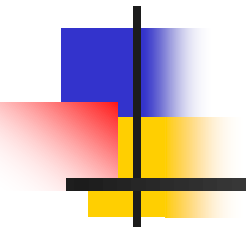
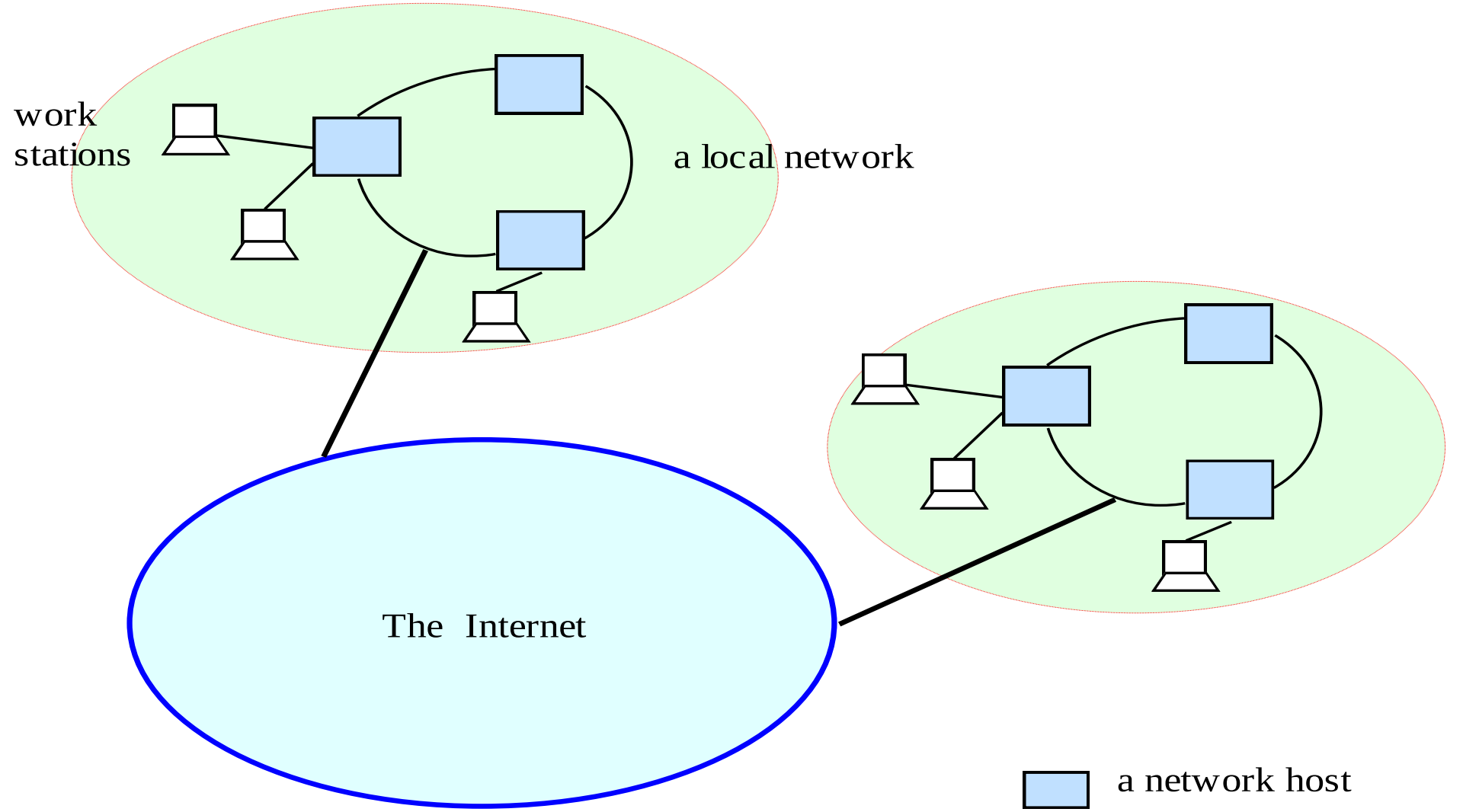


Java RMI
(Remote Method Invocation)

Distributed system, distributed computing

- Early computing was performed on a single processor. Uni-processor computing can be called *centralized computing*.
- A *distributed system* is a collection of independent computers, interconnected via a network, capable of collaborating on a task.
- *Distributed computing* is computing performed in a distributed system.

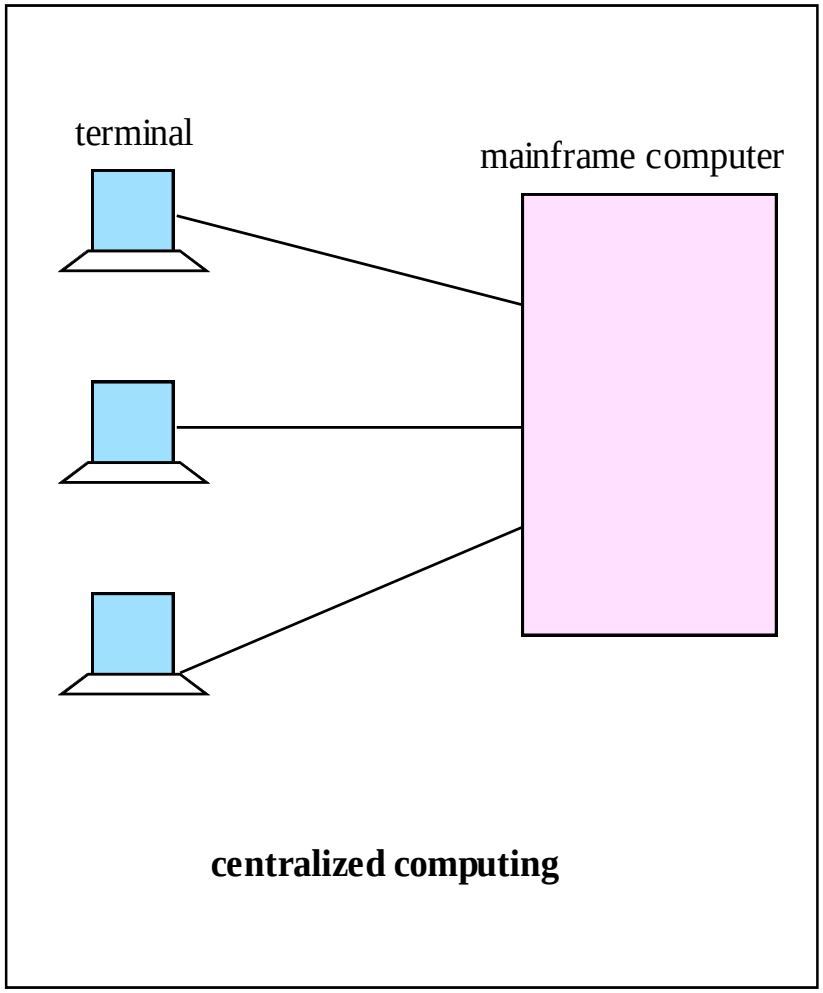
Distributed Systems

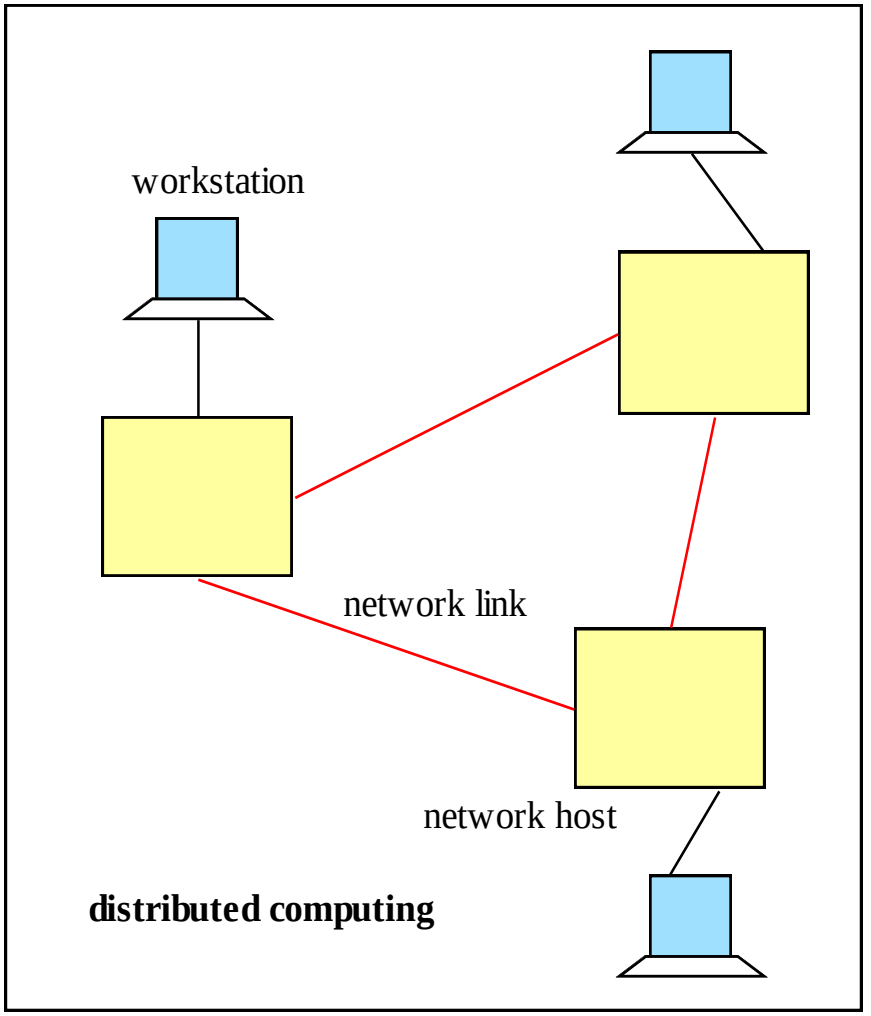


Examples of Distributed systems

- Network of workstations (NOW): a group of networked personal workstations connected to one or more server machines.
- The Internet
- An intranet: a network of computers and workstations within an organization, segregated from the Internet via a protective device (a firewall).

Centralized vs. Distributed Computing





Why distributed computing?

- **Economics**: distributed systems allow the pooling of resources, including CPU cycles, data storage, input/output devices, and services.
- **Reliability**: a distributed system allow replication of resources and/or services, thus reducing service outage due to failures.
- The Internet has become a universal platform for distributed computing.

Dist. Computing: Weaknesses and Strengths

In any form of computing, there is always a trade off in advantages and disadvantages. Some of the reasons for the popularity of distributed computing :

- **The affordability of computers and availability of network access**
- **Resource sharing**
- **Scalability**
- **Fault Tolerance**

Dist. Computing: Weaknesses and Strengths

The disadvantages of distributed computing:

- **Multiple Points of Failures:** the failure of one or more participating computers, or one or more network links, can spell trouble.
- **Security Concerns**: In a distributed system, there are more opportunities for unauthorized attack.

