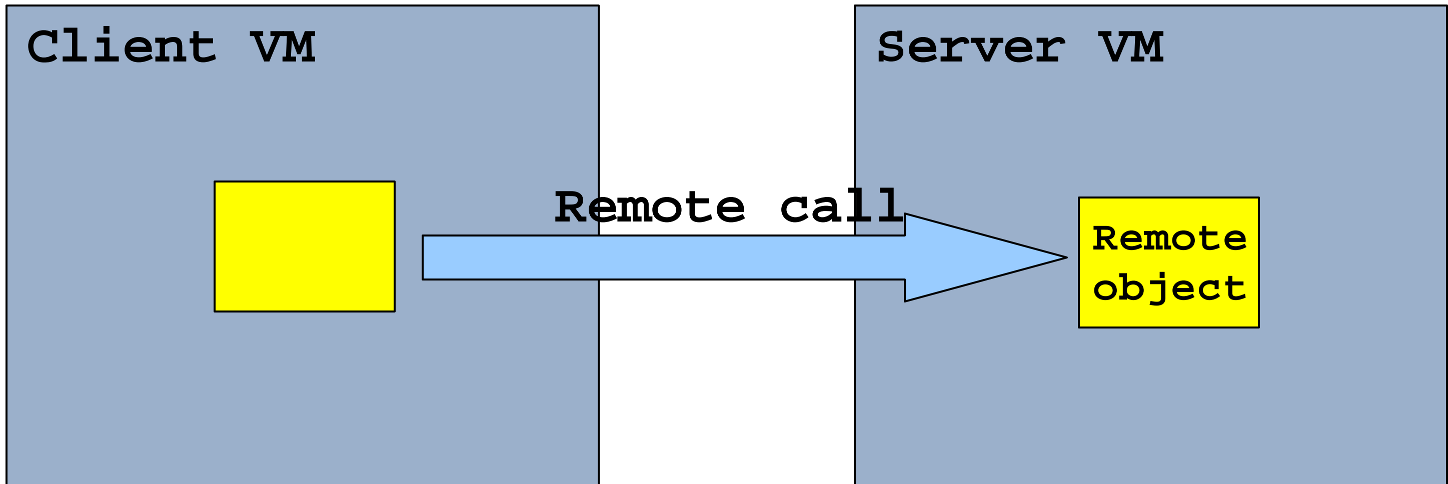
Remote Method Invocation

RMI

- Mechanism that allows one to invoke a method on an object that exists in another address space (on the same of different machine)
- RMI is basically an object-oriented RPC mechanism
- RMI supports *distributed object applications*



RMI Components

- Server: creates remote objects, makes references to them accessible, and waits for clients to invoke methods on these remote objects
- Client: gets a remote reference to one or more remote objects in the server and then invokes methods on them
- Object Registry is a name server that relates objects with names. Objects are *registered* with the Object Registry. Once an object has been registered, one can use the Object Registry to obtain access to a remote object using the name of the object

Native Methods

- Java Native Interface (JNI) supports the ability to call native methods written in languages such as C/C++
- The native keyword is used to identify methods that have implementation defined in such shared object files:

```
public void native sayHello();
```

- JNI provides a means of providing cross-platform native code, but only if appropriate libraries are generated for each machine (Code is portable, but not WORA--write once, run anywhere.)

