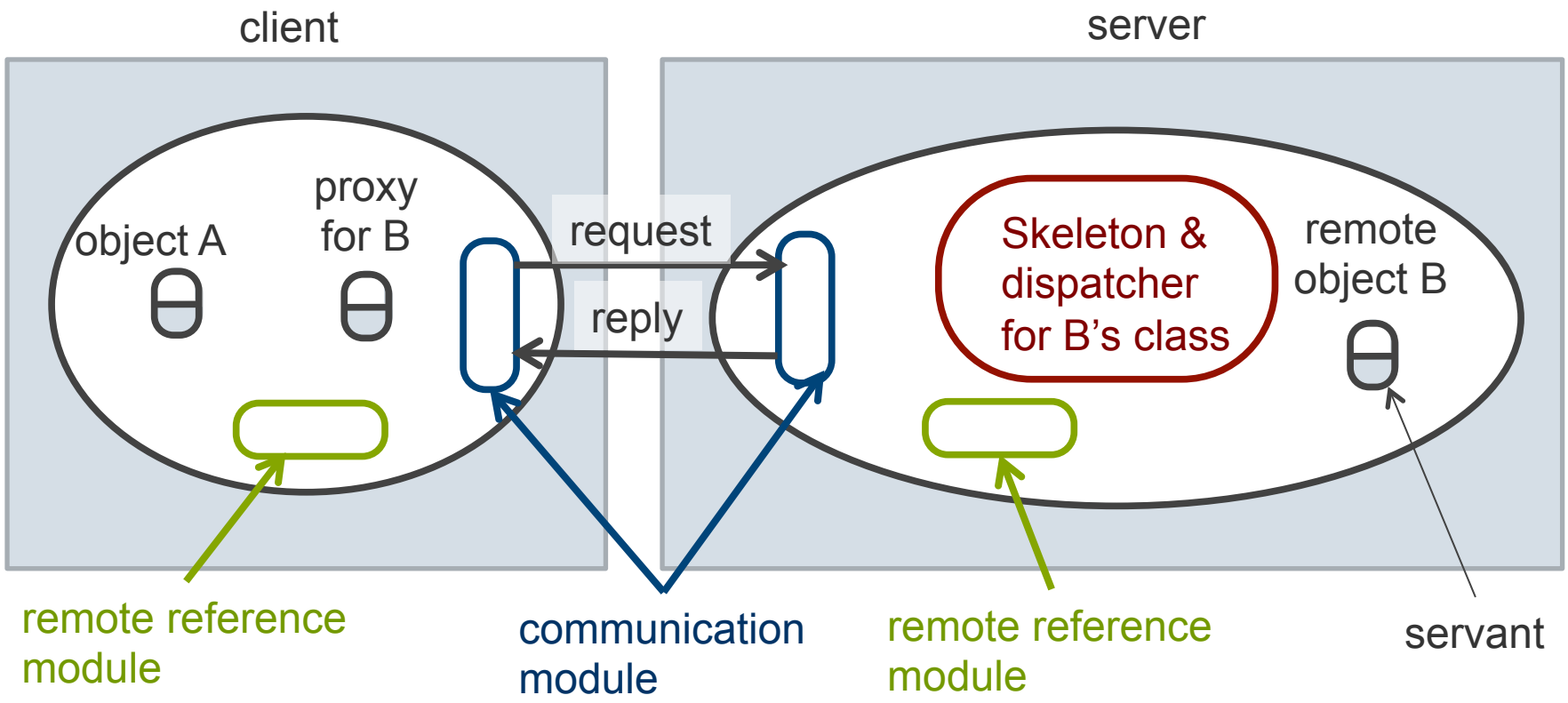
Remote Reference Layer

Transport Layer

Continuing with our Java RMI example: shared whiteboard

Java RMI architecture

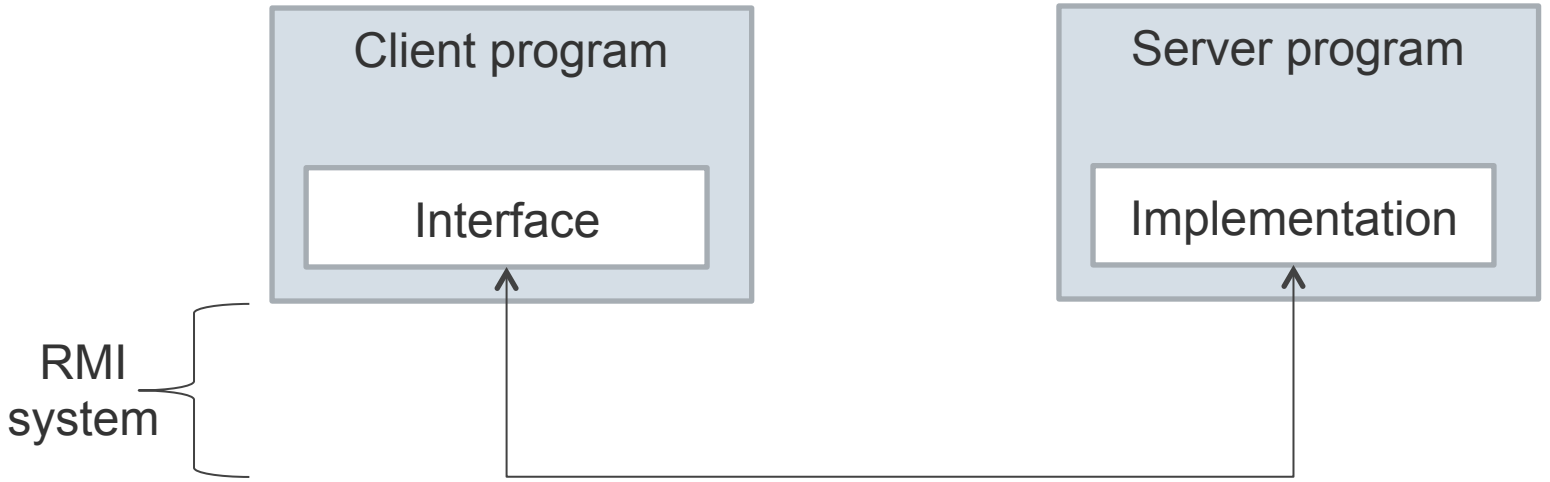
Components of the RMI architecture



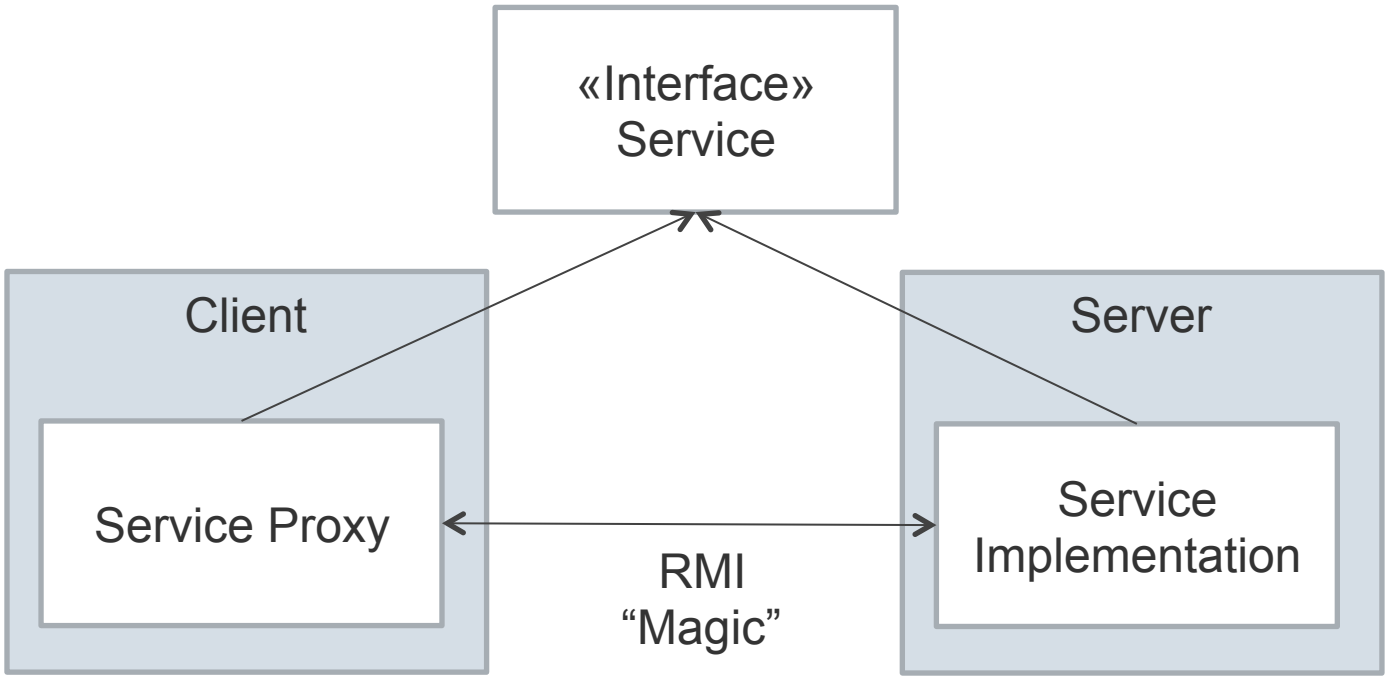
Separation of concerns

RMI architecture is based on one important principle: *the **definition of behavior** and the **implementation of that behavior** are separate concepts*.

RMI allows the code that defines the behavior and the code that implements the behavior to remain separate and to run on separate JVMs.