Related Technologies

- **RPC** ("Remote Procedure Calls")
  - Developed by Sun
  - Platform-specific
- **CORBA** ("Common Object Request Broker Architecture")
  - Developed by OMG
  - Access to non-Java objects (as well as Java)
- **DCOM** ("Distributed Component Object Model")
  - Developed by Microsoft
  - Access to Win32 objects
- **LDAP** ("Lightweight Directory Access Protocol")
  - Finding resources on a network

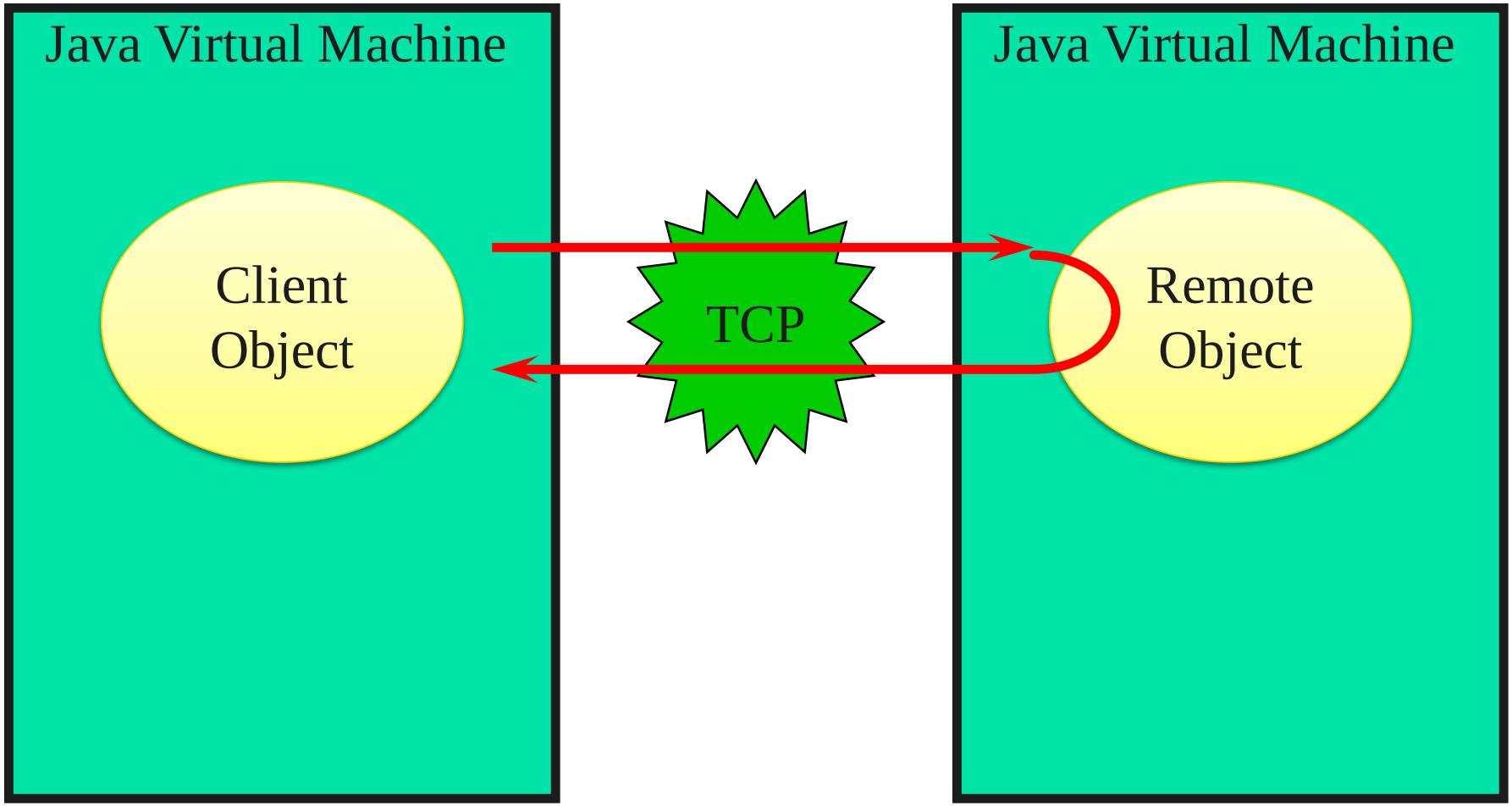
RMI Overview

- Remote Method Invocation (RMI) is a distributed systems technology that allows one Java Virtual Machine (JVM) to invoke object methods that will be run on another JVM located elsewhere on a network.
- RMI is a Java technology that allows one JVM to communicate with another JVM and have it execute an object method. Objects can invoke methods on other objects located remotely as easily as if they were on the local host machine
- Each RMI service is defined by an interface, which describes object methods that can be executed remotely. This interface must be shared by all developers who will write software for that service—it acts as a blueprint for applications that will use and provide implementations of the service.

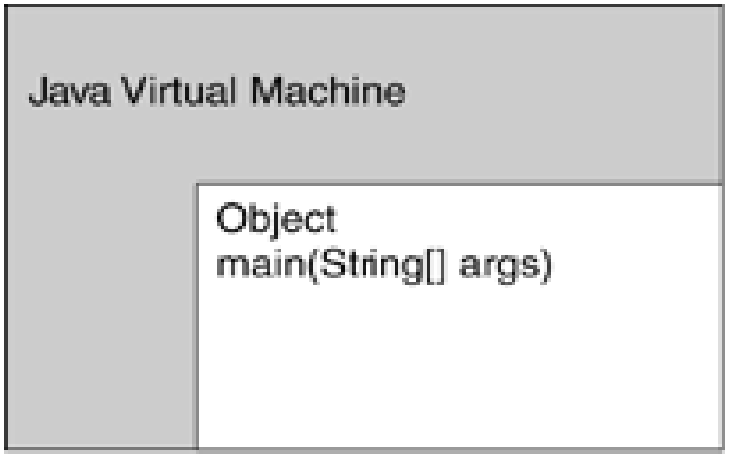
What Is RMI?

- Access to Remote Objects
- Java-to-Java only
- Client-Server Protocol
- High-level API
- Transparent
- Lightweight

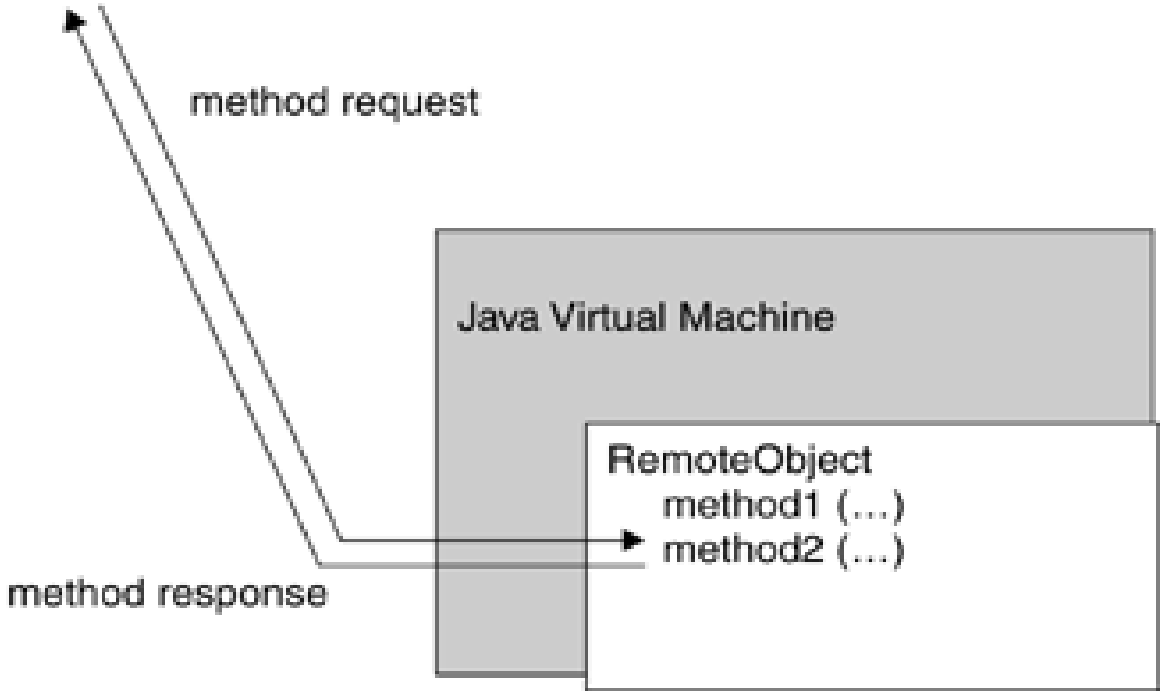
Remote Objects (Diagram)



