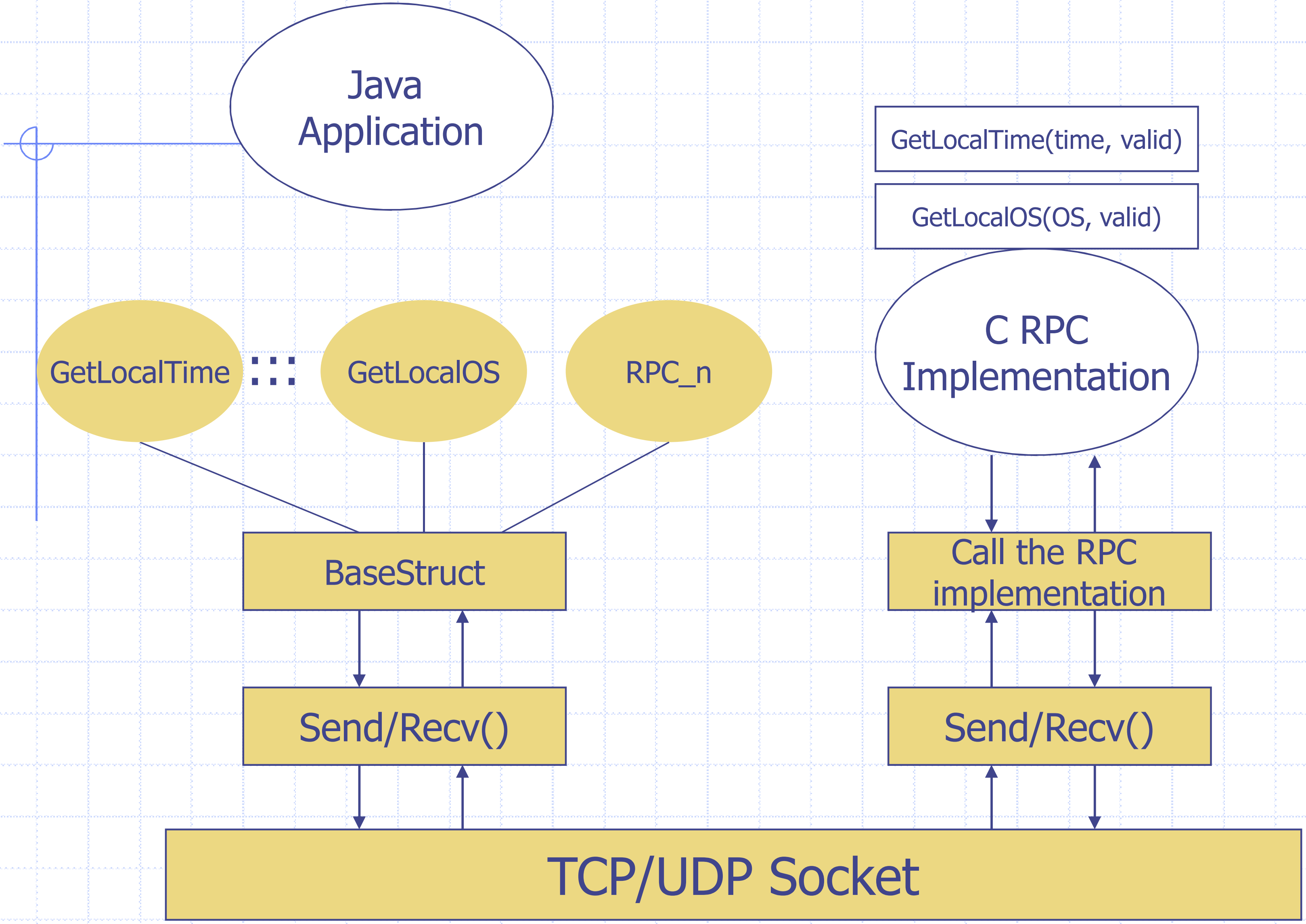
Keys to J2C RPC

- Define a generic RPC model
  - That can represent any possible data structure
- RPC Implementation replies only on the generic model
  - Parameter marshalling
  - Execution
  - Parameter unmarshalling
- Based on an RPC definition, we need to generate only its corresponding RPC class


