Remote Method Invocation (RMI)

❖ ***Topics***

- Architecture
- Method Parameters
- Simple Example
- Pass by Reference
- Distributed Garbage Collection
- Dynamic Class Loading
- Distributed Polymorphism
- Invocation Traffic

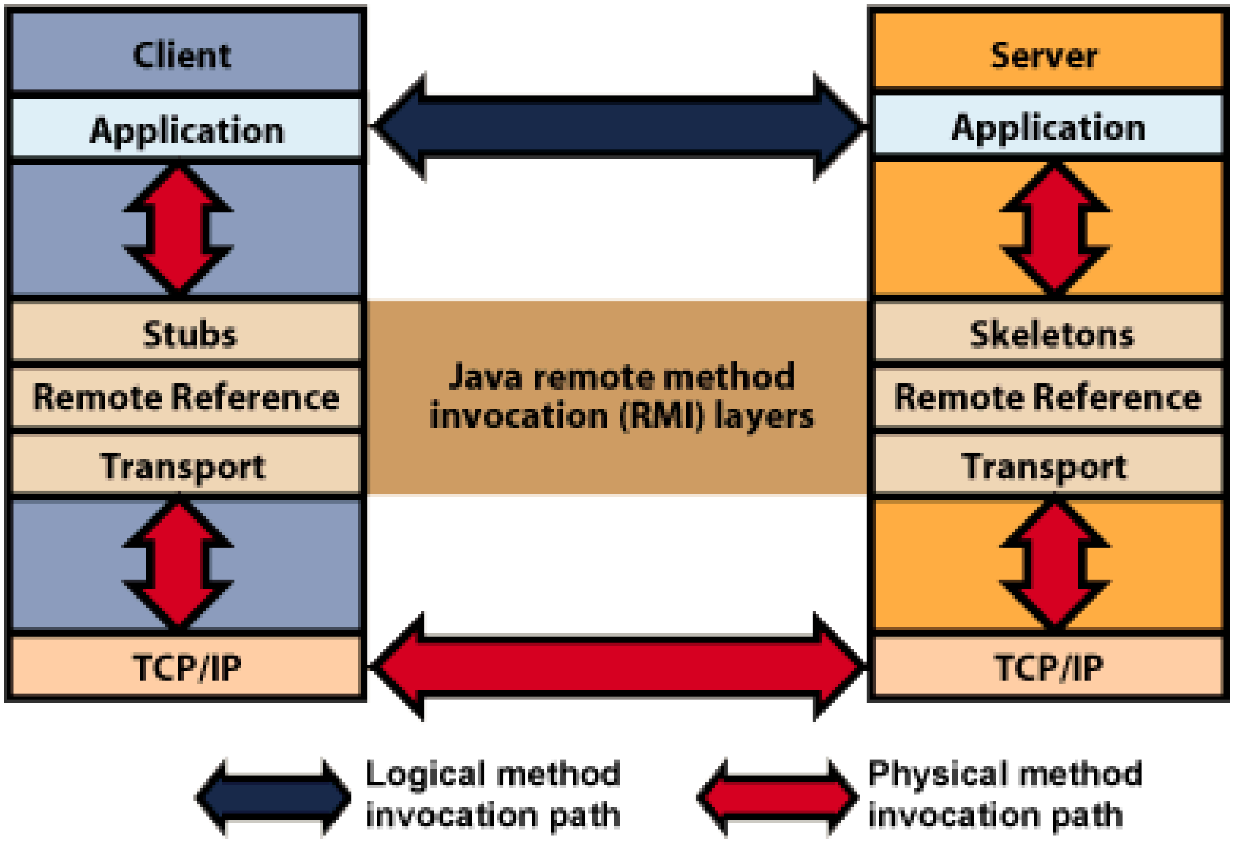
Java Remote Method Invocation

- The Java language's native mechanism for writing distributed objects.
  - **JAVA RMI is Java-centric.**
  - Enables objects to communicate across Java Virtual Machines and physical devices.
- An ***alternative to CORBA, Distributed Component Object Model (DCOM) and SOAP***.
- ***RPC:*** A procedural invocation from a *process* on one machine to a *process* on another.
  - RMI takes this concept one step further and allows for ***distributed object communication***.
  - RMI allows us to *invoke methods on remote objects*.
  - Can build your networking code as *full objects*.
- Pandora's Box of OO programming opportunities such as *distributed polymorphism*.

RMI Architecture

- Any object whose methods can be invoked from another JVM is called a ***Remote Object***.
  - Remote objects are networked objects which expose methods that can be invoked from remote clients.
- Physical location is ***not important*** (location transparency).
  - Client can invoke methods on an object running on a different VM on the same machine or on a remote machine.
  - Client running in the same address space as a remote object can also invoke its methods.
    - To the remote object, both invocations appear to be the same.
- RMI is a ***rich architecture:***
  - Transparent networking (local/remote transparency).
  - Distributed Garbage Collection.
  - Dynamic Class Loading.
  - Security

RMI Architecture



Interfaces v Implementation

- An interface *defines the exposed information about an object*.
  - The names of its methods, parameters, return types.
- ***The client deals with the interface, not with the remote object***.
  - Interface *masks the implementation* from the client's viewpoint.
  - Clients need only be concerned with the end-result: the methods exposed.
- The ***implementation*** is the core programming logic that an object provides - specific algorithms, logic and data.
- ***Why use an interface?***
  - By separating interface from implementation, we can vary an object proprietary logic without changing any client code.
  - E.g. Can plug-in a different algorithm that performs some task more efficiently.