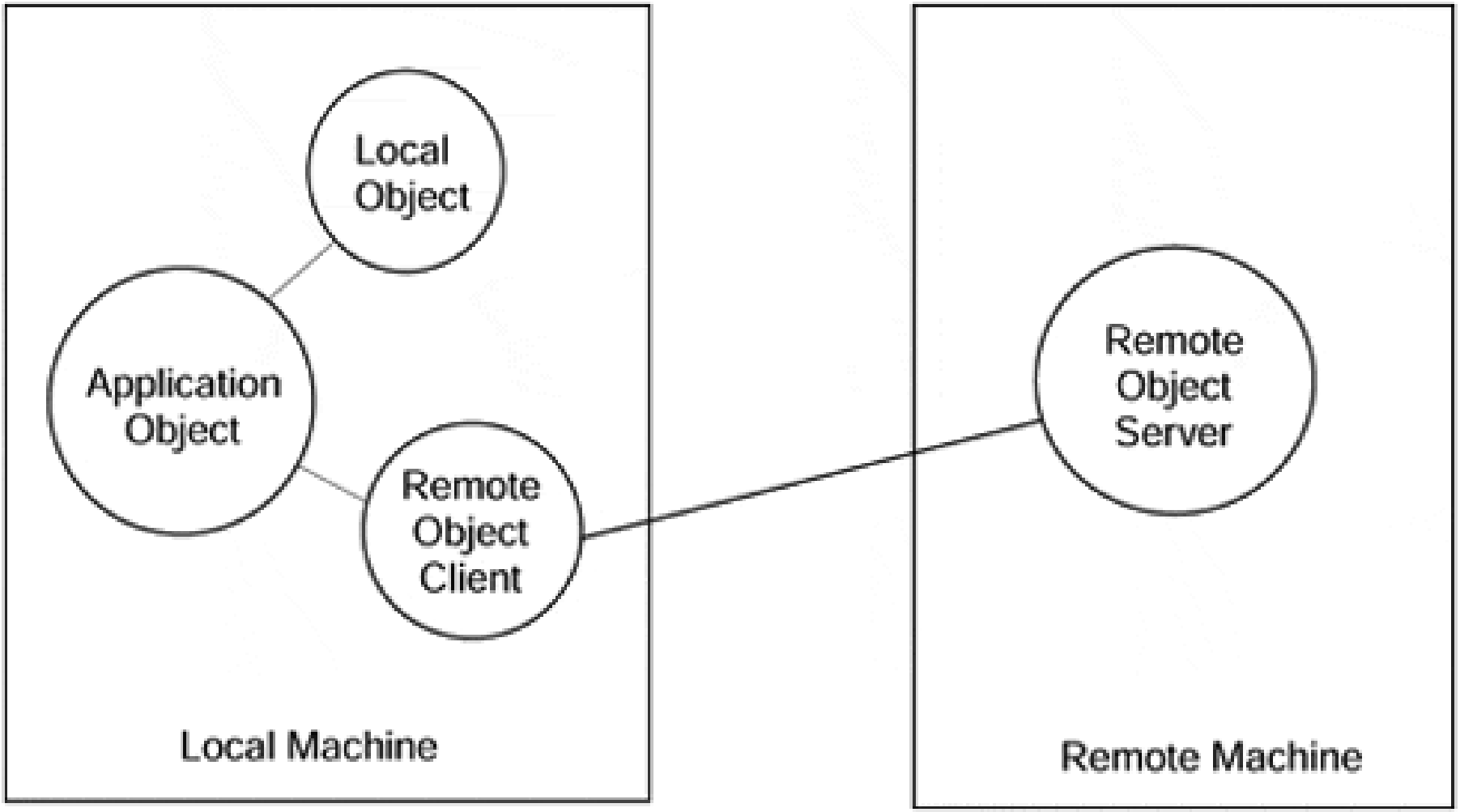
Remote method invocation



Invocation of a method on a remote object, executing on a remote machine

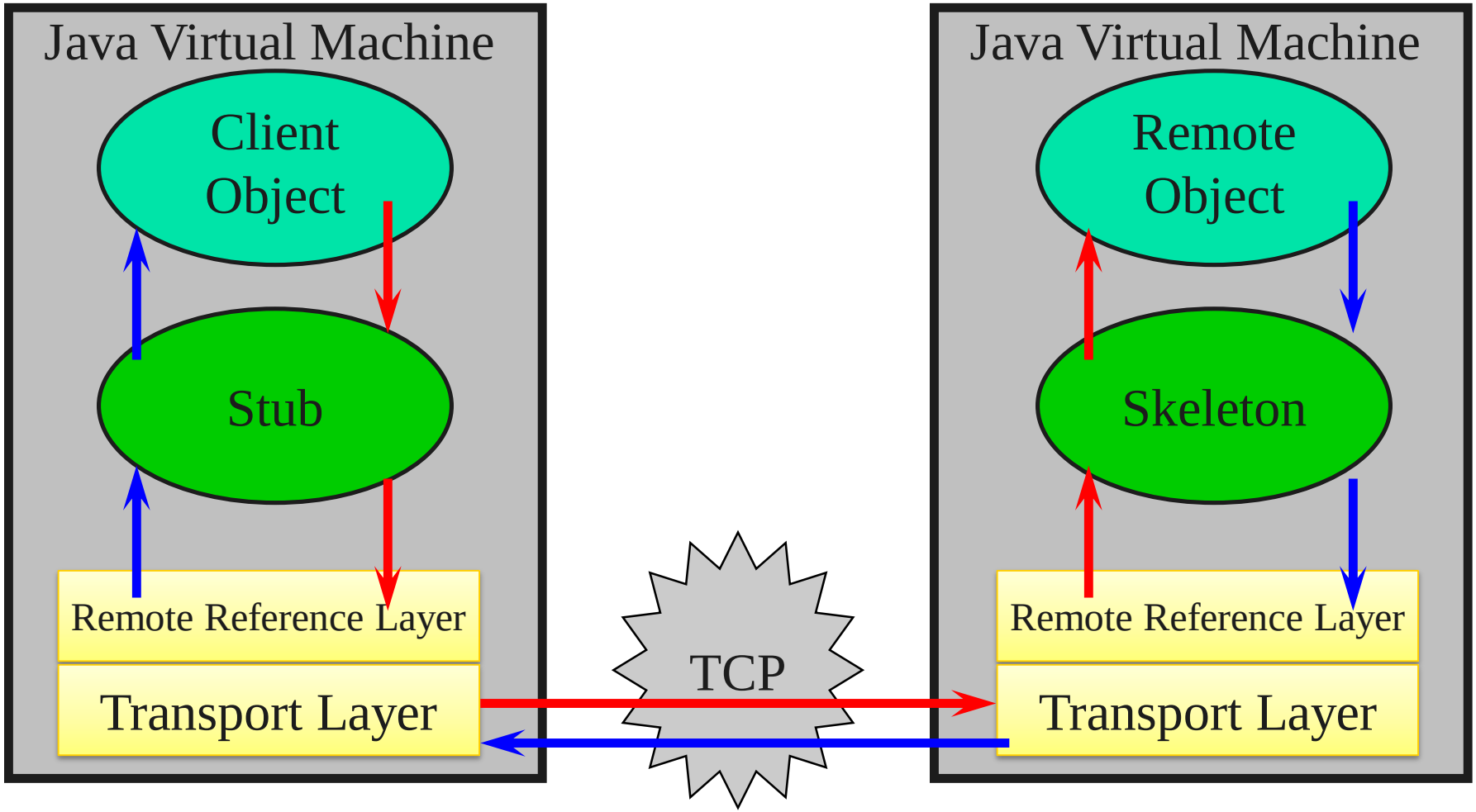


Local object – Remote object



Invocations on remote objects appear the same as invocations on local objects.

RMI Architecture - Layers



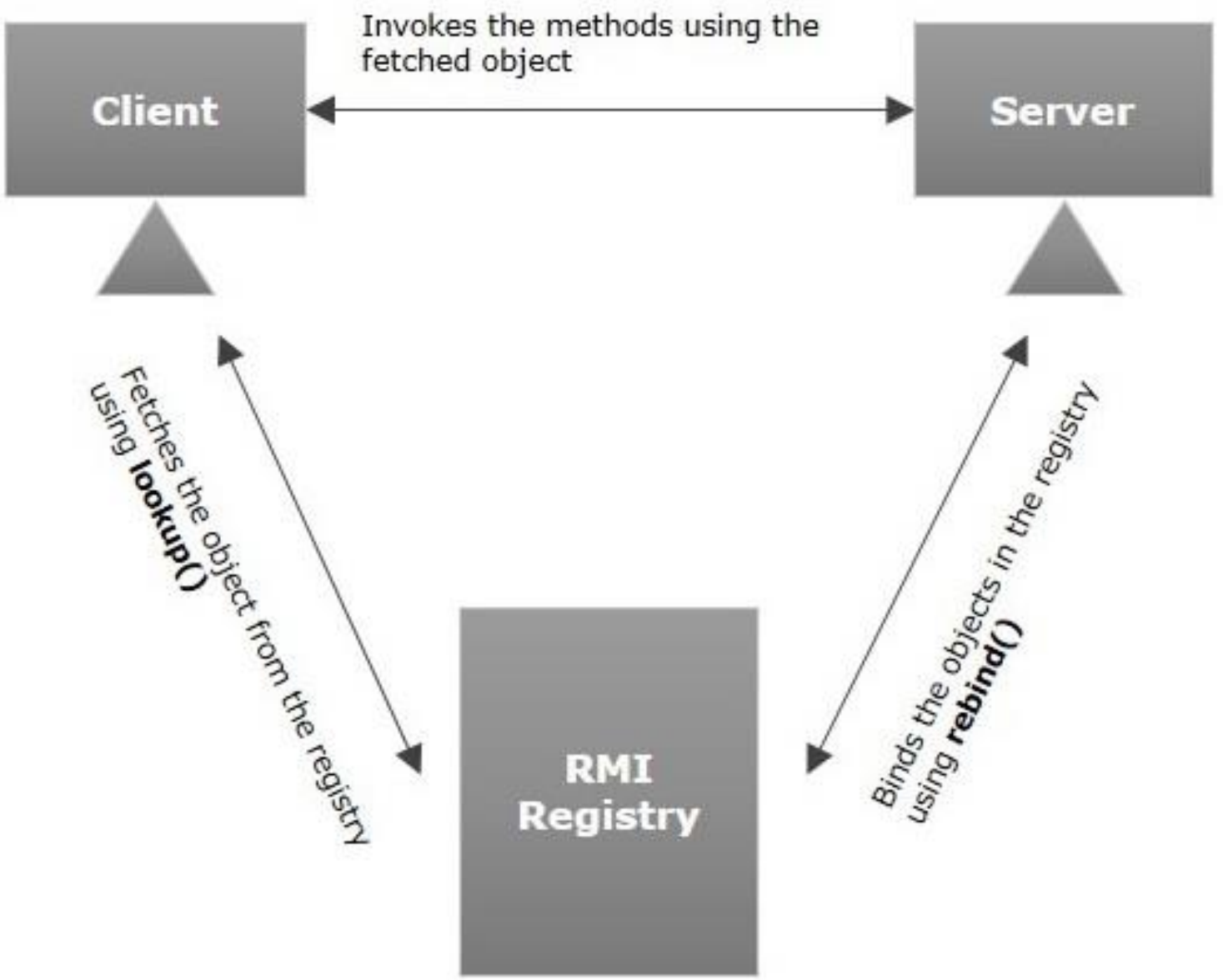
RMI Architecture - Layers

- **Transport Layer** – This layer connects the client and the server. It manages the existing connection and also sets up new connections.
- **Stub** – A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program.
- **Skeleton** – This is the object which resides on the server side. **stub** communicates with this skeleton to pass request to the remote object.
- **RRL(Remote Reference Layer)** – It is the layer which manages the references made by the client to the remote object.

RMI Registries

- Name and look up remote objects
- Servers can register their objects
- Clients can find server objects and obtain a remote reference
- A registry is a running process on a host machine

RMI Registries



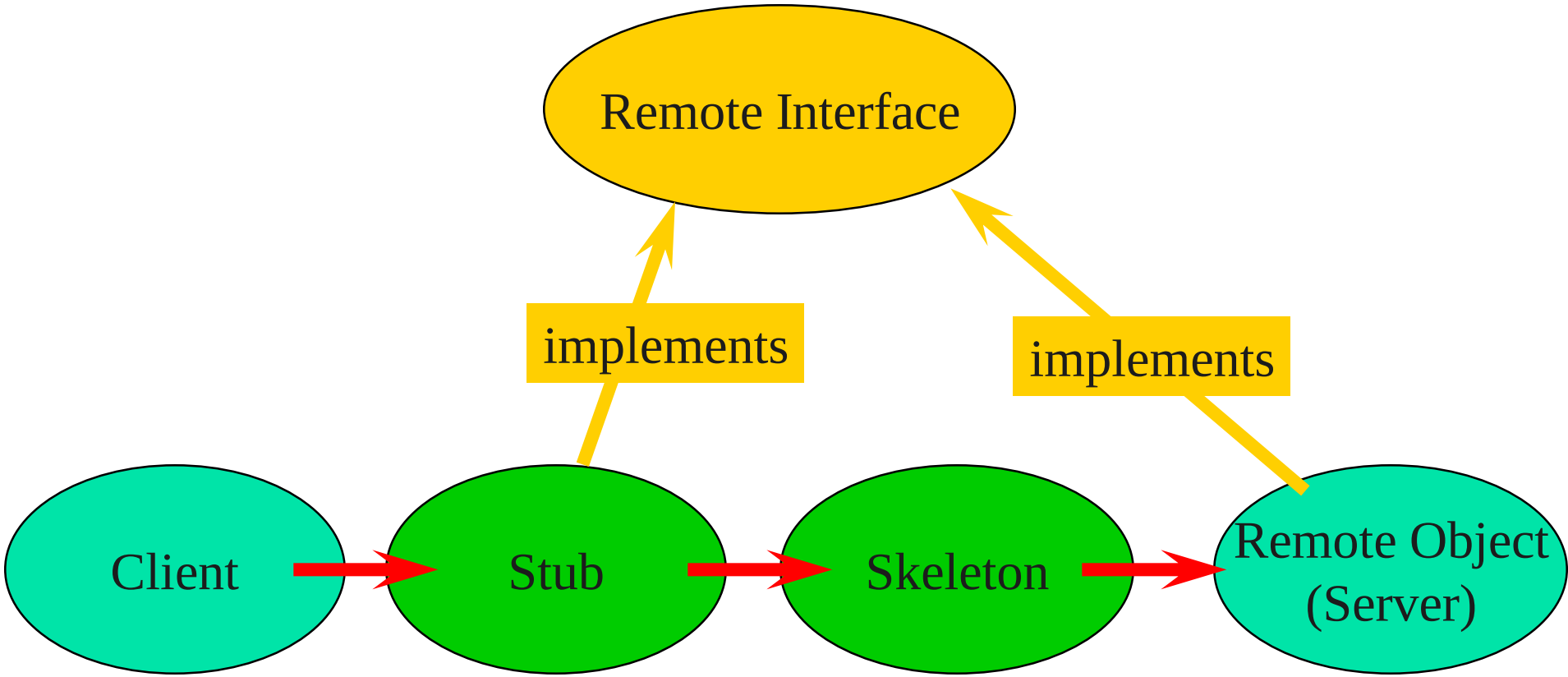
Remote References and Interfaces

- Remote References
  - Refer to remote objects
  - Invoked on client exactly like local object references
- Remote Interfaces
  - Declare exposed methods
  - Implemented on client
  - Like a proxy for the remote object

