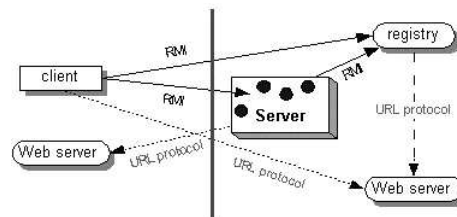


## A Typical RMI Application

- Client and Server run on different machines
- Remote Object(s) registered in rmiregistry by Server
- Remote Object(s) look'd up by Client
- When necessary, code transferred from web server to point of use
  - Both Client and Server can make code network accessible
- Operations on Remote Objects carried out by RMI

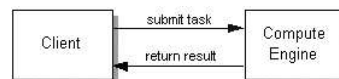


## Case Study

- This example taken directly from the Java RMI tutorial
  - <http://java.sun.com/docs/books/tutorial/rmi/>
- Editorial note:
  - Please do yourself a favor and work through the tutorial yourself
  - If you get the tutorial to work, you'll have no problems with project 5 or with the final exam

## Compute Server Application

- Goal
  - Execute object methods on a remote machine
  - Often because local resources aren't sufficient
- Real-life example:
  - Police car stops a motorist for speeding
  - Officer wants to check driver for warrants
  - Places request with a central computer (i.e., the database is not in the police car)



## Compute Interface

```
package compute;
```

```
import java.rmi.Remote;  
import java.rmi.RemoteException;
```

```
public interface Compute extends Remote {  
    Object executeTask(Task t) throws RemoteException;  
}
```

- Any class that implements Compute is a remote object
  - Its remote methods can be called from any JVM
  - Its implementation does not leave the JVM in which it was created
- executeTask() is a remote method
  - It must throw RemoteException

## Task Interface

```
package compute;

import java.io.Serializable;

public interface Task extends Serializable {
    Object execute();
}
```

- Task implementations implement Serializable, rather than Remote
  - Why?
- execute() method returns an Object
  - Client must downcast result
  - This method not required to throw RemoteException

## Implementing Compute Engine

- Our implementation of compute interface will called *ComputeEngine*
- In general, the implementer of a remote interface should at least
  1. Declare the remote interfaces being implemented
  2. Define the constructor for the remote object
  3. Implement each remote method in the remote interfaces

## Further Requirements for Servers

- The server needs to create and to install the remote objects.
  - The setup procedure often done in main method of the remote object
    - but can be done anywhere
- The setup procedure should
  1. Create and install a security manager
  2. Create one or more instances of a remote object
  3. Register at least one of the remote objects with the RMI registry

## Declare the Remote Interfaces

- The ComputeEngine class is declared as
  - `public class ComputeEngine extends UnicastRemoteObject implements Compute`
- This declaration states that
  - the class implements the Compute remote interface
  - extends the class `java.rmi.server.UnicastRemoteObject`
    - Much magic happens here!

## Define the Constructor

- The ComputeEngine class has a single, 0-argument constructor.  

```
public ComputeEngine() throws RemoteException {  
    super();  
}
```
- The no-argument constructor for Compute-Engine must declare RemoteException
  - because the UnicastRemoteObject constructor does
  - A RemoteException can occur during construction if the attempt to export the object fails
    - for example, if communication resources are unavailable or the appropriate stub class is not found

## Implement Each Remote Method

- The Compute interface contains a single remote method, executeTask, which is implemented as follows:  

```
public Object executeTask(Task t) {  
    return t.execute();  
}
```
- Client provides the ComputeEngine with a Task object,
  - which has an implementation of the task's execute method
- The ComputeEngine executes the Task and returns the result
- Client must downcast the result

## Implement the Setup Procedure

- Create and install a security manager
- Create one or more instances of a remote object
- Register at least one of the remote objects with the RMI registry

## Create and Install a Security Manager

- The security manager determines whether downloaded code has access to the local file system or can perform any other privileged operations.
- All programs using RMI must install a security manager, or RMI will not download classes (other than from the local class path) for objects received as parameters, return values, or exceptions in remote method calls

```
if (System.getSecurityManager() == null) {
    System.setSecurityManager(new RMISecurityManager());
}
```
- We will use a policy file that grants more permissions