Implementation of the interface



Freie Universität Berlin

Java RMI architecture

RMI Architecture Layers

Abstraction layers in the RMI implementation

1. **Stub and Skeleton layer**

   Intercepts method calls made by the client to the interface reference variable and redirects these calls to a remote RMI service

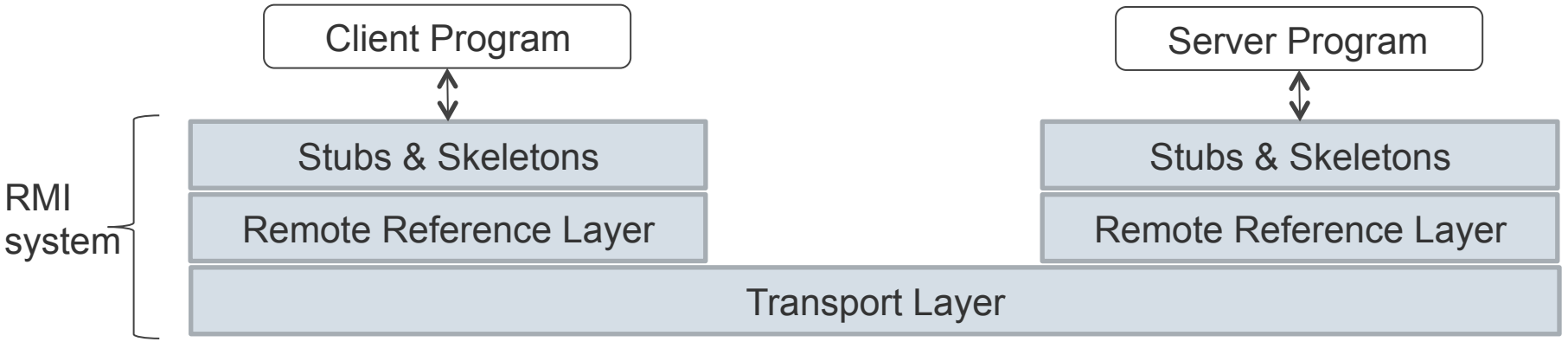
2. **Remote Reference Layer**

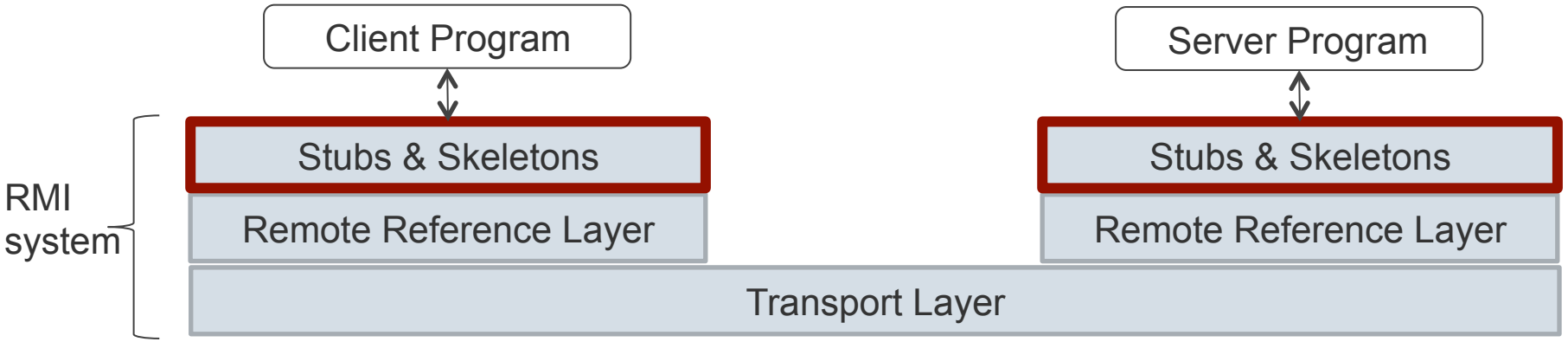
   Interpret and manage references made from clients to the remote service objects