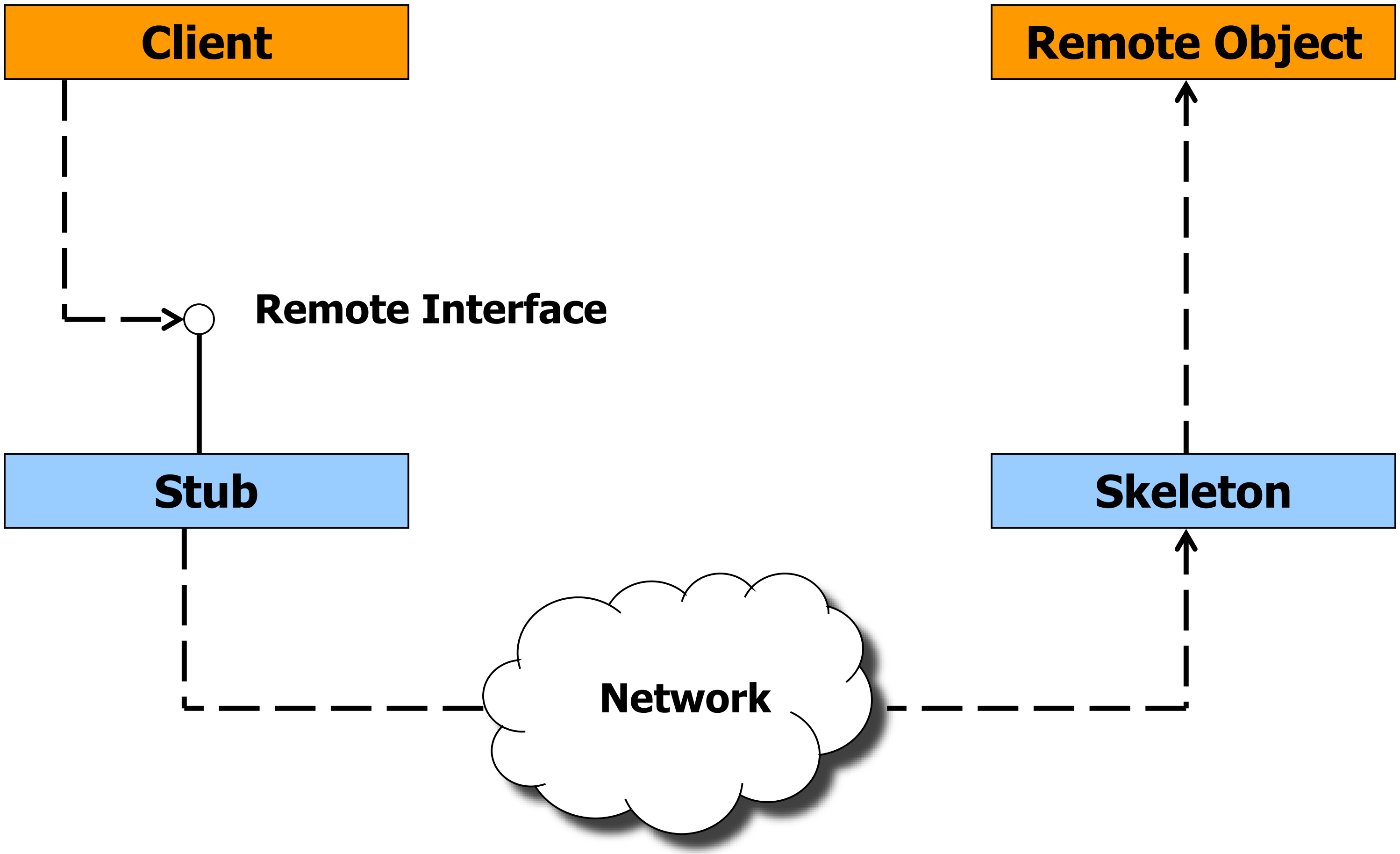
Interfaces v Implementation

- RMI makes *extensive use of interfaces*.
  - All networking code we write is ***applied to an interface***, not an implementation.
  - Impossible to perform a remote invocation directly on an object's implementation.
- *To designate that an object can be accessed remotely, the class must implement the interface* **`java.rmi.Remote`**.
  - Done by creating our own custom interface extending ***`java.rmi.Remote.`***
- Interface contains definition of each method the remote object exposes.
  - *Each method must also throw a java.rmi.RemoteException*.
  - Can be thrown for a variety of reasons – network, registry, class implementation etc…

Stubs and Skeletons

- To mask the fact that you are invoking a remote object, RMI needs to simulate that object locally.
  - Called a ***Stub***.
- *A stub is responsible for accepting method calls locally and* ***delegating*** *them to their actual object implementations*.
- The remote object in turn accepts calls from a ***skeleton*** that is local to the remote object.
  - ***Skeletons are responsible for receiving calls over the network (from a stub) and delegating them to the remote object implementation***.
- Stubs and skeletons are ***also responsible for marshalling*** – done using Java object serialization.
- ***Note: the use of skeletons is not required from Java2 onwards.***

Stubs and Skeletons



Bootstrapping / RMI Registry

- Before a client and server can start talking, they need some way to connect.
  - ***Acquiring this connection is called bootstrapping.***
- Provided by the ***<u>RMI Registry</u>***.
- To make a remote object accessible, register it with the RMI Registry (yellow pages).
- Registry will then route all incoming requests for that object name to our object.
  - Like a giant hashtable. Just a specific type of naming service.
- When a remote client wants to access an object registered with a particular registry, the client issues a lookup request to the registry.
  - RMI registry is by default ***bound to port 1099*** (can change this if we want).