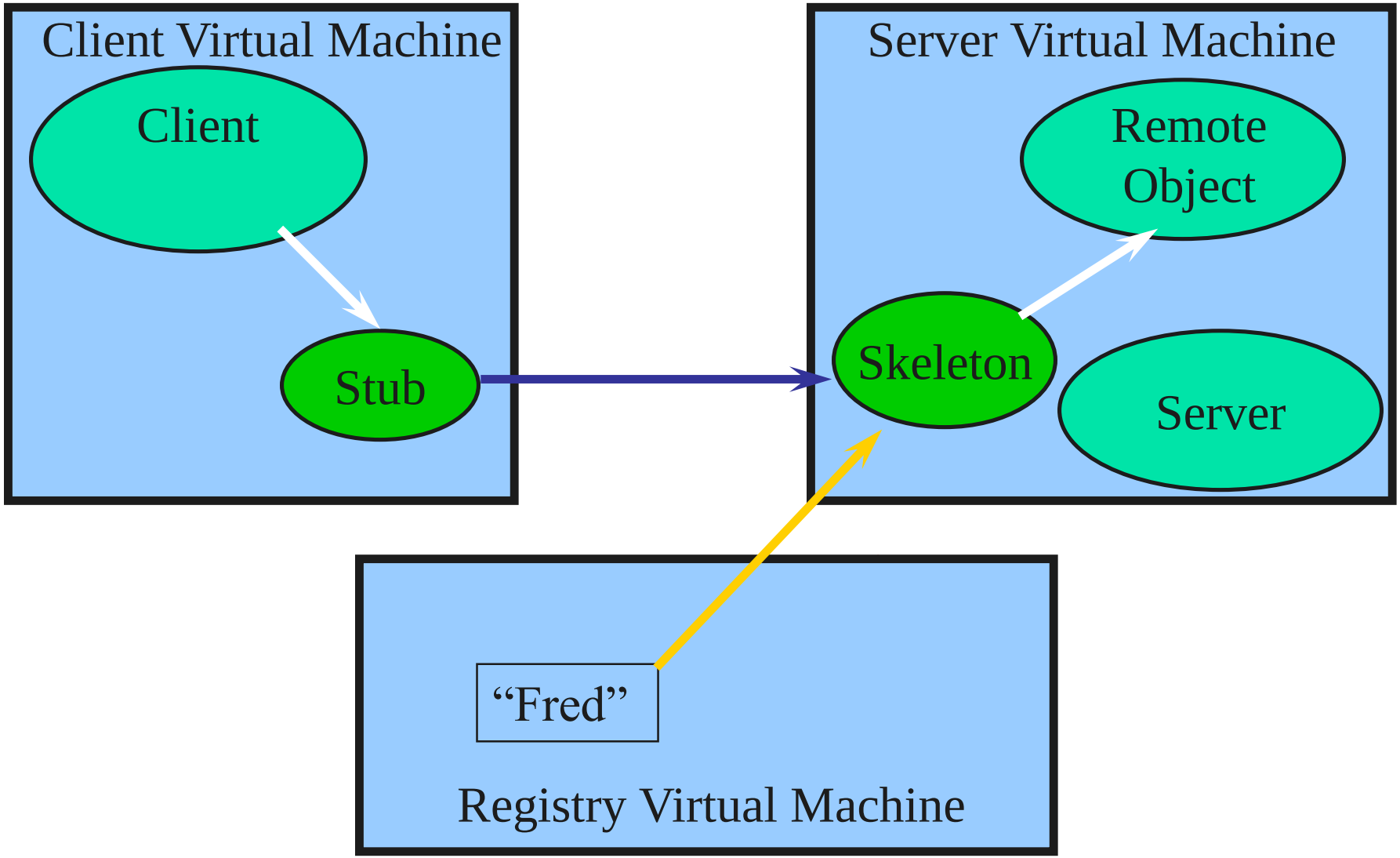
Stubs and Skeletons

- Stub
  - lives on client
  - pretends to be remote object
- Skeleton
  - lives on server
  - receives requests from stub
  - talks to true remote object
  - delivers response to stub