3. **Transport layer**

   Is based on TCP/IP connections between machines in a network

   Provides basic connectivity, as well as some firewall penetration strategies





RMI Architecture Layers

Stub and Skeleton Layer

Stub and Skeleton Layer

RMI uses the **Proxy design pattern**

- Stub class is the *proxy*
- Remote service implementation class is the *RealSubject*

Skeleton is a helper class

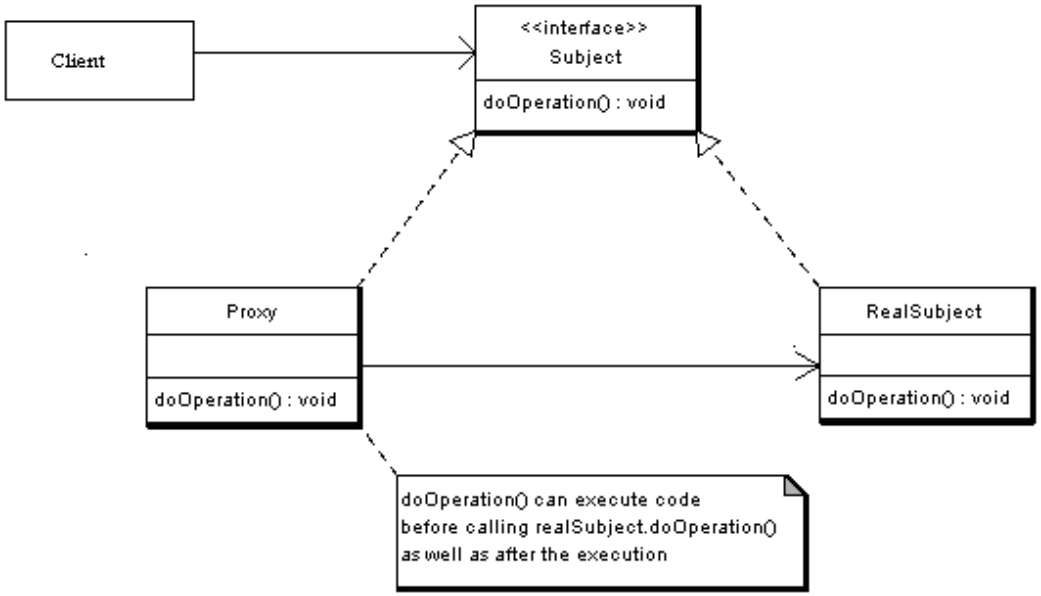
- Carries on a conversation with the stub
- Reads the parameters for the method call → makes the call to the remote service implementation object → accepts the return value → writes the return value back to the stub
- Please note: In the Java 2 SDK implementation of RMI, the new wire protocol has made skeleton classes obsolete. RMI uses ***reflection*** to make the connection to the remote service object.

Proxy design pattern

Motivation

Provide a surrogate or placeholder for another object to control access to it

Implementation



Proxy design pattern: Applications

Virtual Proxies: delaying the creation and initialization of expensive objects until needed, where the objects are created on demand.

Remote Proxies: providing a local representation for an object that is in a different address space. A common example is Java RMI stub objects. The stub object acts as a proxy where invoking methods on the stub would cause the stub to communicate and invoke methods on a remote object (called skeleton) found on a different machine.

Protection Proxies: where a proxy controls access to RealSubject methods, by giving access to some objects while denying access to others.

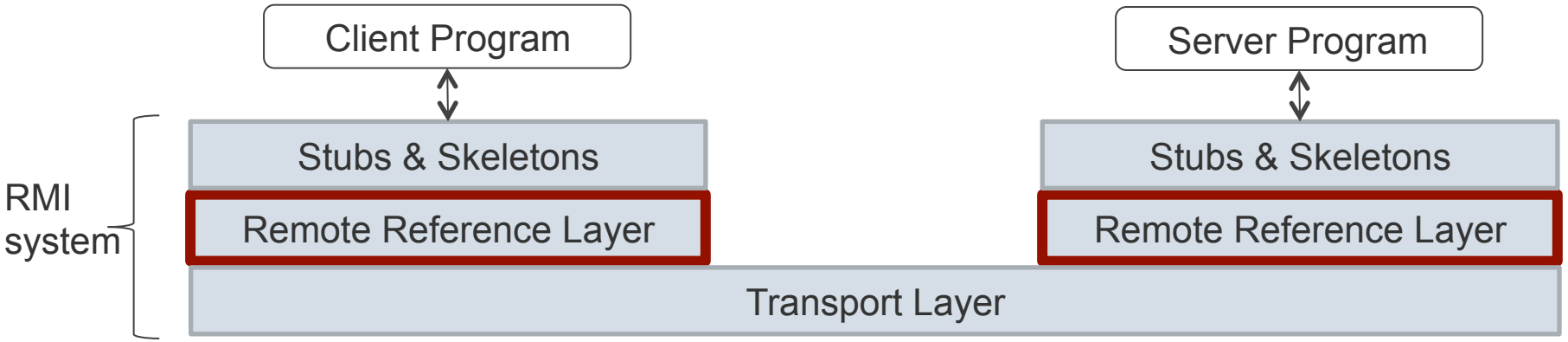
Smart References: providing a sophisticated access to certain objects such as tracking the number of references to an object and denying access if a certain number is reached, as well as loading an object from database into memory on demand.

Reflections

- Reflection enables Java code to discover information about the fields, methods and constructors of loaded classes, and
- To use reflected fields, methods, and constructors to operate on their underlying counterparts on objects, within security restrictions.
- More information: http://download.oracle.com/javase/tutorial/reflect/

Using Reflection in RMI

- Proxy has to marshal information about a method and its arguments into a request message.
- For a method it marshals an object of class *Method* into the request. It then adds an array of *objects* for the method's arguments.
- The dispatcher unmarshals the *Method* object and its arguments from request message.
- The remote object reference is obtained from remote reference module.
- The dispatcher then calls the *Method* object's "invoke" method, supplying the target object reference and the array of argument values.
- After the method execution, the dispatcher marshals the result or any exceptions into the reply message.