RMI URLs

- A Java String that is used to located remote objects on another JVM.
- ***Conventions:***
  - URL must start with *rmi://*
  - Need to specify the target machine. Defaults to *localhost* if omitted.
  - Append the name of the remote object we want to use:

    rmi://www.gmit.ie:1014/myObjectName
- URL is mapped to an object in the RMI Registry using the *java.rmi.Naming* class.
  - Use the ***lookUp()*** which takes the URL as a parameter to find an object.
  - Object is initially registered with the RMI Registry using the ***reBind()*** method.
- JNDI is an enterprise extension of the RMI Registry. No need to hard-code URLs – dynamic binding.

The RMIC Compiler

- Stubs and skeletons provide a screen to block networking issues.
  - Both ***must implement*** the remote object's interfaces.
- Use the rmic compiler to generate the stubs and skeletons for us.

>rmic RemoteObjectName

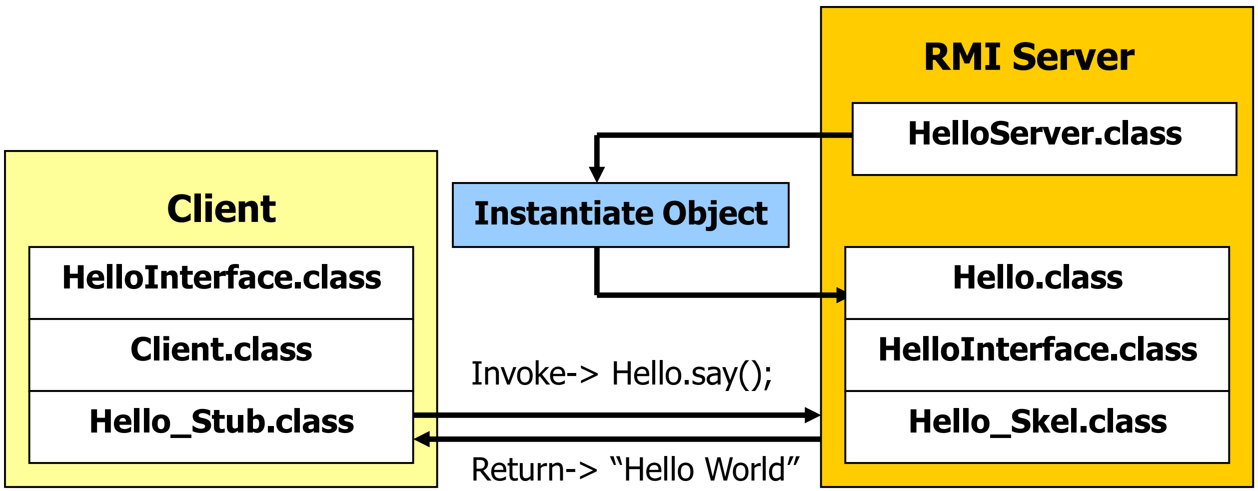
  - Can use the *–keepgenerated* option to view the Java source code.
  - Note: Since Java 5, it is not necessary to call rmic separately to compile RMI stubs

Object Serialization & Params

- When invoking a method using RMI, all parameters to the remote method are ***passed by value***.
  - i.e. all parameters are copied from one machine to another when the target method is invoked.
- ***Problem:*** If passing an object over the network and that object has references to other objects, how are those references resolved on the target machine?
  - Use object serialization to handle this.
- Serialization is the conversion of a java object to a byte stream representation.
  - Once serialized, it can be sent anywhere – network, disk.
- When you want to use the object again, you must de-serialize it.

Simple Example – Hello World

- We will use RMI to invoke a method called ***say()*** on an object called ***Hello*** and print out the response “Hello World”.
- An overview of the architecture looks like this:



1: Define the Remote Interface

- The Remote interface must have the following properties:
  1. The interface must be ***public***.
  2. The interface must extend the interface ***`java.rmi.Remote`***.
  3. Every method in the interface must declare that it throws ***`java.rmi.RemoteException`***. Other exceptions may also be thrown.

```
import java.rmi.*;

public interface HelloInterface extends Remote{
  public String sayHello(String strName) throws ➔
  RemoteException;
}
```

2: Implement the Interface

- The implementation of our remote interface is the remote object – the object whose method we wish to invoke.
- The Remote class itself has the following properties:
  1. It must implement a ***`Remote`*** interface.
  2. It should extend the ***`java.rmi.server.UnicastRemoteObject`*** class. This handles the socket and transport issues.
     - ***Question***: Why Unicast?
     - We also can export an object directly in our code using: ***`java.rmi.server.UnicastRemoteObject.exportObject();`***
     - What is the benefit of doing this?
  3. It **can** have methods that are ***not in its Remote interface***. These can only be invoked locally – why?

2: Implement the Interface

```
import java.rmi.*;
import java.rmi.server.*;
public class HelloWorld extends UnicastRemoteObject ➔
   implements HelloInterface{
   private String strMessage;

   public HelloWorld(String msg) throws RemoteException{
        strMessage = msg;
   }

   public String sayHello(String strName) throws ➔
   RemoteException{
        return strMessage + " " + strName;
   }
}
```

3: Create Stubs and Skeletons

- After compiling the remote object, we can create the stubs and skeletons using the rmic compiler:

```
c:>rmic HelloWorld
```

- This generates the following files:
  - HelloWorld_Skel.class
  - HelloWorld_Stub.class
- Recall that we can use the *–keepgenerated* switch to generate the .java files for our perusal.

4: Instantiate the Remote Object

- So far we have defined and implemented a remote interface.
- But a client can only invoke a method on an object that exists.
  - ***Our object has not been instantiated yet.***
- Also, we need to register our object with the RMI Registry so a client can find it.
- Therefore we use a “server” or factory class to create our remote object for us – recall: an object is an instance of a class.

