Remote Method Invocation (RMI)

- Remote Method Invocation (RMI) allows us to get a reference to an object on a remote host and use it as if it were on our virtual machine. We can invoke methods on the remote objects, passing real objects as arguments and getting real objects as returned values. (Similar to Remote Procedure Call (RPC) in C).

Remote Method Invocation (RMI)

- Remote Method Invocation (RMI) allows us to get a reference to an object on a remote host and use it as if it were on our virtual machine. We can invoke methods on the remote objects, passing real objects as arguments and getting real objects as returned values. (Similar to Remote Procedure Call (RPC) in C).
- RMI uses object serialization, dynamic class loading and security manager to transport Java classes safely. Thus we can ship both code and data around the network.

Remote Method Invocation (RMI)

- Remote Method Invocation (RMI) allows us to get a reference to an object on a remote host and use it as if it were on our virtual machine. We can invoke methods on the remote objects, passing real objects as arguments and getting real objects as returned values. (Similar to Remote Procedure Call (RPC) in C).
- RMI uses object serialization, dynamic class loading and security manager to transport Java classes safely. Thus we can ship both code and data around the network.
- *Stubs and Skeletons.* Stub is the local code that serves as a proxy for a remote object. The skeleton is another proxy that lives on the same host as the real object. The skeleton receives remote method invocations from the stub and passes them on to the object.

Remote Interfaces and Implementations

- A remote object implements a special remote interface that specifies which of the object's methods can be invoked remotely. The remote interface must extend the java.rmi.Remote interface. Both the remote object and the stub implement the remote interface.

```
public interface MyRemoteObject extends java.rmi.Remote {
    public Widget doSomething() throws  java.rmi.RemoteException;
    public Widget doSomethingElse() throws java.rmi.RemoteException;
}
```

Remote Interfaces and Implementations

- A remote object implements a special remote interface that specifies which of the object's methods can be invoked remotely. The remote interface must extend the java.rmi.Remote interface. Both the remote object and the stub implement the remote interface.

```
public interface MyRemoteObject extends java.rmi.Remote {
    public Widget doSomething() throws  java.rmi.RemoteException;
    public Widget doSomethingElse() throws java.rmi.RemoteException;
}
```

- The actual implementation must extend java.rmi.server.UnicastRemoteObject. It must also provide a constructor. This is the RMI equivalent of the Object class.

```
public class RemoteObjectImpl implements MyRemoteObject
   extends java.rmi.server.UnicastRemoteObject {
    public RemoteObjectImpl() throws java.rmi.RemoteException {...}
    public Widget doSomething() throws  java.rmi.RemoteException {...}
    public Widget doSomethingElse() throws java.rmi.RemoteException {.
    // other non-public methods
}
```

RMI Hello Server

```
--Hello.java--
public interface Hello extends java.rmi.Remote {
        String sayHello() throws java.rmi.RemoteException;
}
--HelloServer.java---
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.UnicastRemoteObject;

public class HelloServer extends UnicastRemoteObject implements Hello
{
    private String name;
    private static int registryPort = 1099;

    public HelloServer(String s) throws RemoteException {
        super();
        name = s;
    }
    public String sayHello() throws RemoteException {
        return  "Hello World!";
    }
    public static void main(String args[])
    {
        if (args.length > 0) {
            registryPort = Integer.parseInt(args[0]);
        }
        // Create and install a security manager
        System.setSecurityManager(new RMISecurityManager());
        try {
            Registry registry = LocateRegistry.getRegistry(registryPort);
            HelloServer obj = new HelloServer("//HelloServer");
            registry.rebind("HelloServer", obj);
            System.out.println("HelloServer bound in registry");
        } catch (Exception e) {
            System.out.println("HelloServer err: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

RMI Hello Client

```java
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class HelloClient {
    private HelloClient() {}

    public static void main(String[] args) {
        if (args.length < 1) {
            System.err.println("Usage: java HelloClient <host> [<registry-port>]");
            System.exit(1);
        }
        String host = null;
        int registryPort = 1099;
        if (args.length == 1) {
            host = args[0];
        } else  {
            host = args[0];
            registryPort = Integer.parseInt(args[1]);
        }

        try {
            Registry registry = LocateRegistry.getRegistry(host, registryPort);
            Hello stub = (Hello) registry.lookup("HelloServer");
            String response = stub.sayHello();
            System.out.println("response: " + response);
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