RMI vs. Corba

- CORBA is a language-independent standard (therefore supports multiple languages)
- CORBA includes many other mechanisms in its standard (such as a standard for TP monitors) none of which are part of Java RMI
- CORBA separates interface definitions from implementation
- RMI allows optimization of protocols for communication

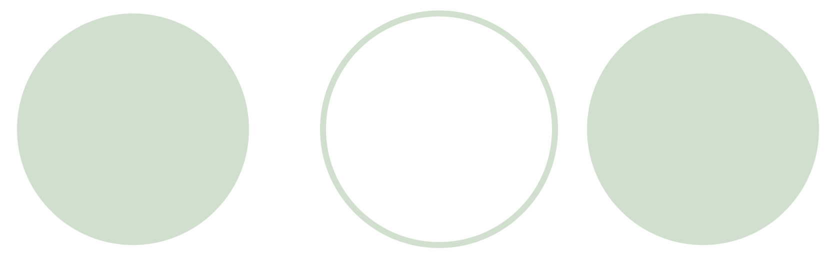
RMI over IIOP

- With RMI-IIOP, programmers can create applications in RMI that include CORBA connections
- Developed by IBM and Sun, with the cooperation of the Object Management Group (OMG)
- CORBA RMI is (relatively) easier to implement than CORBA

Remote Interfaces, Objects, and Methods

- An object becomes remote by implementing a *remote interface:*
  - A remote interface extends interface java.rmi.Remote
  - Each method of the interface declares java.rmi.RemoteException in its throws clause
  - RMI passes a remote *stub* for a remote object. The stub is basically a remote reference
- The client invokes a method on the local stub, which is responsible for carrying out the method call on the remote object

