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.
- Enhanced version of Remote Procedure Call (RPC).
- 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. The stubs and skeletons are managed behind the scenes by the Java Virtual Machine.

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.

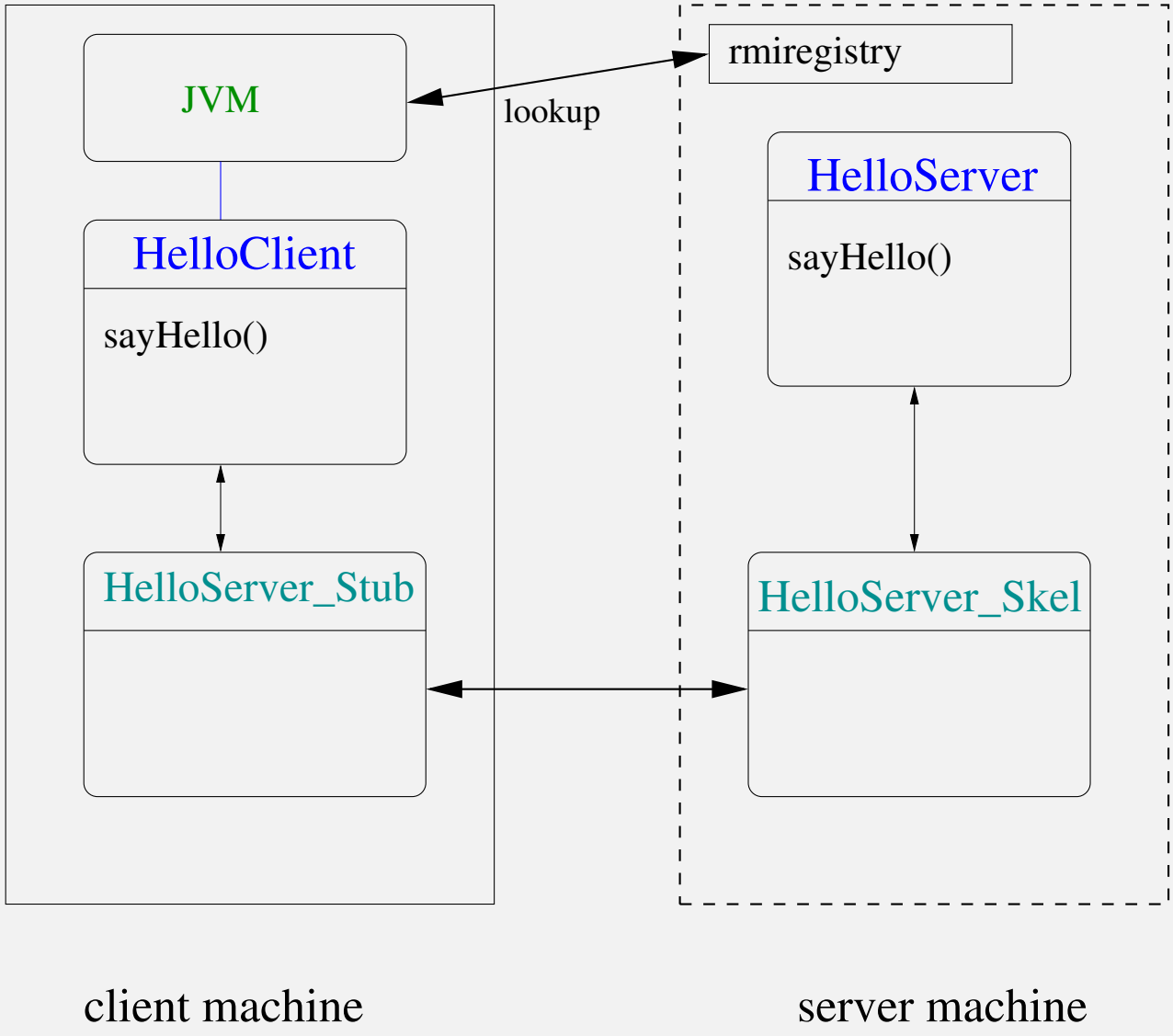
  ```
  export CLASSPATH=$(pwd):$CLASSPATH
  rmiregistry [registryPort] &
  ```
- Then we start up the server as follows.

  ```
  java hello.server.HelloServer &
  ```
- Now run the client as follows.

  ```
  java hello.client.HelloClient hostname [registryPort]
  ```
- Once you are done, kill the server (use `Ctrl-c`) and the `rmiregistry` as shown below.

  ```
  killall -9 rmiregistry
  ```

Example 1: RMI Hello World



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: Object Server

- This example is in the folder rmi/ex3-Object-Server
- A RMI version of the sockets-based Object server from earlier examples.
- Compare the two approaches.
- Two packages `synchronous.server` and `synchronous.client`.
  - 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.

Example 3 (Object Server): 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;
}

--StringEnumerationRequest.java--
import java.rmi.*;
public interface StringEnumerationRequest 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;
    StringEnumerationRequest getList() throws java.rmi.RemoteException;
    void asyncExecute( WorkRequest work, WorkListener listener )
                throws java.rmi .RemoteException;
}
```

Example 3 (Object Server): 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 StringEnumerationRequest 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[]) {
                try {
                        Server server = new MyServer();
                        Naming.rebind("NiftyObjectServer", server);
                        System.out.println("bound");
                } catch (java.io.IOException e) {
                        System.out.println("// Problem registering server");
                        System.out.println(e);
                }
    }
}
```

Example 3 (Object Server): Running the Example

This example is in the folder rmi/ex3-Object-Server. Note that the client and server need access to classes from both packages.
Then we start up the rmiregistry and the server as follows:

```
./run-server.sh
```

Now run the client as follows:

```
./run-client.sh
```

or

```
./test-multiple-clients.sh
```

Once you are done, kill the server and the rmiregistry.

In-class Exercise: RMI Object Server/Client

- Run the object RMI server example on your machine.
- Modify `MyServer.java` class to use 8 threads for the RMI thread pool (see comment in code that shows us how to do set pool size). Then run 8 clients simultaneously using the `test-multiple-clients.sh` script.

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 RMI server so we have to explicitly end it when we are done.
- See example: rmi/ex4-Asynchronous-Server

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.
- The example below demonstrates the problem and a solution: rmi/ex6-Thread-Safety

More Examples

- rmi/ex7-PassingArgsInRMI
- rmi/ex8-with-timeout
- rmi/ex9-HelloServer-2-interfaces
- rmi/ex10-HelloServer-timeout

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 1 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 a production system!
  - Use a custom RMI socket factory and fix the port used in it.
    - Official Java guide for custom socket factories: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/javax/net/SocketFactory.html