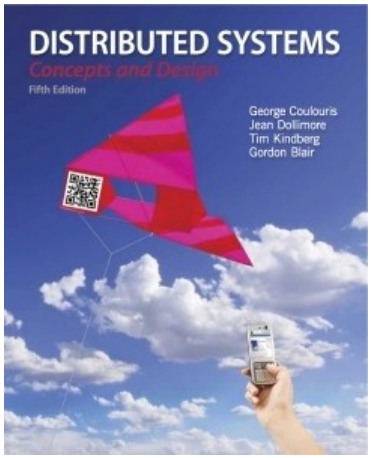
Week 3
Distributed Objects and Remote Invocation



References:

Chapter 5

Distributed Systems: Concepts and Design

Coulouris, Dollimore, Kindberg and Blair

Edition 5, © Addison Wesley 2011

Java RMI Tutorial, SUN Microsystems

http://docs.oracle.com/javase/tutorial/rmi/overview.html

Learning Objectives

- Describe distributed object applications in terms of:
  - Objects and distributed objects
  - Remote object reference
  - Features of distributed object models
- Interpret Remote Method Invocation (RMI) in terms of:
  - Remote interface
  - Remote Invocation Semantics
  - Client and server programs using RMI
- Develop Java RMI applications by a case study: the ***computeEngine***

Distributed Object Models

- The discussion context
  - This discussion is concerned with programming models for distributed applications.
  - Distributed applications are composed of cooperating programs running in several different processes.
  - A process need to be able to invoke operations in other processes that are often running in different computers.
  - This discussion is based on the distributed object model, which consists of geographically distributed objects.

Distributed Object Models

- The object-oriented model has the following features:
  - A program consists of a collection of interacting objects.
  - An object consists of a set of data and a set of methods.
  - Data encapsulated in an object should be accessed via its methods.
  - An object communicates with other objects by invoking their methods – passing arguments and receiving results.

Distributed Object Models

- In object-oriented model, a program is logically partitioned into separate parts, each of which is associated with an object.
- The distributed object model extends the logical partition to physical partition by physical distribution of objects into different computers.
- In object-oriented systems, an object must have a reference to another object in order to invoke its method.

```
aSocket=new DatagramSocket();
aSocket.send(request);
```

Distributed Object Models

- In distributed object systems, invocation among objects crosses the boundary of computers and involves network communication.
- In distributed object systems, other objects can invoke the methods of a remote object if they have access to its *remote object reference*.
- A remote object reference is an identifier that can be used throughout a distributed system to identify a remote object.

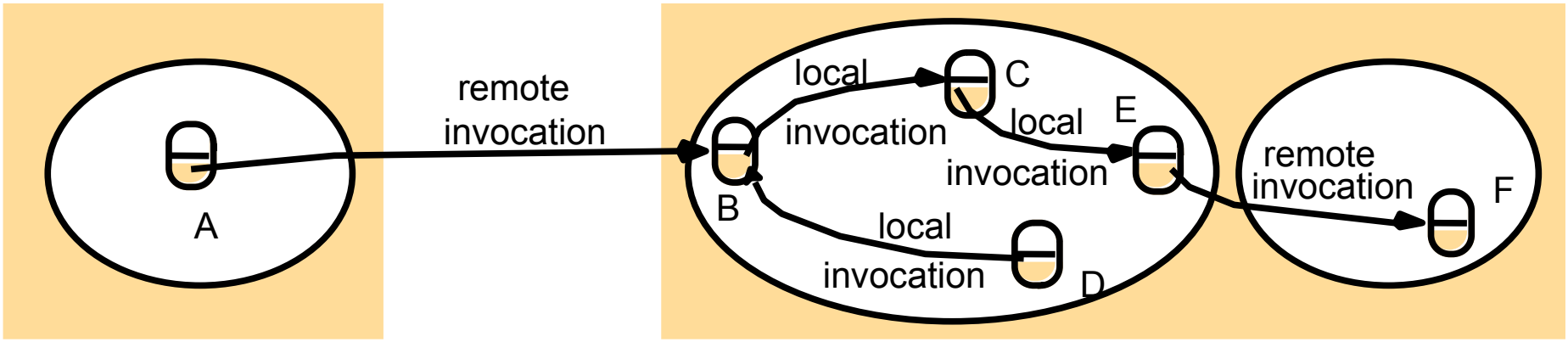
Distributed Object Models

- A remote object reference has the following structure:
  - IP address of the host and port number of the hosting process of the remote object.
  - Time of creation and a local object number of the remote object.
  - Information about the interfaces provided by the remote object.

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object number	interface of remote object