Remote Interfaces and Stubs

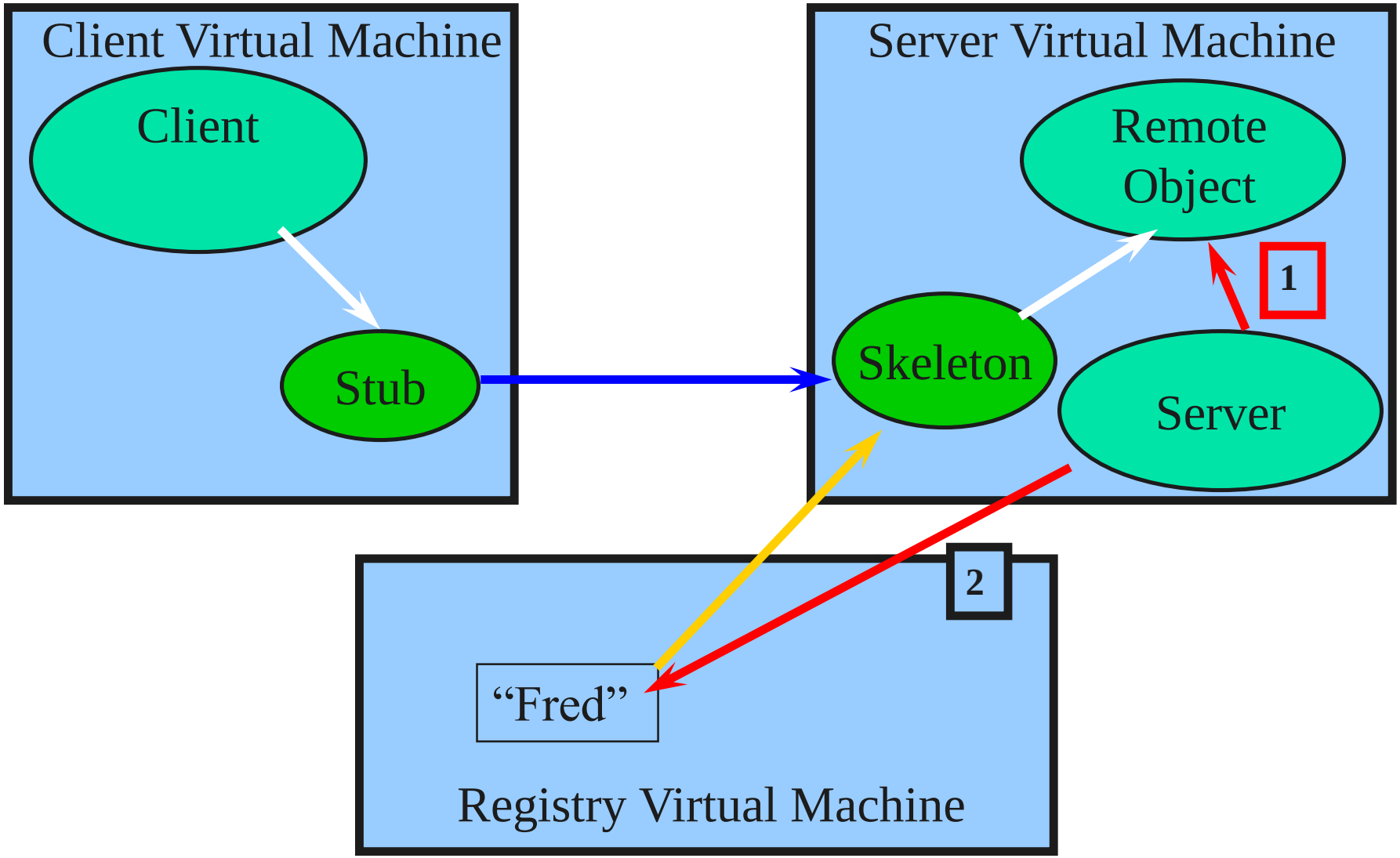


RMI System Architecture



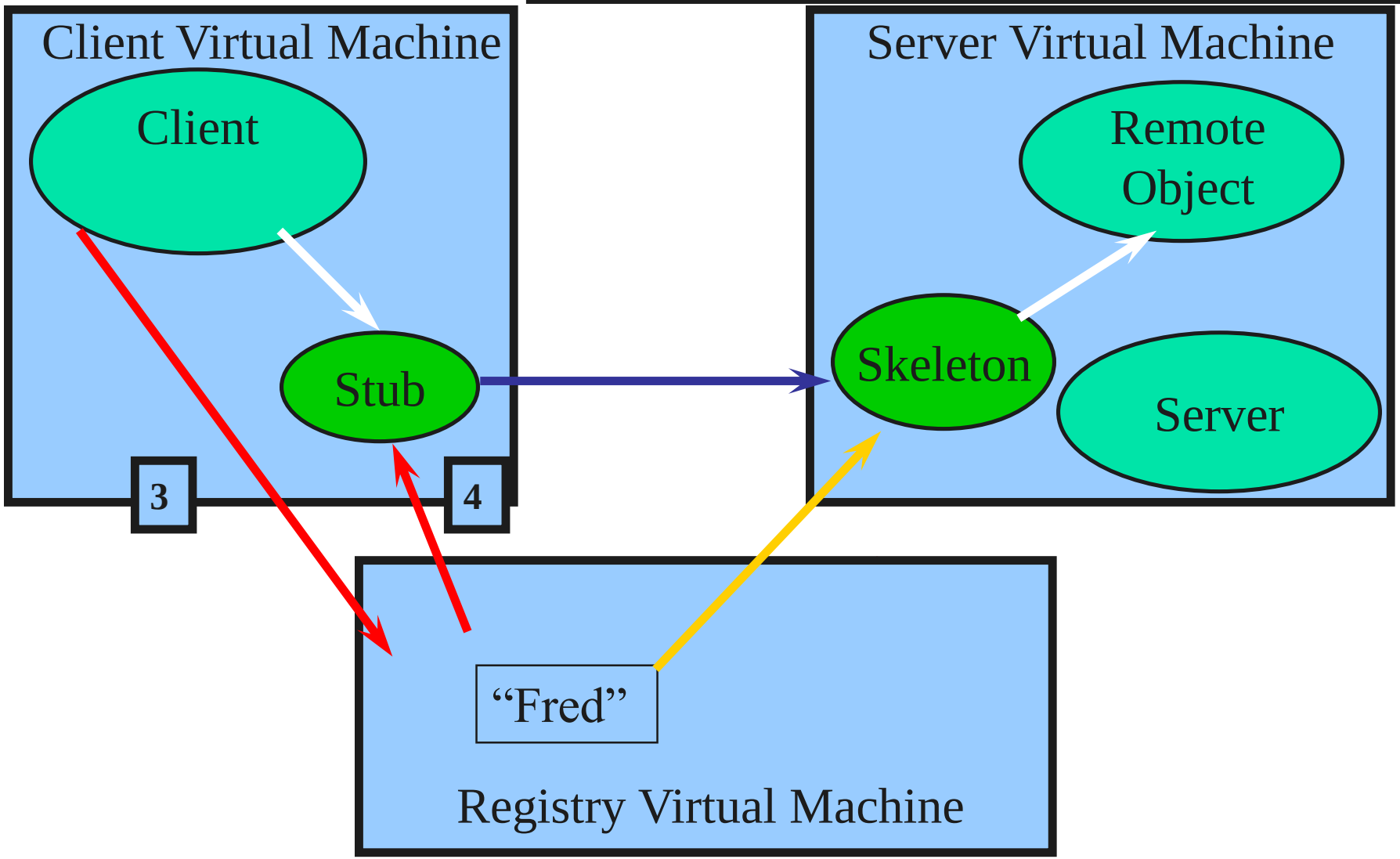
RMI Flow

1. Server Creates Remote Object
2. Server Registers Remote Object



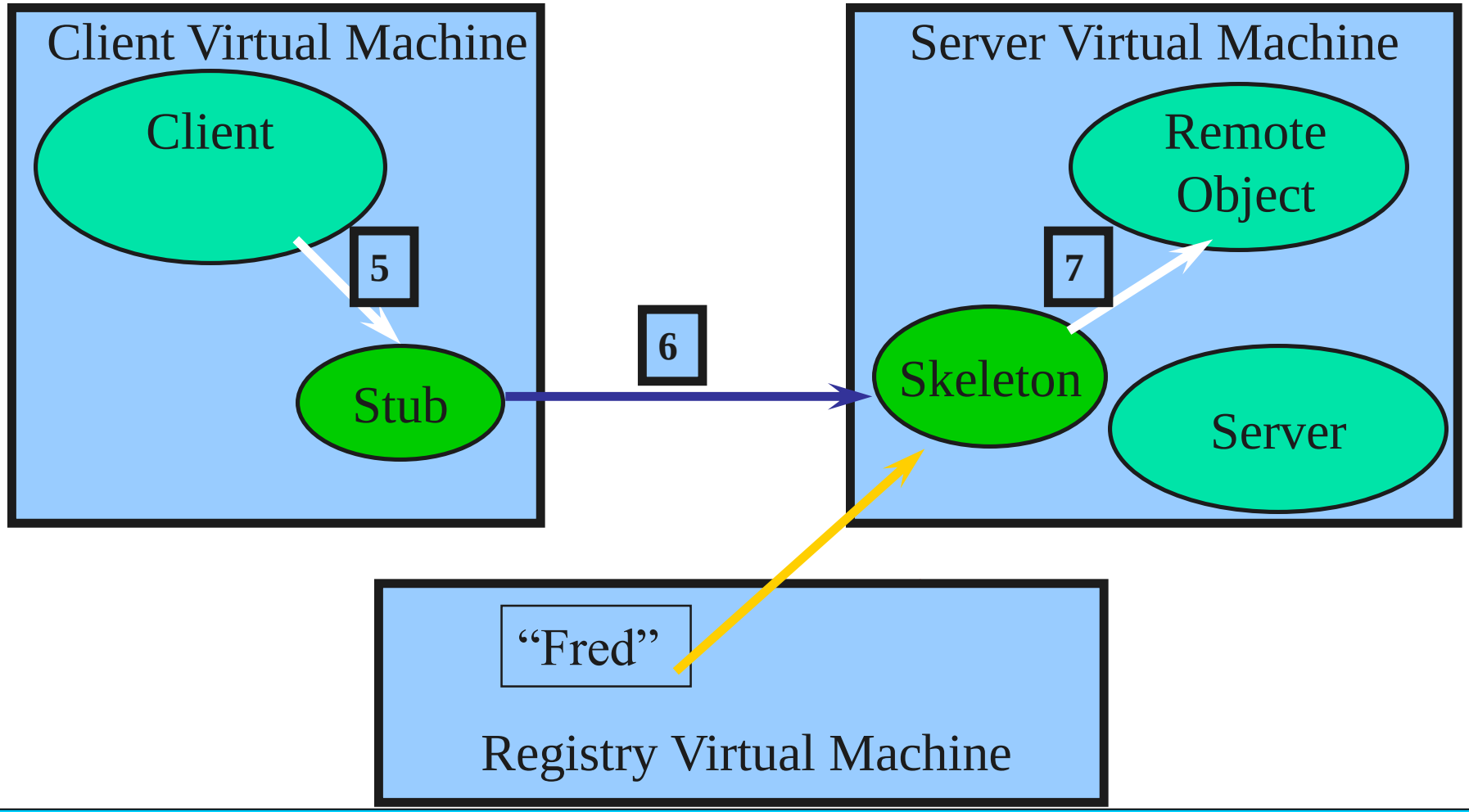
RMI Flow

3. Client requests object from Registry
4. Registry returns remote reference (and stub gets created)

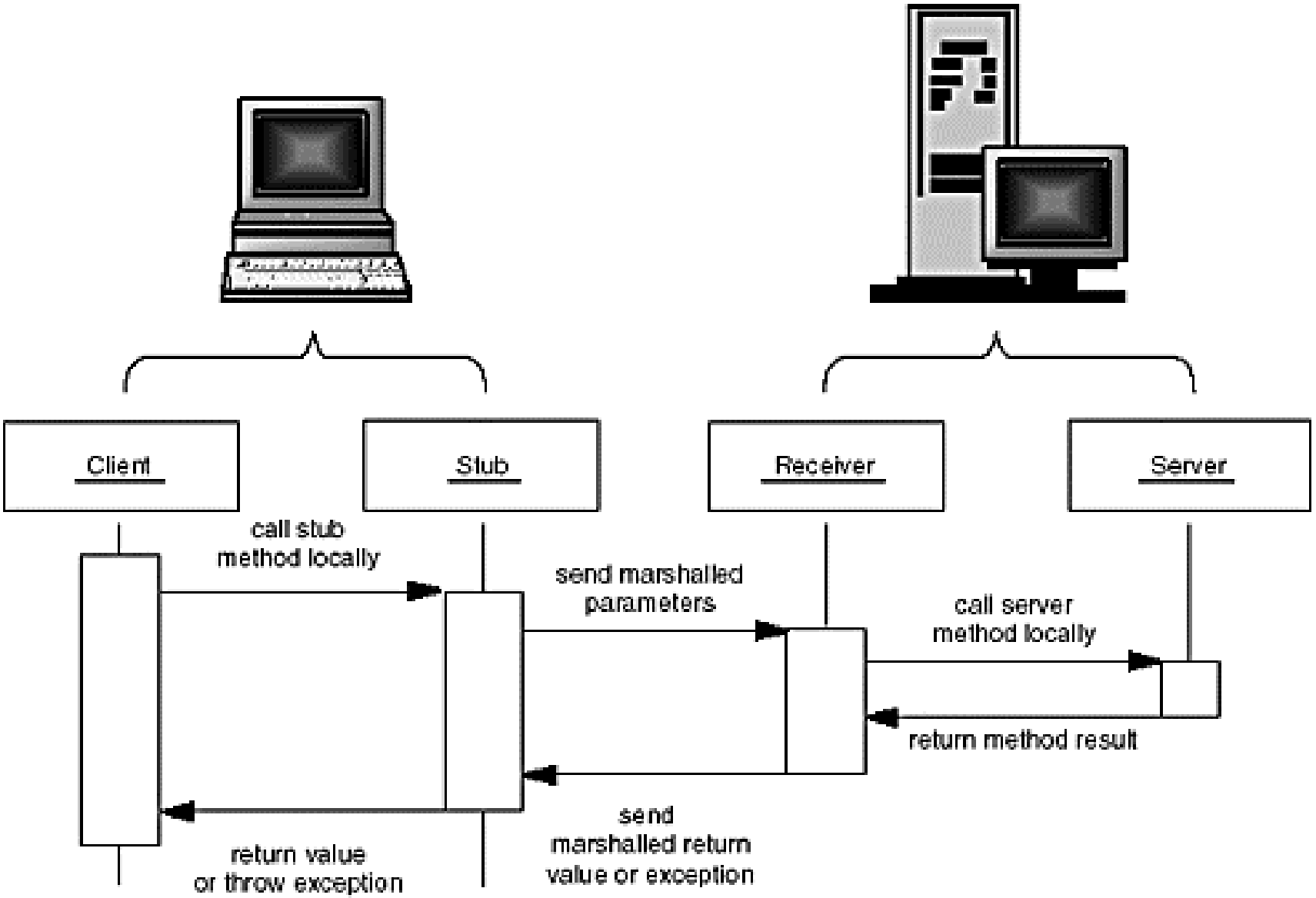


RMI Flow

5. Client invokes stub method
6. Stub talks to skeleton
7. Skeleton invokes remote object method



Parameter marshalling



RMI Applications

- RMI applications are often comprised of two separate programs: a server and a client.
- A typical server application creates some remote objects, makes references to them accessible, and waits for clients to invoke methods on these remote objects.
- A typical client application gets a remote reference to one or more remote objects in the server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth.
- Such an application is sometimes referred to as a *distributed object application*.

Java RMI Application

- Define the remote interface
- Develop the implementation class (remote object)
- Develop the server program
- Develop the client program
- Compile the application
- Execute the application

Defining the Remote Interface

- A remote interface provides the description of all the methods of a particular remote object. The client communicates with this remote interface.

1. Create an interface that extends the predefined interface **Remote** which belongs to the package.
2. Declare all the business methods that can be invoked by the client in this interface.
3. Since there is a chance of network issues during remote calls, an exception named **RemoteException** may occur; throw it.

- Define a Remote Interface
  - extends java.rmi.Remote

Designing a Remote Interface

```
package rmidemo;
import java.rmi.*;
public interface IProduct extends Remote{
    String getDescription() throws
                        RemoteException;
}
```

Developing the Implementation Class

- In general the implementation class of a remote interface should at least
  - Declare the remote interfaces being implemented
  - Define the constructor for the remote object
  - Provide an implementation for each remote method in the remote interfaces
- Define a class that implements the Remote Interface
  - extends java.rmi.RemoteObject
  - or java.rmi.UnicastRemoteObject

Implementing a Remote Interface

```
import java.rmi.*;
import java.rmi.server.*;
public class ProductImpl  extends UnicastRemoteObject
                                  implements IProduct {
 private String name;
 public ProductImpl(String n) throws RemoteException {
       super();
       name = n;
   }
 public String getDescription() throws RemoteException {
      return "I am a " + name + ". Buy me!";
   }
}
```

Compiling Remote Classes

- Compile the Java class
  - **javac**
    - reads .java file
    - produces .class file
- Compile the Stub and Skeleton
  - **rmic**
    - reads .class file
    - produces _Skel.class and _Stub.class

Naming conventions for RMI classes

- `IProduct` A remote interface
- `ProductImpl` A server class implementing that interface
- `ProductImpl_Stub` A stub class that is automatically generated by the `rmic` program
- `ProductImpl_Skel` A skeleton class that is automatically generated by the `rmic` program needed for SDK 1.1
- `ProductServer` A server program that creates server objects
- `ProductClient` A client program that calls remote methods

RMI Applications from JDK 1.5

- Pregenerating a stub class for a remote object's class is only required if the remote object needs to support pre-5.0 clients. As of the 5.0 release, if a pregenerated stub class for a remote object's class cannot be loaded when the remote object is exported, the remote object's stub class is generated dynamically.
  - Designing a Remote Interface
  - Implementing a Remote Interface
  - Creating a RMI Server
  - Createing a RMI Client

RMI Applications from JDK 1.5

Creating a RMI Server

- Creates and exports a Registry instance on the local host that accepts requests on the specified port

```
Registry RMIReg =
        LocateRegistry.createRegistry(1099);
```

- Creates a remote object

```
ProductImpl EOS350D =
        new ProductImpl("Canon EOS 350D");
```

- Binds a remote reference to the specified name in this registry

```
RMIReg.bind("EOS350D", EOS350D);
```

RMI Applications from JDK 1.5

Creating a RMI Client

- Getss a reference to the remote object Registry on the specified host and port. If host is null, the local host is used.

```
Registry RMIReg =
    LocateRegistry.getRegistry("localhost",1099);
```

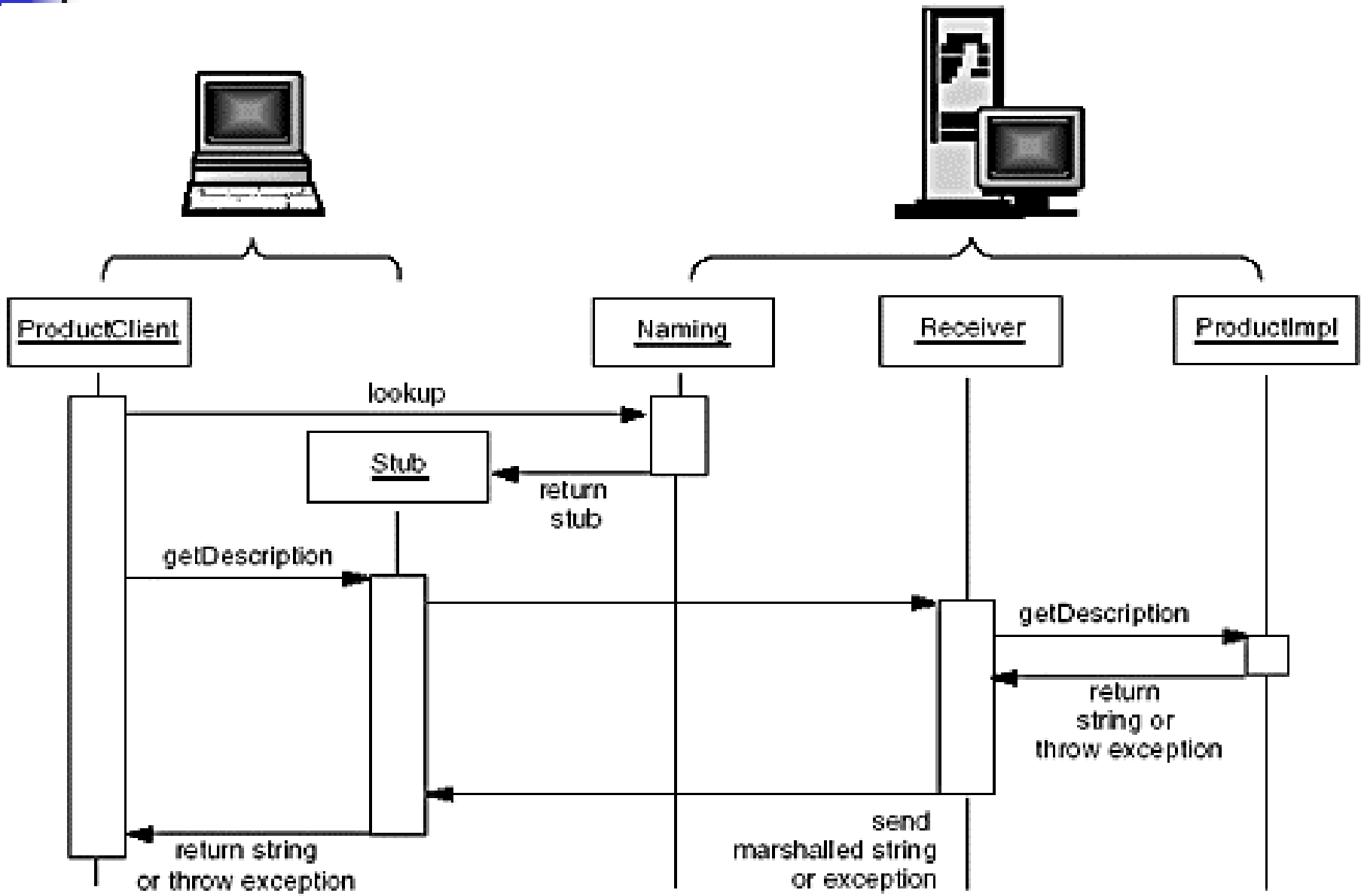
- Lookups the service in the registry, and obtain a remote service

```
Remote remoteService1 =
                    RMIReg.lookup("EOS350D");
```

