Remote Method Invocation

Middleware Programming Models

Commonly used models:

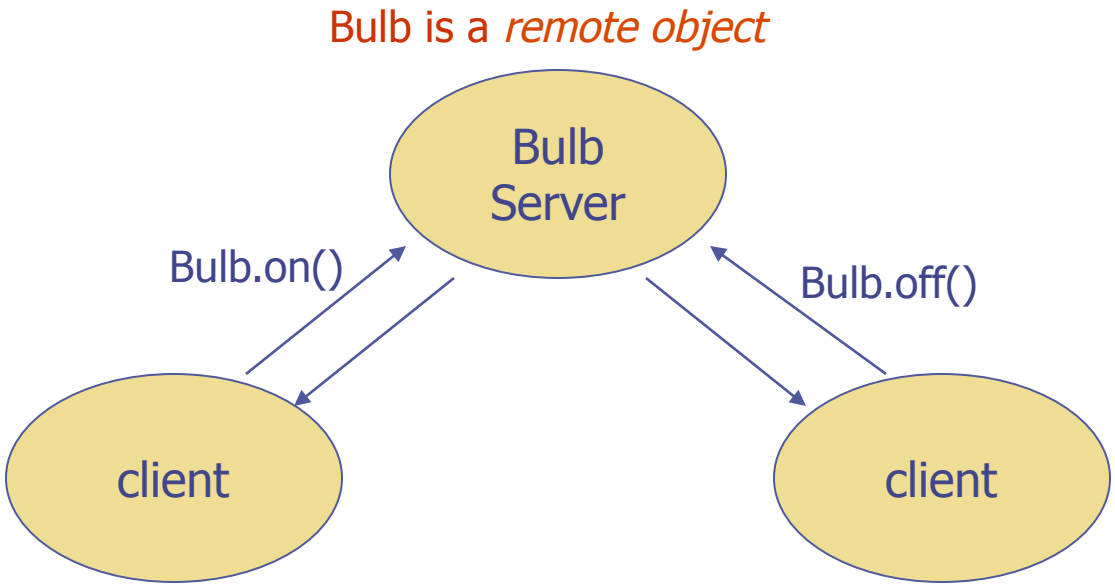
- Distributed objects and remote method invocation *(Java RMI, Corba)*
- Remote Procedure Call *(Web services)*
- Remote SQL access *(JDBC, ODBC)*
- Distributed transaction processing

CORBA:

- provides remote object invocation between
  - a client program written in one language and
  - a server program written in another language
- commonly used with C++

Introduction of RMI

- What is Remote Method Invocation?



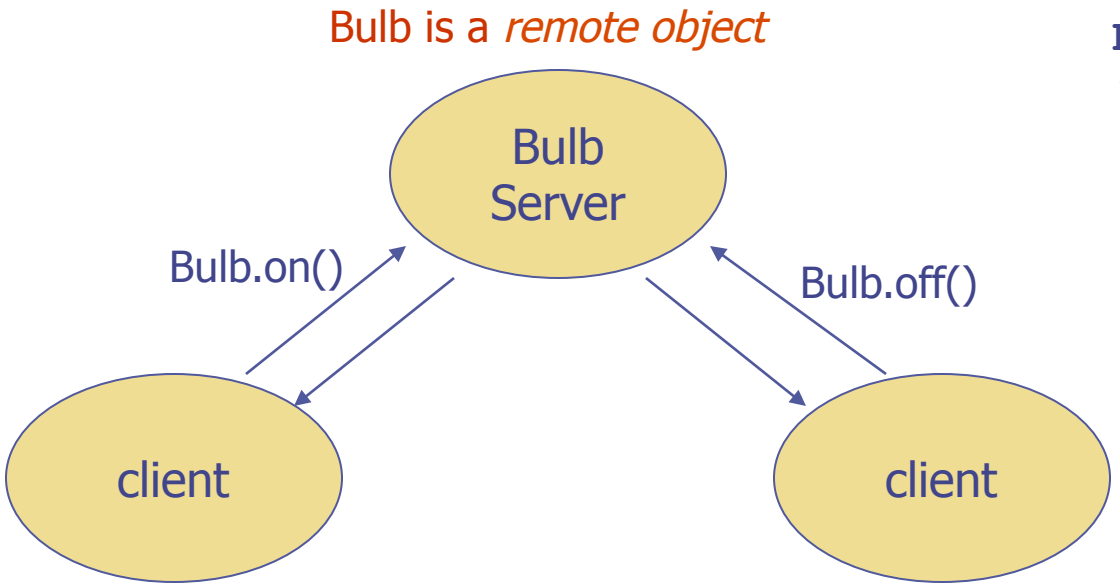

```
public class LightBulb
{
    private boolean lightOn;
    public void on()
    {
        lightOn = true;
    }

    public void off()
    {
        lightOn = false;
    }

    public boolean isOn()
    {
        return lightOn;
    }
}
```

Introduction of RMI

- What is Remote Method Invocation?