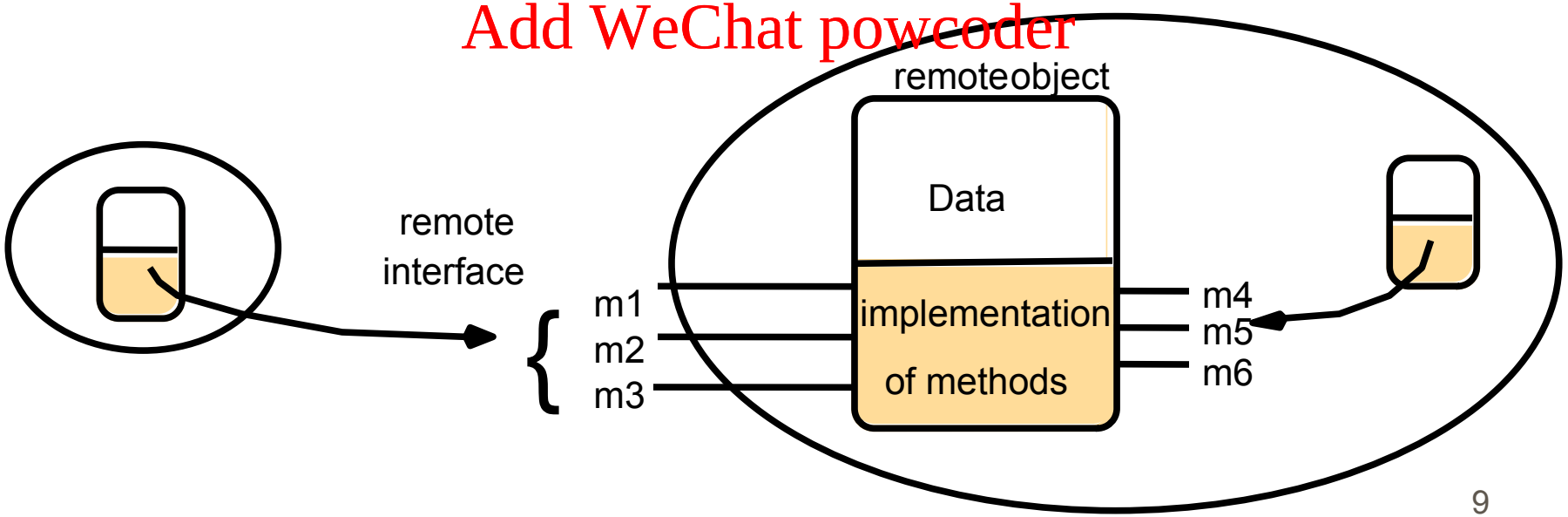
Distributed Object Models

- In distributed systems, each process contains objects, some of which can receive remote invocations.
- Those that can receive remote invocations are called *remote objects.*
- The *remote interface* specifies which methods can be invoked remotely.



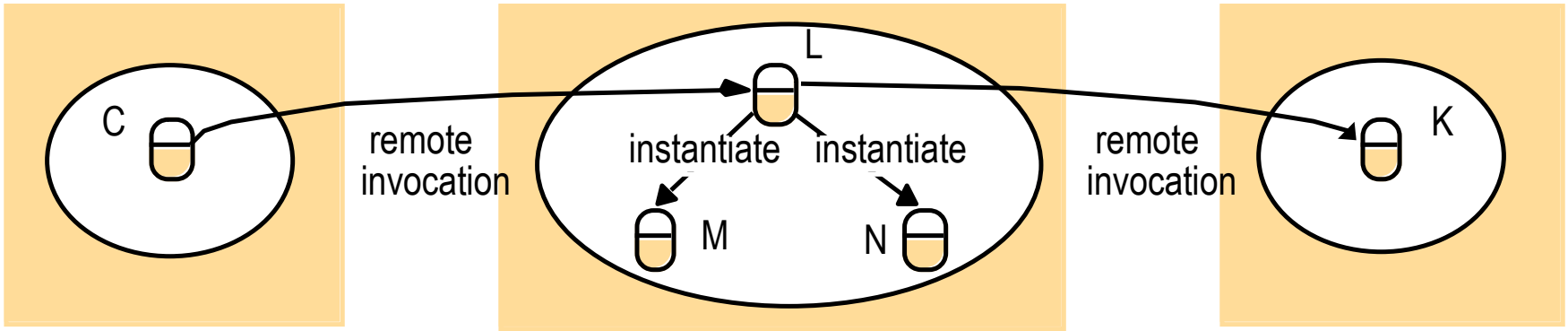
Distributed Object Models

- An interface provides a definition of a set of methods, the types of their arguments, return values and exceptions.
- The *remote interface* specifies which methods can be invoked remotely.



Distributed Object Models

- An invocation of a method can have three effects:
  - The state of the receiver may be changed.
  - New objects may be instantiated.
  - Further invocation may take place.



Remote Method Invocation

- A client object may invoke a method on a remote object, residing in another process, on another host.
- RMI layer is above TCP/UDP, providing communication between objects.
- In RMI, the processes that host remote objects are servers and the processes that host their invokers are clients.
- RMI uses request-reply protocol. Because of unreliable network, for all request-reply protocols, messages may get lost.

Remote Method Invocation

- Solutions for lost /retransmitted messages are:
  - Retry request message
  - Duplicate filtering
  - Retransmission of results
- In local object-oriented system, all methods are invoked exactly once per request.
- In distributed object system, we need to know what has happened if we do not hear results from a remote object.
  - The request is lost.
  - The response is lost.

Remote Method Invocation

- There are 3 different types of invocation semantics.

Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

Remote Method Invocation

- Maybe semantics
  - If requests are sent in unacknowledged messages, there is no certainty that the requests ever reached the server. In this case the invocation may have taken place or not.
- At-least-once semantics
  - If requests may be retransmitted due to communication failures (e.g. no reply) but duplicates are not filtered by the server, the retransmission of a request may result in the re-execution of the method or procedure.

Remote Method Invocation

- At-most-once semantics
  - If the system supports retransmission of requests and duplicate filtering at the server, we can be sure that re-execution does not happen.
  - Java RMI uses at-most-once semantics.
- The transparency of RMI aims at:
  - Making no distinction in syntax between a local invocation and a remote invocation.
  - Making objects of remote invocations be able to recover from failure.
  - Aborting a remote invocation to have no effect on the server.

