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;
}
```

- The actual implementation would 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
}
```

Example 1: RMI Hello World

- Example is in the `rmi/ex1-HelloServer` folder.
- The remote interface: Hello.java
- The server that implements the remote interface: HelloServer.java
- A sample client: HelloClient.java
- Example 1: Running the RMI Hello World Example
- Start up the rmiregistry if it isn't already running. It runs on port 1099 by default. Choose a different port if you want to run your own copy (required in the onyx lab). make sure the class path is set correctly so rmiregistry can find your classes.

```
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.

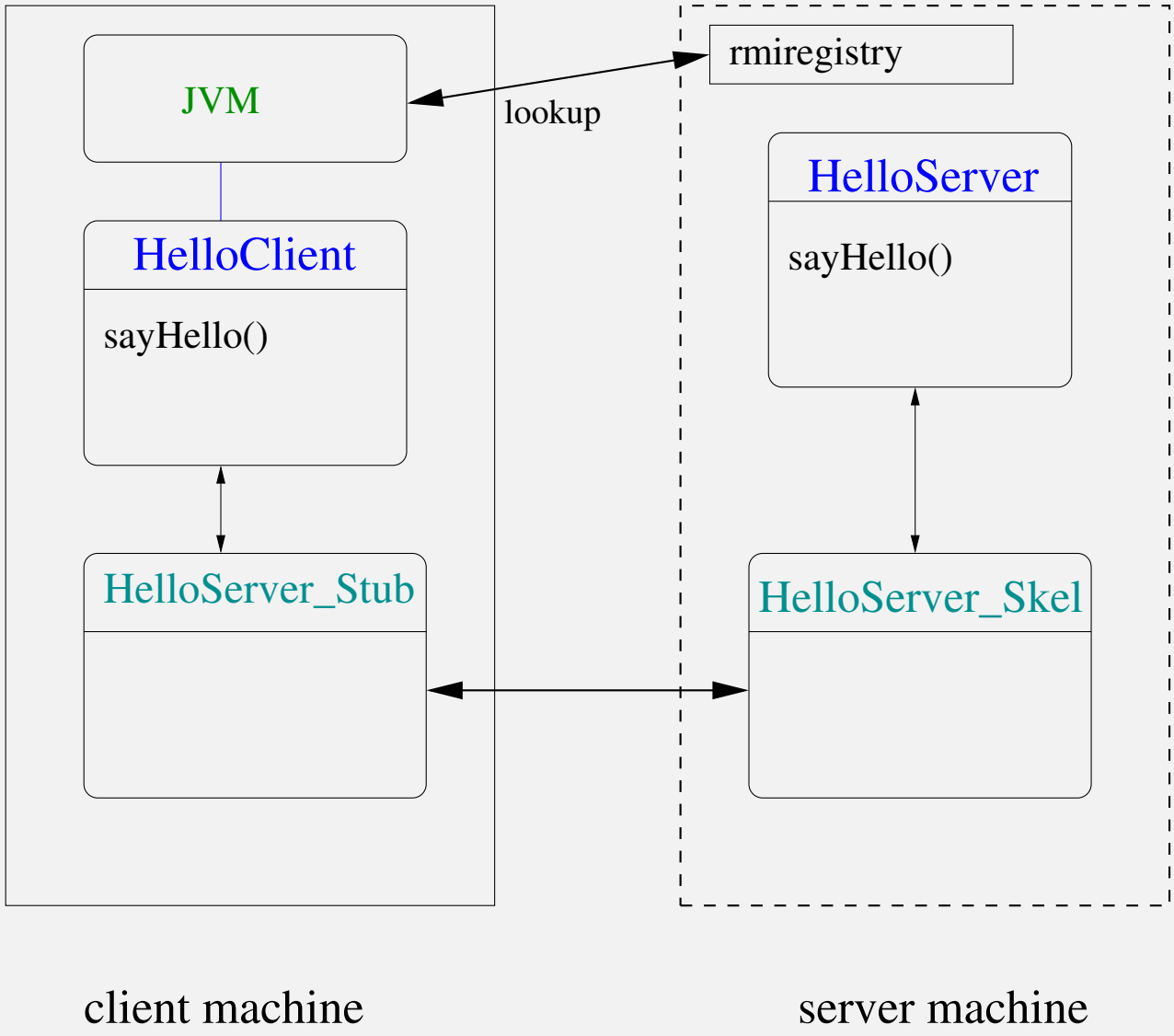
- Now run the client as follows.

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

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