Running the RMI Hello World Example

This example is in the folder rmi-rpc/rmi/ex1-HelloServer. First we need to start up the rmiregistry. It runs on port 1099 by default. Choose a different port if you want to run your own copy.

```
export CLASSPATH=`pwd`:$CLASSPATH
rmiregistry [registryPort] &
```

Then we start up the server as follows:

```
java -Djava.security.policy=mysecurity.policy hello.server.HelloServer &
```

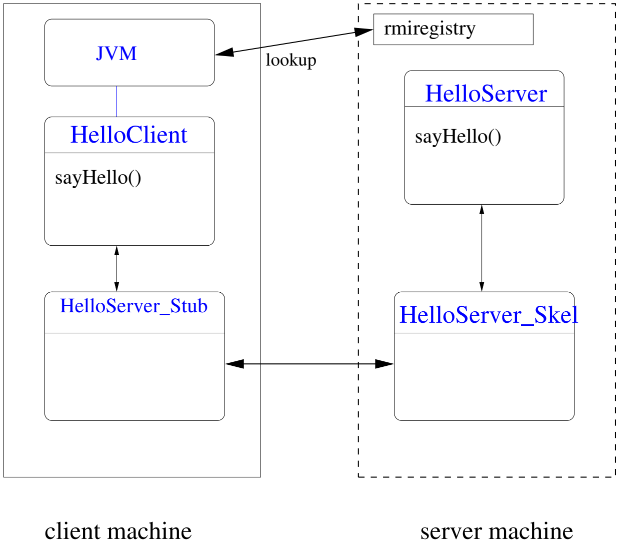
Here mysecurity.policy is the security policy that is required. Now run the client as follows:

```
java hello.client.HelloClient hostname [registryPort]
```

Once you are done, kill the server and the rmiregistry as shown below.

```
killall -9 rmiregistry
fg <Ctrl-C>
```

RMI Hello World example



RMI Hello World: Alternate Examples

- **rmi-rpc/rmi/ex1-HelloServer-without-rmic**: Shows how we can build the RMI example code without using the `rmic` RMI compiler. The java VM just runs it on the fly for us to generate the stubs.

RMI Hello World: Alternate Examples

- **rmi-rpc/rmi/ex1-HelloServer-without-rmic**: Shows how we can build the RMI example code without using the `rmic` RMI compiler. The java VM just runs it on the fly for us to generate the stubs.
- **rmi-rpc/rmi/ex1-HelloServer-with-packages**: Shows the setup for the example when server/client code is in packages.

Example 2: RMI Square Server

This example is in the folder `rmi-rpc/rmi/ex2-SquareServer`

```
--Square---
public interface Square extends java.rmi.Remote {
        long square(long arg) throws java.rmi.RemoteException;
}
```

This example runs *n* calls to a remote square method, so that we can time the responsiveness of remote calls.

Example 3: Client Callback

This example is in the folder `rmi-rpc/rmi/ex3-Client-Callback`

```
--Request--
// Could hold basic stuff like authentication, time stamps, etc.
public class Request implements java.io.Serializable { }

--WorkRequest.java--
public class WorkRequest extends Request {
        public Object execute() { return null; }
}

--WorkListener.java--
public interface WorkListener extends Remote {
        public void workCompleted( WorkRequest request, Object result )
                        throws RemoteException;
}

--StringEnumeration.java--
import java.rmi.*;
public interface StringEnumeration extends Remote {
        public boolean hasMoreItems() throws RemoteException;
        public String nextItem() throws RemoteException;
}

--Server.java--
import java.util.*;
public interface Server extends java.rmi.Remote {
    Date getDate() throws java.rmi.RemoteException;
    Object execute( WorkRequest work ) throws java.rmi.RemoteException;
    StringEnumeration getList() throws java.rmi.RemoteException;
    void asyncExecute( WorkRequest work, WorkListener listener )
               throws java.rmi .RemoteException;
}
```