- Cast to a Remote (Product) interface

```
IProduct c1 = (IProduct) remoteService1;
```

Calling the remote getDescription method



Remote Interfaces vs. Remote Classes

- Remember that the reference is to an interface
- You must make references, arrays, etc. out of the interface type, not the implementation type
- You can't cast the remote reference to a normal reference
- So name your Remote Objects with "Impl" (so you don't get confused)

Parameter Passing

- **Primitive types**
  - passed by value
- **Remote objects**
  - passed by reference
- **Non-remote objects**
  - passed by value
  - uses **Java Object Serialization**

Java Serialization

- Writes object as a sequence of bytes
- Writes it to a Stream
- Recreates it on the other end
- Creates a brand new object with the old data

java.io.Serializable

- Objects that implement the java.io.Serializable interface are marked as serializable
- Also subclasses
- Magically, all non-static and non-transient data members will be serialized
- Actually, it's not magic, it's *Reflection* (it's done with mirrors)
- empty interface - just a marker
- It's a promise

Not All Objects Are Serializable

- Any object that doesn't implement Serializable
- Any object that would pose a security risk
  - e.g. FileInputStream
- Any object whose value depends on VM-specific information
  - e.g. Thread
- Any object that contains a (non-static, non-transient) unserializable object (recursively)

NotSerializableException

- thrown if you try to serialize or unserialize an unserializable object
- maybe you subclassed a Serializable object and added some unserializable members

Serial Version UID

- The serialVersionUID is a universal version identifier for a Serializable class. Deserialization uses this number to ensure that a loaded class corresponds exactly to a serialized object. If no match is found, then an InvalidClassException is thrown

Guidelines for serialVersionUID :

- always include it as a field, for example: "private static final long serialVersionUID = 7526472295622776147L; " include this field even in the first version of the class, as a reminder of its importance
- do not change the value of this field in future versions, unless you are knowingly making changes to the class which will render it incompatible with old serialized objects
- new versions of Serializable classes may or may not be able to read old serialized objects; it depends upon the nature of the change;

Incompatible Changes

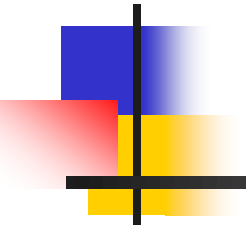
- Adding or removing members
- Changing a nonstatic field to static or a nontransient field to transient
- Changing the declared type of a primitive field
- . . . . . .
- java.io.InvalidClassException thrown if you try to deserialize an incompatible object stream

Serial Version

- If the changes were actually compatible
- find out the Serial Version UID of the original class
  - use the **`serialver`** utility
- add a member variable to the changed class

```
protected static final long serialVersionUID =
  -2215190743590612933L;
```

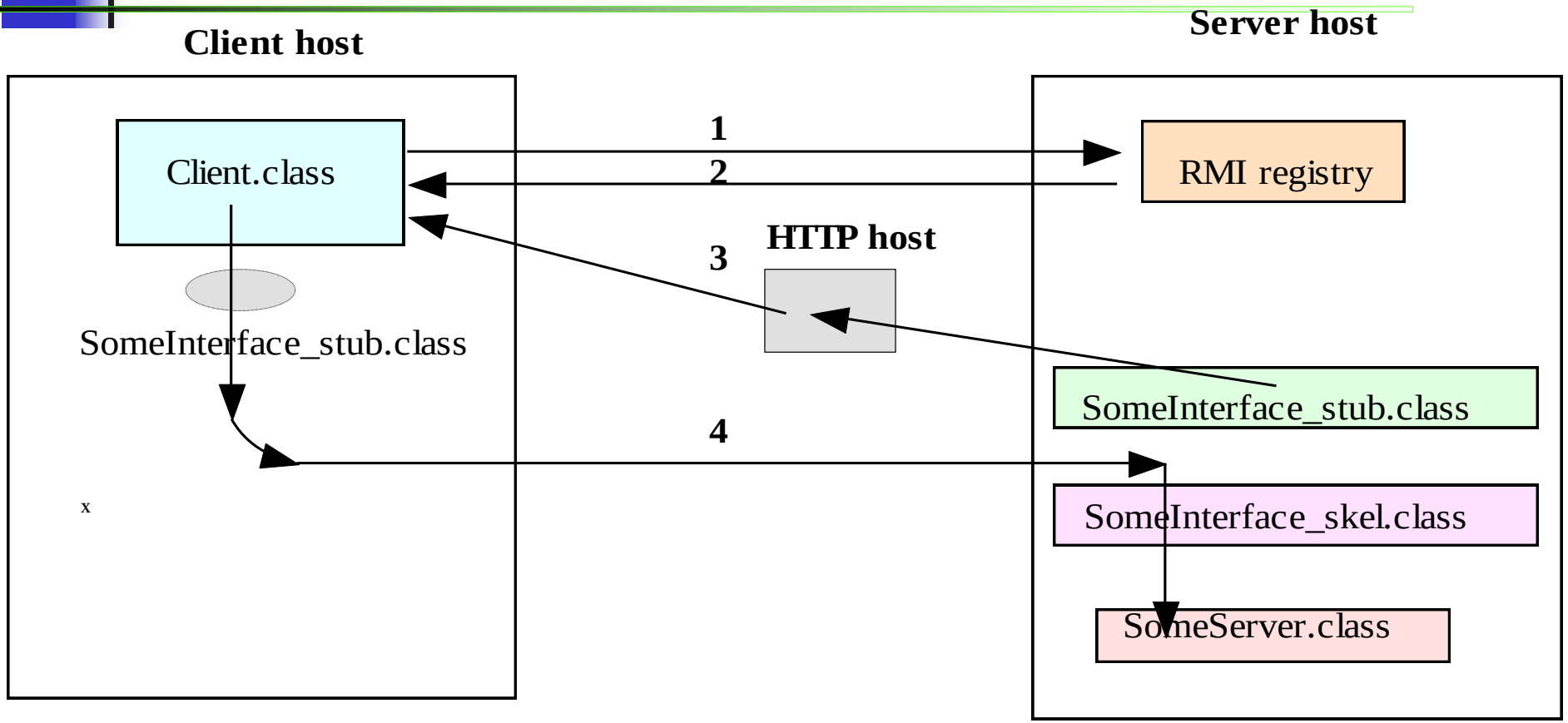
- now it's marked as compatible with the old class



Using Remote Method Invocation to Implement Callbacks

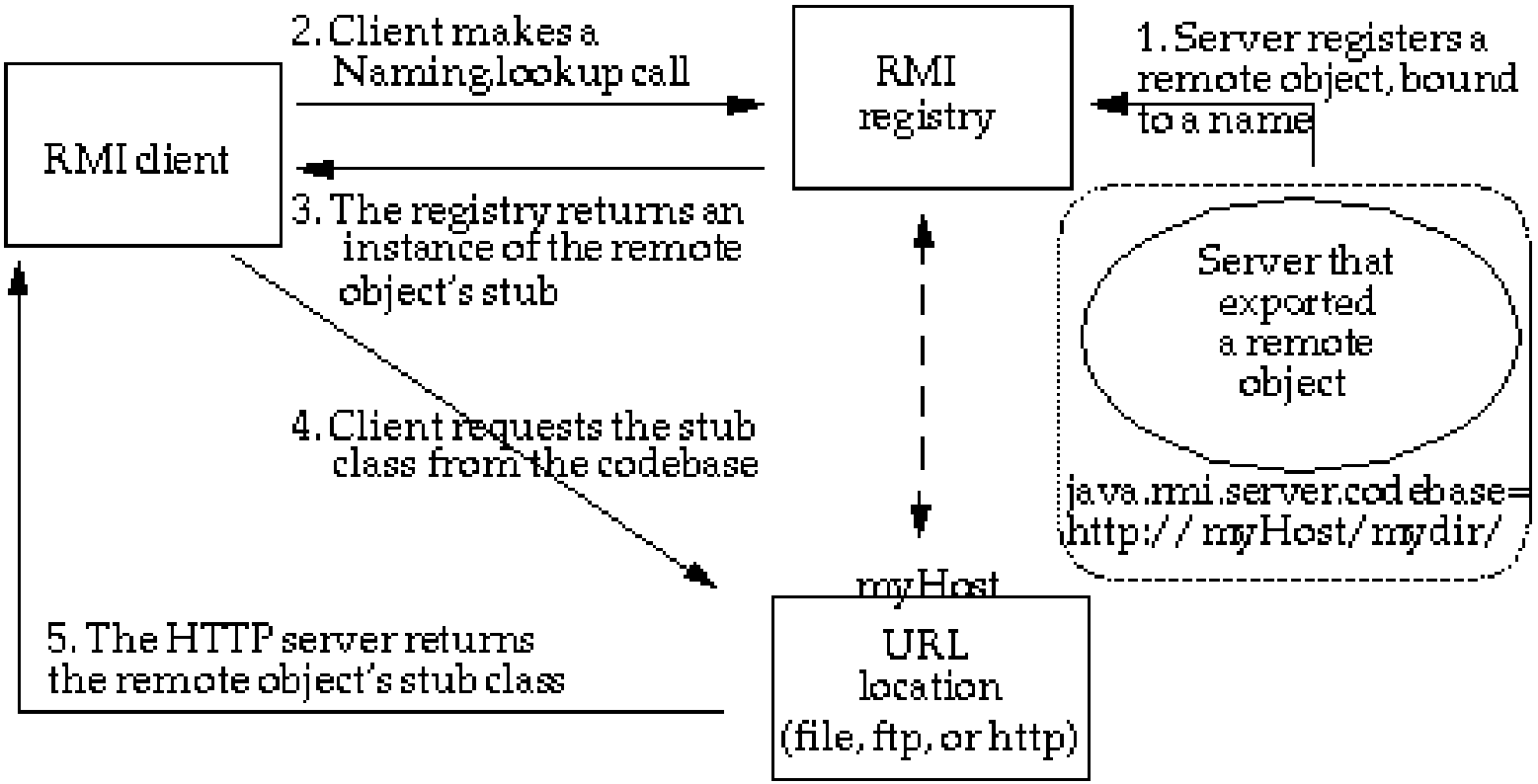
Self study

Java RMI Client Server Interaction



1. Client looks up the interface object in the RMIregistry on the server host.
2. The RMIRegistry returns a remote reference to the interface object.
3. If the interface object's stub is not on the client host and if it is so arranged by the server, the stub is downloaded from an HTTP server.
4. Via the server stub, the client process interacts with the skeleton of the interface object to access the methods in the server object.