RMI Architecture Layers

Remote Reference Layer

Remote Reference Layer

Defines and supports the invocation semantics of the RMI connection

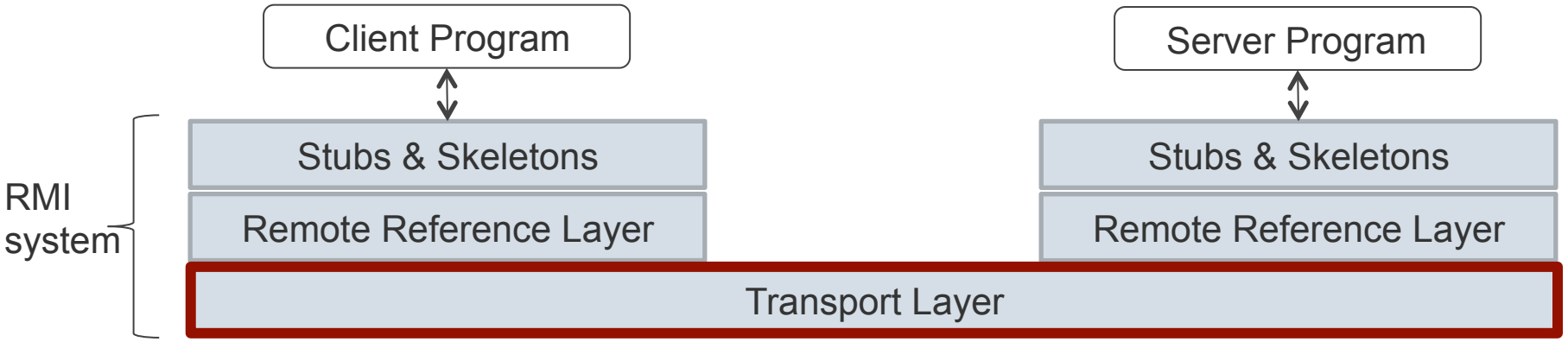
Provides a RemoteRef object that represents the link to the remote service implementation object

JDK 1.1 implementation of RMI

- Provides a unicast, point-to-point connection
- Before a client can use a remote service, the remote service must be instantiated on the server and exported to the RMI system

Java 2 SDK implementation of RMI

- When a method call is made to the proxy for an activatable object, RMI determines if the remote service implementation object is dormant
- If yes, RMI will instantiate the object and restore its state from a disk file

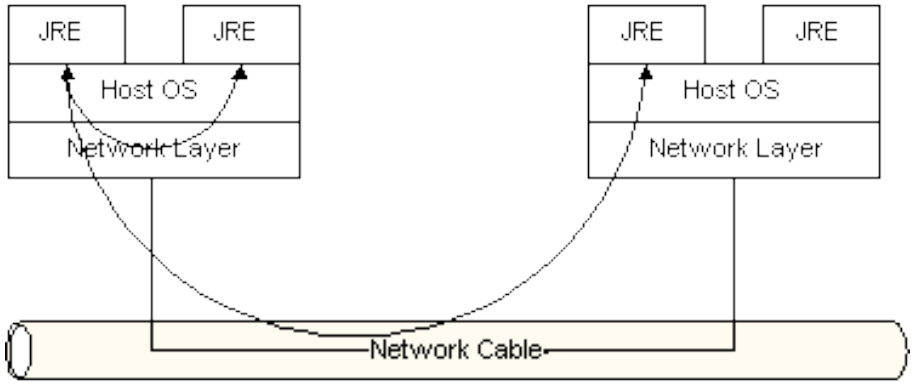


RMI Architecture Layers

Transport Layer

Transport Layer

The Transport Layer makes the connection between JVMs. All connections are stream-based network connections that use TCP/IP.



On top of TCP/IP, RMI uses a wire level protocol called Java Remote Method Protocol (JRMP). JRMP is a proprietary, stream-based protocol that is only partially specified in two versions:

- First version was released with the JDK 1.1 version of RMI and required the use of Skeleton classes on the server.
- Second version was released with the Java 2 SDK. It has been optimized for performance and does not require skeleton classes.

Freie Universität Berlin

Java RMI architecture

Naming Remote Objects

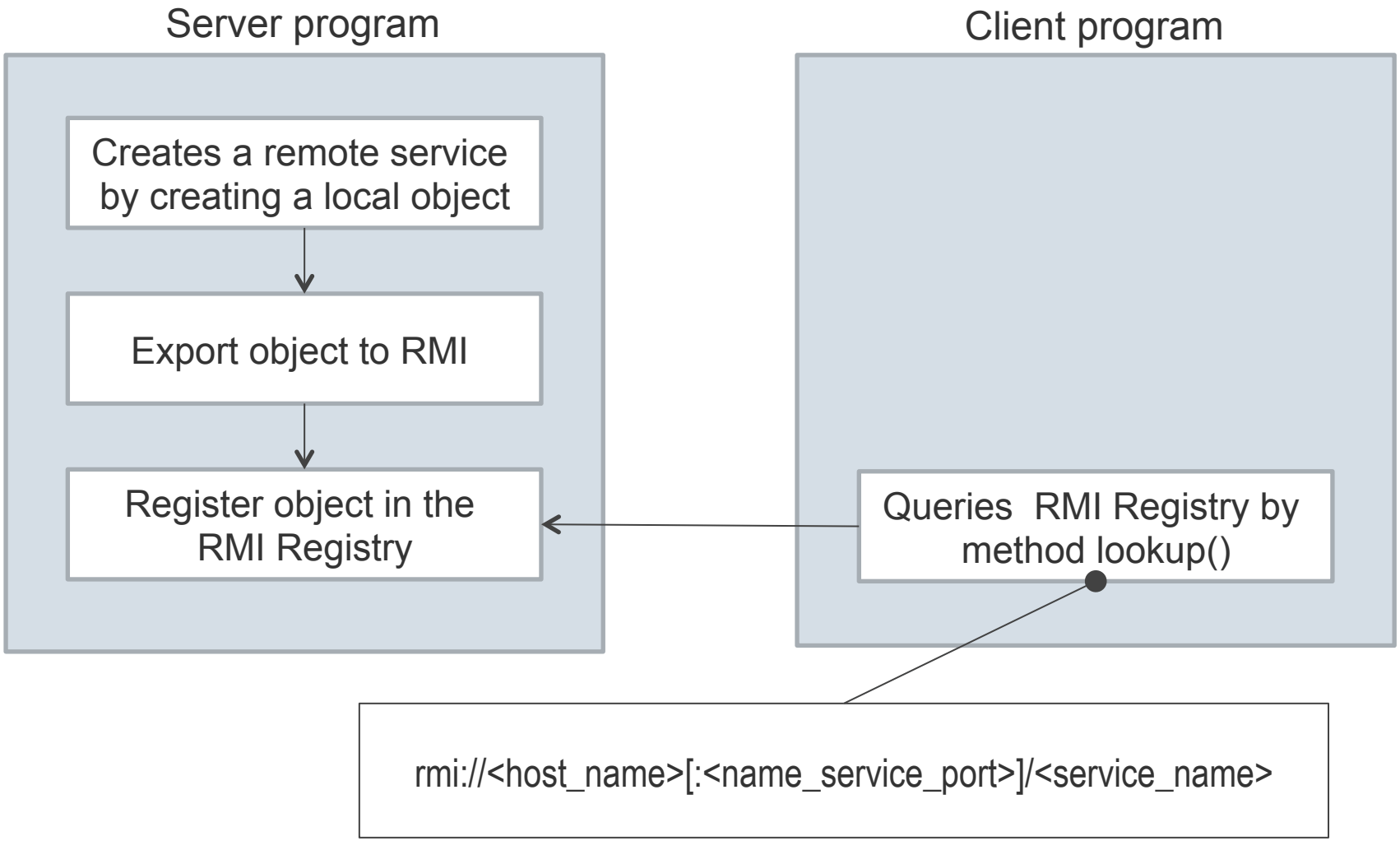
Naming Remote Objects

How does a client find a RMI remote service?