Example 3: Server

```
import java.rmi.*;
import java.util.*;
public class MyServer
  extends java.rmi.server.UnicastRemoteObject implements Server {
    public MyServer() throws RemoteException { }
    // Implement the Server interface
    public Date getDate() throws RemoteException {
                return new Date();
        }
    public Object execute( WorkRequest work ) throws RemoteException {
                return work.execute();
        }
    public StringEnumeration getList() throws RemoteException {
                return new StringEnumerator(
                        new String [] { "Foo", "Bar", "Gee" } );
        }
    public void asyncExecute( WorkRequest request , WorkListener listener )
                throws java.rmi.RemoteException {

                Object result = request.execute();
                System.out.println("async req");
                listener.workCompleted( request, result );
                System.out.println("async complete");
        }
    public static void main(String args[]) {
                System.setSecurityManager(new RMISecurityManager());
                try {
                        Server server = new MyServer();
                        Naming.rebind("NiftyServer", server);
                        System.out.println("bound");
                } catch (java.io.IOException e) {
                        System.out.println("// Problem registering server");
                        System.out.println(e);
                }
    }
}
```

Example 3: RMI Client

```
import java.rmi.*;
public class MyClient
 extends java.rmi.server.UnicastRemoteObject implements WorkListener {
  public static void main(String [] args) throws RemoteException {
      System.setSecurityManager(new RMISecurityManager());
      new MyClient( args[0] );
  }
  public MyClient(String host) throws RemoteException {
     try {
         Server server = (Server)Naming.lookup("rmi://"+host+"/NiftyServer");

         System.out.println( server.getDate() );
         System.out.println( server.execute( new MyCalculation(2)));
         StringEnumeration se = server.getList();
         while ( se.hasMoreItems() )
             System.out.println( se.nextItem() );
             server.asyncExecute( new MyCalculation(100), this );
      } catch (java.io.IOException e) {
         // I/O Error or bad URL
         System.out.println(e);
      } catch (NotBoundException e) {
         // NiftyServer isn't registered
         System.out.println(e);
      }
  }
  public void workCompleted( WorkRequest request, Object result)
     throws RemoteException
  {
     System.out.println("Async work result = " + result);
     System.out.flush();
  }
}
```

Running the RMI Client-Callback Example

This example is in the folder `rmi-rpc/rmi/ex3-Client-Callback`. Note that the client needs a copy of `MyServer_Stub.class` whereas the server needs a copy of `MyCalculation.class` and `MyClient_Stub.class`.

```
rmiregistry [registryPort] &
```

Then we start up the server as follows:

```
java -Djava.security.policy=mysecurity.policy HelloServer &

or if running over actual internet use

java -Djava.rmi.server.codebase="http://onyx.boisestate.edu/~amit/rmi/ex3/"
      -Djava.security.policy=mysecurity.policy  MyServer
```

Here `mysecurity.policy` is the security policy that is required. Now run the client as follows:

```
java -Djava.security.policy=mysecurity.policy MyClient hostname [registryPort]
```

Note that since the server calls back the client via RMI, the client also needs to have a security policy. Once you are done, kill the server and the rmiregistry.

Creating a Asynchronous Server/Client

- To convert the Example 3 into a true asynchronous server, we would need a spawn off a thread for each asynchronous request that would execute the request and then call the client back with the result.
- On the client side, we need to keep track of number of asynchronous requests out to the server so the client doesn't quit until all the results have come back. Since the client is also potentially multi-threaded, we will need to use synchronization.
- See example `rmi-rpc/rmi/ex4-Asynchronous-Server`.

Example 4: Asynchronous Server

```
public class MyServer
    extends java.rmi.server.UnicastRemoteObject implements Server
{
...
public void asyncExecute( WorkRequest request , WorkListener listener )
    throws java.rmi.RemoteException
{
    new AsyncExecuteThread(request, listener).start();
    System.out.println("Started thread to execute request.");
}
...
}

class AsyncExecuteThread extends Thread
{
    WorkRequest request;
    WorkListener listener;

    public AsyncExecuteThread(WorkRequest request, WorkListener listener)
    {
        this.request = request;
        this.listener = listener;
    }

    public void run()
    {
        try {
            Object result = request.execute();
            System.out.println("async req");
            listener.workCompleted( request, result );
            System.out.println("async complete");
        } catch (RemoteException e) {
            System.out.println("AsyncExecuteThread:"+e);
        }
    }
}
```