Java Application
GetLocalTime(time, valid)
GetLocalOS(OS, valid)
GetLocalTime
GetLocalOS
RPC_n
C RPC Implementation
BaseStruct
Call the RPC implementation
Send/Recv()
Send/Recv()
TCP/UDP Socket

What Defines a Data Structure

- struct = name + a list of fields
- What can be changed?
  - Name of data structure (i.e., RPC)
  - Number of fields
  - Each field
    - Data type
    - Variable name

```
typedef struct
{
    int     *time;
    char    *valid;
} GET_LOCAL_TIME;
```

What Defines a Field

- Field = type + name
- Primitive data type
  - int (4 bytes)
  - short (2 bytes)
  - char (1 bytes)
  - etc.
- Complex data type
  - data structure
  - array

```
typedef struct
{
    int    x;
    char   y;
    short  z[20];
} DS1;

typedef struct
{
    DS1    x1[100];
    DS2   *x2;
} DS2;
```

Generic Data Structure

```
public abstract class BaseStruct
{
     String        Name;
     BaseField     Field[] = null;
}

public abstract class BaseField
{
     String           Name;

     BaseType         BType           = null;
     BaseType         BTypeArray[]    = null;
     BaseStruct       BStruct         = null;
     BaseStruct       BStructArray[]  = null;
}
```

```
typedef struct
{
    int     x;
    char    y;
    short   z[20];
} DS1;

typedef struct
{
    DS1     x1[100];
    DS2    *x2;
} DS2;
```