4: Instantiate the Remote Object

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

class HelloServer{
   public static void main(String args[]){
        try{
             Naming.rebind("HelloWorld", new ➔
             HelloWorld("Howday"));
             System.out.println("The HelloWorld server ➔
             is ready...");
        }catch(Exception e){
             System.out.println("Error creating ➔
             HelloWorld object.");
        }
   }
}
```

5: Create the Client

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

public class Client{
  public static void main(String args[]){
      try{
          HelloInterface hello = (HelloInterface)➔
          Naming.lookup("//localhost/Howday");
          System.out.println(hello.sayHello(args[0]));
      }catch (Exception e){
          System.out.println("Error: looking up ➔
          object in RMI registry." + e);
      }
  }
}
```

The result of the `Naming.lookup` method **must be cast** to the type of the remote interface

6: Start the RMI Registry

- Before starting the Server, we need to start the Object Registry, and leave it running in the background:

rmiregistry

- It takes a second or so for the Object Registry to start running and to start listening on its socket (what port number?).
- The server should then be started with the java command.
- The client can then be started and make its remote method invocation.

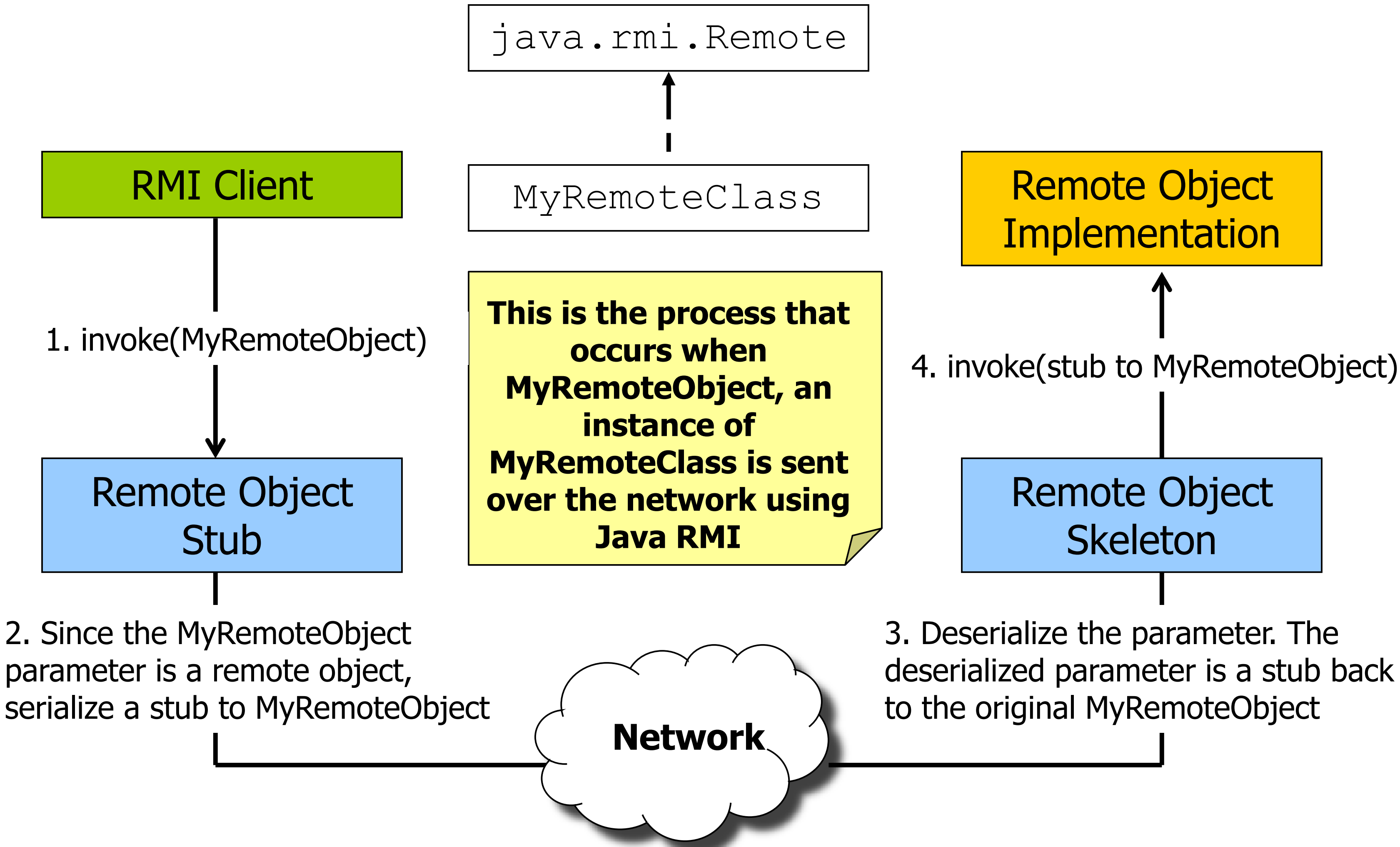
Pass by Reference

- So far we have looked at passing parameters by value.
- An entire graph of objects has to be serialized and sent across the network.
  - What happens if graph is very large?
  - What if we have lots of state information to send?
  - Performance degradation!
- RMI allows us to get around this by simulating a pass by reference – a copy of an object parameter is not sent.
  - The remote method works with the original object on the client.
- ***To do this, the object parameter must itself be a remote object.***
- When the remote host performs an action on the object parameter, this action occurs on the local host.
  - ***i.e. The operation is itself a remote method invocation***.