Example 4: Asynchronous Client

```
public class MyClient extends java.rmi.server.UnicastRemoteObject
                      implements WorkListener
{
    private Semaphore sem;
    private static final long serialVersionUID = -6314695118464643327L;

    public static void main(String [] args) throws RemoteException
    {
        System.setSecurityManager(new RMISecurityManager());
        new MyClient(args[0]);
    }

    public MyClient(String host) throws RemoteException
    {
        sem = new Semaphore(-3);
        try {
            ...
            server.asyncExecute( new MyCalculation(10000), this );
            server.asyncExecute( new MyCalculation(1000), this );
            server.asyncExecute( new MyCalculation(10), this );
            server.asyncExecute( new MyCalculation(50), this );

            sem.Down();
            System.exit(0);

        } catch (...) {...}
    }

    public void workCompleted( WorkRequest request, Object result )
        throws RemoteException
    {
        //.. process result
        sem.Up();
    }
}
```

Example 5: Load classes over the network

See example in folder rmi-rpc/rmi/ex4-Load-Remote-Class.

```
import java.rmi.RMISecurityManager;
import java.rmi.server.RMIClassLoader;
import java.net.*;

public class LoadClient
{
    public static void main(String[] args)
    {
        System.setSecurityManager(new RMISecurityManager());
        try {
                URL url = new URL(args[0]);
                Class cl = RMIClassLoader.loadClass(url,"MyClient");
                System.out.println(cl);
                Runnable client = (Runnable)cl.newInstance();
                client.run();
                System.exit(0);
        } catch (Exception e) {
                System.out.println("Exception: " + e.getMessage());
                e.printStackTrace();
        }
    }
}
```

```
java LoadClient http://onyx.boisestate.edu/~amit/codebase/rmi/ex4/
```

Class Loading in Java

- The AppletClassLoader is used to download a Java applet over the net from the location specified by the codebase attribute on the web page that contains the <applet> tag. All classes used directly in the applet are subsequently loaded by the AppletClassLoader.
- The default class loader is used to load a class (whose main method is run by using the java command) from the local `CLASSPATH`. All classes used directly in that class are subsequently loaded by the default class loader from the local `CLASSPATH`.

Class Loading in Java (contd.)

- The RMIClassLoader is used to load those classes not directly used by the client or server application: the stubs and skeletons of remote objects, and extended classes of arguments and return values to RMI calls. The RMIClassLoader looks for these classes in the following locations, in the order listed:
  - The local `CLASSPATH`. Classes are always loaded locally if they exist locally.

Class Loading in Java (contd.)

- The RMIClassLoader is used to load those classes not directly used by the client or server application: the stubs and skeletons of remote objects, and extended classes of arguments and return values to RMI calls. The RMIClassLoader looks for these classes in the following locations, in the order listed:
  - The local `CLASSPATH`. Classes are always loaded locally if they exist locally.
  - For objects (both remote and nonremote) passed as parameters or return values, the URL encoded in the marshal stream that contains the serialized object is used to locate the class for the object.

Class Loading in Java (contd.)

- The RMIClassLoader is used to load those classes not directly used by the client or server application: the stubs and skeletons of remote objects, and extended classes of arguments and return values to RMI calls. The RMIClassLoader looks for these classes in the following locations, in the order listed:
  - The local `CLASSPATH`. Classes are always loaded locally if they exist locally.
  - For objects (both remote and nonremote) passed as parameters or return values, the URL encoded in the marshal stream that contains the serialized object is used to locate the class for the object.
  - For stubs and skeletons of remote objects created in the local virtual machine, the URL specified by the local `java.rmi.server.codebase` property is used.

Dynamic downloading of classes by the RMI Class Loader

- Start up the rmiregistry. Make sure you do not start it from the server folder. If you do start the rmiregistry and it can find your stub classes in CLASSPATH, it will not remember that the loaded stub class can be loaded from your server's code base, specified by the java.rmi.server.codebase property when you started up your server application. Therefore, the rmiregistry will not convey to clients the true code base associated with the stub class and, consequently, your clients will not be able to locate and to load the stub class or other server-side classes.

Dynamic downloading of classes by the RMI Class Loader

- Start up the rmiregistry. Make sure you do not start it from the server folder. If you do start the rmiregistry and it can find your stub classes in CLASSPATH, it will not remember that the loaded stub class can be loaded from your server's code base, specified by the java.rmi.server.codebase property when you started up your server application. Therefore, the rmiregistry will not convey to clients the true code base associated with the stub class and, consequently, your clients will not be able to locate and to load the stub class or other server-side classes.
- The server and client machines both need to be running a web server. The folders containing the server and client code both need to be accessible via the web.

