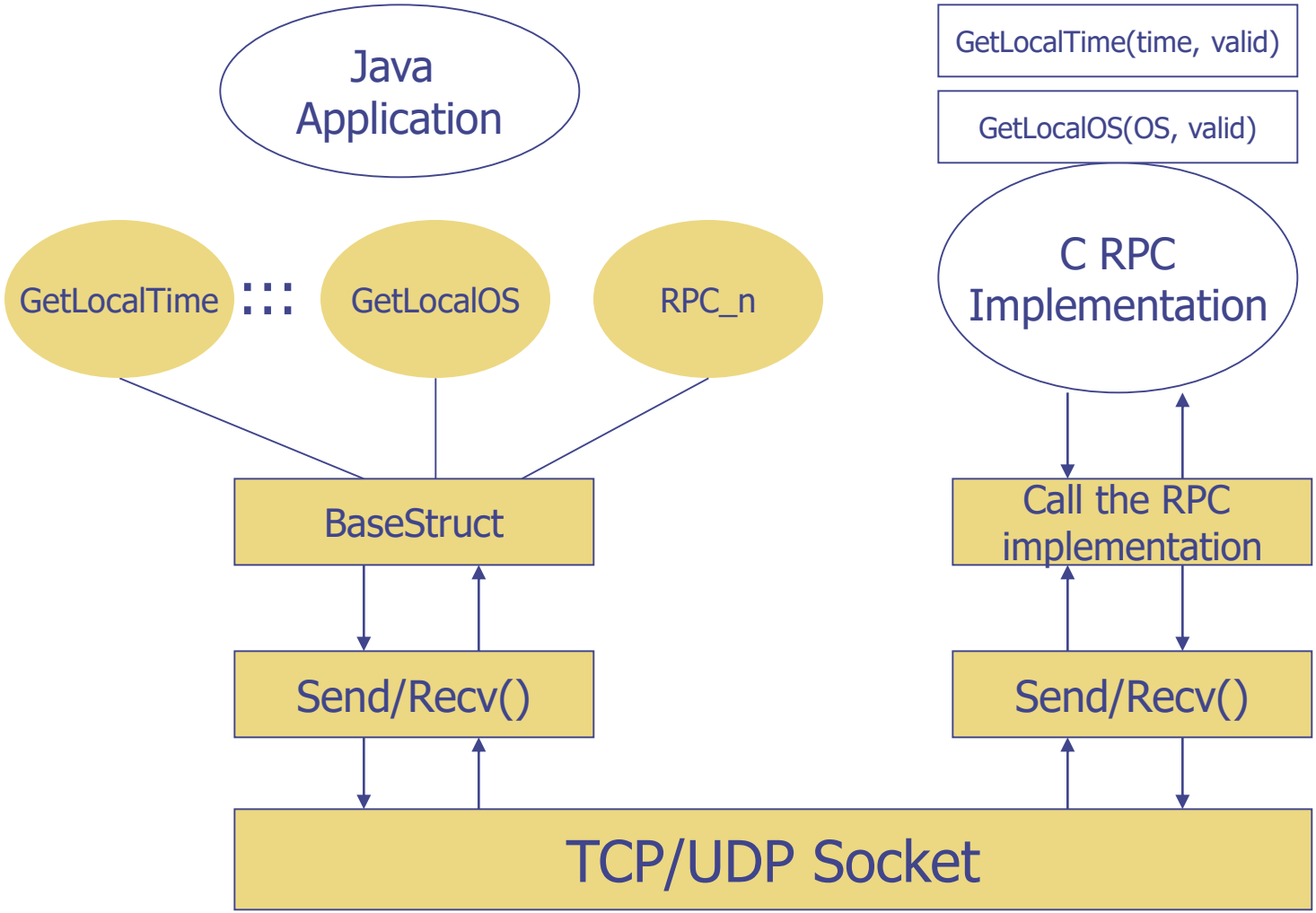
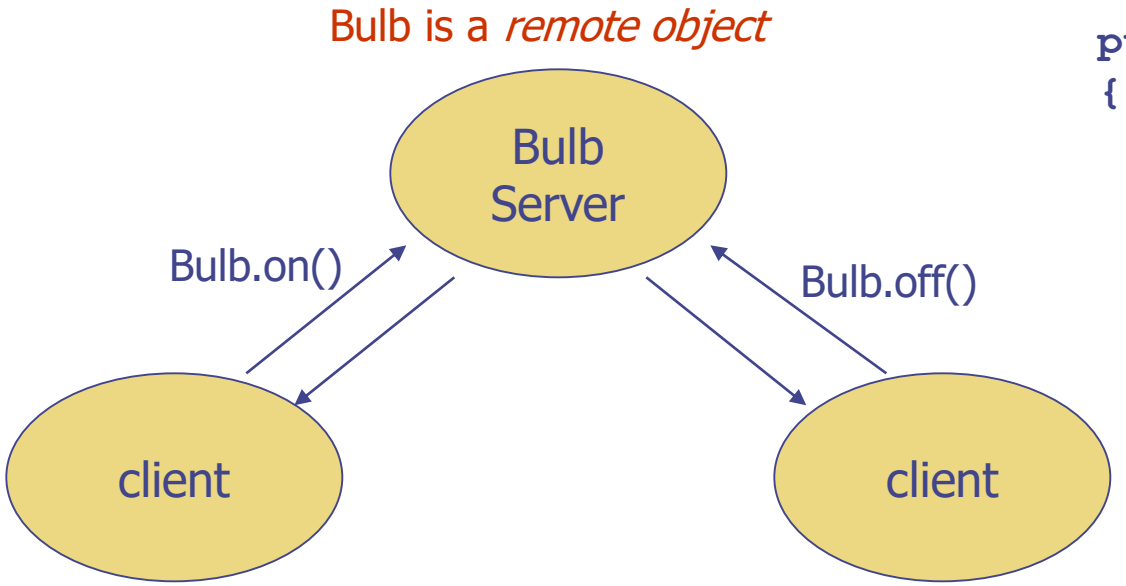