Primitive Type Abstraction

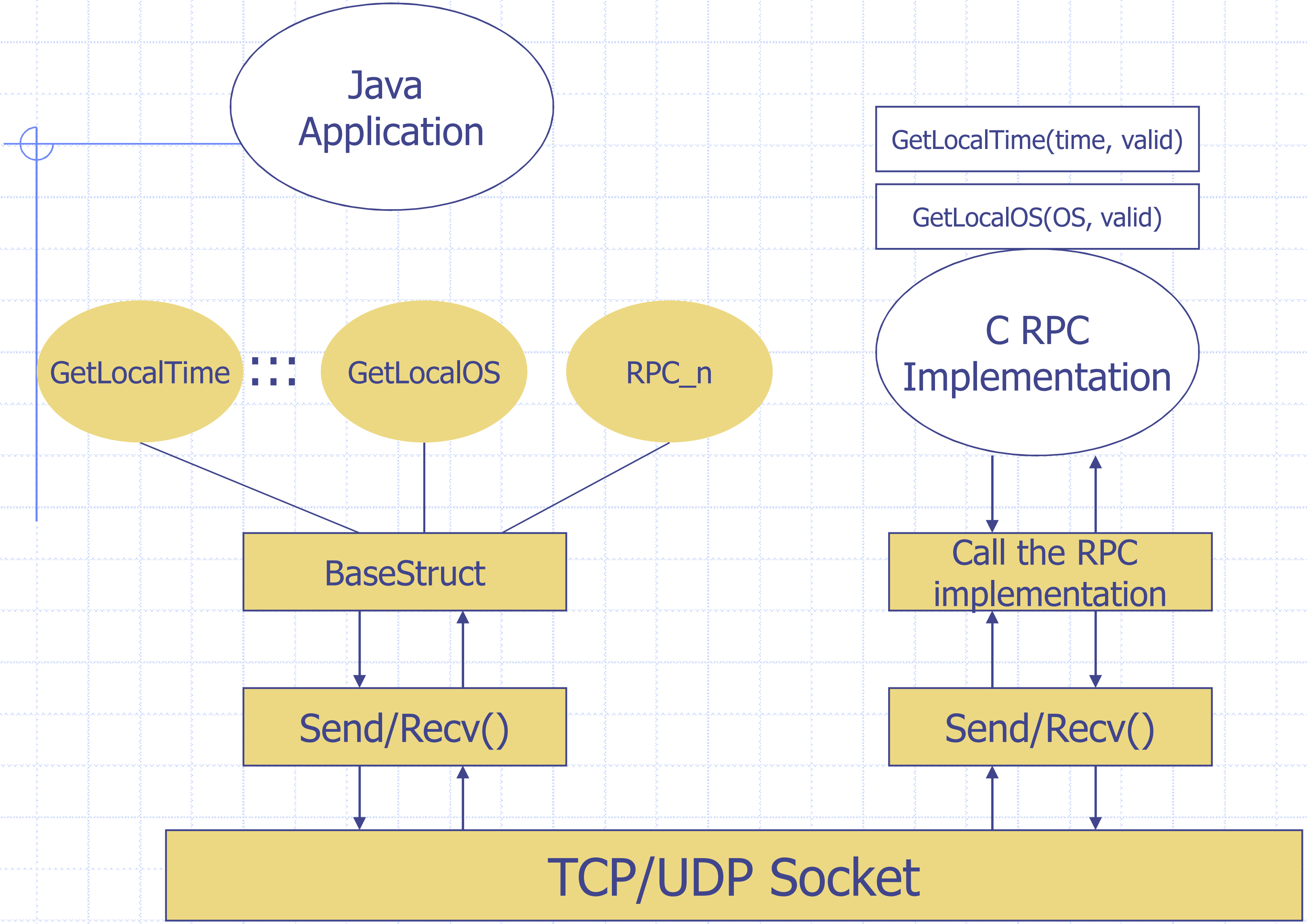
```
public abstract class BaseType
{
    byte buffer[];
    int  myType;

    public byte[] toByte();
    public byte[] setvalue(byte buf[]);
    public getSize();
}
```

```
public class U8 extends BaseType
{
    public U8(char value)
    {
       buffer = new byte[1];
       buffer[0] = value;
       myType = TYPE_U8;
    }
}
```

Primitive Array Abstraction

```
public class BaseArray extends BaseType
{
    int ArrayType;
    public BaseArray(int type, int array_size);
    public int getSize();
}
```

```
public class U8_ARRAY extends BaseArray
{
    public U8_ARRAY(int size)
    {
       super(TYPE_U8_ARRAY, size);
    }
}
```


Java Application
GetLocalTime(time, valid)
GetLocalOS(OS, valid)
C RPC Implementation
GetLocalTime
GetLocalOS
RPC_n
BaseStruct
Call the RPC implementation
Send/Recv()
Send/Recv()
TCP/UDP Socket

Implementation of DS.Execute()

- Create a binary buffer

```
int length = 100;
for (int i=0; i<ds.getFieldNumber(); i++)
{
    length = length + ds.field[i].getsize();
}
byte[] buf = new byte[4+length];
```

- Marshall parameters into the buffer

```
buf[0, 4] = length; offset = 4;
buf[offset, 100] = ds.getName(); offset = offset + 100;
for (int i=0; i<ds.getFieldNumber(); i++)
{
    buf[offset, ds.filed[i].getSize()] = ds.field[i].toByte();
    offset = offset + ds.filed[i].getSize();
}
```

- Send/receive the buffer to/from the RPC server

```
s = CreateSocket(IP, port);
SendPacket(s, buf, buf.length());
RecvPacket(s, buf, buf.length());
```

- Set parameters according to the buffer

```
offset = 100;
for (int i=0; i<ds.getFieldNumber(); i++)
{
    Ds.field[i].setValue(buf, offset);
    offset = offset + ds.field[i].getSize();
}
```