CodeBase



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

CodeBase

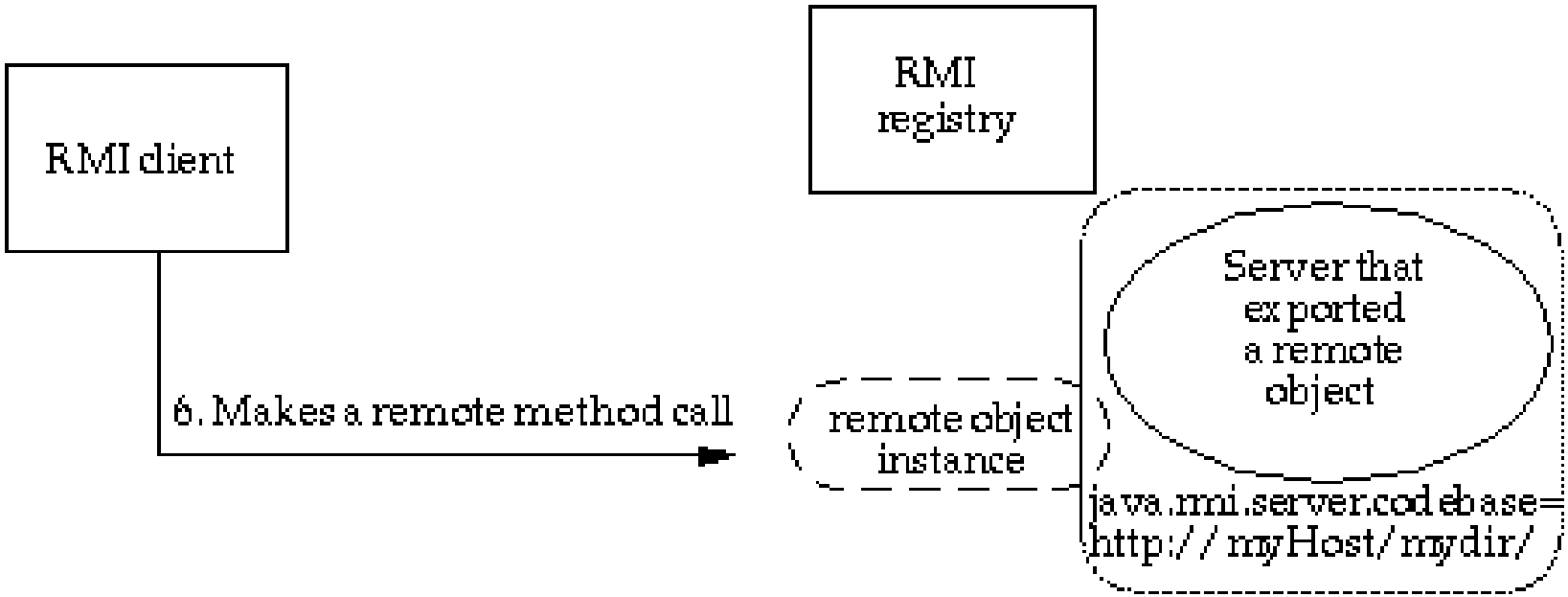
1. The remote object's codebase is specified by the remote object's server by setting the java.rmi.server.codebase property.The codebase set on the server VM is annotated to the remote object reference in the Java RMI registry.
2. The Java RMI client requests a reference to a named remote object. The reference (the remote object's stub instance) is what the client will use to make remote method calls to the remote object.
3. The Java RMI registry returns a reference (the stub instance) to the requested class. If the class definition for the stub instance can be found locally in the client's CLASSPATH , which is always searched before the codebase, the client will load the class locally. However, if the definition for the stub is not found in the client's CLASSPATH, the client will attempt to retrieve the class definition from the remote object's codebase.

CodeBase

4. The client requests the class definition from the codebase. The codebase the client uses is the URL that was annotated to the stub instance when the stub class was loaded by the registry.
5. The class definition for the stub (and any other class(es) that it needs) is downloaded to the client.
6. Now the client has all the information that it needs to invoke remote methods on the remote object. The stub instance acts as a proxy to the remote object that exists on the server;

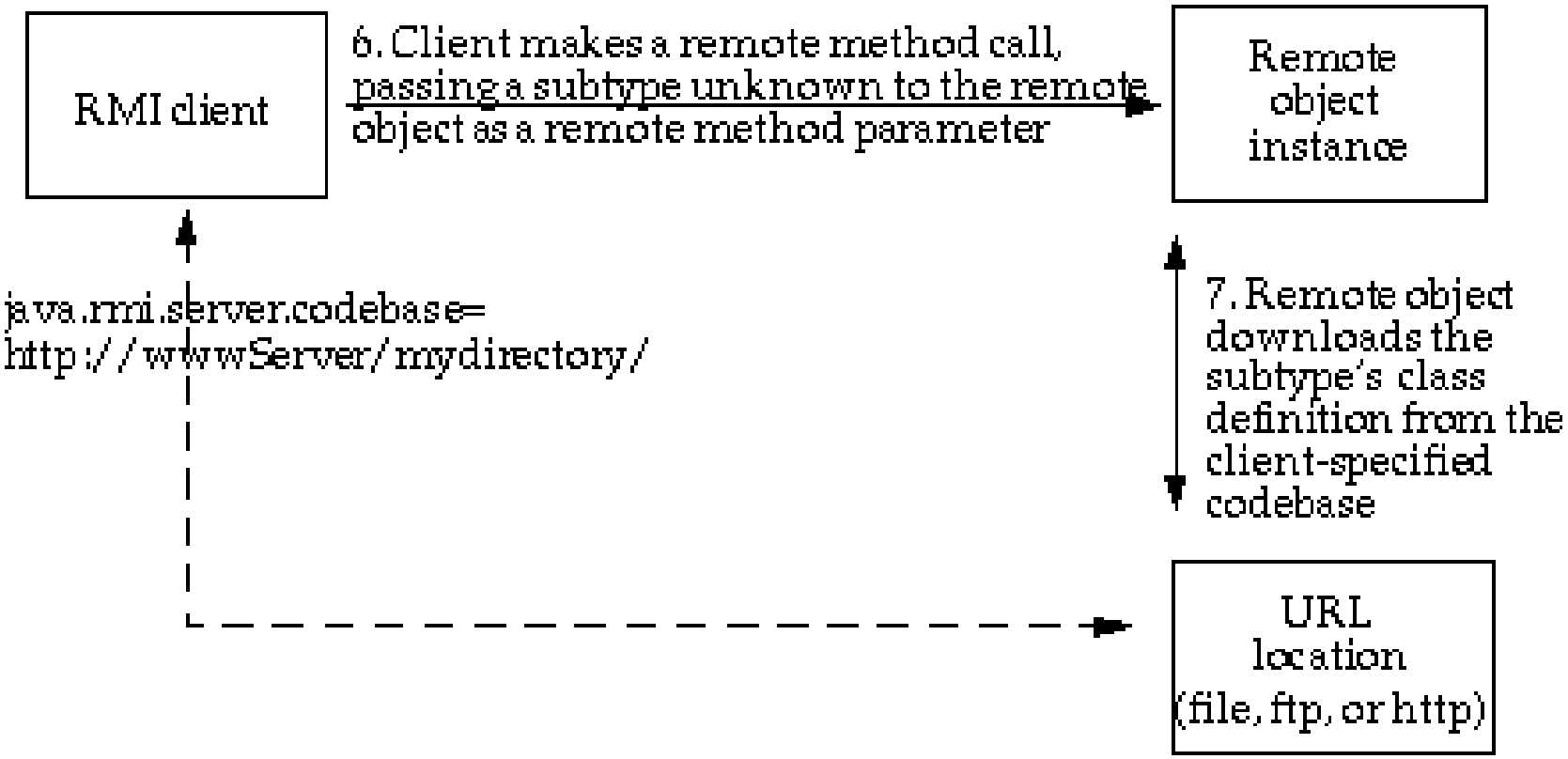
- In addition to downloading stubs and their associated classes to clients, the java.rmi.server.codebase property can be used to specify a location from which any class, not only stubs, can be downloaded.

CodeBase



Java RMI client making a remote method call

CodeBase



Java RMI client making a remote method call, passing an unknown subtype as a method parameter

Preparing for Deployment

- Deploying an application that uses RMI can be tricky because so many things can go wrong and the error messages that you get when something does go wrong are so poor. Separate the class files into three subdirectories:
- **`server`**
- **`download`**
- **`client`**
- The server directory contains all files that are needed to run the server. You will later move these files to the machine running the server process. In our example, the server directory contains the following files:
- **`server/`**

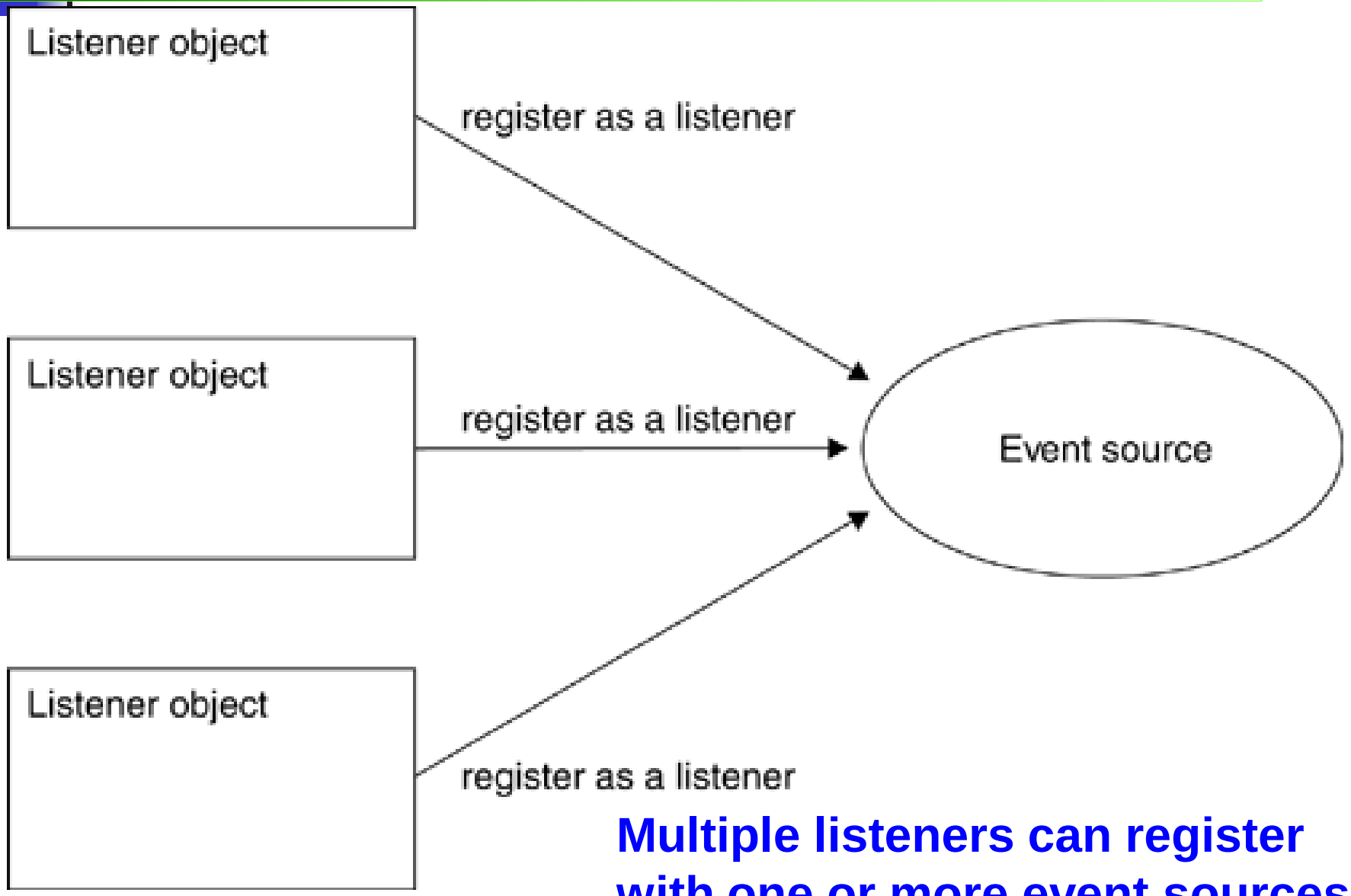
```
ProductServer.class
ProductImpl.class
Product.class
```

Preparing for Deployment

- The client directory contains the files that are needed to start the client
- **client/**
  ProductClient.class
  Product.class
  client.policy
- You will deploy these files on the client computer. Finally, the download directory contains those class files needed by the RMI registry, the client, and the server, as well as the classes they depend on. In our example, the download directory looks like this:
- **download/**
  ProductImpl_Stub.class