1. How to identify a remote object and its methods?
2. How to invoke a method of a remote object (e.g., parameters passing, result returning)?
3. How to locate a remote object?

```
public class LightBulb
{
    private boolean lightOn;
    public void on()
    {
        lightOn = true;
    }

    public void off()
    {
        lightOn = false;
    }

    public boolean isOn()
    {
        return lightOn;
    }
}
```

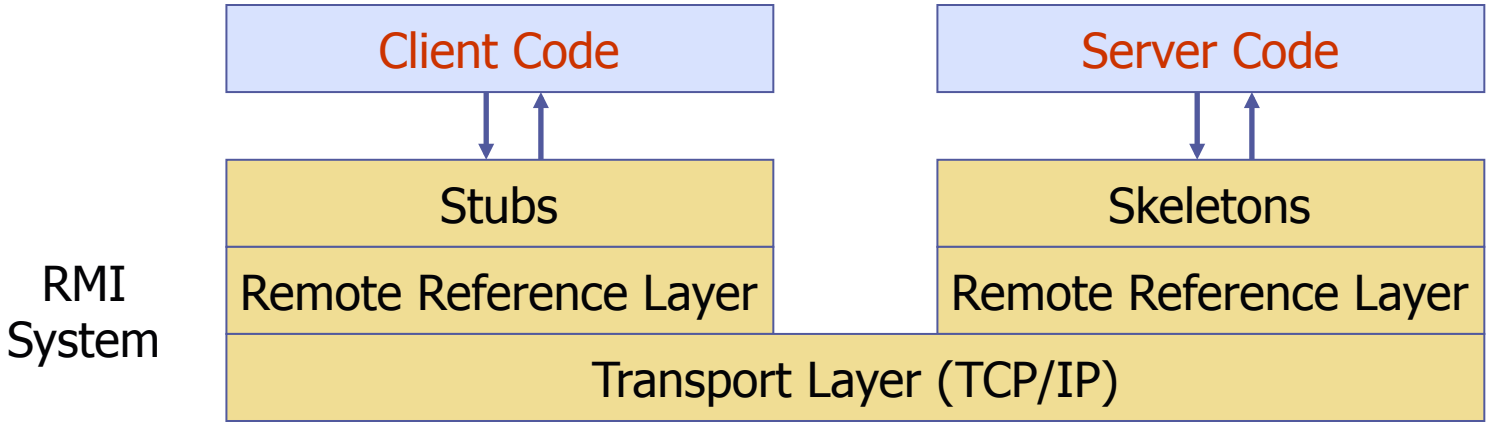
Introduction of RMI

- Primary goal of RMI
  - Allow programmers to develop distributed Java programs with the same syntax and semantic used for non-distributed programs
- RMI vs. RPC
  - RMI is for Java only, allowing Java objects on different JVM to communicate each other
  - RMI is object-oriented
    - Input parameters could be objects
      - These objects could be executed in a remote host
    - Return value could be an object as well

