RMI

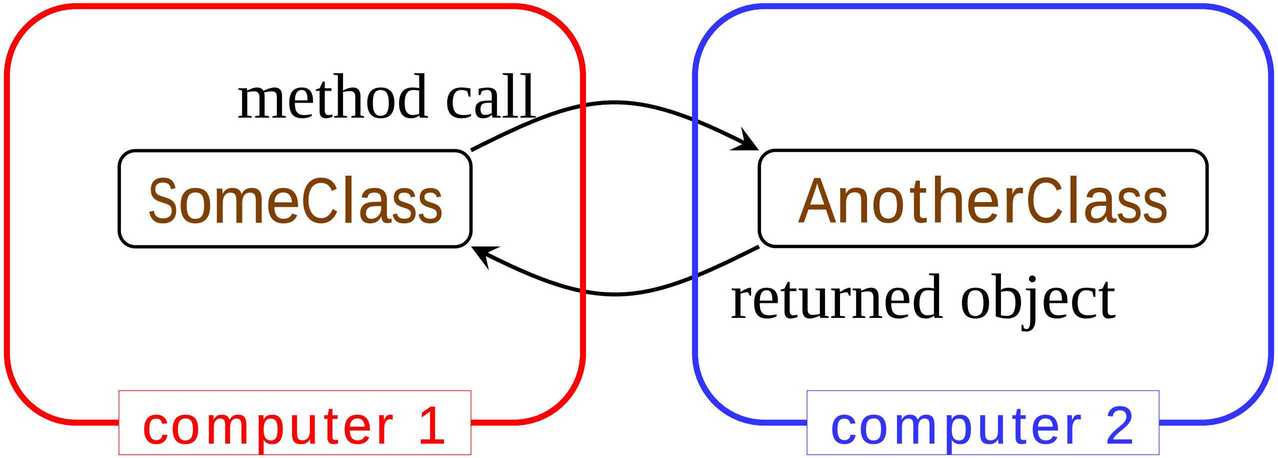
Remote Method Invocation

Written by: Dave Matuszek

appeared originally at:
http://www.cis.upenn.edu/~matuszek/cit597-2003/

“The network is the computer”*

- Consider the following program organization:



- If the network *is* the computer, we ought to be able to put the two classes on different computers
- RMI is one technology that makes this possible

* For an opposing viewpoint, see http://www.bbspot.com/News/2001/04/network.html

RMI and other technologies

- CORBA (Common Object Request Broker Architecture) has long been king
  - CORBA supports object transmission between virtually any languages
  - Objects have to be described in IDL (Interface Definition Language), which looks a lot like C++ data definitions
  - CORBA is complex and flaky
- Microsoft supported CORBA, then COM, now .NET
- RMI is purely Java-specific
  - Java to Java communications only
  - As a result, RMI is much simpler than CORBA

What is needed for RMI

- Java makes RMI (Remote Method Invocation) *fairly* easy, but there are some extra steps
- To send a message to a remote “server object,”
  - The “client object” has to *find* the object
    - Do this by looking it up in a registry
  - The client object then has to marshal the parameters (prepare them for transmission)
    - Java requires Serializable parameters
    - The server object has to unmarshal its parameters, do its computation, and marshal its response
  - The client object has to unmarshal the response
- Much of this is done for you by special software