Remote Method Invocation

CS587x Lecture 6
Department of Computer Science
Iowa State University

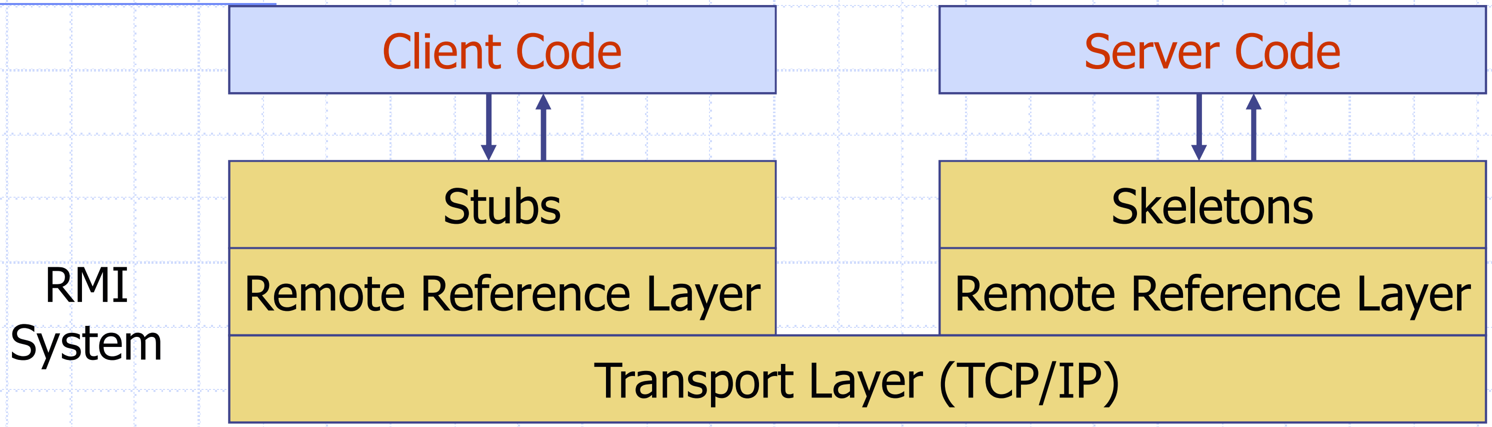
Introduction of RMI

- 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 objects
    - Return value could be an object as well

RMI Architecture

- The definition of behavior and the implementation of that behavior are two separate concepts
  - Clients are concerned about the definition of a service
    - Coded using a Java interface
    - Interfaces define behavior
  - Servers are focused on providing the service
    - Coded using a Java class
    - Classes define implementation

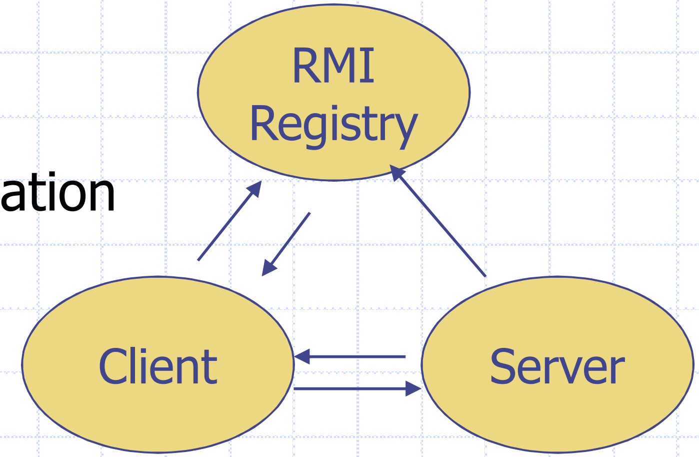
RMI Layers



- A stub is the proxy of an object while the remote service implementation class is the real object
- A skeleton handles the communication with the stub 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