RMI Architecture

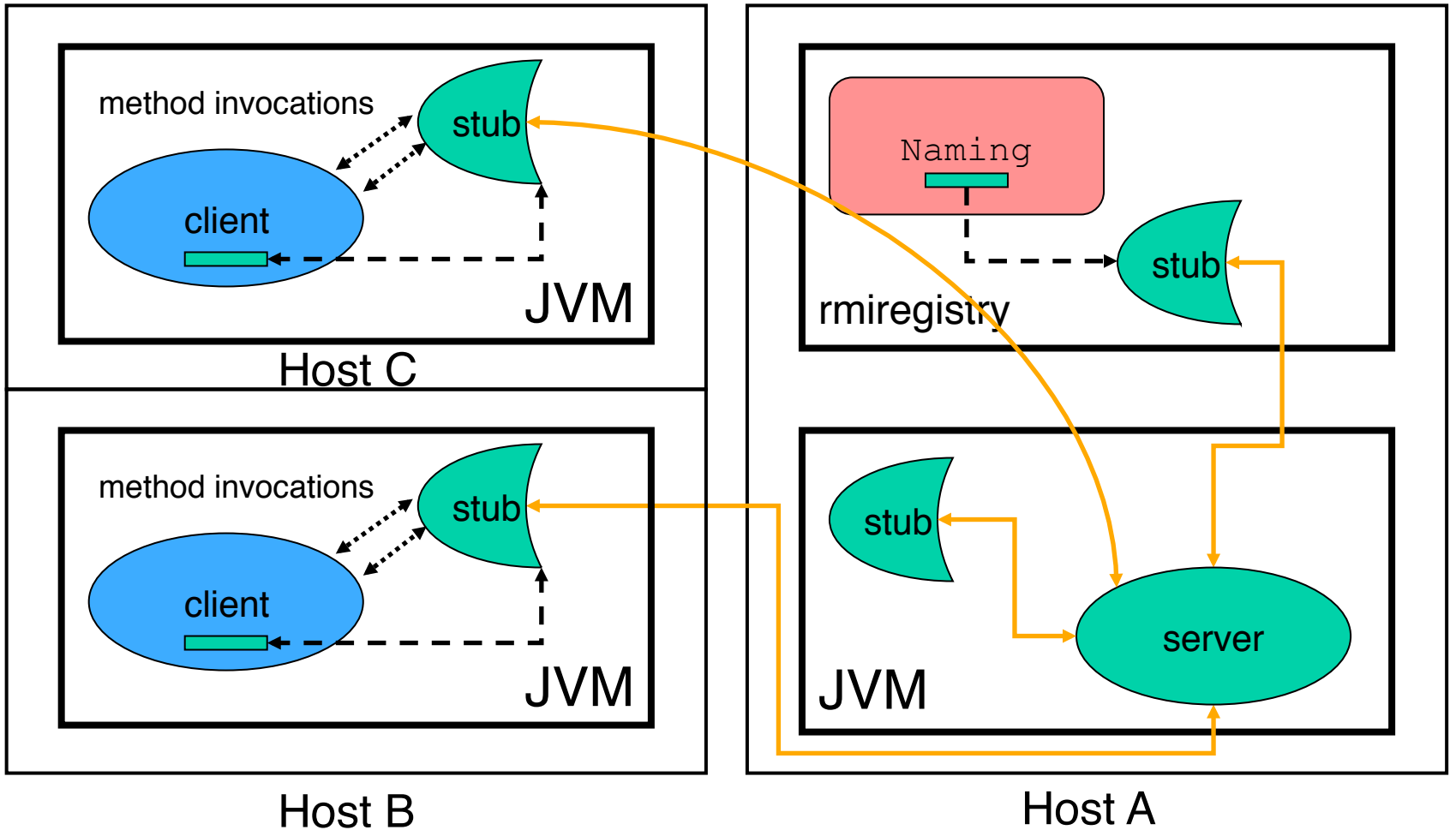
- Each remote object has two separate parts
  - Definition of its behavior
    - Clients are concerned about the definition of a service
    - Coded using a Java interface, which defines the behavior
  - Implementation of its behavior
    - Servers are focused on providing the service
    - Coded using a Java class, which defines the implementation

RMI Layers



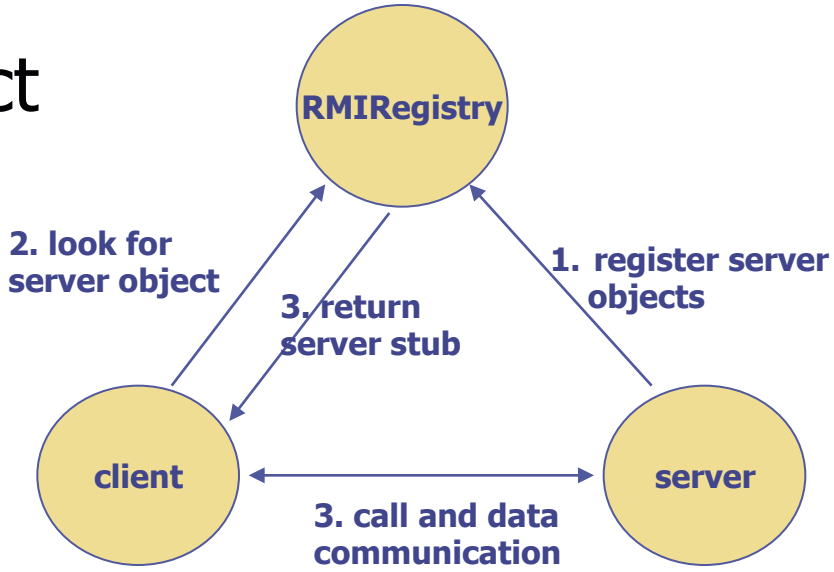
- Each remote object has two interfaces
  - Client interface – a stub/proxy of the object
  - Server interface – a skeleton of the object
- The communication of stub and skeleton is done across the RMI link
  - Read parameters/make call/accept return/write return back to the stub
- Remote reference layer defines and supports the invocation semantics of the RMI connection