RMI includes a simple service called the RMI Registry, *rmiregistry*.

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

Naming Remote Objects (*cont.*)



Freie Universität Berlin

Java RMI architecture

Factory Design Pattern

Examples

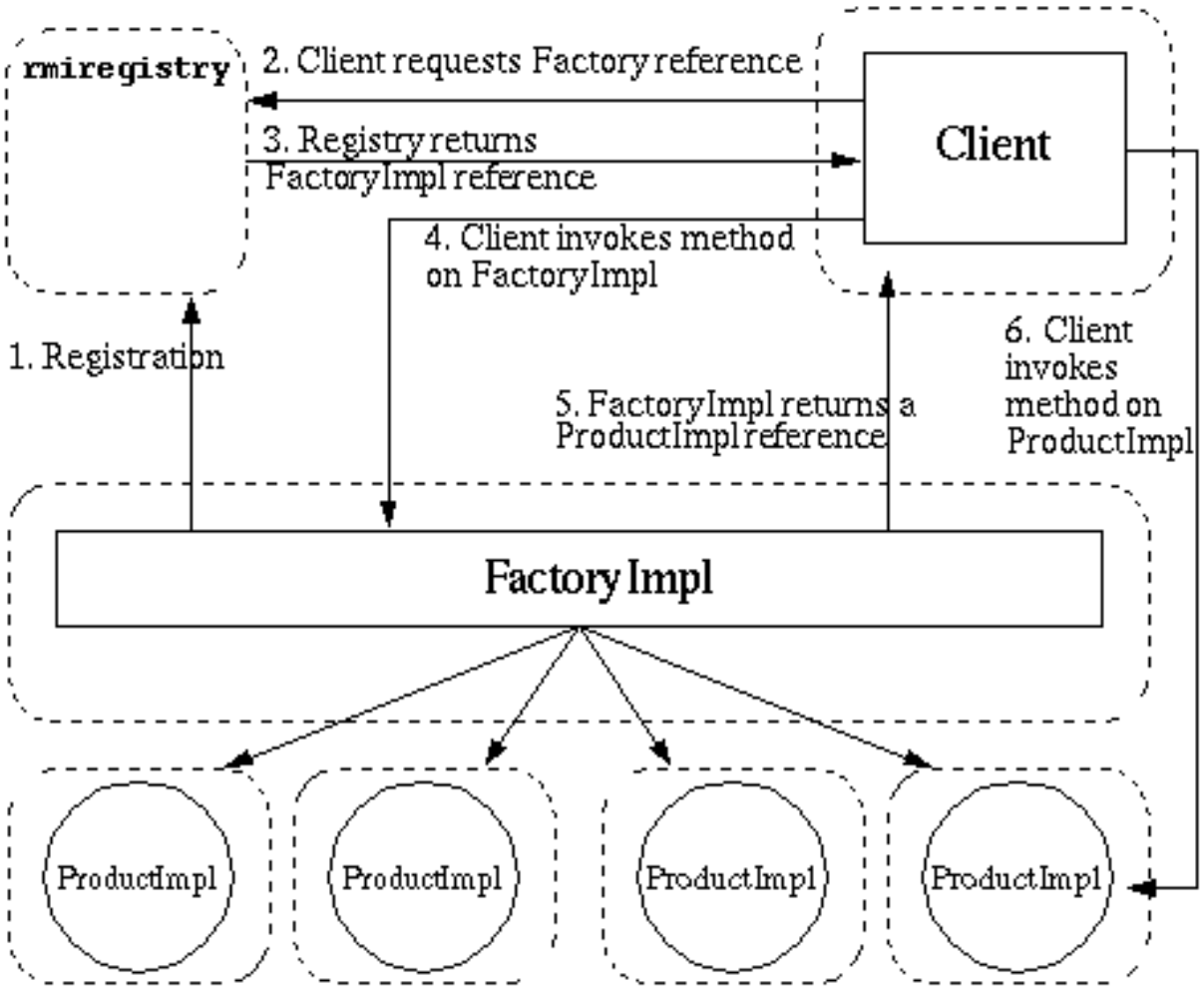
1

http://www.flickr.com/photos/23065375@N05/

2

http://www.flickr.com/photos/dunedinpubliclibraries/

Factory Design Pattern



How could the examples be implemented in Java RMI?

1

```
public interface AccountManager extends Remote {
    public Account createAccount() throws RemoteException;
}

public interface Account extends Serializable {
    public depositMoney();
    …
}
```

2

```
public interface Librarian extends Remote {
    public LibraryCard createLibraryCard() throws RemoteException;
}

public interface LibraryCard extends Serializable {
    …
    }
```

Java RMI architecture

Dynamic code downloading using RMI

Dynamic code loading

The Java VM interprets bytecode or compiles it on the fly and can load and run code dynamically.

Bytecode loading is encapsulated in a ClassLoader

- Developers can write custom ClassLoaders
- Can load bytecode from anywhere
- Specifically from the network

URLClassLoader (“out of the box”)

- Loads from a Uniform Resource Locator (URL) (per file://, ftp://, http://)

Well-known example are Java Applets.

```
<applet height=100 width=100 codebase="myclasses/" code="My.class">
<param name="ticker">
</applet>
```

What is the codebase

A codebase can be defined as a source, or a place, from which to load classes into a Java virtual machine.

CLASSPATH is your "local codebase", because it is the list of places on disk from which you load local classes. When loading classes from a local disk-based source, your CLASSPATH variable is consulted.