RMI: Tools provided

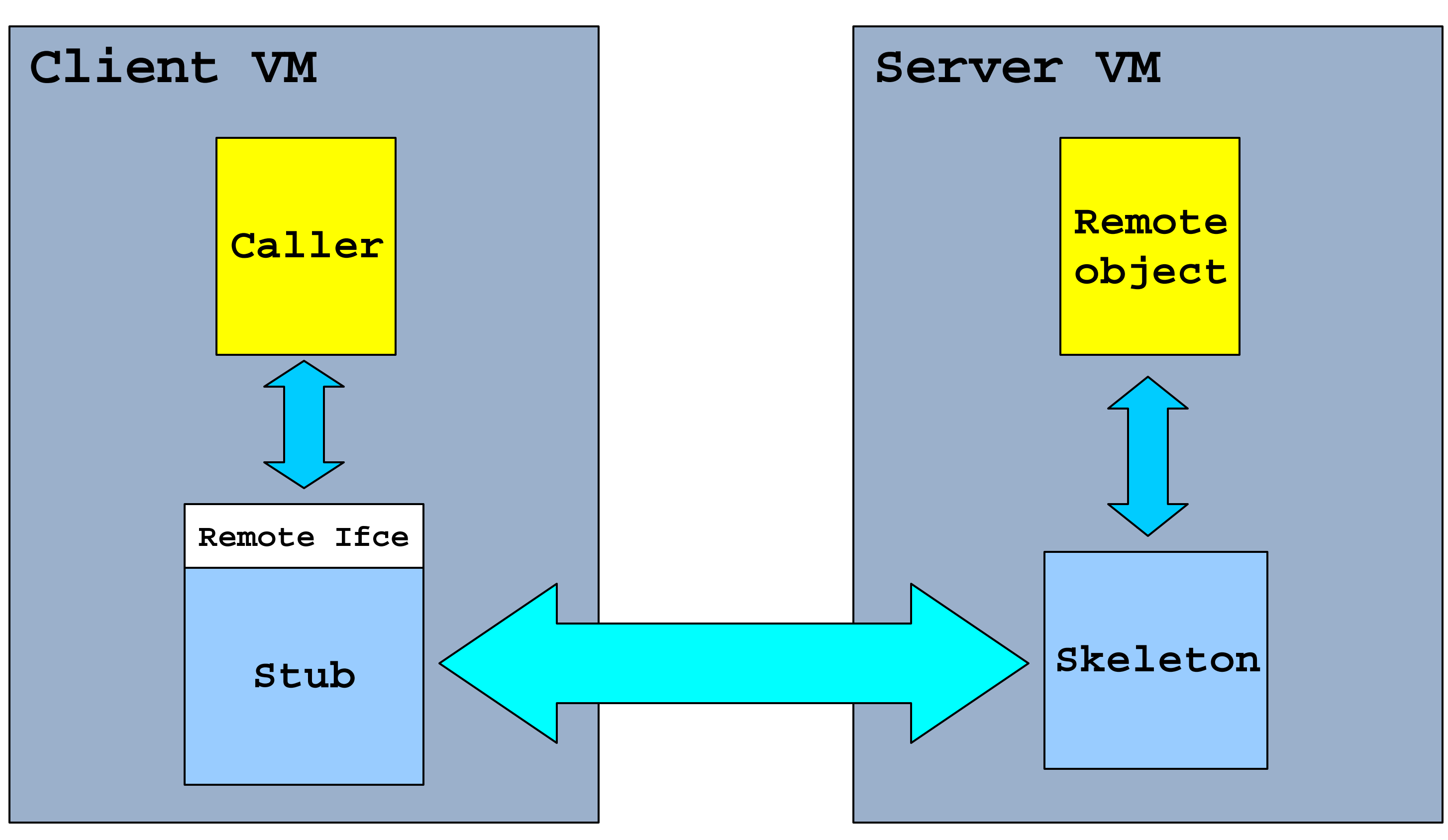


- RMIC
  - Java RMI Stub compiler
- RMIRegistry
  - Java RMI remote object registry
- RMID
  - Java RMI activation system daemon

Stub and Skeleton

- RMI Stub
  - Resides in caller’s local address space
  - Represents remote object to caller
    - Implementation of Remote interface
    - Caller invokes methods of RMI Stub locally
  - Connects to the remote object
  - Sends arguments and receive results from remote object
    - Performs marshaling and unmarshaling
- RMI Skeleton
  - Resides in server’s address space
  - Receives arguments from caller (RMI Client's Stub) and send results back to caller
  - In JDK 1.3, RMI Skeleton gets created automatically via reflection

RMI Communication Model



RMI Flow Control

- Client
  - Invokes a method of a remote object
- Stub of remote object
  - Intercepts method call
  - Marshals the arguments
  - Make calls to remote objects
- Remote Object
  - Receives calls via Skeleton
  - Unmarshalls arguments
  - Executes the call locally
  - Marshalls the result
  - Sends the result to the client

Creating Distributed Applications Using RMI

To develop a distributed application using RMI, follow these general steps:

1. Design and implement the components of your distributed application.
2. Compile sources and generate stubs
3. Make classes network accessible
4. Start the application

1. Design and Implement the Application Components

- *Define remote interfaces*:

  A remote interface specifies the methods that can be invoked remotely by a client.

- *Implement remote objects*:

  Remote objects must implement one or more remote interfaces. The remote object class may include implementations of other interfaces (either local or remote) and other methods (which are available only locally).

- *Implement the clients*:

  Clients can be implemented after the remote interfaces are defined

2. Compile Sources and Generate Stubs

A two-step process:

- Use **`javac`** to compile source files: implementation of remote interfaces, server classes, and client classes.
- Use the **`rmic`** compiler to create stubs for the remote objects.