Structure of an RMI application



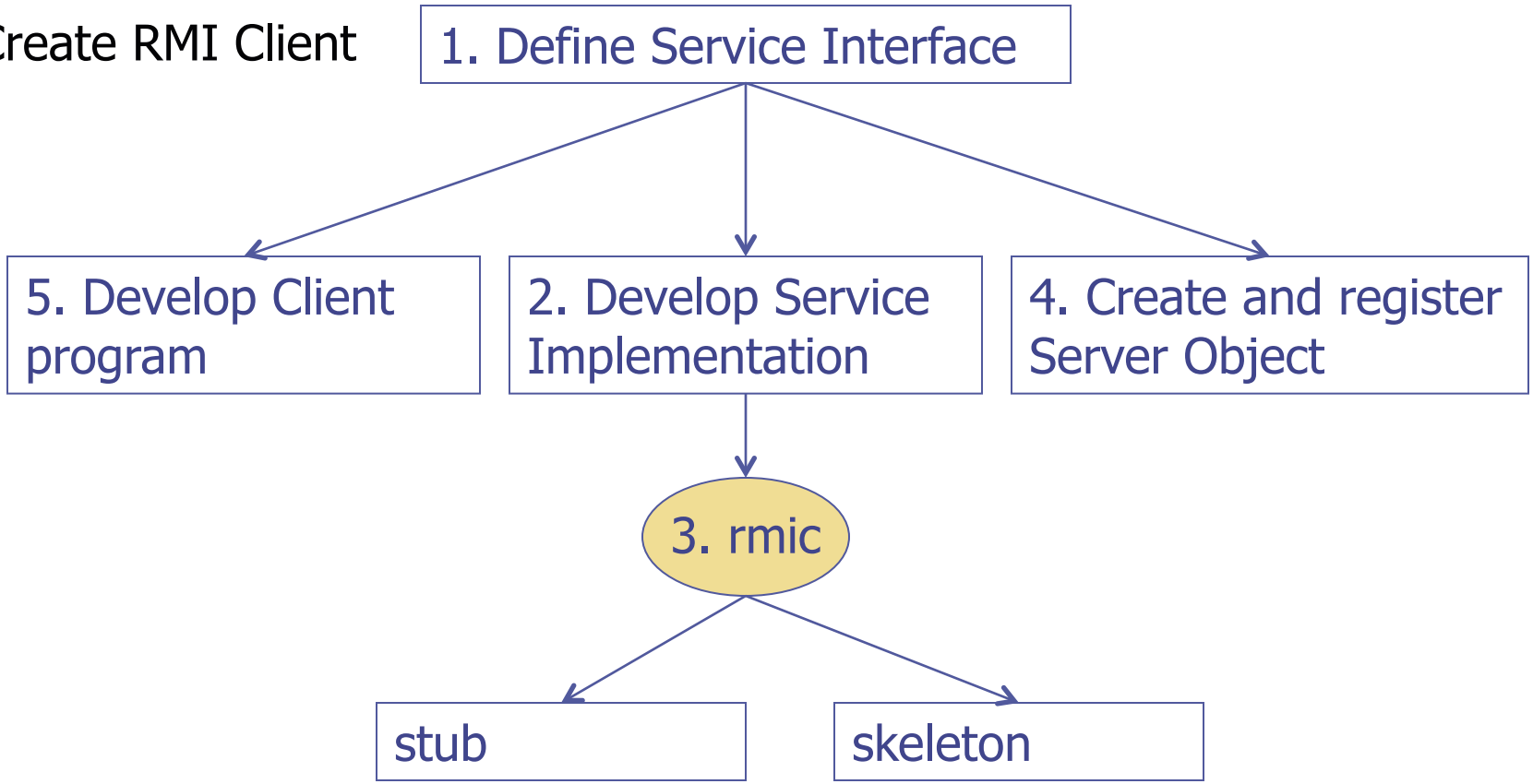
RMI Components

- RMI registry
  - Each remote object needs to register their location
  - RMI clients find remote objects via the lookup service
- Server hosting a remote object
  - Construct an implementation of the object
  - Provide access to methods via skeleton
  - Register the object to the RMI registry
- Client using a remote object
  - Ask registry for location of the object
  - Construct stub
  - Call methods via the object's stub