```
killall -9 rmiregistry
```

Example 1: RMI Hello World



Example 1: RMI Hello World, Where is the Stub?

- **rmi/ex1-HelloServer-with-rmic**: Normally, the Java runtime generates the stubs using dynamic proxy objects. We can also build them statically using the `rmic` tool. However, this is deprecated. This example, same as the first example, uses `rmic` in the `Makefile` to generate static stubs.

RMI Server

- Instead of extending java.rmi.server.UnicastRemoteObject class, we can use the static method UnicastRemoteObject.exportObject to create a remote server. See below for a code snippet.

```
public class HelloServer implements Hello
{
  /* ... */
  public static void main(String args[]) {
    if (args.length > 0) {
      registryPort = Integer.parseInt(args[0]);
    }
    try {
      HelloServer obj = new HelloServer("HelloServer");
      Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
      Registry registry = LocateRegistry.getRegistry(registryPort);
      registry.bind("HelloServer", obj);
      System.out.println("HelloServer bound in registry");
    } catch (Exception e) {
          System.out.println("HelloServer err: " + e.getMessage());
    }
  }
}
```

- The port parameter is 0, which means it will pick a random available port for RMI server port. For a firewalled environment, we could choose a specific port number here and allow it through the firewall.

Example 2: RMI Square Server

- This example is in the folder rmi/ex2-SquareServer
- The server interface is in server/Square.java

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

- The server implementation is in server/SquareServer.java
- A sample client is in client/SquareClient.java
- 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/ex3-Client-Callback
- A RMI version of the sockets-based Object server from earlier examples. Compare the two approaches.
- Two packages `callback.server` and `callback.server`.
  - The server interface is in Server.java and the server implementation is in MyServer.java. The other classes in the server package are the same as in the Object server example from before.
  - The main client class is in MyClient.java. The work class that client uses is MyCalculation.java. *Note that client is also a remote server to allow callback from the server.*

Example 3 (Client Callback): Server Interface

```
--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 (Client Callback): Server Implementation

```
public class MyServer
  extends java.rmi.server.UnicastRemoteObject implements Server {
    public MyServer() throws RemoteException { }
    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 (Client Callback): Running the Example

This example is in the folder `rmi/ex3-Client-Callback`. Noet that the client and server need access to classes from both packages.

```
rmiregistry [registryPort] &
```

Then we start up the server as follows:

```
java -Djava.security.policy=mysecurity.policy callback.server.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.

Example 4: 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.
  - We introduce a new class `AysncExecuteThread` that is called from the `asynExecute` method.
- 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.
  - We use an object for synchronization on counting outstanding requests.
  - Note also the client is now a server so we have to explicitly end it when we are done.
- See example: rmi/ex4-Asynchronous-Server

Example 5: Load classes over the network

See example in folder `rmi/ex5-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();
        }
    }
}
```

Class Loading in Java

- 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.
- 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.
- 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 3 (revisited): Dynamic downloading of classes

See example in the folder `rmi/ex3-Client-Callback`

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

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

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

The following shows a sample client start up, where the client is running on the host `cs`, which also has a webserver running on it.

```
java \
-Djava.rmi.server.codebase=\
 "http://cs.boisestate.edu/~amit/teaching/455/rmi/ex3-Client-Callback/"\
-Djava.security.policy=mysecurity.policy  callback.client.MyClient onyx
```

Example 6: 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.
- See example : rmi/ex6-Thread-Safety demonstrates the problem and a solution.

RMI through Firewalls

- Here is what ports RMI uses:
  - The RMI Registry uses port 1099 (or whatever port you specified to it when you started it). This is what the clients will use to make the initial connection.
  - Client and server (stubs, remote objects) communicate over random ports. The communication is started via a socket factory which uses 0 as starting port, which means "use any port that's available" between 0 and 65535.
- Here is how we can get this work with firewalls.
  - Simply open all non-privileged ports on the server. This is fine for testing but obviously not desirable for production system!
  - Use a custom RMI socket factory and fix the port used in it.
    - See the javadocs for RMISocketFactory
    - Official Java guide for custom socket factories: http://docs.oracle.com/javase/8/docs/technotes/guides/rmi/socketfactory/

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.