RMI Components

- RMI registry service
  - New RMI servers register their location
  - RMI clients find server(s) location via the lookup service
- Servers
  - Construct an implementation of an interface
  - Provide access to methods via skeleton
  - Register location with registry
- Clients
  - Ask registry for location of implementation
  - Construct stub
  - Call methods on server via 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 all methods defined by service interface
- Example

```
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
- Example:

```
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

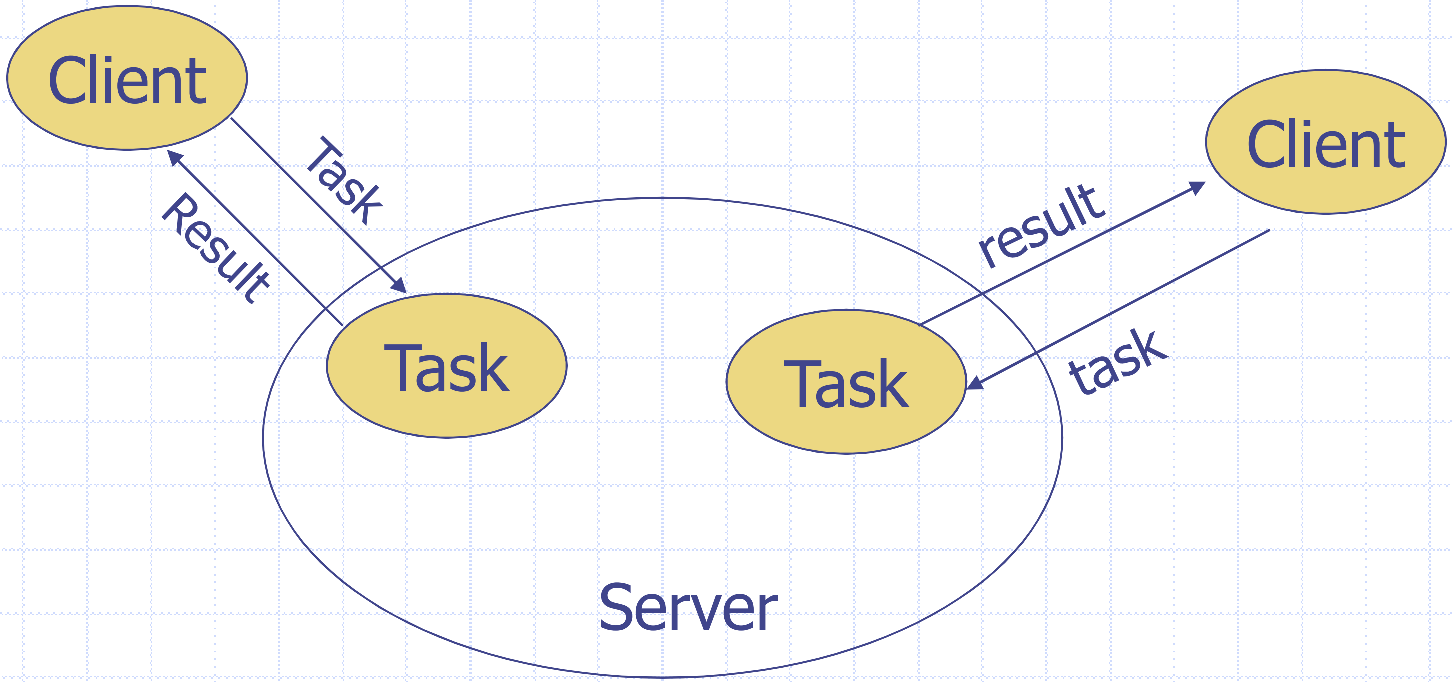
- Obtain a reference to the remote interface
- Invoke desired methods on the reference

```
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`**

Another Example: Compute Server



An Example of Corporate Server

Task interface

```
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);
    }
}
```

A Task Example