1. Define C interface procedure/data structure
2. Implement C interface
3. Run C2J on the interface
4. Use C-equivalent Java code

Remote Method Invocation

CS587x Lecture
Department of Computer Science
Iowa State University

Remote Method Invocation

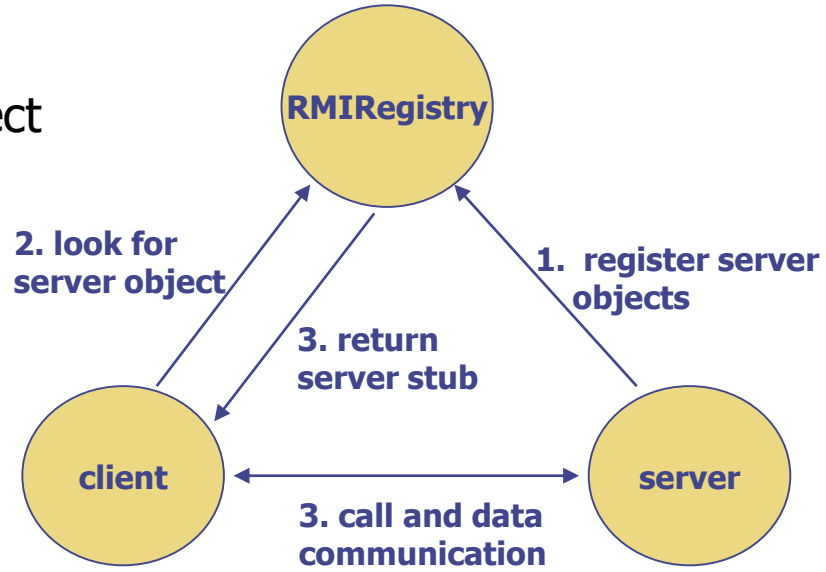



```
public class LightBulb
{
    private boolean lightOn;
    public void on()
    {
        lightOn = true;
    }

    public void off()
    {
        lightOn = false;
    }

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

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?

RMI Components

- RMI registry : recording all remote objects
  - 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



Remote Object ID	Hosting Server
RMILightBulb	12.34.56.78:4000
:::	:::

1. Defining a Remote 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 the Remote 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"; // where the registry server locates
   if (args.length >=1) {
    registry = args[0];
   }
   String registration = "rmi://" + registry + "/RMILightBulb";
   Naming.rebind( registration, bulbService );
 } catch (Exception e) { System.err.println ("Error - " + e); } } }
```