Steps of Using RMI

1. Create Service Interface
2. Implement Service Interface
3. Create Stub and Skeleton Classes
4. Create RMI Server
5. Create RMI Client



1. Defining RMI Service Interface

- Declare an Interface that extends java.rmi.Remote
  - Stub, skeleton, and implementation will implement this interface
  - Client will access methods declared in the interface
- Example

```
public interface RMILightBulb extends java.rmi.Remote {
  public void on ()      throws java.rmi.RemoteException;
  public void off()      throws java.rmi.RemoteException;
  public boolean isOn() throws java.rmi.RemoteException;
}
```

2. Implementing RMI Service Interface

- Provide concrete implementation for each methods defined in the interface

```
public class RMILightBulbImpl extends
    java.rmi.server.UnicastRemoteObject implements RMILightBulb
{
    public RMILightBulbImpl() throws java.rmi.RemoteException
    {setBulb(false);}
    private boolean lightOn;
    public void on() throws java.rmi.RemoteException { setBulb(true); }
    public void off() throws java.rmi.RemoteException {setBulb(false);}
    public boolean isOn() throws java.rmi.RemoteException
     { return getBulb(); }
    public void setBulb (boolean value) { lightOn = value; }
    public boolean getBulb () { return lightOn; }
}
```

3. Generating Stub & Skeleton Classes

- Simply run the `rmic` command on the implementation class
- Example:
  - `rmic RMILightBulbImpl`
  - creates the classes:
    - RMILightBulbImpl_Stub.class
      - Client stub
    - RMILightBulbImpl_Skeleton.class
      - Server skeleton

4. Creating RMI Server

- Create an instance of the service implementation
- Register with the RMI registry (binding)

```
import java.rmi.*;
import java.rmi.server.*;
public class LightBulbServer {
 public static void main(String args[]) {
  try {
   RMILightBulbImpl bulbService = new RMILightBulbImpl();
   RemoteRef location = bulbService.getRef();
   System.out.println (location.remoteToString());
   String registry = "localhost";
   if (args.length >=1) {
    registry = args[0];
   }
   String registration = "rmi://" + registry + "/RMILightBulb";
   Naming.rebind( registration, bulbService );
 } catch (Exception e) { System.err.println ("Error - " + e); } } }
```

5. Creating RMI Client

- Obtain a reference to the remote interface
- Invoke desired methods on the reference

```
import java.rmi.*;
public class LightBulbClient {
 public static void main(String args[]) {
  try { String registry = "localhost";
   if (args.length >=1) { registry = args[0]; }
   String registration = "rmi://" + registry + "/RMILightBulb";
   Remote remoteService = Naming.lookup ( registration );
   RMILightBulb bulbService = (RMILightBulb) remoteService;
   bulbService.on();
   System.out.println ("Bulb state : " + bulbService.isOn()  );
   System.out.println ("Invoking bulbservice.off()");
   bulbService.off();
   System.out.println ("Bulb state : " + bulbService.isOn() );
  } catch (NotBoundException nbe) {
   System.out.println ("No light bulb service available in registry!");
  } catch (RemoteException re) { System.out.println ("RMI - " + re);
  } catch (Exception e) { System.out.println ("Error - " + e); }
 }
}
```

Steps of Running RMI

- Make the classes available in the server host's, registry host's, and client host's classpath
  - Copy, if necessary
- Start the registry
  - **`rmiregistry`**
- Start the server
  - **`java LightBulbServer reg-hostname`**
- Start the client
  - **`java LightBulbClient reg-hostname`**

Summary

- RMI is a Java middleware to deal with remote objects based on RPC communication protocol
  - Interface defines behaviour and class defines implementation
  - Remote objects are pass across the network as stubs and nonremote objects are copies
- RMI will not replace CORBA since a JAVA client may require to interact with a C/C++ server
- RMI fits well in n-tier architectures since it can intermix easily with servlets