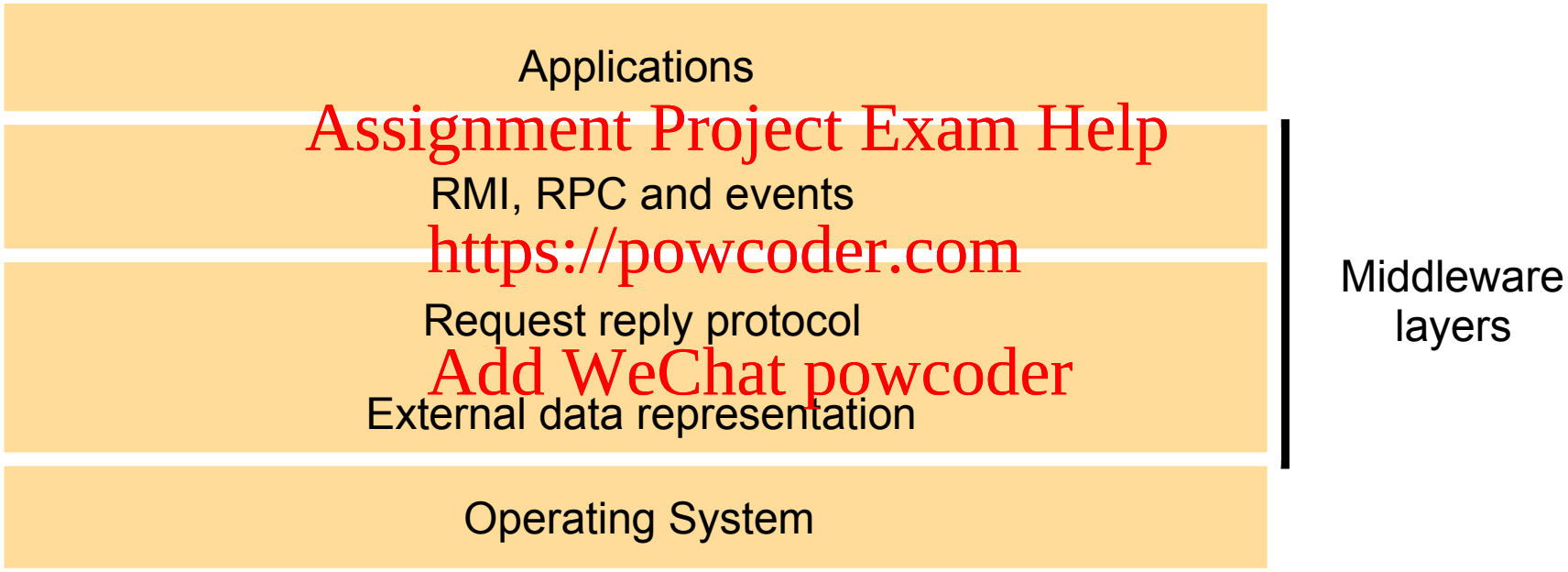
Remote Method Invocation

- Such transparency requires:
  - Automated marshalling and unmarshalling of arguments and return values.
  - Exception handling capability of distributed objects.
  - State consistency maintenance of distributed objects.
- Middleware provides a high-level abstraction above the basic building blocks of processes and message passing.

Remote Method Invocation

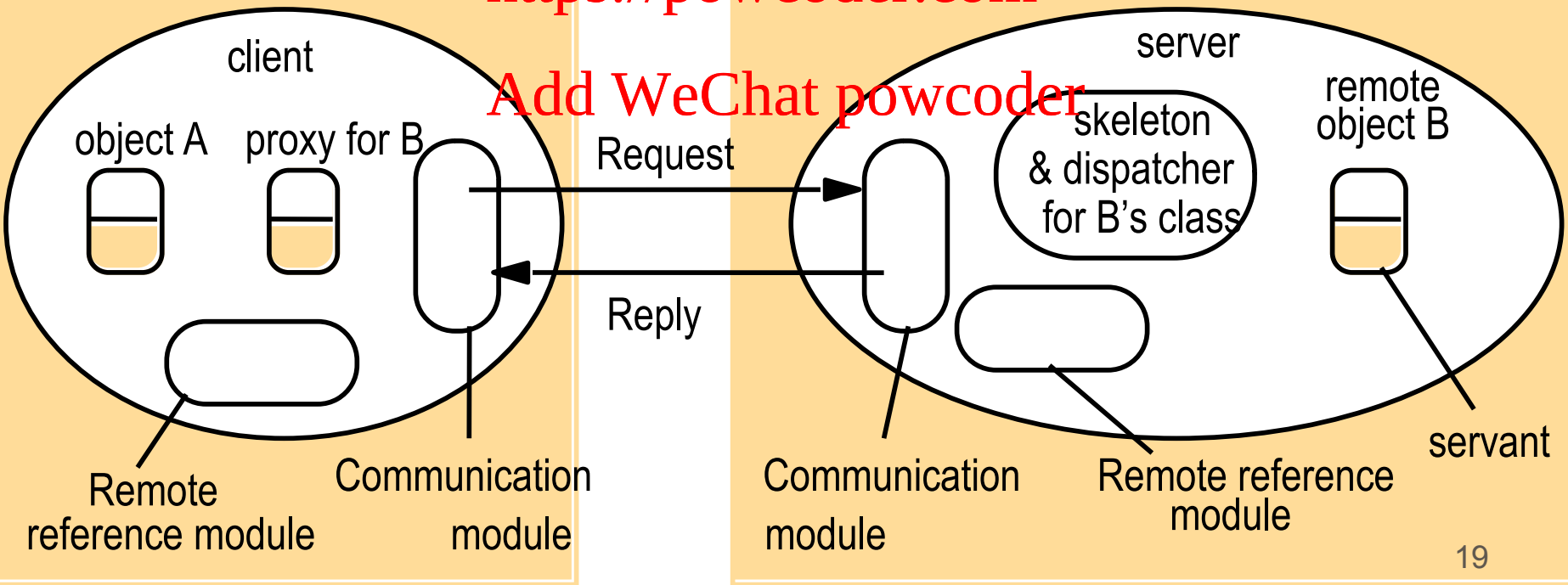
- Middleware consists of a programming paradigm and runtime environment, providing
  - Location transparency of distributed objects.
  - Independence of
    - Communication protocols
    - Computer hardware
    - Operating systems
    - Programming languages
- Distributed object middleware includes CORBA, SUN RPC, and Java RMI etc.

Remote Method Invocation



Remote Method Invocation

- The architecture of RMI consists of the following components.
- The classes (codes) for the proxy, dispatcher and skeleton are generated automatically by an interface compiler.



Remote Method Invocation

- The Communication Module
  - is responsible for transmitting the requests and replies between the client and the server.
- The Remote Reference Module
  - maintains a *remote object table* which has an entry for each remote object held locally, and each local proxy for a remote object.
- The proxy
  - makes the remote invocation appear as if it were transparent.
  - offers a method corresponding to each method of the interface of the remote object.
  - marshalls an invocation into a request message and waits for the reply and unmarshalls the returning results.

Remote Method Invocation

- The dispatcher
  - receives incoming requests from the server communication module.
  - selects the appropriate method in the skeleton passing on the request message which still contains the marshalled arguments.
- The skeleton
  - unmarshalls the arguments in the request message and invokes (locally) the corresponding method in the remote object.
  - marshalls the result, together with any exceptions, in a reply message which is sent to the proxy.