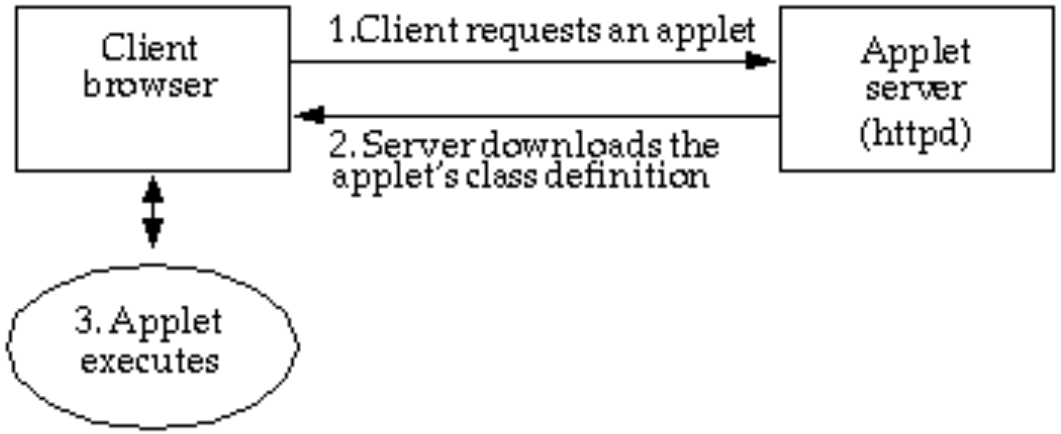
So just as CLASSPATH is a kind of "local codebase", the codebase used by applets and remote objects can be thought of as a "remote codebase". But other codebases must be supported by a ClassLoader.

ClassLoaders form a hierarchy

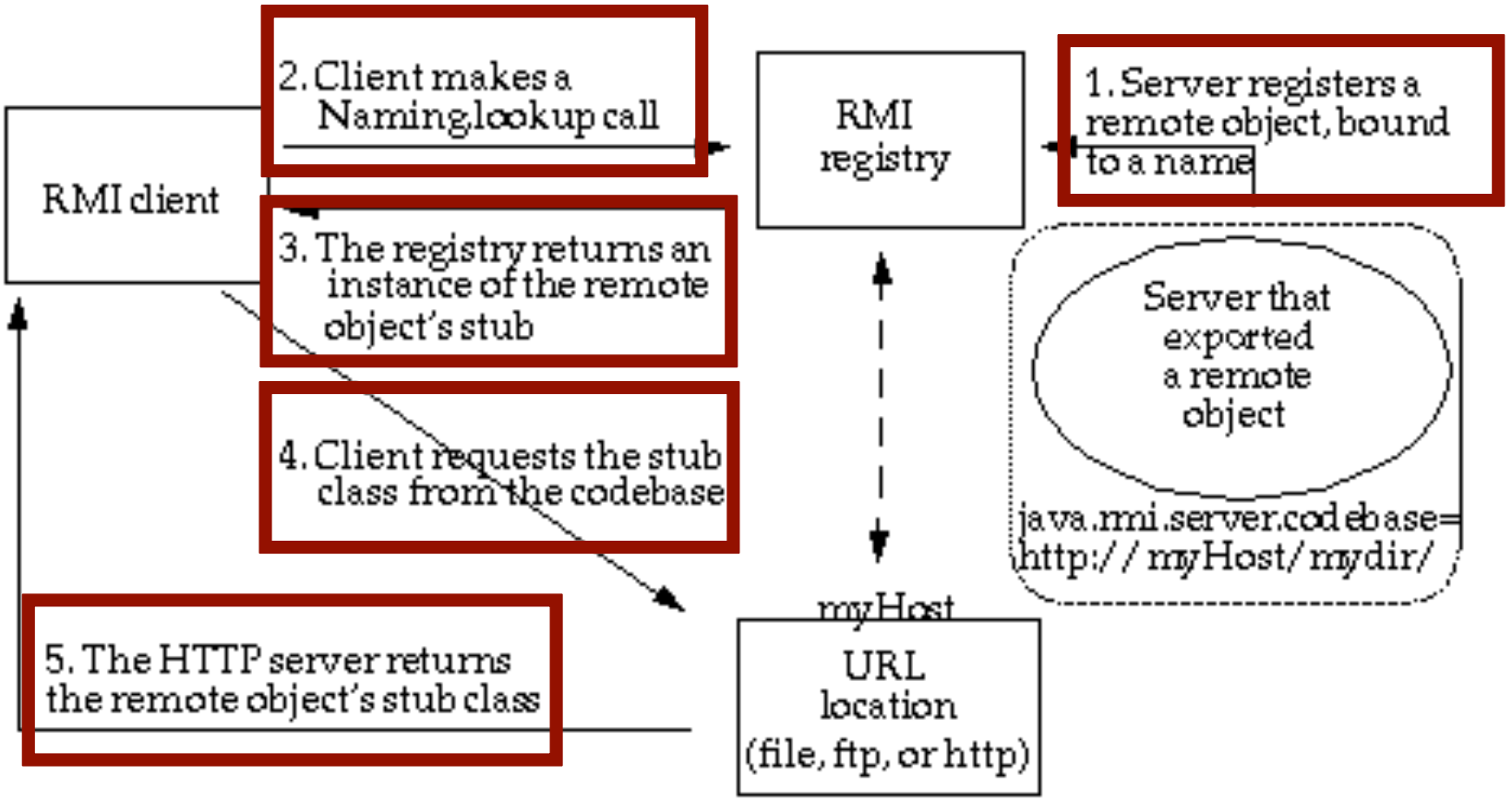
- Asks parent ClassLoader for class first, if not found then
- Loads the class itself