3. Make Classes Network Accessible

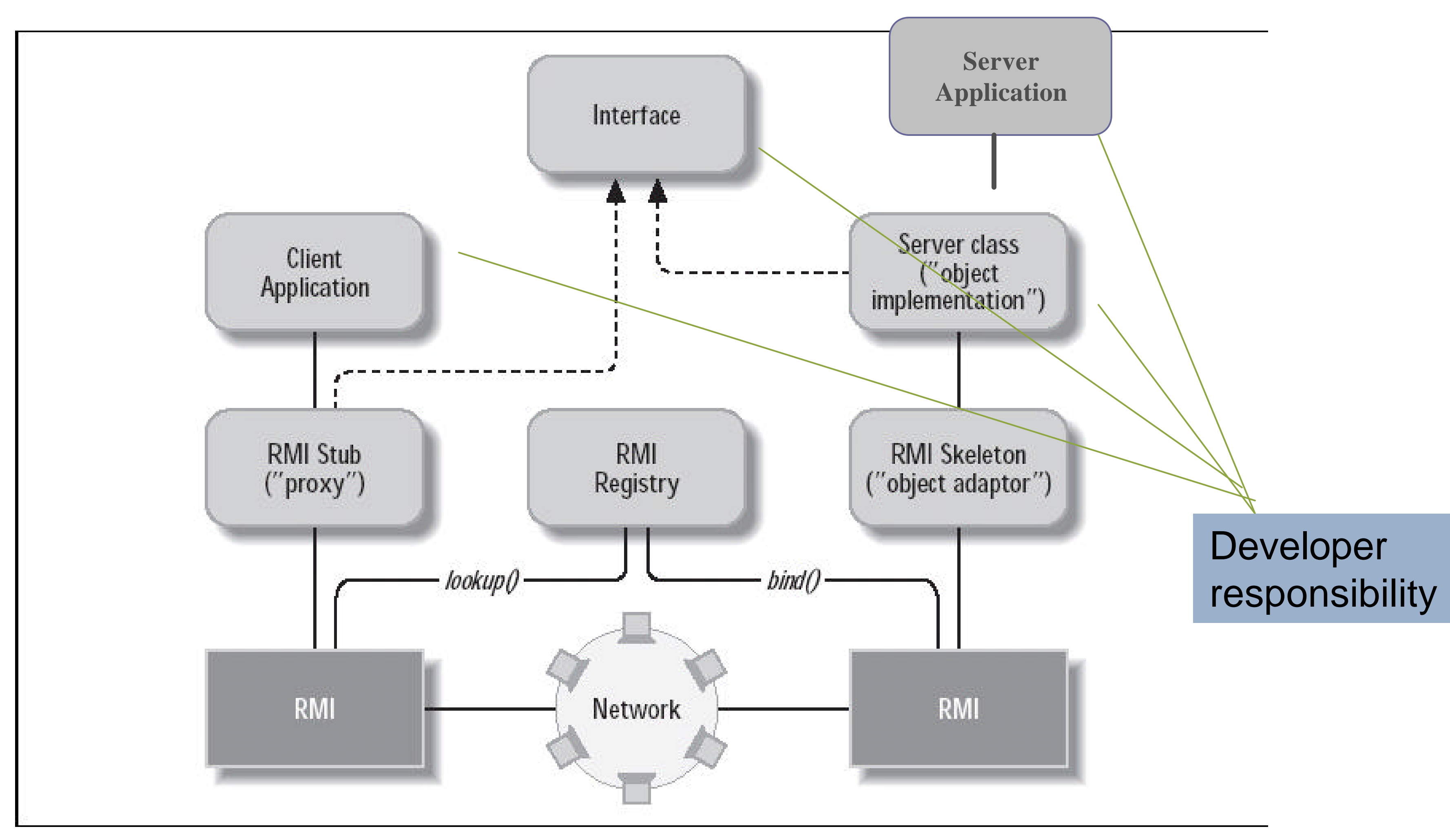
Make accessible to clients (e.g. via a web or ftp server):

- class files associated with the remote interfaces
- Stubs
- other useful classes

4. Start the Application

- Run the RMI remote object registry
- Run the server
- Run the client

How to make a distributed Java object



Example 1

- *Hello World* example:
  - Server side:
    - HelloInterface.java
    - HelloImpl.java
  - Client side:
    - HelloClient.java

- Run rmiregistry from server directory

HelloInterface.java

```
import java.rmi.*;

public interface HelloInterface extends Remote {

  public String say() throws RemoteException;

}
```