Pass by Reference

- What is actually passed to the remote object?
  - The stub for our local object (which itself is remote with reference to the remote object).
- In Java we can have references to many objects.
- RMI is an extension of this allowing us references to remote objects.
- `java.rmi.RemoteStub` objects are the manifestation of this.
- They are serializable and can be passed over the network – class loading.
- In summary, Java RMI simulates a pass by reference by passing a serialized stub, rather than passing a serialized object.

Pass by Reference



Distributed Garbage Collection

- RMI uses a reference-counting garbage collection algorithm similar to Modula-3's Network Objects.
  - RMI runtime keeps track of ***all live references*** within each JVM.
- Described by ***java.rmi.dgc.DGC*** interface methods: ***dirty()*** and ***clean()***.
  - ***Dirty call*** is made when a remote reference is unmarshaled in a client (the client is indicated by its VMID). ***Reference count*** is incremented
  - Corresponding ***clean call*** is made when no more references to the remote reference exist in the client.
  - ***As live references are found to be unreferenced in the local VM, the count is decremented.***
- A remote object not referenced by any client is called a ***weak reference***.
  - Allows the JVM's garbage collector to discard the object if no other local references to the object exist.

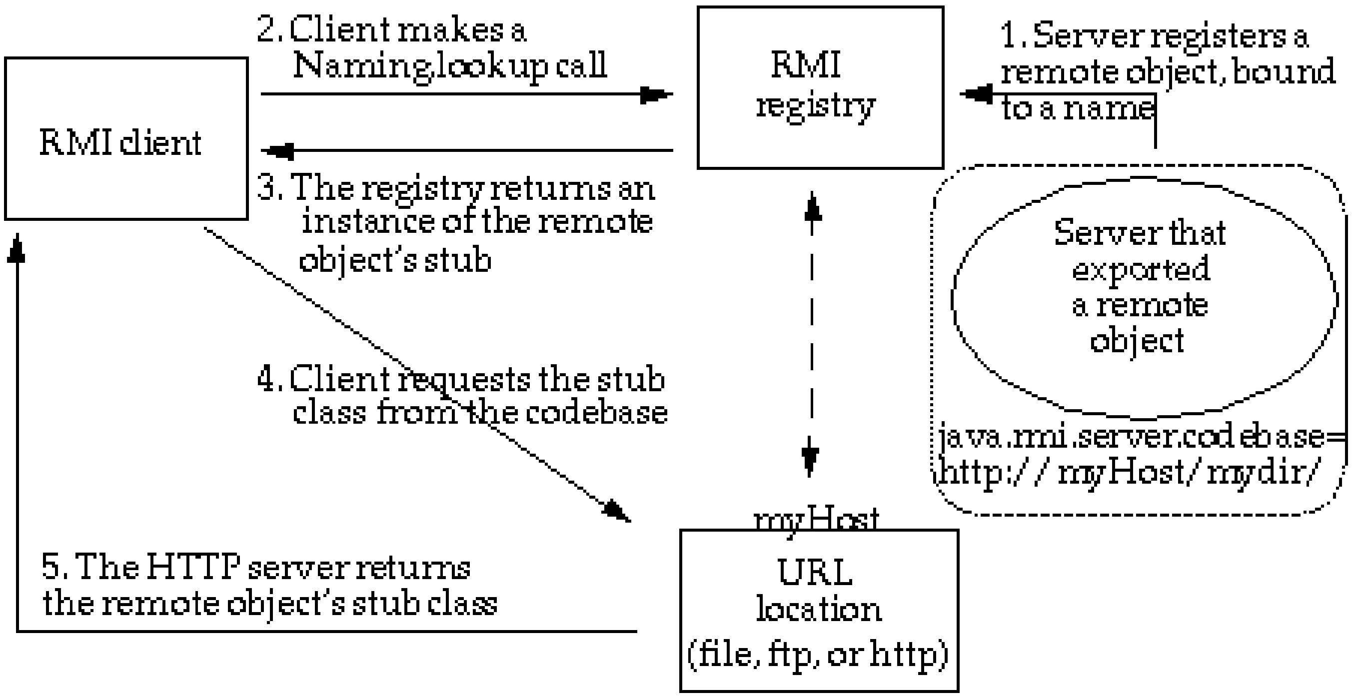
Distributed Garbage Collection

- As long as a local reference to a remote object exists, it cannot be garbage-collected.
  - Can be passed in remote calls or returned to clients.
- Passing a remote object ***adds the identifier*** for the VM to which it was passed to the referenced set.
  - A remote object needing unreferenced notification must implement the ***java.rmi.server.Unreferenced*** interface.
  - ***unreferenced()*** method invoked when references no longer exist.
  - ***finalized()*** (if implemented) will be invoked when the GC is about to reclaim an objects memory space.
- ***Note:*** References cannot guarantee ***referential integrity***: possible that a remote reference may in fact not refer to an existing object.
  - Transport might think that application has crashed.

Dynamic Class Loading

- RMI allows parameters, return values and exceptions passed in RMI calls to be any object that is ***serializable***.
  - Annotates the serialized call stream with the ***appropriate location information*** so that the class definition files can be loaded at the receiver.
- ***Require class definitions*** for unmarshalled parameters/return types.
  - Unmarshalling process first attempts to resolve classes by name in its ***local CLASSPATH***.
  - Can also ***dynamically load class definitions*** over a network (including remote stub classes).
- Done using special subclasses of ***ObjectOutputStream*** and ***ObjectInputStream***. (***RMIClassLoader***)
  - Subclasses override the ***annotateClass()*** of ***ObjectOutputStream*** and ***resolveClass()*** of ***ObjectInputStream*** .
  - Communicates locations for class definitions of *class descriptors in the stream*.