## Create the Remote Object

- The main method creates an instance of the ComputeEngine
  - `Compute engine = new ComputeEngine();`
- Note that engine's type is Compute, not ComputeEngine
  - The interface is available to clients, not the implementation
  - At runtime, you'll pass the stub, not the actual implementation

## Make the Remote Object Accessible

- To invoke a remote object method caller must have a reference to it
  - Can get it from the program
    - e.g., as part of the return value of a method or as part of a data structure that contains such a reference, or
  - You can look it up in an RMI registry
- The RMI registry is a simple remote object name service that allows remote clients to get a reference to a remote object by name
- Start the registry
  - From the command line as a separate process, or
  - From within your Server program (See T1J)

## Add Remote Object to Registry

- The `java.rmi.Naming` interface is used as a front-end API for binding, or registering, and looking up remote objects in the registry
- The `ComputeEngine` class creates a name for the remote object  
`String name = "//host/Compute";`
- The `ComputeEngine` adds remote object to the registry  
`Naming.rebind(name, engine);`
- Note the following about the arguments to the call to `Naming.rebind`.
  - The first parameter is a URL-formatted `java.lang.String` representing the location and the name of the remote object
  - If no port number is given in name, defaults to 1099
- Note that for security reasons, an application can bind, unbind, or rebind remote object references only with a registry running on the same host
- Once the server has registered the remote object, the setup procedure exits

## Creating a Client Program

- Two separate classes make up the client in our example.
  - `ComputePi`
  - `Pi`
- `ComputePi` must obtain a reference to a `Compute` object, create a `Task` object, and then request that the task be executed
- `Pi` implements the `Task` interface, calculating `Pi` to some degree of precision

## ComputePi

- Begins by installing a security manager
- Constructs a name used to look up a Compute remote object.
- Uses Naming.lookup() to look up the remote object by name in the remote host's registry
- Creates a new Pi object
- Invokes executeTask() on the Compute remote object
- executeTask() returns an object of type java.math.BigDecimal, so the program downcast casts the result and stores it
- Program prints out the result

## Pi

- Calculates Pi

## Compiling

- Application has 4 directory trees
- Server
  - Application directory – (server code written and compiled here)
  - Web accessible location – (client downloads server code from here)
- Client
  - Application directory (client code written and compiled here)
  - Web accessible location -- (server downloads client code from here)
- Editorial note:
  - You have to put all the code in the right places each time you make changes
    - *So use a makefile!*
  - For testing purposes keep client and server code in separate directory trees / separate machines
    - *Otherwise you may not know if things are really working*

## Compiling

- Compile interface classes, build a jar file
  - Move jar file to developer-accessible locations
  - Move jar file to web-accessible locations and unpack it
  - Everyone shares these files – don't change them
- Build Server classes
  - (add classpath info to the following command lines)
  - `cd ServerDevDir`
  - `javac engine/ComputeEngine.java`
  - `rmic -d . engine.ComputeEngine`
  - `mkdir ServerWebDir/engine`
  - `cp engine/ComputeEngine_*.class ServerWebDir/engine`
- Stubs and Interfaces are now web-accessible

## Compiling

- Build the Client classes
  - cd ClientDevDir
  - javac client/ComputePi.java
  - javac -d ClientWebDir client/Pi.java
- Serializable class and Interfaces are now web-accessible

## Running Application

- Copy policy file to home directory
  - On Unix I put the file in ~/.java.policy
- Start the RMI registry
  - mkdir EmptySubDir
  - cd EmptySubDir
  - unsetenv CLASSPATH
  - rmiregistry portNum &
- Start the server

```
java -classpath ServerDevDir/ \
-Djava.rmi.server.codebase=http://webHost/WebServerDir/ \
-Djava.rmi.server.hostname=ServerName \
-Djava.security.policy=java.policy \
engine.ComputeEngine
```

## Running Application

- Start the client (on another machine)  
java -classpath ClientDevDir/ \  
-Djava.rmi.server.codebase=http://ClientWebServer/ClientWebDir/ \  
-Djava.security.policy=java.policy \  
client.ComputePi serverName 20
- Should produce
  - 3.14159265358979323846