```
public class MyTask implements Task, Serializable
{
    double data[];
    SubTask st;
    void setTask(SubTask newTask) { st = newTask; }
    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);
    }
}
```

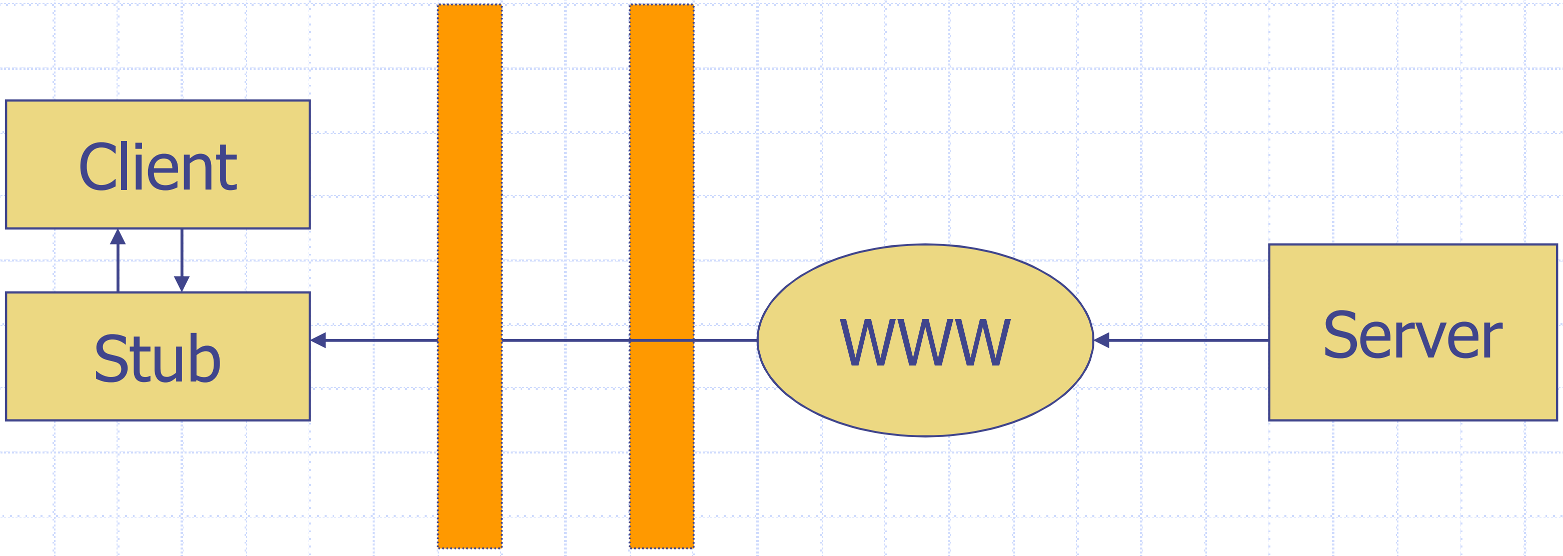
RMI Safety and Security

- 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;
}
```

Firewalls

- Firewalls block all network traffic, with the exception of those intended for certain “well-known” ports
- RMI traffic is typically blocked by firewall
    - RMI transport layer opens dynamic socket connections between the client and the server to facilitate communication



Firewalls

RMI Solutions

- The sequence of trying to make connections:
  - Communicate directly to the server's port using sockets
  - If this fails, build a URL to the server's host and port and use an HTTP post request on that URL, sending the information to the skeleton as the body of the POST.
    - need to set system property http.proxyhost
  - If this also fails, build a URL to the server's host using port 80, the standard HTTP port, using a CGI script that will forward the posted RMI request to the server.
    - java-rmi.cgi script needs to be install
    - java.rmi.server.hostname = host.domain.com
    - A more efficient solution is using servlet
  - If all fails, RMI fails.

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 better technology for n-tier architectures since it intermix easily with servlets

References

- http://java.sun.com/marketing/collateral/javarmi.html
- http://developer.java.sun.com/developer/onlineTraining/rmi/RMI.html