Dynamic Class Loading



Dynamic Class Loading

- Normally dynamically load classes over **HTTP or FTP**.
  - ***java.rmi.server.RMIClassLoader*** performs this function.
- Controlled by ***properties*** specified when each JVM is run:

  java –D<propertyName>=<propertyValue> + className

  - Use ***java.rmi.server.codebase*** property to specify a URL (same as applets).
  - Points to a HTTP/FTP service that supplies classes for objects <u>***sent from this JVM.***</u>
  - This URL is sent in serialized byte stream to client along with object.
- ***Note:*** RMI does <u>***not***</u> send class files along with serialized objects.
  - If remote JVM needs to load a class file for an object, it looks for the embedded URL and contacts server at that location for the class.
- If *java.rmi.server.useCodebaseOnly* property is set to true, JRE will load classes only from location specified by the *CLASSPATH* or a URL specified by *java.rmi.server.codebase* property.
  - This is an easy way of imposing ***additional security*** to the behaviour of ***RMIClassLoader***.

Class Loading Configurations

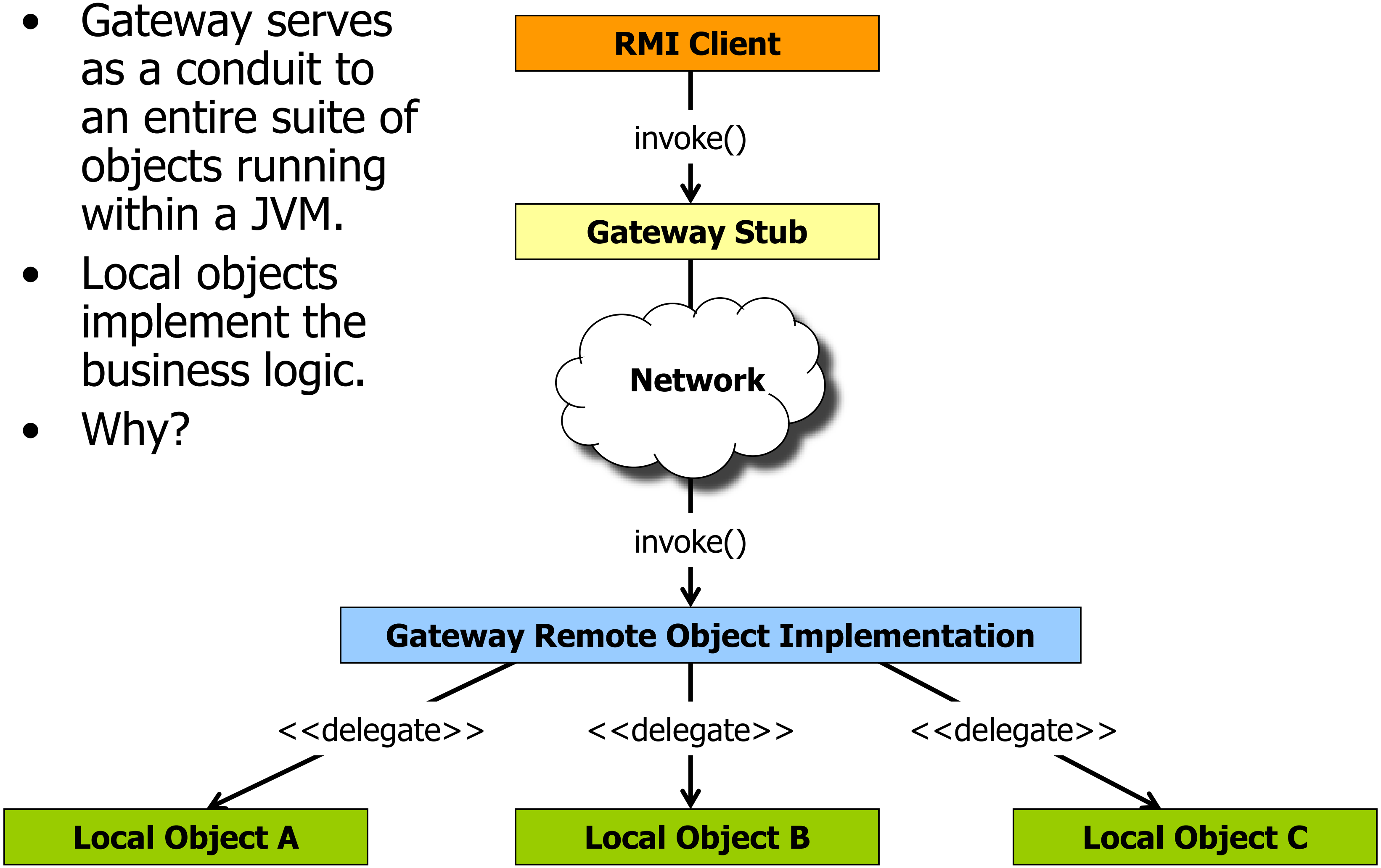
- ***Closed***: All classes used by clients and server must be located on JVM and referenced by the *CLASSPATH* environment variable. No dynamic class loading is supported.
- ***Server based:*** Client is loaded from the server's CODEBASE along with all supporting classes. Similar to the way applets are loaded.
- ***Client dynamic:*** Primary classes loaded by referencing the *CLASSPATH* environment variable of the JVM for the *client*. Supporting classes loaded by *java.rmi.server.RMIClassLoader* via HTTP/FTP from *server specified* location.
- ***Server-dynamic:*** Primary classes loaded by referencing the *CLASSPATH* environment variable of the JVM for the *server*. Supporting classes loaded by *java.rmi.server.RMIClassLoader* via HTTP/FTP from client specified server.

Class Loading Configurations

- ***Bootstrap client:*** All client code is loaded via HTTP/FTP server across the network. Only code residing on the client machine is a small bootstrap loader.
- ***Bootstrap server.*** All of the server code is loaded via HTTP/FTP server located on the network. Only code residing on the server machine is a small bootstrap loader.