Case Study of Java RMI

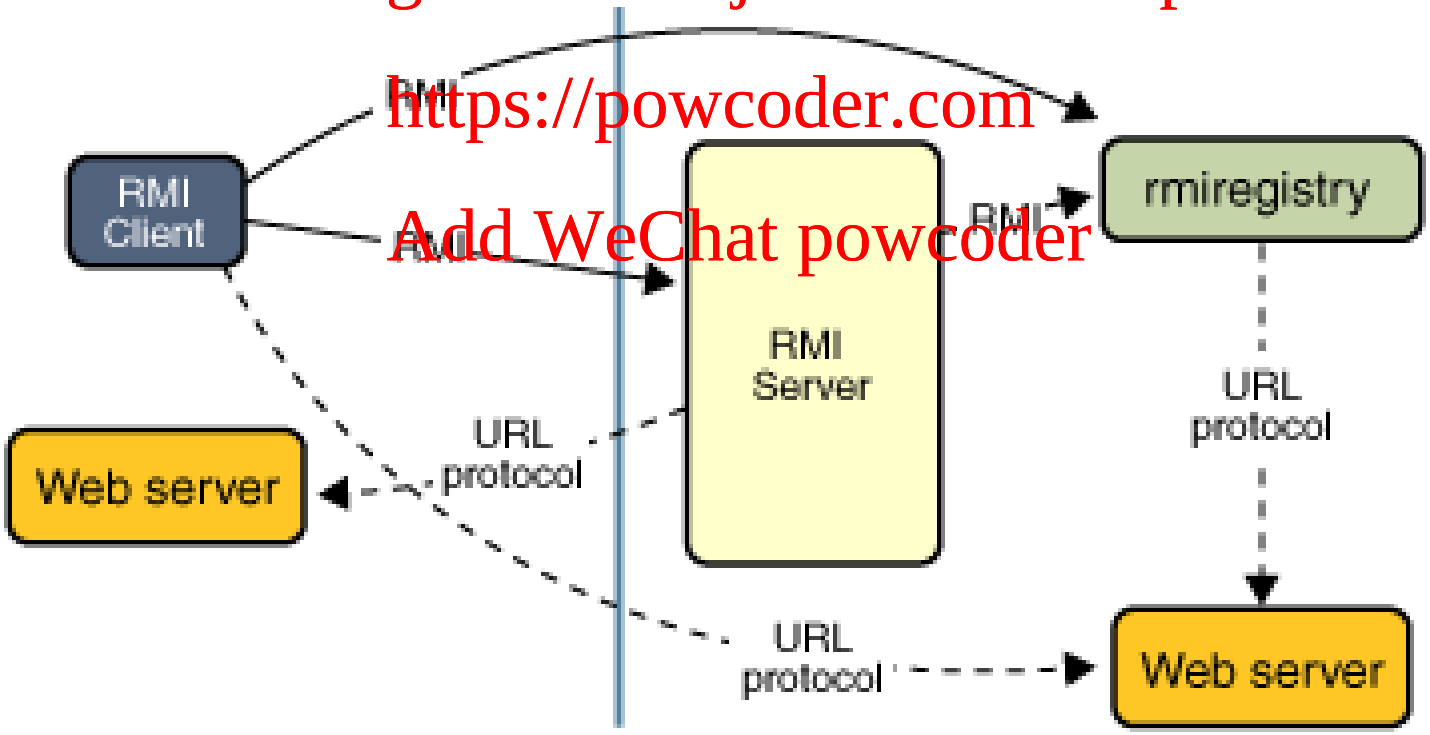
- This section is based on RMI tutorial from SUN Microsystems.
  - The original materials are from **http://docs.oracle.com/javase/tutorial/rmi/overview.html**
  - The author of these lecture slides would like to acknowledge SUN Microsystems for using the materials.
  - The adoption of the above materials aims at providing a comprehensive and easy understanding case study of Java RMI.

Case Study of Java RMI

- Java RMI distributed object applications comprise two separate programs:
  - A server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects.
  - A client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them.
- Java RMI provides the mechanism by which the server and the client communicate and pass information back and forth.

Case Study of Java RMI

- The Architecture of Java RMI is illustrated in the following figure.



Case Study of Java RMI

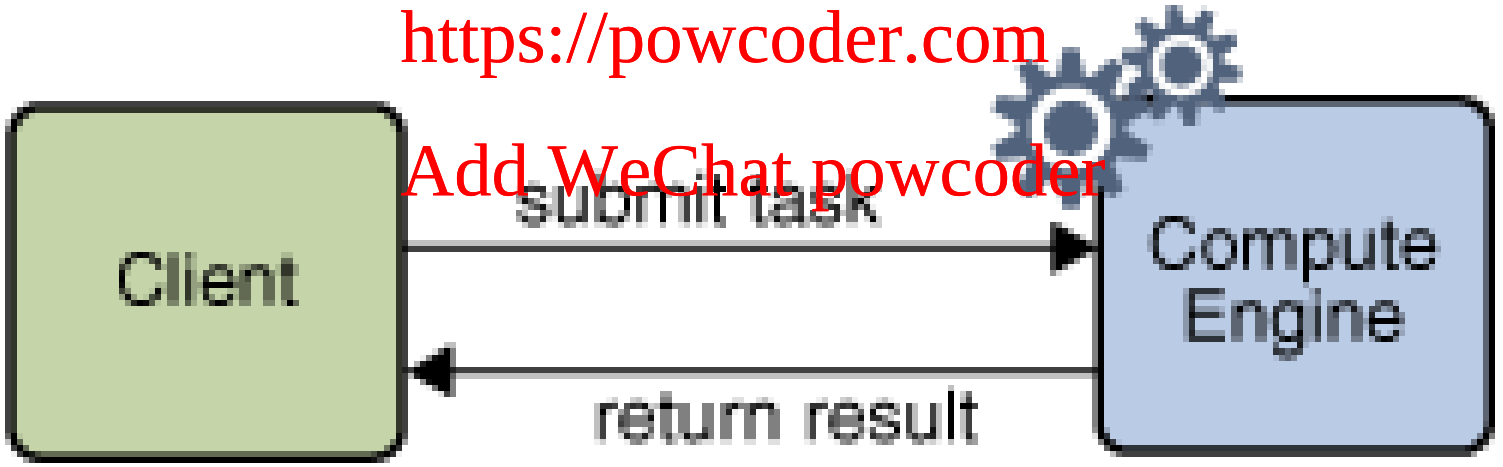
- The server calls the registry to associate (or bind) a name with a remote object.
- The client looks up the remote object by its name in the server's registry and then invokes a method on it.
- Java RMI uses an existing web server to load class definitions, from server to client and from client to server, for objects when needed.