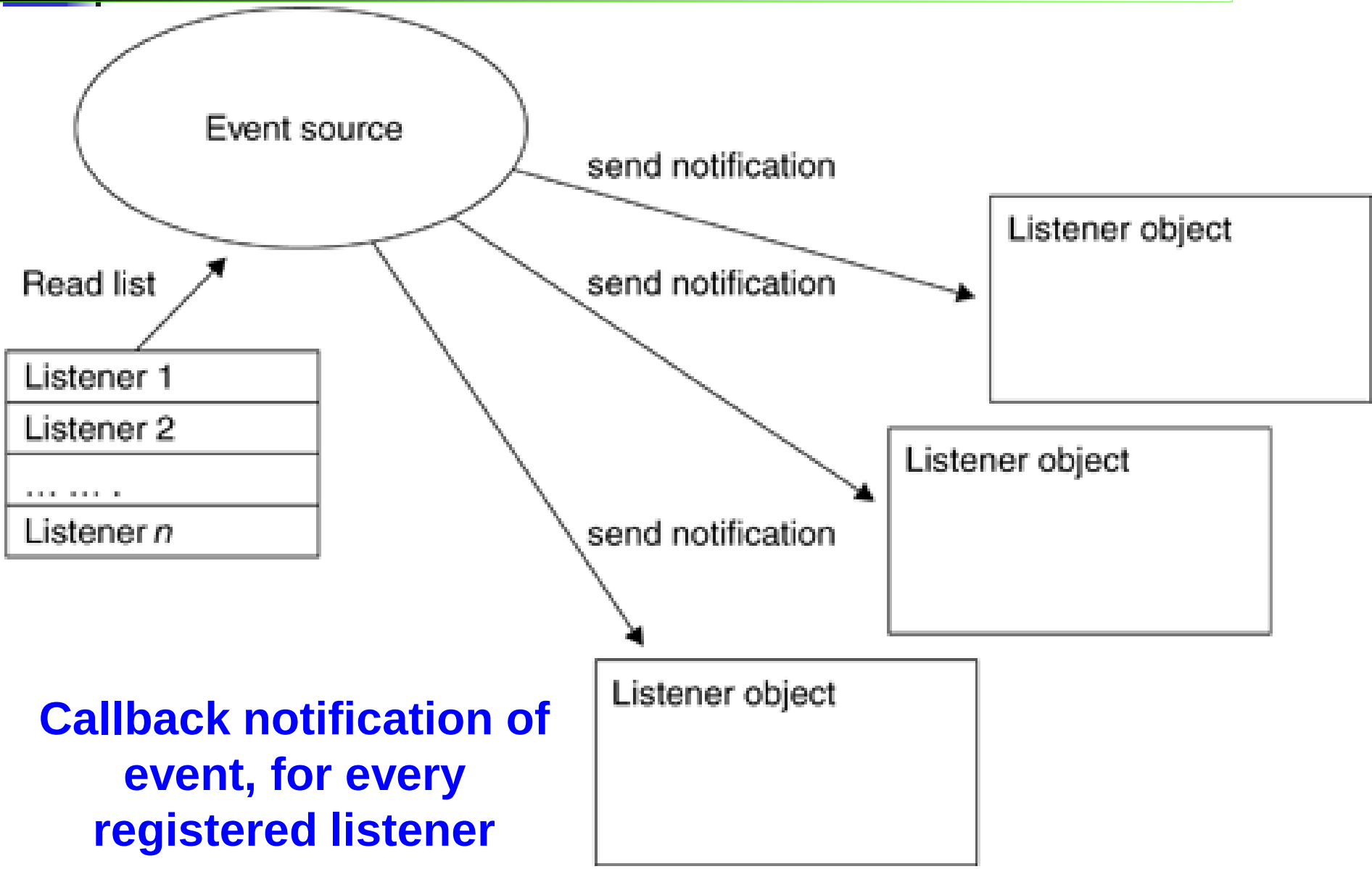
java - Djava.rmi.server.codebase = http://localhost:8080/ download/ ProductServer &

Using RMI to Implement Callbacks



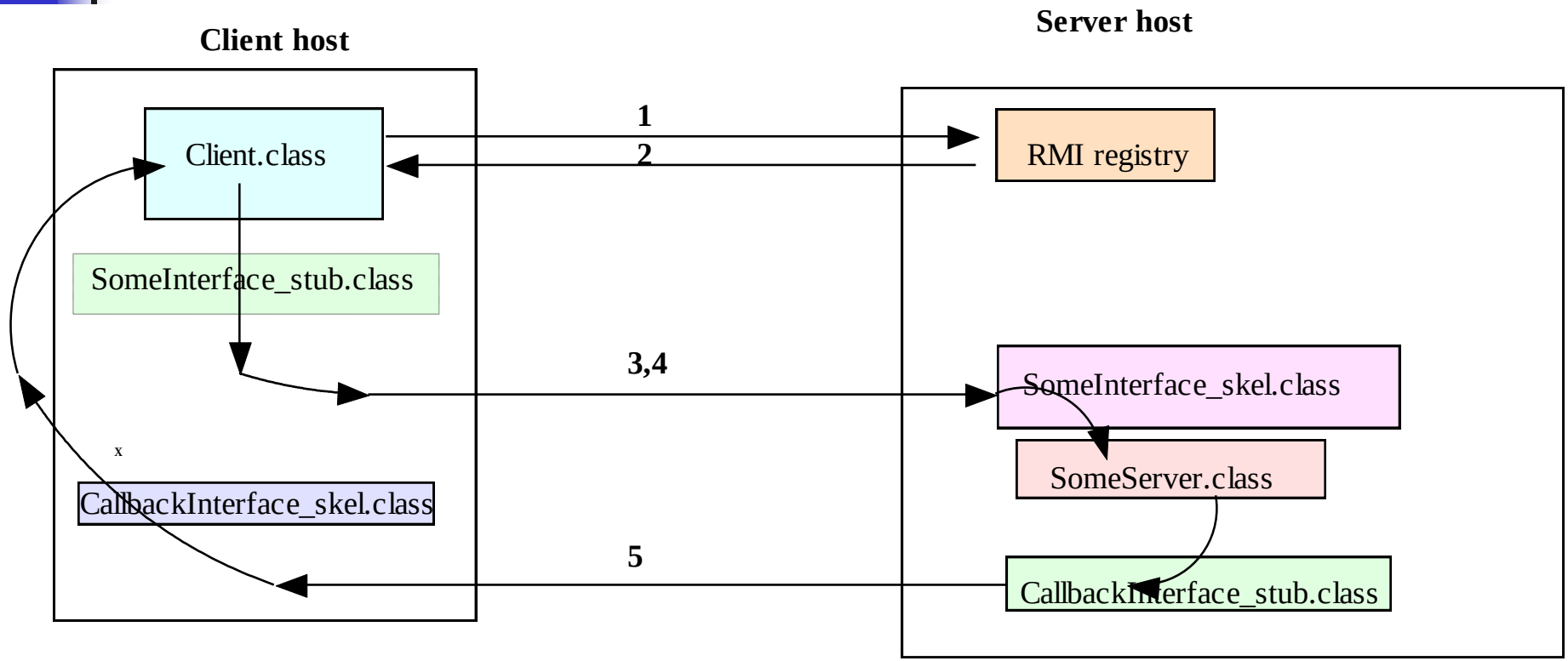
Multiple listeners can register with one or more event sources.

Using RMI to Implement Callbacks



Callback notification of event, for every registered listener

Callback Client-Server Interactions



1. Client looks up the interface object in the RMIregistry on the server host.
2. The RMIRegistry returns a remote reference to the interface object.
3. Via the server stub, the client process invokes a remote method to register itself for callback, passing a remote reference to itself to the server. The server saves the reference in its callback list.
4. Via the server stub, the client process interacts with the skeleton of the interface object to access the methods in the interface object.
5. When the anticipated event takes place, the server makes a callback to each registered client via the callback interface stub on the server side and the callback interface skeleton on the client side.

Defining the Listener Interface

- The listener interface defines a remote object with a single method. This method should be invoked by an event source whenever an event occurs, so as to act as notification that the event occurred. The method signifies a change in temperature, and allows the new temperature to be passed as a parameter.

```
interface TemperatureListener extends
Remote {
     public void temperatureChanged(double
temperature)
          throws java.rmi.RemoteException;
}
```

Defining the Event Source Interface

- The event source must allow a listener to be registered and unregistered, and may optionally provide additional methods. In this case, a method to request the temperature on demand is offered.

```
interface TemperatureSensor extends
java.rmi.Remote{
        public double getTemperature()
                        throws  java.rmi.RemoteException;
        public void addTemperatureListener
                        (TemperatureListener listener )
                        throws java.rmi.RemoteException;
        public void removeTemperatureListener
                        (TemperatureListener listener )
                        throws java.rmi.RemoteException;
}
```

Implementing the Event Source Interface

- A **`TemperatureSensorServerImpl`** class is defined, which acts as an RMI server.
- To notify registered listeners as a client (The server must extend **`UnicastRemoteObject`**, to offer a service, and implement the **`Temperature Sensor`** interface.
- To create an instance of the service and registering it with the **`rmiregistry`**
- **`To launch a new thread, responsible for updating the value of the temperature`**, based on randomly generated numbers.
- As each change occurs, registered listeners are notified, by reading from a list of listeners stored in a **`java.util.Vector`** object. This list is modified by the remote **`addTemperatureListener(TemperatureListener)`** and **`removeTemperatureListener(TemperatureListener)`** methods.

Implementing the Event Source Interface

```
public class TemperatureSensorImpl extends UnicastRemoteObject
                                implements  TemperatureSensor, Runnable {
    private volatile double temp; private Vector list = new Vector();
    public TemperatureSensorImpl() throws java.rmi.RemoteException {
       super();
       temp = 98.0; // Assign a default setting for the temperature
    }
    public double getTemperature() throws java.rmi.RemoteException {
        return temp;
    }
    public void addTemperatureListener(TemperatureListener listener)
                                                  throws java.rmi.RemoteException {
        System.out.println("adding listener -" + listener);
        list.add(listener);
    }
    public void removeTemperatureListener(TemperatureListener listener)
                                                  throws java.rmi.RemoteException {
        System.out.println("removing listener -" + listener);
        list.remove(listener);
    }
```

Implementing the Event Source Interface

```
public void run() {
        Random r = new Random();
        for (; ; ) {
            try {
                // Sleep for a random amount of time
                int duration = r.nextInt() % 10000 + 2000;
                // Check to see if negative, if so, reverse
                if (duration < 0) duration = duration * -1;
                Thread.sleep(duration);
            } catch (InterruptedException ie) {}
            // Get a number, to see if temp goes up or down
            int num = r.nextInt();
            if (num < 0)  temp += 0.5; else  temp -= 0.5;
           notifyListeners(); // Notify registered listeners
}}
```

Implementing the Event Source Interface

```
private void notifyListeners() {
    // Notify every listener in the registered list
    for (Enumeration e = list.elements();
                                        e.hasMoreElements();) {
        TemperatureListener listener = (TemperatureListener)
                                        e.nextElement();
        // Notify, if possible a listener
        try {
            listener.temperatureChanged(temp);
        } catch (RemoteException re) {
            System.out.println("removing listener -" + listener);
            // Remove the listener
            list.remove(listener);
    } }
} }
```

Implementing the Listener Interface

- The temperature monitor client must implement the **`TemperatureListener`** interface, and register itself with the remote temperature sensor service, by invoking the **`TemperatureSensor.addTemperatureListener (Temperature Listener)`** method.
- By registering as a listener, the monitor client will be notified of changes as they occur, using a remote callback. The client waits patiently for any changes, and though it does not ever remove itself as a listener, functionality to achieve this is supplied by the

```
TemperatureSensor.removeTemperatureListener(
    TemperatureListener)
```

method.

Implementing the Listener Interface

```
import java.rmi.*;
import java.rmi.server.*;
public class TemperatureListenerImpl extends
UnicastRemoteObject implements  TemperatureListener {
    // Default constructor throws a RemoteException
    public TemperatureListenerImpl() throws RemoteException {
        super();
  }

    public void temperatureChanged(double temperature)
                throws java.rmi.RemoteException {
        System.out.println("Temperature change event : " +
                                                temperature);
    }
}
```

TemperatureSensorServer

```
public class TemperatureSensorServer{
    public static void main(String args[]) {
        System.out.println("Loading temperature service");
        try {
            // Load the service
            TemperatureSensorImpl sensor = new  TemperatureSensorImpl();
            // Register with service so that clients can find us
            String registry = "localhost";
            String registration = "rmi://" + registry + "/TemperatureSensor";
            Naming.rebind(registration, sensor);
            // Create a thread, and pass the sensor server.  This will activate the
           //run()  method, and  trigger regular temperature changes.
            Thread thread = new Thread(sensor);
            thread.start();
        } catch (RemoteException re) {
            System.err.println("Remote Error - " + re);
        } catch (Exception e) {
            System.err.println("Error - " + e);
        }
}}
```

TemperatureMonitor

```
public static void main(String args[]) {
   System.out.println("Looking for temperature sensor");
    try {
        // Lookup the service in the registry, and obtain a remote service
        String registry = "localhost";
        String registration = "rmi://" + registry +  "/TemperatureSensor";
        Remote remoteService = Naming.lookup(registration);
        // Cast to a TemperatureSensor interface
        TemperatureSensor sensor = (TemperatureSensor) remoteService;
        // Get and display current temperature
        double reading = sensor.getTemperature();
        System.out.println("Original temp : " + reading);
        // Create a new monitor and register it as a listener with remote sensor
        TemperatureListenerImpl monitor = new TemperatureListenerImpl();
        sensor.addTemperatureListener(monitor);
    } catch (RemoteException re) {
          System.out.println("RMI Error - " + re);
    } catch (Exception e) {
          System.out.println("Error - " + e);
} }
```

Configuring a RMID