HelloImpl.java

```
import java.rmi.*;
import java.rmi.server.*;

public class HelloImpl extends UnicastRemoteObject implements
   HelloInterface {
   private String message;

   public HelloImpl(String m) throws RemoteException {
        message = m;
   }
   public String say() throws RemoteException {
        return message;
   }
   public static void main (String[] argv) {
       try {
        Naming.rebind ("//localhost/Hello", new
                                     HelloImpl("Hello, world!"));
       }   catch (Exception e) { System.out.println(e); }
   }
}
```

HelloClient.java

```
import java.rmi.*;

public class HelloClient {

   public static void main (String[] argv)
   {
     try {
       HelloInterface h =(HelloInterface)
               Naming.lookup("//localhost/Hello");
       System.out.println (h.say());
     }
     catch (Exception e) {
   System.out.println(e);
     }
   }
}
```

Compile and Run

- Server Side:
  - `javac HelloImpl.java HelloInterface.java`
  - `rmic HelloImpl`
  - `rmiregistry`

    Note: make sure application classes are referenced in `CLASSPATH`
  - `java HelloImpl`
- Client Side:
  - Grab a copy of `HelloInterface.class` and `HelloImpl_Stub.class`
  - `javac HelloClient.java`
  - `java HelloClient`

Classes in Java RMI

1. A *Remote* class is one whose instances can be used remotely. An object of such a class can be referenced in two different ways:
   - Within the address space where the object was constructed, the object is an ordinary object which can be used like any other object
   - Within other address spaces, the object can be referenced using an *object handle*. While there are limitations on how one can use an object handle compared to an object, for the most part one can use object handles in the same way as an ordinary object

For simplicity, an instance of a Remote class will be called a *remote*

Classes in Java RMI (cont'd)

2. A *Serializable* class is one whose instances can be copied from one address space to another
   - An instance of a Serializable class will be called a *serializable object (*is one that can be marshaled)
   - If a serializable object is passed as a parameter (or return value) of a remote method invocation, then the value of the object will be copied from one address space to the other
   - If a remote object is passed as a parameter (or return value), then the object handle will be copied from one address space to the other.

Marshaling and Unmarshaling

- Marshaling: encoding objects for transmission on the wire
- Unmarshaling: decoding from wire and placing object in address space
- Java uses **`serialization`** to perform marshaling and unmarshaling

Example2: Serializable Class

- Example 2 is example 1 but returning an object instead of a string
- The object will be an instance of class myMessage

myMessage.java/HelloInterface.java

```
import java.io.Serializable;

public class myMessage implements Serializable {
   String from, to, body;
   public myMessage(String f, String t, String b) {
        from = f;  to = t;  body =b;
   }
   public String getFrom() { return from; }
   public String getTo()   { return to; }
   public String getBody() { return body; }
}
```

```
import java.rmi.*;
public interface HelloInterface extends Remote {
        public myMessage say() throws RemoteException;
}
```

HelloImpl.java

```
import java.rmi.*;
import java.rmi.server.*;

public class HelloImpl extends UnicastRemoteObject implements
   HelloInterface {
   private myMessage message;

   public HelloImpl(String f, String t, String b) throws
   RemoteException {
        message = new myMessage(f, t, b);
   }
   public myMessage say() throws RemoteException {
        return message;
   }
   public static void main (String[] argv) {
       try {
        Naming.rebind ("//localhost/Hello",
          new HelloImpl("me", "you", "Hello, world!"));
       }
       catch (Exception e) { System.out.println(e); }
}
```

HelloClient.java

```
import java.rmi.*;

public class HelloClient {

   public static void main (String[] argv)
   {
     try {
       HelloInterface h =(HelloInterface)
        Naming.lookup("//localhost/Hello");
       myMessage mes = h.say();
       System.out.println ("From: " + mes.getFrom() +
          " To: " + mes.getTo() + " Body: " + mes.getBody()
       );
     }
     catch (Exception e) { System.out.println(e);  }
   }
}
```