Using Gateway to Mask RMI

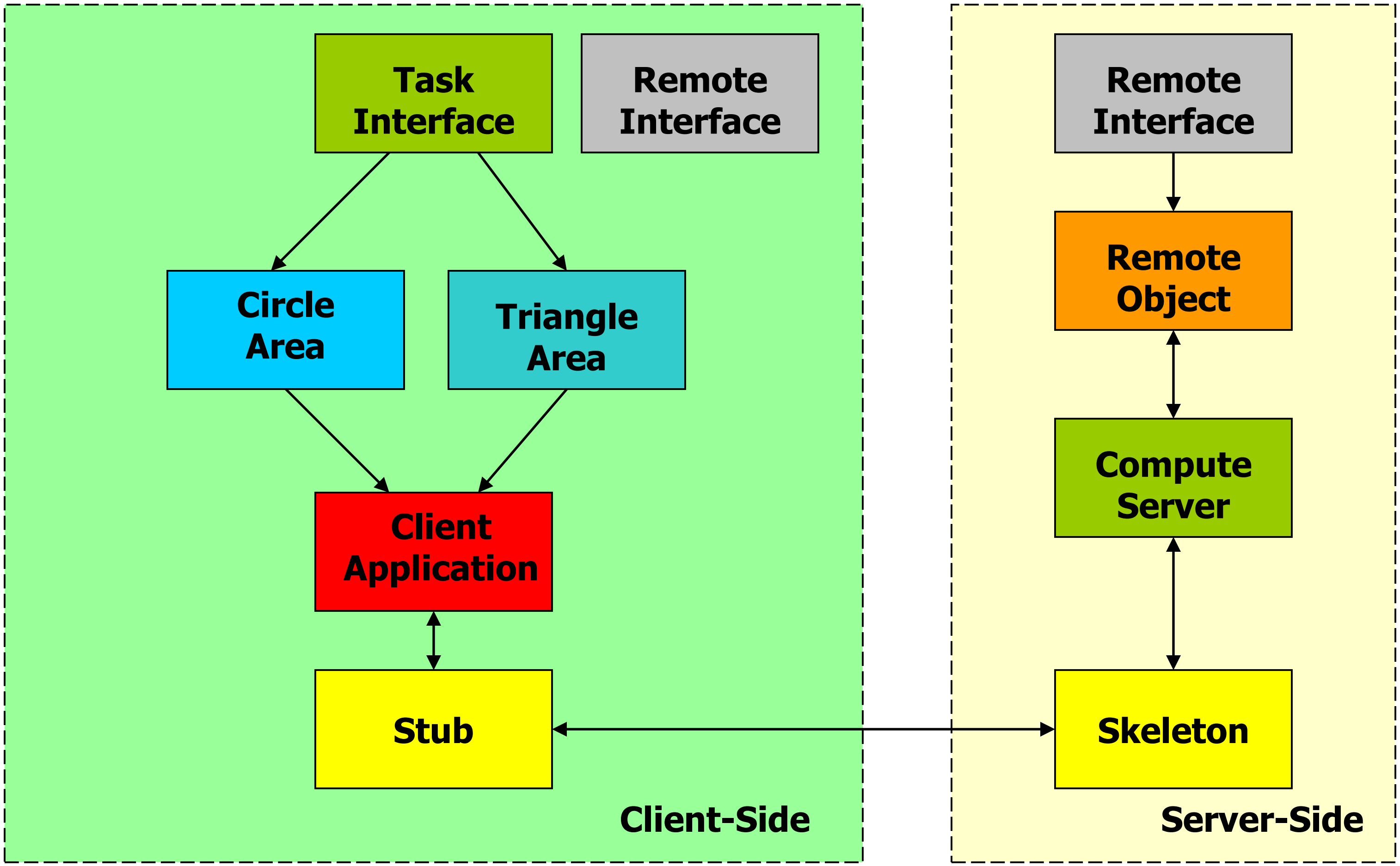
- Gateway serves as a conduit to an entire suite of objects running within a JVM.
- Local objects implement the business logic.
- Why?



Distributed Polymorphism

- ***The ability to recognise at runtime, the actual implementation type for a particular interface.***
- We will use an example of a remote object that calculates arbitrary tasks:
  - A client application sends a task object to a compute server.
  - The compute server runs the task and returns a result.
  - RMI loads the task code dynamically on server.
- This example shows polymorphism on the server.
  - Will also work on the client.
- *Server returns a particular interface implementation*.

Distributed Polymorphism



The Task and Remote Interfaces

```
package gmit;
import java.io.Serializable;

public interface Task extends Serializable{
    public Object run();
}



package gmit;
import java.rmi.Remote;
import java.rmi.RemoteException;

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

The Remote Object & Server

```
package gmit;
import java.rmi.*;
import java.rmi.server.*;
public class ComputeServer extends UnicastRemoteObject implements Compute{
    public ComputeServer() throws RemoteException{}

    public Object runTask(Task t) throws RemoteException{
        return t.run();
    }

    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        try{
            ComputeServer cs = new ComputeServer();
            Naming.rebind("Computer", cs);
        }catch(Exception e){}
    }
}
```

We haven't even written the code that the server is going to execute yet!

The CircleArea Task

```
package gmit;

public class CircleArea implements Task{

    private double radius;

    public Object run(){
        Object value = (Object) new Double(radius * radius * Math.PI);

        return (value);
    }

    public void setRadius(double rad){
        radius = rad;
    }
}
```

The TriangleArea Task