Case Study of Java RMI

- The ***computeEngine*** application
  - This case study is based on a distributed object application: ***computeEngine***.
  - The ***computeEngine*** is a remote object on the server that takes tasks from clients, runs the tasks, and returns any results.
  - The tasks are run on the machine where the server is running.
  - This type of distributed application can enable a number of client machines to make use of a particularly powerful machine or a machine that has specialized hardware.

Case Study of Java RMI

- The ***computeEngine*** application



Case Study of Java RMI

- Remote Interface
  - Defines methods that can be invoked remotely.
  - Extends the interface ***java.rmi.Remote***.
  - Each method of the interface declares ***java.rmi.RemoteException*** in its throws clause, in addition to any application-specific exceptions.


```
package compute;
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface Compute extends Remote {
  <T> T executeTask(Task<T> t) throws RemoteException;
}
```

Case Study of Java RMI

- The second interface needed for the ***computeEngine*** is the ***Task*** interface, which is the type of the parameter to the ***executeTask()*** method in the *ComputeEngine* interface


```
package compute;
public interface Task<T> {
   T execute();
}
```


- Remote object
  - Objects with methods that can be invoked across Java virtual machines are called *remote objects*.
  - An object becomes remote by implementing a *remote interface*.

Case Study of Java RMI

- Server program
  - Remote objects are normally defined in a server program.
  - An RMI server program needs to create and install a security manager.
  - An RMI server program needs to create and export one or more remote objects.
  - An RMI server program needs to register at least one remote object with the ***RMI registry***.

Case Study of Java RMI

```
package engine;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import compute.Compute;
import compute.Task;
public class ComputeEngine implements Compute {
  public ComputeEngine() {
    super();
}
  public <T> T executeTask(Task<T> t) {
    return t.execute();
}
```


☞ This program continues on the next slide

Case Study of Java RMI

```
  public static void main(String[] args) {
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new SecurityManager());
    }
    try {
      String name = "Compute";
      Compute engine = new ComputeEngine();
      Compute stub=(Compute)
               UnicastRemoteObject.exportObject(engine, 0);
      Registry registry = LocateRegistry.getRegistry();
      registry.rebind(name, stub);
System.out.println("ComputeEngine bound");
    } catch (Exception e) {
        System.err.println("ComputeEngine exception:");
        e.printStackTrace();
      }
  }
}
```


The end of this program

Case Study of Java RMI

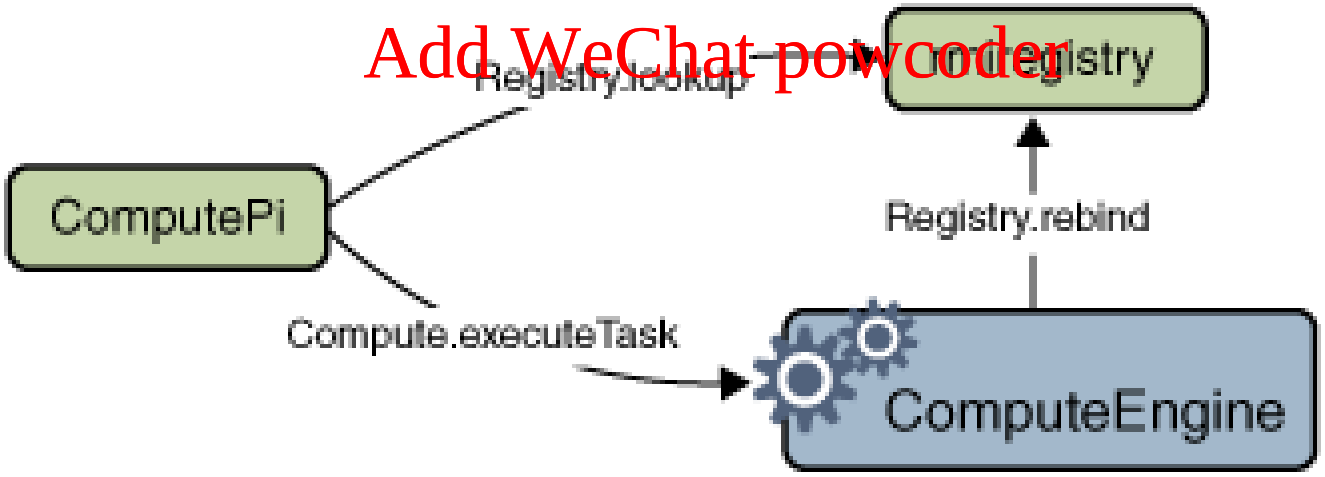
- Client program
  - A client program contains invokers of the remote objects.
  - Two separate classes make up the client in this application.
    - The first class, ***ComputeEngine***, looks up and invokes a ***Compute*** object.
    - The second class, ***Pi***, implements the ***Task*** interface and defines the work to be done by the ***computeEngine***.
    - The job of the ***Pi*** class is to compute the value of π to some number of decimal places.

Case Study of Java RMI

- Client program
  - The client program begins by installing a security manager.
  - The client invokes the ***lookup*** method on the registry to look up the remote object by name.
  - The client creates a new ***Pi*** object
  - The client invokes the ***executeTask*** method of the ***Compute*** remote object, which returns an object of type ***BigDecimal***.
  - The program prints the result finally.

Case Study of Java RMI

- The flow of messages among the client: ***ComputePi***, the RMI registry: ***rmiregistry***, and the server: ***ComputeEngine***.



Case Study of Java RMI

```
package client;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.math.BigDecimal;
import compute.Compute;
public class ComputePi {
  public static void main(String args[]) {
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new SecurityManager());
    }
```


☞ This program continues on the next slide

Case Study of Java RMI

```
    try {
      String name = "Compute";
      Registry registry=
              LocateRegistry.getRegistry(args[0]);
      Compute comp = (Compute) registry.lookup(name);
      Pi task = new Pi(Integer.parseInt(args[1]));
      BigDecimal pi = comp.executeTask(task);
      System.out.println(pi);
      } catch (Exception e) {
            System.err.println("ComputePi exception:");
            e.printStackTrace();
      }
   }
}
```


The End of this program

Case Study of Java RMI

- The ***Pi*** class implements the ***Task*** interface and computes the value of π to a specified number of decimal places.
- For this application, the actual algorithm of ***Pi*** is unimportant.
- What is important is that the algorithm is computationally expensive.
- It should be executed on a capable server.