Example 6: Dynamic downloading of classes

See example in the folder `rmi-rpc/rmi/ex6-Dynamic-Classloading`.

The following shows a sample server start up, where the server is running on `onyx`.

```
currdir=`pwd`
cd /
rmiregistry &
cd $currdir

java -Djava.rmi.server.codebase="http://onyx.boisestate.edu/~amit/rmi/ex6/"
    -Djava.security.policy=mysecurity.policy  MyServer
```

The following shows a sample client start up, where the client is running on the host `cs`.

```
java \
-Djava.rmi.server.codebase="http://cs.boisestate.edu/~amit/teaching/555/lab/ex6/"\
-Djava.security.policy=mysecurity.policy  MyClient onyx
```

RMI and Thread Safety

- The default RMI implementation is multi-threaded. So if multiple clients call the server, all the method invocations can happen simultaneously, causing race conditions. The same method may also be run by more than one thread on behalf of one or more clients. Hence we must write the server to be thread-safe.
- Use the `synchronized` keyword to ensure thread-safety.
- The example `rmi-rpc/rmi/ex7-RMI-Thread-Safety` demonstrates the problem and a possible solution.

Example 7: RMI Thread Safety

```
import java.rmi.*;

public interface RMIThreadServer extends Remote {

    public void update() throws RemoteException;
    public int read() throws RemoteException;
}
```

Example 7 contd.: Server

```
import ...

public class RMIThreadServerImpl extends UnicastRemoteObject implements RMIThreadServer
{
    private volatile int counter = 0;
    private final int MAXCOUNT = 900000;
    public RMIThreadServerImpl () throws RemoteException {
        super();
    }

    /*public synchronized void update() {*/
    public void update() {
        int i;
        Thread p = Thread.currentThread();

        System.out.println("[server] Entering critical section: " + p.getName());
        for (i = 0; i < MAXCOUNT; i++)
            this.counter++;
        for (i = 0; i < MAXCOUNT; i++)
            this.counter--;
        System.out.println("[server] Leaving critical section: " + p.getName());

    }
    /*public synchronized int read() {*/
    public int read() {
        return this.counter;
    }
    public static void main (String[] args) {
        try {
            RMIThreadServerImpl localObject = new RMIThreadServerImpl( );
            Naming.rebind("//localhost:5130/RMIThreadServer", localObject);
            System.err.println("DEBUG: RMIThreadServerImpl RMI listener bound\n");
        } catch (RemoteException e) {
            System.err.println("RemoteException: " + e);
        } catch (MalformedURLException e) {
            System.err.println("MalformedURLException: " + e);
        }
    }
}
```

Example 7 contd.: Client

```
import java.io.*;
import java.net.*;
import java.rmi.*;

public class RMIThreadClient {

    /* RMI Parameters */
    String host = "localhost:5130";
    RMIThreadServer remObj;

    public static void main (String[] args) {
        RMIThreadClient myself = new RMIThreadClient();
        System.exit(0);
    }
    /** Constructor */
    public RMIThreadClient() {
        try {
            remObj = (RMIThreadServer) Naming.lookup("//" + host + "/RMIThreadServer");
            System.out.println("[client] (before) counter = " + remObj.read() );
            remObj.update();
            System.out.println("[client] (after)  counter = " + remObj.read() );
        } catch (java.rmi.NotBoundException e) {
            System.err.println("RMI endpoint not bound: "+e);
            System.exit(2);
        } catch (java.net.MalformedURLException e) {
            System.err.println("RMI endpoint URL not formed correctly: "+e);
            System.exit(2);
        } catch (java.rmi.RemoteException e) {
            System.err.println("RMI RemoteException: "+e);
            System.exit(2);
        }
    }
}
```

RMI: Further Topics

- **RMI Object Persistence.** RMI activation allows a remote object to be stored away (in a database, for example) and automatically reincarnated when it is needed.
- **RMI, CORBA, and IIOP.** CORBA (Common Object Request Broker Architecture) is a distributed object standard that allows objects to be passed between programs written in different languages. IIOP (Internet Inter-Object Protocol) is being used by RMI to allow limited RMI-to-CORBA interoperability.