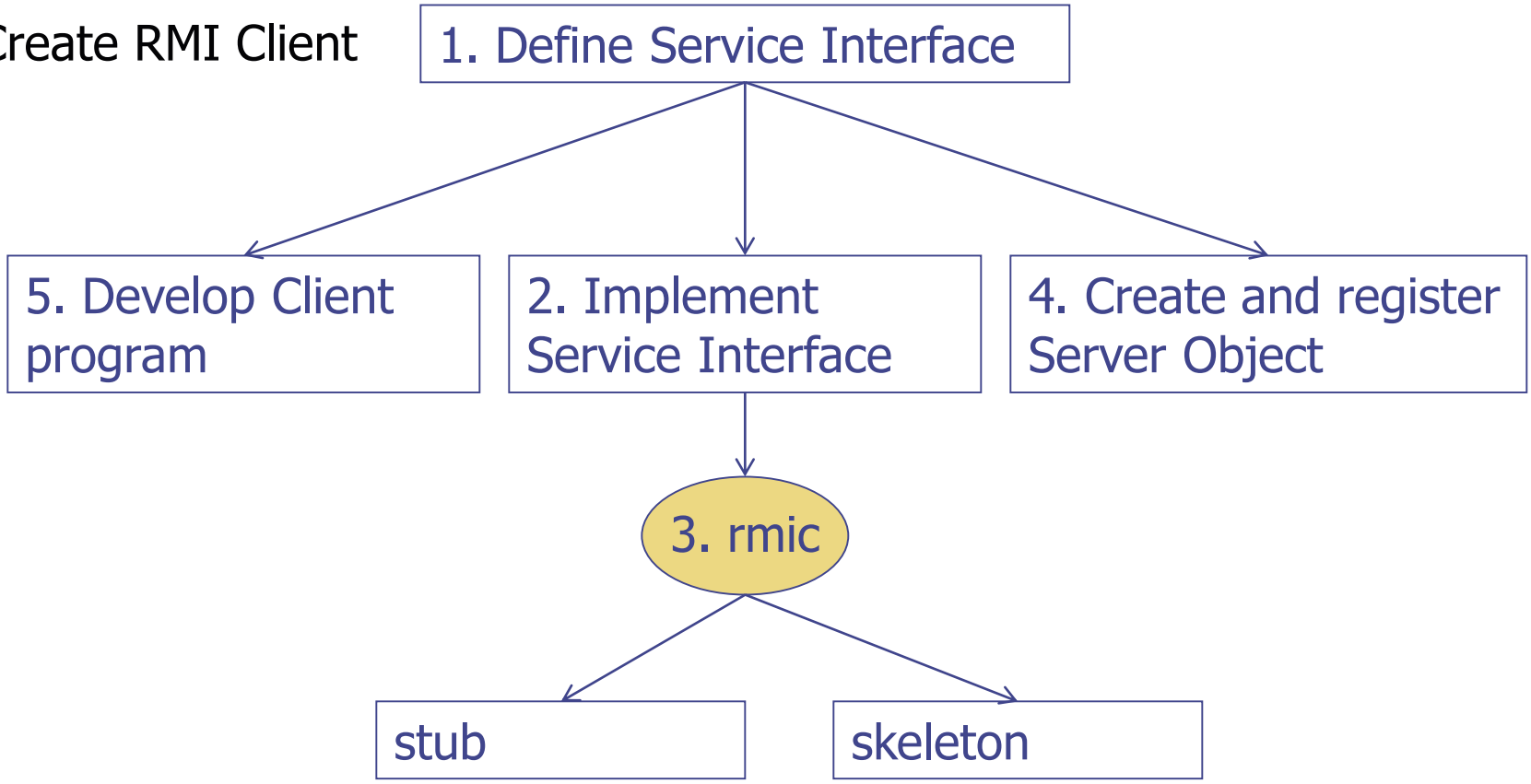
5. Creating RMI Client

- 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"; // the registry server's IP
   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); }
 }
}
```

Review: 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



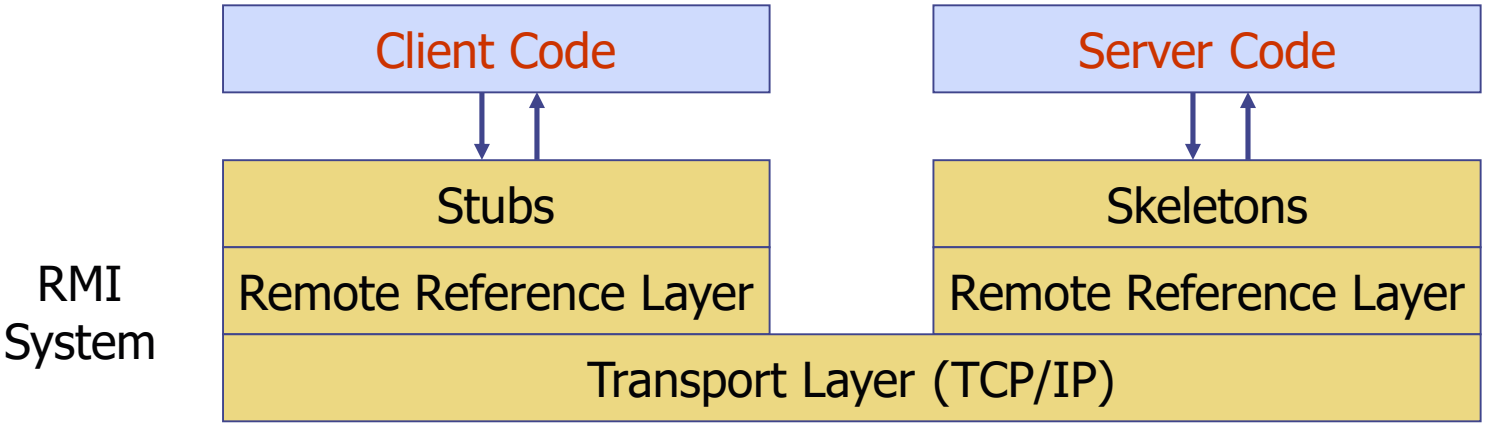
Steps of Running RMI

- 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`**

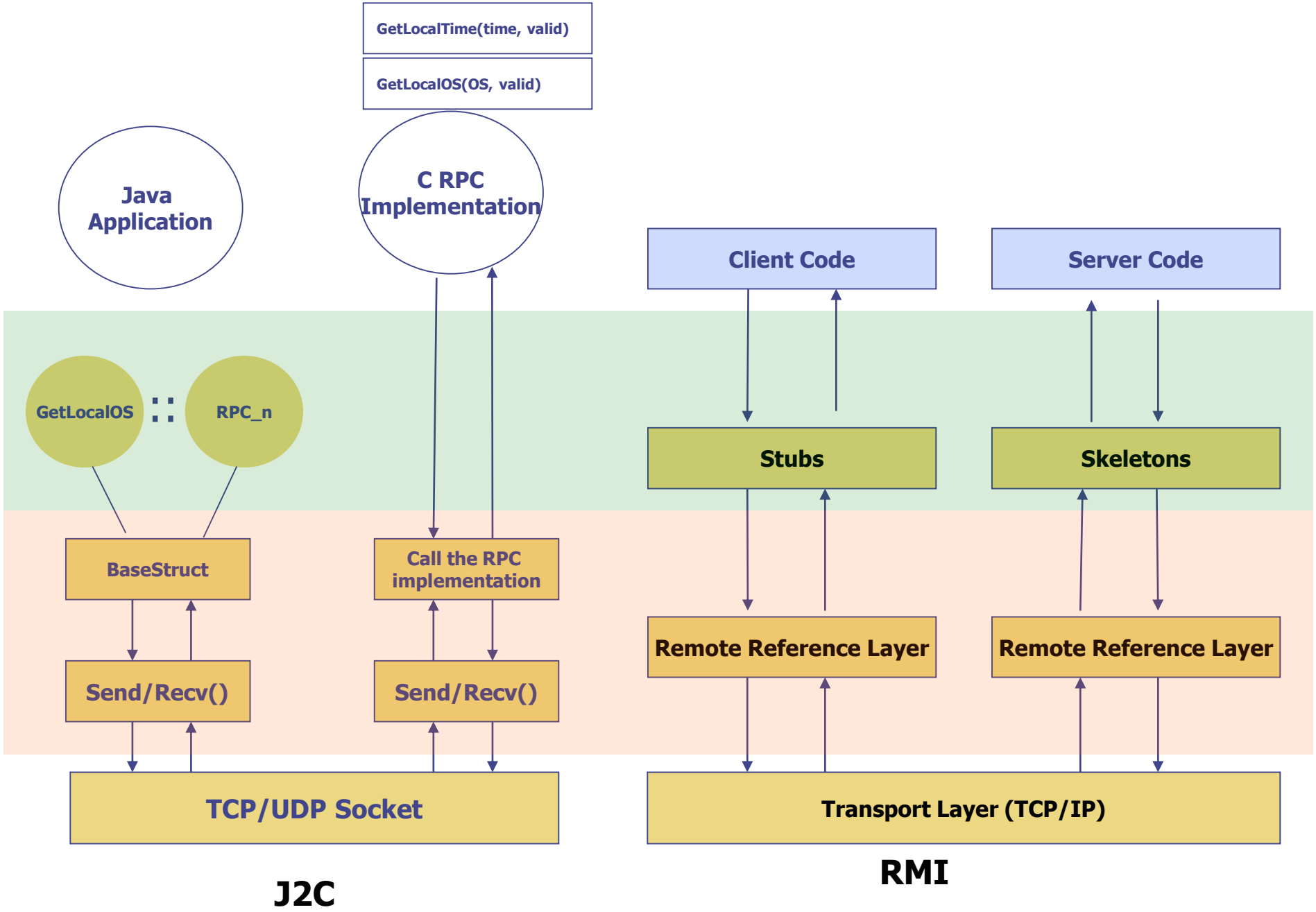