Compile and Run

- Server Side:
  - `javac myMessage.java`
  - `javac HelloImpl.java HelloInterface.java`
  - `rmic HelloImpl`
  - `rmiregistry`

    Note: make sure application classes are referenced in **`CLASSPATH`**
  - `java HelloImpl`
- Client Side:
  - Grab a copy of **`HelloInterface.class, HelloImpl_Stub.class, and myMessage.class`**
  - `javac HelloClient.java`
  - `java HelloClient`

RMI & JVM Security

- Define a RMISecurityManager in the server implementation:

```
if( System.getSecurityManager() == null ) {
 System.setSecurityManager(new RMISecurityManager());
}
```

- Define a policy for the server:

```
grant {
 permission java.net.SocketPermission "127.0.0.1:1024-65535",
  "connect,accept";
};
```

- Run Server:

```
java -Djava.security.policy=policy.txt HelloImpl
```

- See example 3

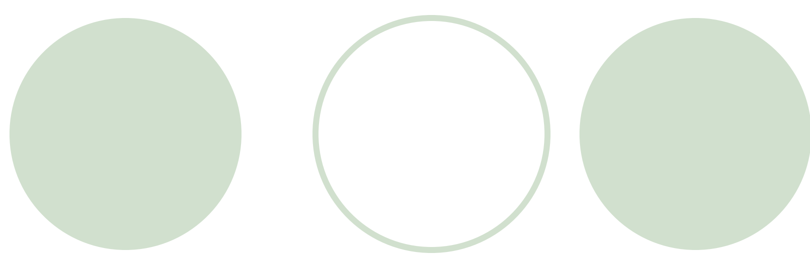
Advanced Techniques

1. Activatable Services
2. Secure connections via SSL
3. Callbacks
4. Dynamic Class Loading

Activatable RMI Services

- Motivation: remote objects may be too numerous to all exist in a server's memory at once
- Solution: activate objects as needed. A remote object activation protocol is then required.
- Rmid is the remote object activation daemon that handles activation

Example



- Server:
  - Hello.java (interface, same as before)
  - HelloImpl.java
  - ActivationSetup.java
  - rmid.policy
- Client:
  - HelloClient.java
- See example 4

HelloImpl.java

```
import java.rmi.*;
import java.rmi.activation.*;

public class HelloImpl extends Activatable implements Hello {

   public HelloImpl( ActivationID id, MarshalledObject data )
   throws RemoteException {
        super( id, 0 );
   }

   public String sayHello() {
        return "Hello, World";
   }
}
```

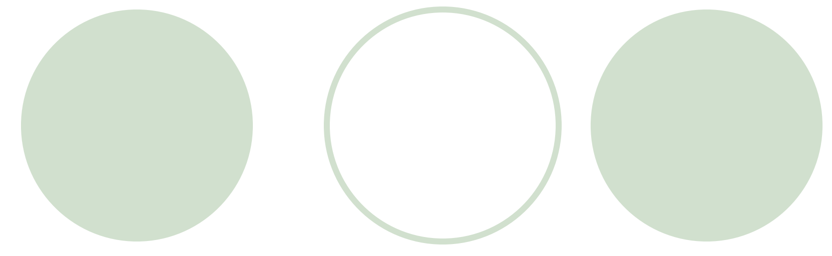
ActivationSetup.java

```
import java.rmi.*;
import java.rmi.activation.*;

public class ActivationSetup {

   public static void main( String[] args ) throws Exception {

   ActivationGroupID agi =
   ActivationGroup.getSystem().registerGroup( new
   ActivationGroupDesc( null, null ) );

   ActivationDesc desc = new ActivationDesc( agi, "HelloImpl",
   null, null );
   Hello helloserver = (Hello) Activatable.register( desc );
   Naming.rebind( "rmi://localhost/HelloServer", helloserver
   );
   System.exit(0);
   }
}
```