Case Study of Java RMI

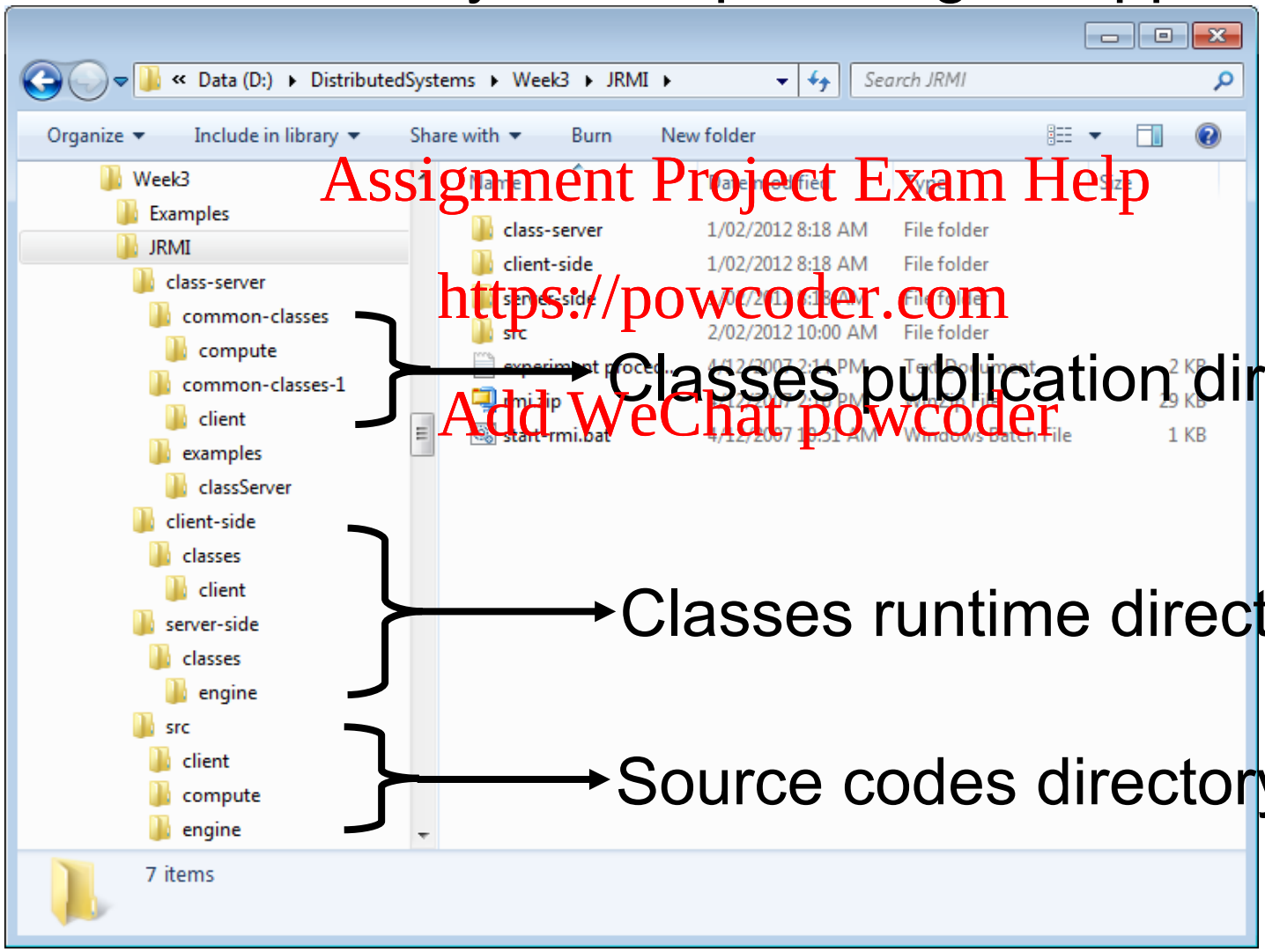
```
package client;
import compute.Task;
import java.io.Serializable;
import java.math.BigDecimal;
public class Pi implements Task<BigDecimal>, Serializable
{

.....
.....
.....


}
```

Case Study of Java RMI

- The hierarchy of compute engine application



Case Study of Java RMI

- Building Classes
  - Under directory: **`d:\DistributedSystems\week3\jrmi\src\`**, using the following commands to compile remote interfaces and package them into a jar file.


```
javac compute\Compute.java compute\Task.java
jar cvf compute.jar compute\*.class
```

  - These two interfaces comprise the interaction protocol between the client and server programs.

Case Study of Java RMI

- Building Classes
  - Under directory: **d:\DistributedSystems\week3\jrmi\src**, using the following command to compile the server program.


```
javac  -cp .\compute.jar engine\ComputeEngine.java
```


  - Under directory: **d:\DistributedSystems\week3\jrmi\src**, using the following command to compile the client program.

```
javac  -cp .\compute.jar client\ComputePi.java
       client\Pi.java
```