RMI Architecture

- 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



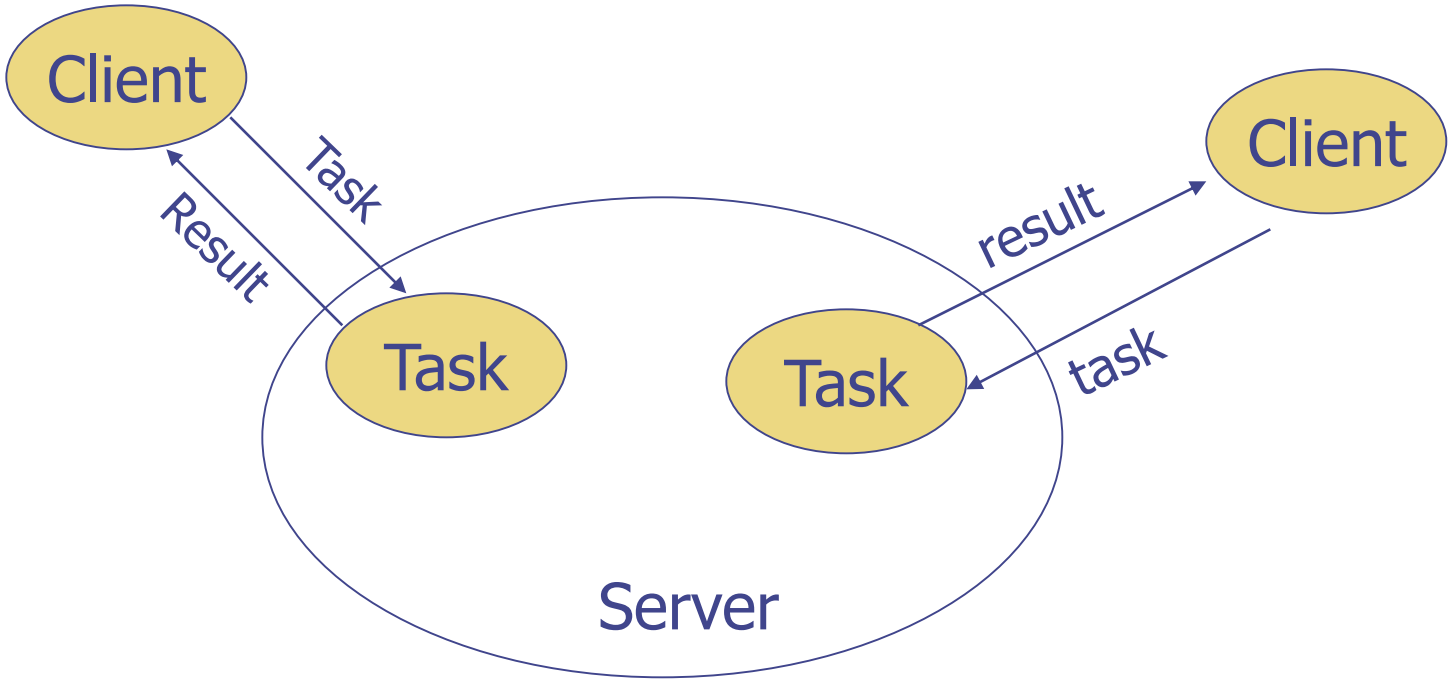
- Each remote object has two interfaces
  - Client interface – a stub/proxy of the object
  - Server interface – a skeleton of the object
- The communication of stub and skeleton is done across the RMI link
  - Read parameters/make call/accept return/write return back to the stub
- Remote reference layer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


GetLocalTime(time, valid)
GetLocalOS(OS, valid)
Java Application
C RPC Implementation
Client Code
Server Code
GetLocalOS
RPC_n
Stubs
Skeletons
BaseStruct
Call the RPC implementation
Remote Reference Layer
Remote Reference Layer
Send/Recv()
Send/Recv()
TCP/UDP Socket
Transport Layer (TCP/IP)
J2C
RMI

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

Another Example: ComputeServer