HelloClient.java

```
import java.rmi.*;

public class HelloClient {

   public static void main( String[] args ) throws Exception {

   Hello server = ( Hello ) Naming.lookup(
        "rmi://localhost/HelloServer" );
   System.out.println( server.sayHello() );
   }
}
```

Run the Example



- **Server side:**
  - `javac *java`
  - `rmic HelloImpl`
  - `rmid -C-Djava.rmi.server.logCalls=true -C-Djava.security.policy=rmid.policy`
  - `rmiregistry  (from server dir)`
  - `java ActivationSetup`
- **Client side:**
  - `java HelloClient`

RMI with SSL

- Remote object implementation:
  - Install RMISecurityManager
  - Call constructor:

```
super(0, new RMISSLClientSocketFactory(),
        new RMISSLServerSocketFactory());
```

- See example 5

Example 5

- Server:
  - Hello.java (interface)
  - HelloImpl.java
  - RMISSLClientSocketFactory.java and RMISSLServerSocketFactory.java
  - Policy file (all.policy)
- Client
  - HelloClient.java

RMISSLServerSocketFactory()

```
public class RMISSLServerSocketFactory
        implements RMIServerSocketFactory, Serializable {

    public ServerSocket createServerSocket(int port)
        throws IOException
        {
         SSLServerSocketFactory ssf = null;
         try {
           SSLContext ctx;
            KeyManagerFactory kmf;
            KeyStore ks;
            char[] passphrase = "storepass".toCharArray();
            ctx = SSLContext.getInstance("TLS");
            kmf = KeyManagerFactory.getInstance("SunX509");
            ks = KeyStore.getInstance("JKS");
            ks.load(new FileInputStream(“server.ks"), passphrase);
            kmf.init(ks, passphrase);
            ctx.init(kmf.getKeyManagers(), null, null);
            ssf = ctx.getServerSocketFactory();
         } catch (Exception e) {}
        return ssf.createServerSocket(port);
   }}
```

RMISSLClientSocketFactory()

```
public class RMISSLClientSocketFactory
        implements RMIClientSocketFactory, Serializable {

    public Socket createSocket(String host, int port)
        throws IOException
        {
            SSLSocketFactory factory =
             (SSLSocketFactory)SSLSocketFactory.getDefault();
            SSLSocket socket =
             (SSLSocket)factory.createSocket(host, port);
            return socket;
        }
}
```

Compile & Run

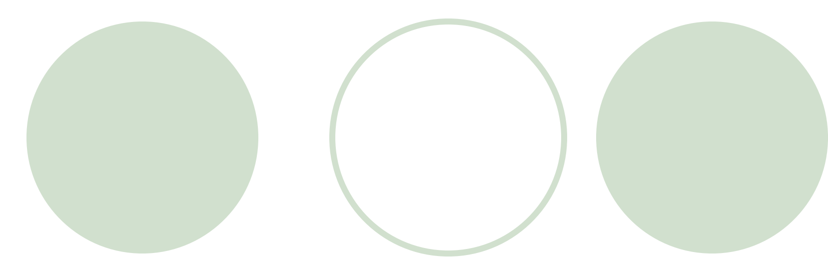
- Server:
    - `javac RMISSLClientSocketFactory.java`
    - `javac RMISSLServerSocketFactory.java`
    - `javac Hello.java`
    - `javac HelloImpl.java`
    - `rmic HelloImpl`
  - Create keys and export certificate:
    - `keytool -genkey -alias rlent -keystore server.ks -storepass storepass`
    - `keytool -export -keystore server.ks -storepass storepass -alias rlent -file server.cer`
  - Make available to client:
    - `Hello.class, HelloImpl_Stub.class, cp RMISSLClientSocketFactory.class, RMISSLServerSocketFactory.class`
    - `server.cer`

- Run rmiregistry (from server dir)

Compile & Run (cont'd)