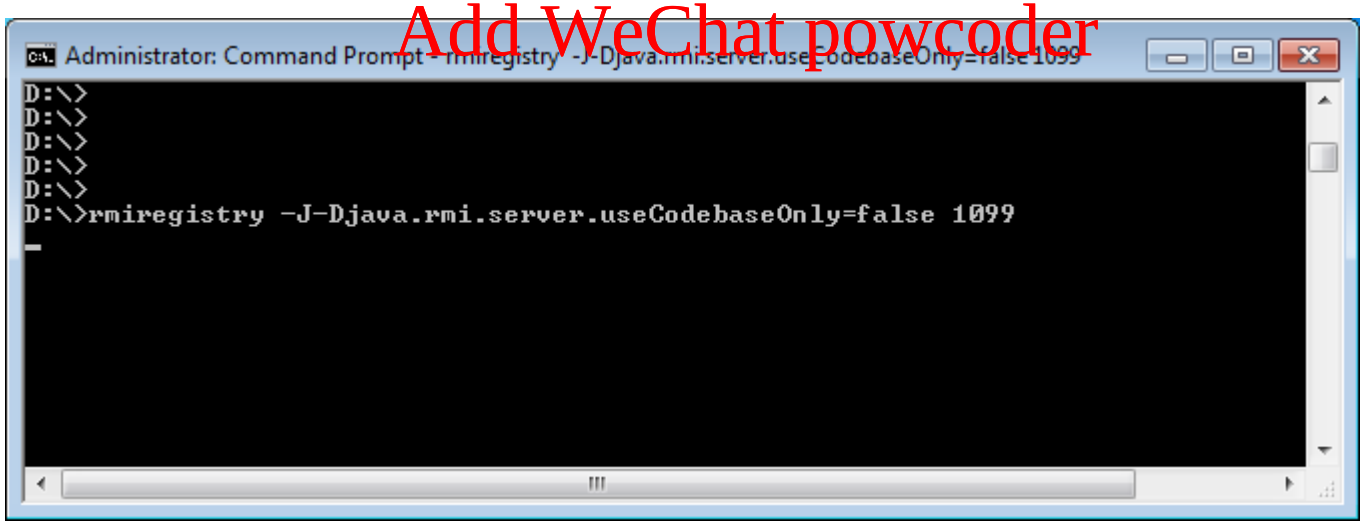
Case Study of Java RMI

- Publish classes at Web servers
  - Configure the root directory of a web server at `d:\DistributedSystems\week3\jrmi\class-server\common-classes\.`
  - Publish ***`Compute.class`*** and ***`Task.class`*** under directory `d:\DistributedSystems\week3\jrmi\class-server\common-classes\compute\.`
  - Configure the root directory of another web server at `d:\DistributedSystems\week3\jrmi\class-server\common-classes-1\.`
  - Publish ***`Pi.class`*** and ***`Task.class`*** under directory: `d:\DistributedSystems\Week3\JRMI\class-server\common-classes-1\client\.`

Case Study of Java RMI

- Start RMI registry by using following command.
- The registry is running at the background and has no outputs.


```
start rmiregistry -J-
Djava.rmi.server.useCodebaseOnly=false 1099
```

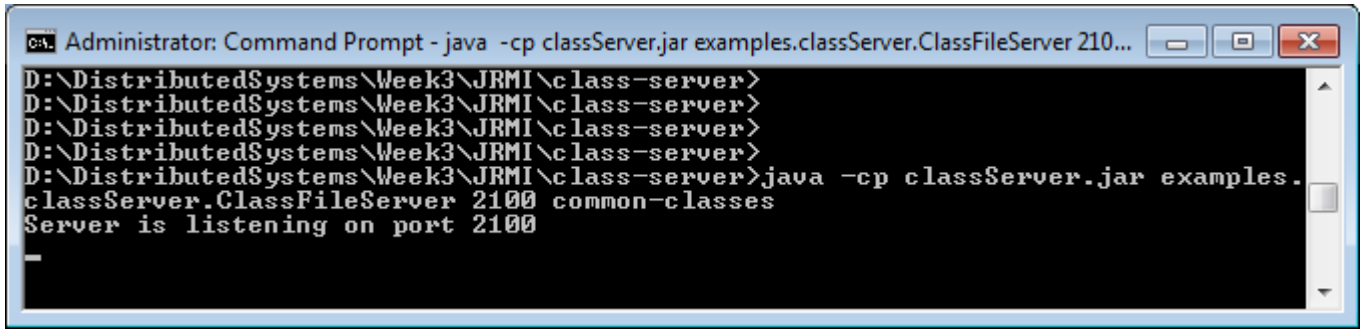



Case Study of Java RMI

- Start the first Web server by using the following command from directory: **`d:\DistributedSystems\week3\jrmi\class-server\`**

- The Web server can be downloaded from Week 3 block of the course site.

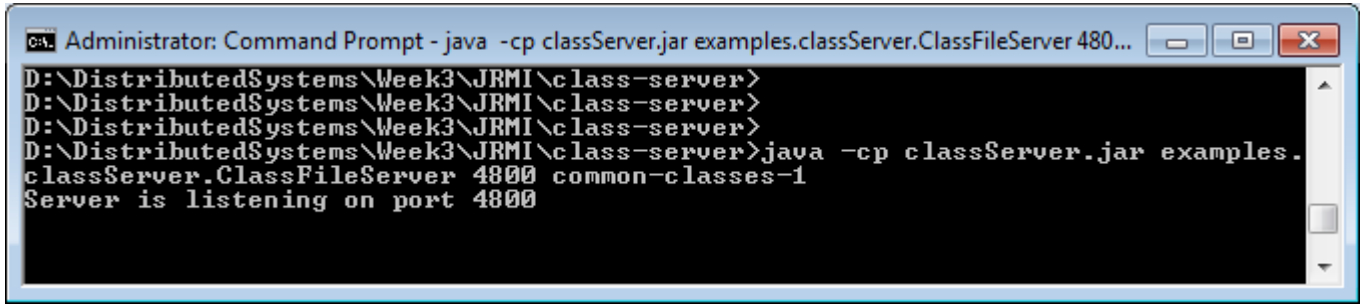

```
java -cp classServer.jar
examples.classServer.ClassFileServer 2100 common-
classes
```



Case Study of Java RMI

- Start the second Web server by using the following command from directory: **`d:\DistributedSystems\week3\jrmi\class-server\.`**


```
java -cp classServer.jar
examples.classServer.ClassFileServer 4800 common-
classes-1
```