```
public interface Task {
        Object run();
}
```

When run is invoked, it does some computation and returns an object that contains the results

Remote Interface of ComputeServer

```
import java.rmi.*

public interface ComputeServer extends Remote

{

        Object compute(Task task) throws RemoteException;

}
```

The only purpose of this remote interface is to allow a client to create a task object and send it to the Server for execution, returning the results

Remote Object ComputeServerImpl

```
import java.rmi.*;
Import java.rmi.server.*;
public class ComputeServerImpl extends UnicastRemoteObject implements ComputeServer
{
    public ComputeServerImpl() throws RemoteException { }
    public Object compute(Task task) { return task.run(); }
    public static void main(String[] args) throws Exception
    {
        ComputeServerImpl server = new ComputeServerImpl();
        Naming.rebind("ComputeServer", server); // by default, registry server is local
    }
}
```

A Task Example

```
public class MyTask implements Task, Serializable
{
    double data[];
    MyTask()
    {
       ReadFile(data, “c:\data.txt”);
    }
    double run()
    {
        //ReadFile(data, “c:\data.txt”);
       // some CPU-intensive operations on data[];
    }
}
```

Submitting a Task

```
public class RunTask
{
    public static void main(String[] args) throws Exception
    {
        Mytask myTask = new MyTask();

        // set the data[] of myTask;

        // submit to the remote compute server and get result back
        Remote cs = Naming.lookup("rmi://localhost/ComputeServer");
        Double result = ((ComputeServer) cs).compute(myTask);
    }
}
```

RMI Safety and Security

- RMISecurityManager imposes restrictions on downloaded objects the same on applets
  - No access to local disk I/O
  - No socket connection except to codebase, etc.

```
public static void main(String[] args) throws Exception
{
        System.setSecurityManager(new RMISecurityManager());
        ComputeServerImpl server = new ComputeServerImpl();
        Naming.rebind("ComputeServer", server);
        return;
}
```

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
- Other development of RMI
  - JINI, Java Remote Method Protocol (JRMP)