```
package gmit;

public class TriangleArea implements Task{
    private double base;
    private double height;

    public Object run(){
        Object value = (Object) new Double((base/2)*height);
        return (value);
    }

    public void setBase(double b){
        base = b;
    }
    public void setHeight(double h){
        height = h;
    }
}
```

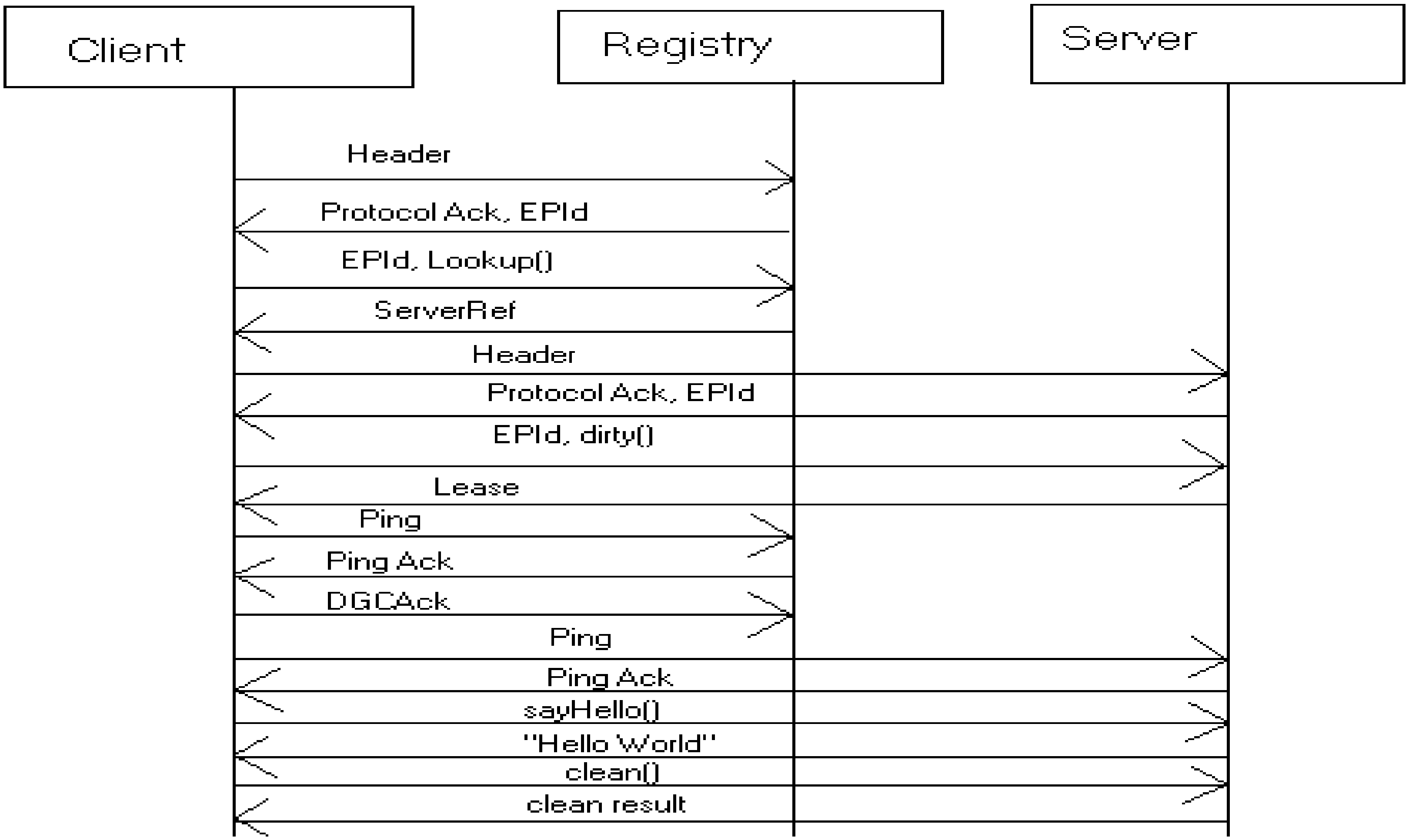
The Client Application

```
package gmit;

import java.rmi.*;

public class PolyClient{
    public static void main(String args[]){
        try{
            Compute comp = (Compute) Naming.lookup("///Computer");

            CircleArea circle = new CircleArea();
            circle.setRadius(1234);

            TriangleArea triangle = new TriangleArea();
            triangle.setBase(1234);
            triangle.setHeight(1234);

            String circleResult = ((Double) comp.runTask(circle)).toString();
            String triangleResult = ((Double) comp.runTask(triangle)).toString();
```

The Client Application

```
            System.out.println("Area of Circle:" + circleResult);
            System.out.println("Area of Triangle:" + triangleResult);

        }catch(NotBoundException nbe){
            System.out.println("NotBoundException Error: " + nbe.getMessage());
            System.exit(0);
        }catch(RemoteException re){
            System.out.println("RemoteException Error: " + re.getMessage());
            System.exit(0);
        }catch(Exception e){
            System.out.println("General Exception Error: " + e);
            System.exit(0);
        }
    }
}
```

Remote Call Message Sequence



Invocation Data Traffic

	Client to server & registry	Server & Registry to Client	Total
Registry Lookup	55(6%)	276(42%)	331(20%)
Invocation Data	41(4%)	37(6%)	78(5%)
DGC Data	831(85%)	305(46%)	1136(69%)
Protocol Overhead	52(5%)	40(6%)	92(6%)
Total	979(100)%	658(100%)	1637(100%)

- From Campadello et al, *"Performance Enhancing Proxies for Java2 RMI over Slow Wireless Links", Proceedings of the Second International Conference on the Practical Application on Java*, 2000.