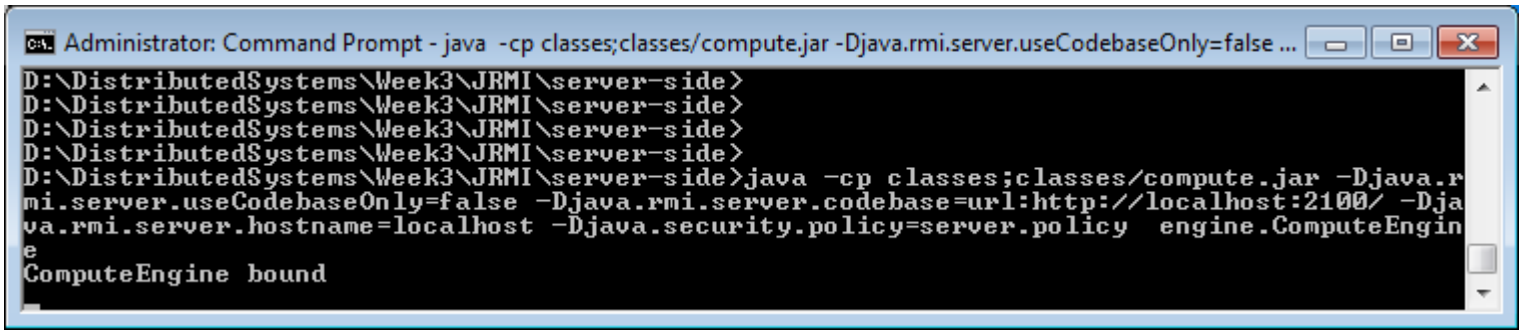
Case Study of Java RMI

- Start the server by using the following command

```
java -cp classes;classes/compute.jar -
Djava.rmi.server.useCodebaseOnly=false -
Djava.rmi.server.codebase=url:http://localhost:2100
/ -Djava.rmi.server.hostname=localhost -
Djava.security.policy=server.policy
engine.ComputeEngine
```


- The security policy is defined as:


```
grant codeBase "file:classes/" {
    permission java.security.AllPermission;
};
```



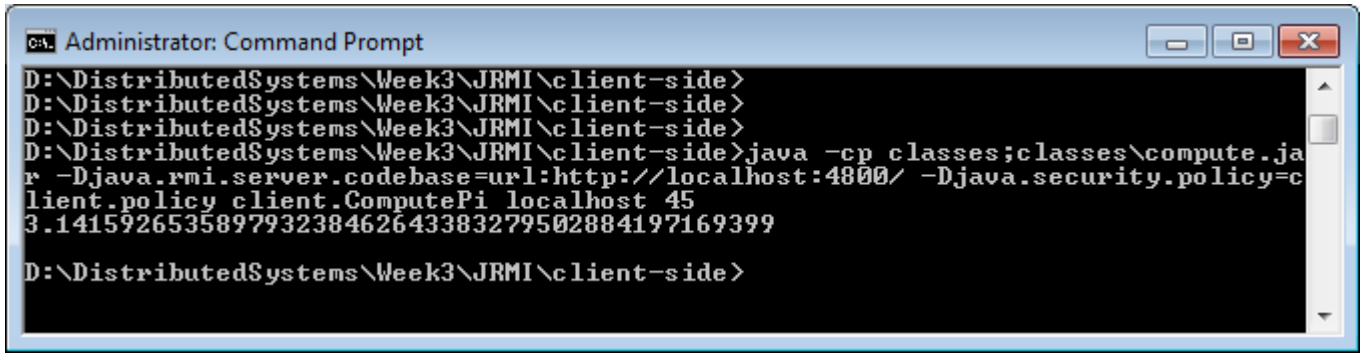
Case Study of Java RMI

- Start the client by using the following command

```
java -cp classes;classes\compute.jar -
Djava.rmi.server.codebase=url:http://localhost:4800
/ -Djava.security.policy=client.policy
client.ComputePi localhost 45
```

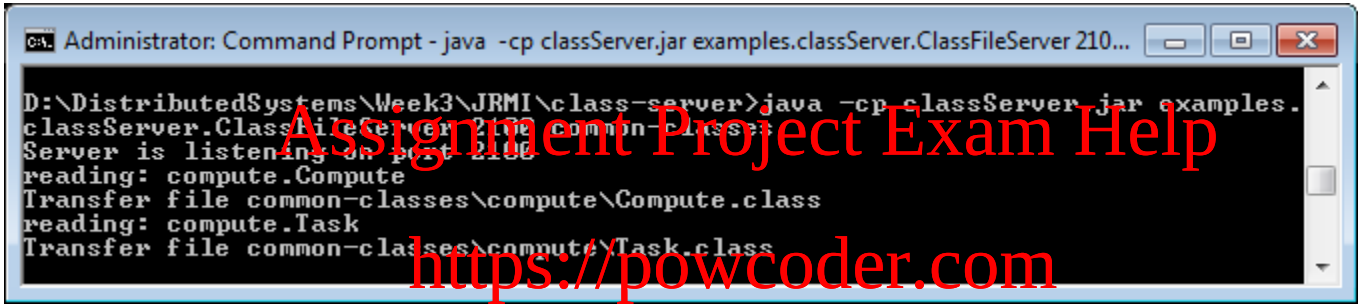

- The security policy is defined as:

```
grant codeBase "file:classes/" {
    permission java.security.AllPermission;
};
```

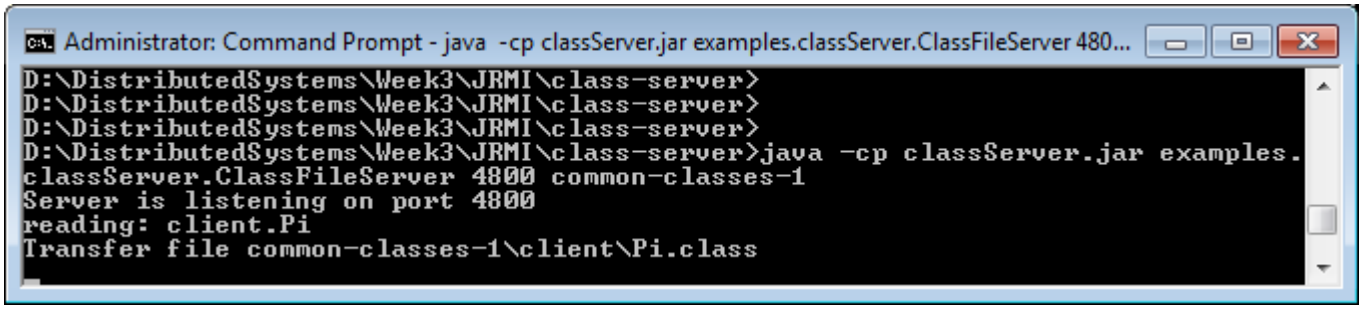


Case Study of Java RMI

- The output of Web server 1



- The output of Web server 2



Summary

- Distributed object applications are composed of cooperating objects running in several different computers.
- The distributed object model extends the logical partition to physical partition by physical distribution of objects into different computers.
- In distributed object systems, other objects can invoke the methods of a remote object if they have access to its *remote object reference*.
- Middleware provides a high-level abstraction and automated marshalling and unmarshalling of arguments and return values for remote invocation.