Distributed Object-Based Systems

- Architecture
  - Distributed Objects
  - Persistent and Transient Objects
  - Enterprise Java Beans
  - RMI example

Distributed Objects

- Invoking remote objects using:
  - CORBA
  - RMI
  - ...

- Compile-Time Objects
  RMI - same language (Java) for the objects

- Runtime Objects
  CORBA
  Interface specification language
  - Different languages for the objects

Persistent and Transient Objects

- Persistent objects will survive even if their server is shut down.
- Transient objects dies if their server is shut down.

- Is it enough with transient objects?
  - some people thinks so since persistent data normally is stored in a database anyway.

Enterprise Java Beans

- Used in many distributed system systems.

- Enterprise Java Beans (EJB)
  Java objects that is hosted in a special server:
  - Stateless session beans
  - Stateful session beans
  - Entity beans
  - Message-driven beans

  can more or less automatically be bound to different services:
  - RMI
  - JDBC (SQL database)
  - JNDI (naming)
  - JMS (messaging)
### Stateless Session Beans

- Transient object:
  - Invoked once
  - Does its work
  - Is then discarded
  - E.g., an SQL query.

### Stateful Session Beans

- An object that maintains client-related state.
  - Remembers client data
    - E.g., what a client clicks and writes on web forms when shopping on Internet.
  - Is discarded when the client is ready.

### Entity Beans

- A long-lived persistent object.
- Contains information that should be stored even after both client and server are shut down.
- Normally put in a database.

### Message-Driven Beans

- Objects that should react to incoming messages.
  - Not directly invoked by clients.
  - Publish-subscribe invocation (call-back).
This is an example with RMI that could not be done using CORBA.

We are working on our local computer and want to compute π

We’ve got a class Pi that carries out the computation with the requested number of decimals:

If we want many decimals we will get bad performance on our local computer.

By using a “super computer” we will get much better performance.

But we want to do it transparently from our local computer.

Send the computation to the super computer using RMI.

RMI Example - Local System

RMI Example - Distributed System

RMI — Remote Method Invocation

RMI — Marshalling

Marshalling

transforms method calls to a format that can be sent to another process.

transforms parameters to a format that can be sent to another process.

transforms return values to a format that can be sent to another process.

Un-Marshalling

transforms back to the original object at the receiver.

Create a proxy (stub).

Serializes the objects into sendable bytes.
Parameters or return values from "remote methods" can be of most types:

- local object,
- "remote" object
- primitive types.

i.e. everything that is a primitive type, a "remote object" or a serializable object.

(implments the interface java.io.Serializable)

Some few object can not be sent, e.g.

- file descriptor,
- encapsulate information that makes sense only within a single address space.

Most of the important classes in Java are serializable.

e.g. in

- java.lang
- java.util

Rules for object transmission

<table>
<thead>
<tr>
<th>Local JVM</th>
<th>Remote JVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>primitive types</td>
<td>by value</td>
</tr>
<tr>
<td>object</td>
<td>by reference</td>
</tr>
<tr>
<td>remote object</td>
<td>by reference</td>
</tr>
</tbody>
</table>

NB. The difference between objects that are sent locally or remote.

- An object of type Vector will be copied if it is sent remote. A copy will be referred to at the remote site.
- An object of type Remote will on the other hand instead be referred remotely.

Creating a "remote" object

An object will be "remote" by implementing a remote interface:

- A remote interface extends the interface java.rmi.Remote.

Pi Example

Compute Engine's interface, Compute:

```java
package compute;
import java.rmi.*;
public interface Compute extends Remote {
    Object executeTask(Task t) throws RemoteException;
}
```
Client’s interface, Task:

```java
package compute;
import java.io.Serializable;
public interface Task extends Serializable
{
    Object execute();
}
```

Task extends java.io.Serializable interface. RMI uses “object serialization mechanism” for the transport of objects among JVMs.

The definition shows that

- an object of type Compute can receive an object of type Task.
- an object of type Task has the method execute().
- Compute then can execute a Task.

Implementing the remote objects

- A remote object must implement a remote interface.
- Additionally it must implement whatever is needed locally.
  - Declare the remote interfaces being implemented
  - Define the constructor for the remote object
  - Provide an implementation for each remote method in the remote interfaces
  - The server needs to create and to install the remote objects.

Implementing Remote Interfaces

Could be done in different ways

- by extending java.rmi.server.UnicastRemoteObject
- or by calling UnicastRemoteObject.exportObject(Remote Object)
- or other.

```java
public class ComputeEngine extends UnicastRemoteObject implements Compute
{
    public ComputeEngine() throws RemoteException
    {
        super(); //Important! Here the class becomes remote
    }

    public Object executeTask(Task t)
    {
        return t.execute();
    }
}
```

Pi Example

```java
package engine;
import java.rmi.*;
import java.rmi.server.*;
import compute.*;
public class ComputeEngine extends UnicastRemoteObject
implements Compute
{
    public ComputeEngine() throws RemoteException
    {
        super(); //Important! Here the class becomes remote
    }

    public Object executeTask(Task t)
    {
        return t.execute();
    }
}
```
public static void main(String[] args)
{
    if (System.getSecurityManager() == null)
    {
        System.setSecurityManager(new RMISecurityManager());
    }
    String name = "//host/Compute";
    try
    {
        Compute engine = new ComputeEngine();
        Naming.rebind(name, engine);
        System.out.println("ComputeEngine bound");
        catch (Exception e)
        {
            System.err.println("ComputeEngine exception: " +
                e.getMessage());
            e.printStackTrace();
        }
    }
    catch (Exception e)
    {
        System.err.println("ComputePi exception: " +
            e.getMessage());
        e.printStackTrace();
    }
}

Implementing the Clients

In our example the class Pi will be sent to be run at the server. Subsequently it must implement the Task interface.

Since it should not be called “remote” it does not have to be made “remote”.

The class ComputePi will contact the server, send the work, and then take care of the result.

```java
package client;
import java.rmi.*;
import java.math.*;
import compute.*;
public class ComputePi
{
    public static void main(String args[])
    {
        if (System.getSecurityManager() == null)
        {
            System.setSecurityManager(new RMISecurityManager());
        }
        try
        {
            String name = "//" + args[0] + "/Compute";
            Compute comp = (Compute) Naming.lookup(name);
            Pi task = new Pi(Integer.parseInt(args[1]));
            BigDecimal pi =
                (BigDecimal) (comp.executeTask(task));
            System.out.println(pi);
        }
        catch (Exception e)
        {
            System.err.println("ComputePi exception: " +
                e.getMessage());
            e.printStackTrace();
        }
    }

    The class Pi

    package client;
    import compute.*;
    import java.math.*;
    public class Pi implements Task
    {
        int digits;
        // ....
        public Pi(int digits)
        {
            this.digits = digits;
        }
        // ....
        public Object execute()
        {
            return computePi(digits);
        }
        public static BigDecimal computePi(int digits)
        {
            // ...
        }
    }
```
Pi example

```
interface Task extends java.lang.Serializable {
    Object execute();
}

interface Compute extends java.rmi.Remote {
    void executeTask(Task t);
}

class Pi extends java.rmi.server.UnicastRemoteObject {
    void execute() {
        Create()
        ComputeEngine
    }
}

class ComputeEngine {
    void executeTask() {
        Java.rmi.server.UnicastRemoteObject
    }
}
```