How codebase is used in applets

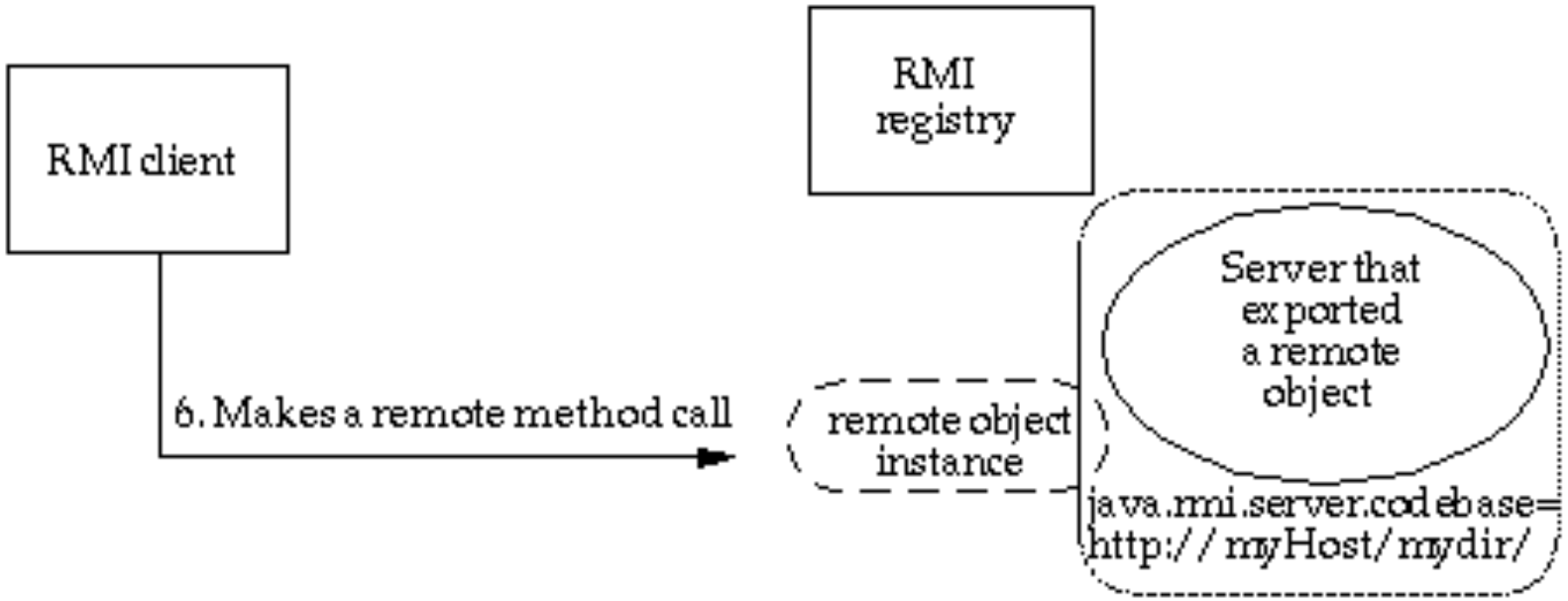
1. Initialize ClassLoader for Applet
2. ClassLoader first looks into CLASSPATH
3. If class not found then load from URL
4. Same for other classes the Applet needs



Codebases for Java RMI



RMI client making a remote method call



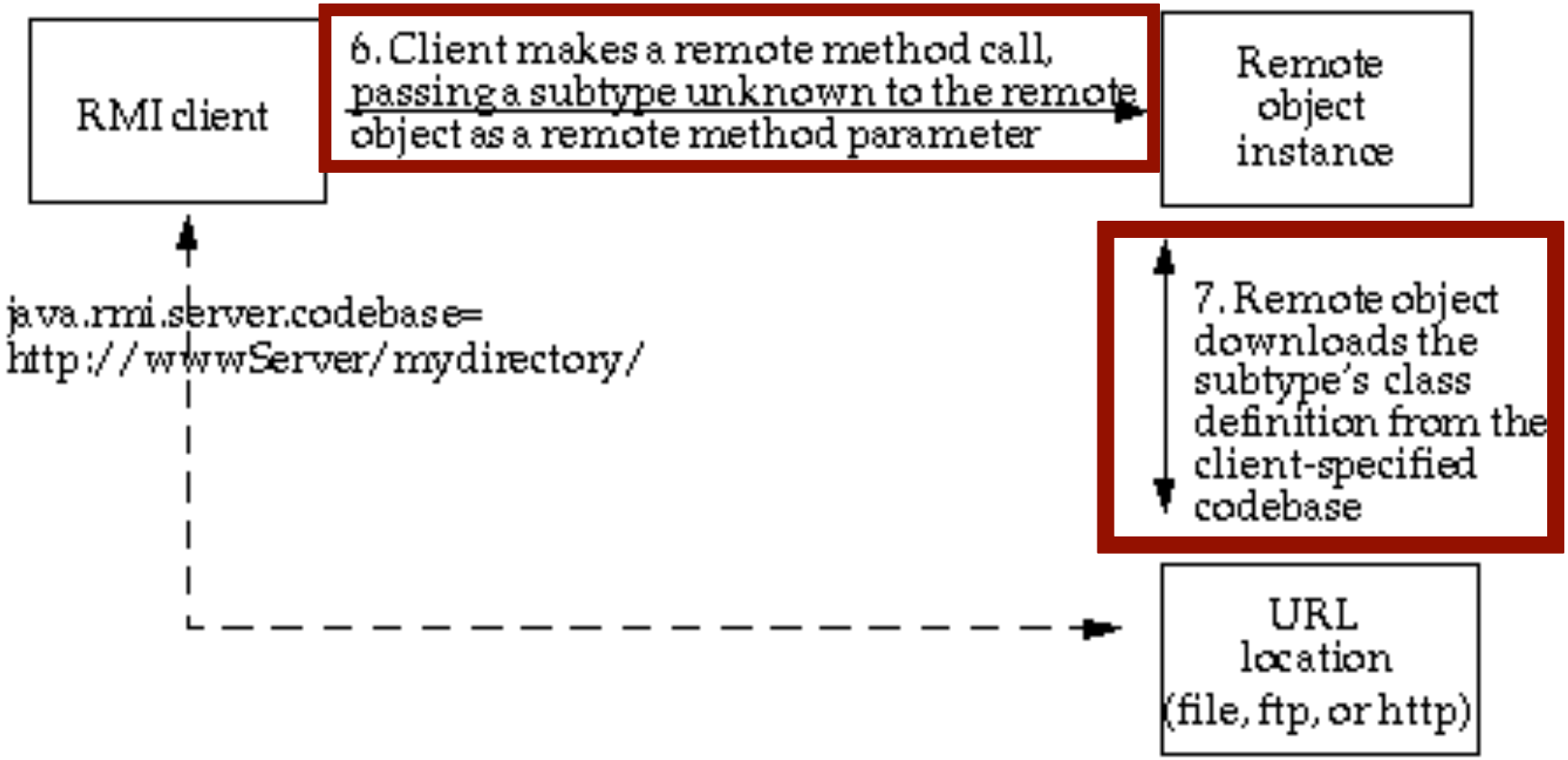
Using codebase in RMI for more than just stub downloading

The client may pass to the remote object:

1. Primitive data types
2. Types that are found in the remote CLASSPATH
3. Types unknown to the remote object
   - Implementation of known interface
   - Subclass of known type

How can the latter case be handled?