- Client:
  - **javac HelloClient.java**
  - **Import certificate:**
    - **keytool -import -file server.cer -alias rlent -keystore client.ks -storepass storepass**
  - **Run client:**
    - **java -Djavax.net.ssl.trustStore=client.ks -Djavax.net.ssl.trustStorePassword=storepass HelloClient**

Callbacks



- Server:
  - HelloInterface.java
  - HelloImpl.java
- Client:
  - HelloClientInterface.java
  - HelloClient.java
- Note: HelloClientInterface.class is required to compile server
- See example 4

HelloInterface.java

```
import java.rmi.*;

public interface HelloInterface extends Remote {

  public String say() throws RemoteException;
  public void Register(HelloClientInterface h) throws
   RemoteException;

}
```

HelloImpl.java

```
…

public class HelloImpl extends UnicastRemoteObject implements
   HelloInterface {
   private String message;
   private HelloClientInterface client;

   public HelloImpl(String m) throws RemoteException {
        message = m;
   }
   public String say() throws RemoteException {
        return message;
   }
   public void Register(HelloClientInterface c) {
      client = c;
      try {
        System.out.println(client.answer());
      } catch (Exception e) {}
   }
…
```

HelloClientInterface.java

```
import java.rmi.*;


public interface HelloClientInterface extends Remote {

  public String answer() throws RemoteException;

}
```

HelloClient.java

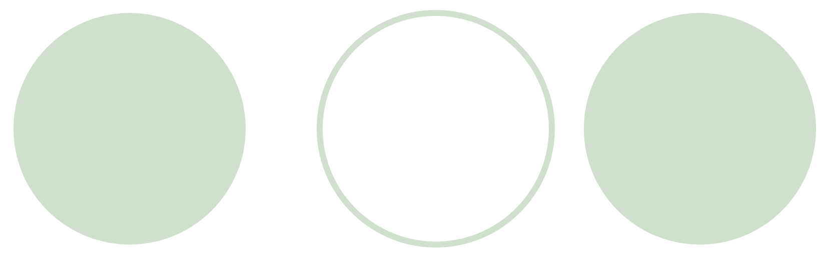
```
public class HelloClient extends UnicastRemoteObject implements
   HelloClientInterface {

    HelloInterface h;
    public HelloClient() throws RemoteException {
     try{
        h =(HelloInterface)    Naming.lookup("//localhost/Hello");
        h.Register(this);
        System.out.println (h.say());
      } catch (Exception e) {}
    }

    public static void main (String[] argv)  {
      try { new HelloClient(); } catch (Exception e) {}
    }

    public String answer() {return "Hello, there";}
}
```

Dynamic Class Loading

- **Dynamic stub loading is used where the client does not have local copies of the remote object**
- **Class bytecodes (class files) get downloaded during runtime (the stubs can even be generated on the fly)**
- **Lookup:**
  1. **Local resolution is attempted via CLASSPATH (no security manager required)**
  2. **Server tells client the stub's URL (requires security manager)**

     `java.rmi.server.codebase`

Example



- Server
  - javac HelloImpl.java HelloInterface.java
  - rmic HelloImpl
  - javac HelloClient.java
- Move HelloInterface.class and HelloImpl_Stub.class to a web server
- Move HelloClient.class to client

Running the Example: rmiregistry

- Server classes must not be in classpath
- Policy must be defined
- rmiregistry -J"-

  Djava.security.policy=rmireg.policy"

Define policy and codebase

- Server:

```
java -Djava.security.policy=all.policy -
Djava.rmi.server.codebase=http://www.cs.ucf.edu/courses
/cop4610/fall2003/LectureSet7/classes/ HelloImpl
```

- Client:

```
java -Djava.rmi.server.useCodebaseOnly=true -
Djava.rmi.server.codebase=http://www.cs.ucf.edu/courses
/cop4610/fall2003/LectureSet7/classes/ -
Djava.security.policy=client.policy HelloClient
```

- Note that URL must end with a trailing ‘/’