Using codebase in RMI for more than just stub downloading *(cont.)*



Freie Universität Berlin

Distributed object component middleware I

Java RMI case study

Case study: shared whiteboard

http://www.flickr.com/photos/36567420@N06/

Steps to develop an RMI application

Design and implement the components of your distributed application

- Define the remote interface(s)
- Implement the remote object(s)
- Implement the client(s)

Compile sources and generate stubs (and skeletons)

Make required classes network accessible

Run the application

Java Remote interfaces *Shape* and *ShapeList*

```
import java.rmi.*;
import java.util.Vector;

public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject  getAllState() throws RemoteException;
}

public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException;
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

Freie Universität Berlin

Java RMI

Building a client and server programs

Server program

The server is a whiteboard server which

- represents each shape as a remote object instantiated by a servant that implements the *Shape* interface
- holds the state of a graphical object as well as its version number

- represents its collection of shapes by using another servant that implements the *ShapeList* interface
- holds a collection of shapes in a *Vector*

Java class *ShapeListServer* with *main* method

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        try{
            ShapeList aShapeList = new ShapeListServant();
            ShapeList stub =
                        (ShapeList) UnicastRemoteObject.exportObject(aShapeList,0);
            Naming.rebind("//bruno.ShapeList", stub);
            System.out.println("ShapeList server ready");
            }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());}
    }
}
```

Java class *ShapeListServant* implements interface *ShapeList*

```
import java.util.Vector;

public class ShapeListServant implements ShapeList {
      private Vector theList;              // contains the list of Shapes
      private int version;
     public ShapeListServant() {...}
     public Shape newShape(GraphicalObject g) {
              version++;
              Shape s = new ShapeServant( g, version);
              theList.addElement(s);
              return s;
     }
     public  Vector allShapes() {...}
     public int getVersion() { ... }
}
```

Java client of *ShapeList*

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;

public class ShapeListClient{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        ShapeList aShapeList = null;
        try{
            aShapeList  = (ShapeList) Naming.lookup("//bruno.ShapeList");
            Vector sList = aShapeList.allShapes();
        } catch(RemoteException e) {System.out.println(e.getMessage());
        }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
    }
}
```

Callbacks

Disadvantages of polling

1. The performance of the server may be degraded by constant polling.
2. Clients cannot notify users of updates in a timely manner.

Procedure callbacks

- The clients creates a remote object that implements an interface that contains a method for the server to call. We refer to this as a callback object.
- The server provides an operation allowing interested clients to inform it of the remote object references of their callback objects. It records these in a list.
- Whenever an event of interest occurs, the server calls the interested clients.

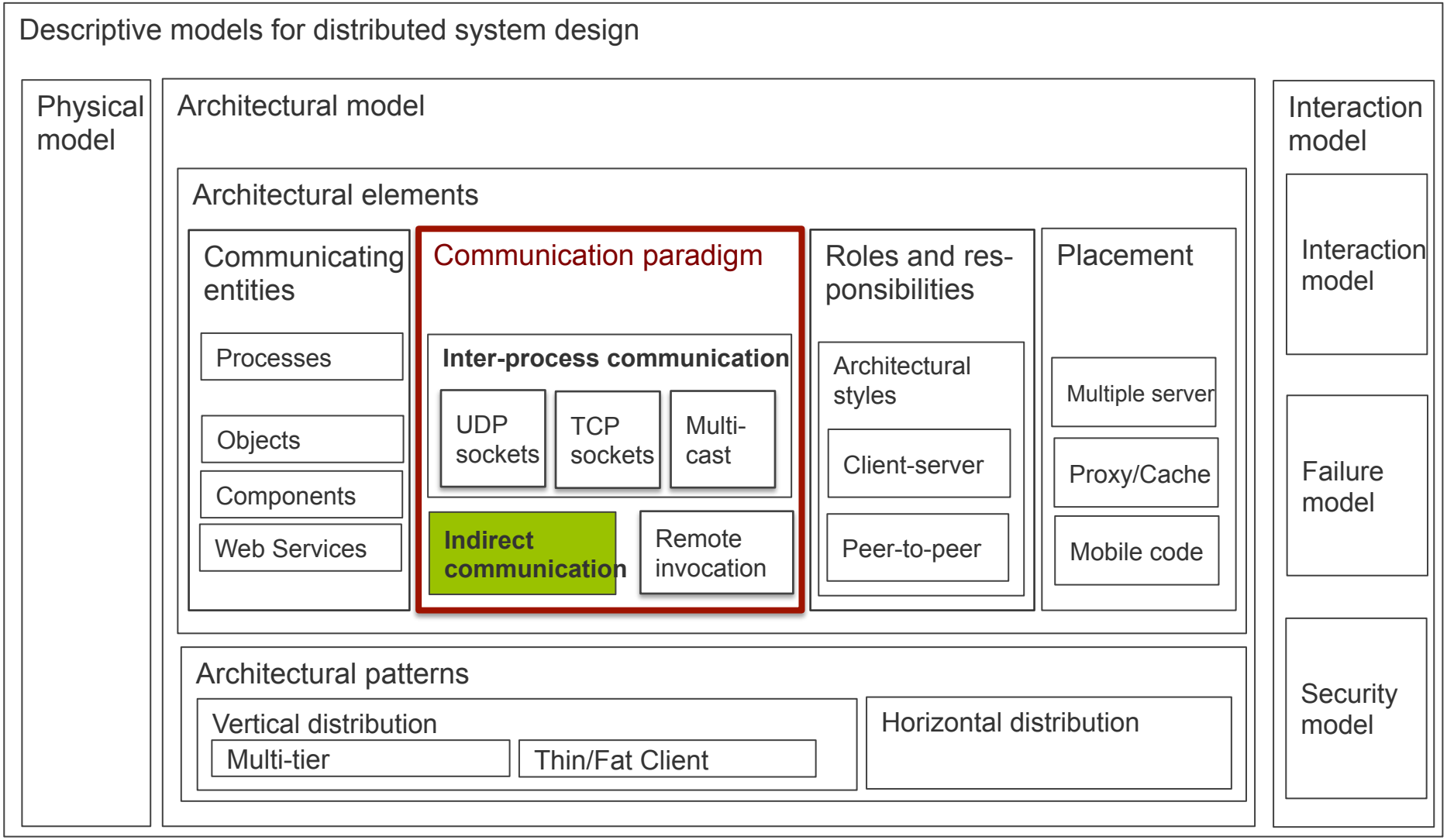
Remote method invocation

Summary

We have we learned?

- General components of an RMI infrastructure
- Core concepts of the RMI infrastructure
- The three RMI architecture layers
- Proxy design pattern
- Reflections
- Process how a client finds a RMI remote service
- Factory design pattern
- Dynamic code loading in RMI

Our topics next week



Freie Universität Berlin

Next class

Indirect Communication

References

Main resource for this lecture:

George Coulouris, Jean Dollimore, Tim Kindberg: *Distributed Systems: Concepts and Design*. 5th edition, Addison Wesley, 2011

Oracle Remote Method Invocation (RMI):

http://java.sun.com/developer/onlineTraining/rmi/RMI.html#BooksAndArticles

Dynamic Code Loading:

http://download.oracle.com/javase/1.4.2/docs/guide/rmi/codebase.html

Security in Java RMI

http://www.eg.bucknell.edu/~cs379/DistributedSystems/rmi_tut.html#secure