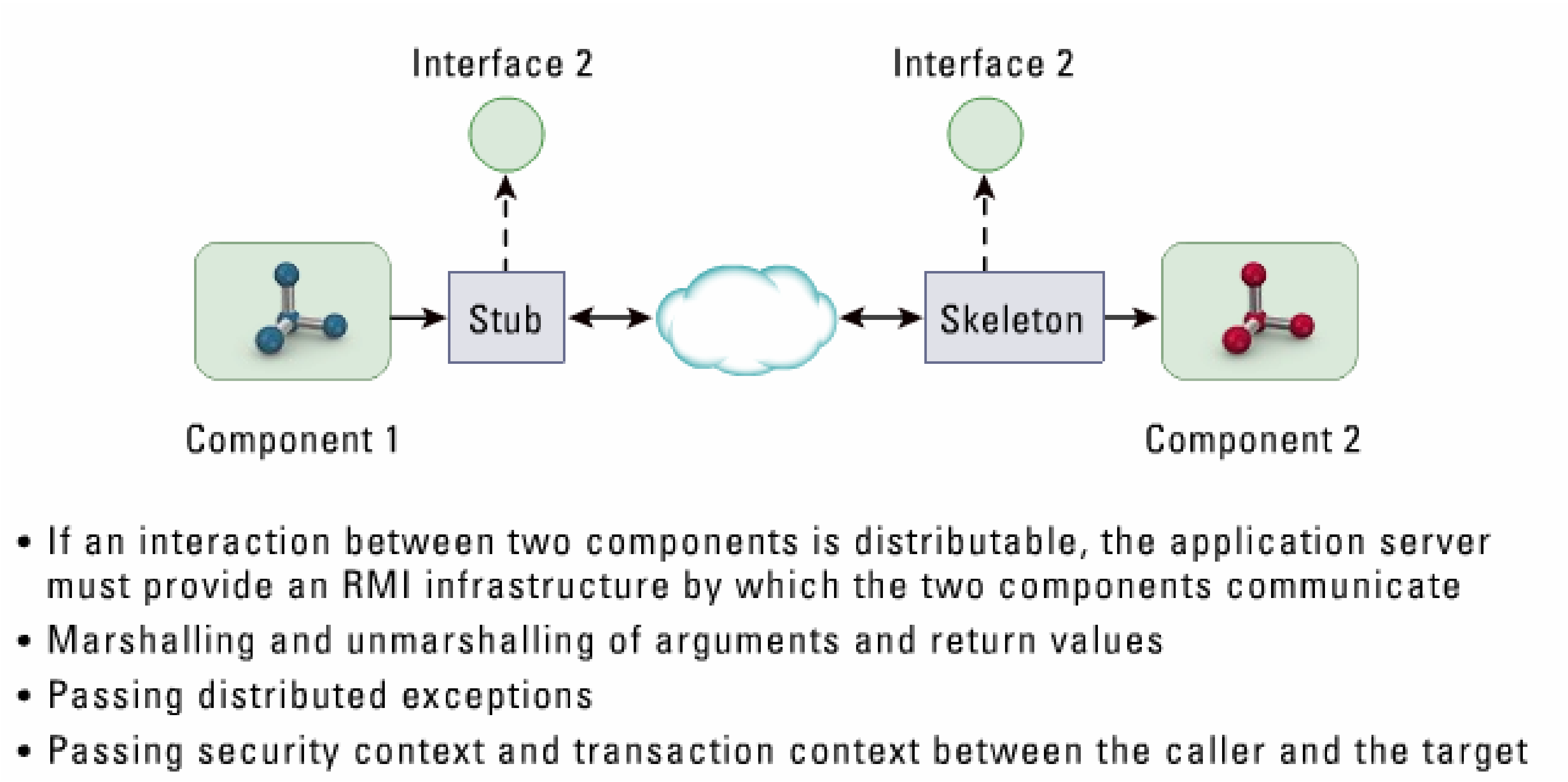
Terminology

- A remote object is an object on another computer
- The client object is the object making the request (sending a message to the other object)
- The server object is the object receiving the request
- As usual, “client” and “server” can easily trade roles (each can make requests of the other)
- The rmiregistry is a special server that looks up objects by name
  - Hopefully, the name is unique!
- rmic is a special compiler for creating stub (client) and skeleton (server) classes

Processes

- For RMI, you need to be running *three* processes
  - The Client
  - The Server
  - The Object Registry, rmiregistry, which is like a DNS service for objects
- You also need TCP/IP active

RMI Architecture



- If an interaction between two components is distributable, the application server must provide an RMI infrastructure by which the two components communicate
- Marshalling and unmarshalling of arguments and return values
- Passing distributed exceptions
- Passing security context and transaction context between the caller and the target

Interfaces

- Interfaces define behavior
- Classes define implementation

- Therefore,
  - In order to use a remote object, the client must know its behavior (interface), but does not need to know its implementation (class)
  - In order to provide an object, the server must know both its interface (behavior) and its class (implementation)
- In short,
  - The interface must be available to both client and server
  - The class should only be on the server

Classes

- A Remote class is one whose instances can be accessed remotely
  - On the computer where it is defined, instances of this class can be accessed just like any other object
  - On other computers, the remote object can be accessed via object handles
- A Serializable class is one whose instances can be marshaled (turned into a linear sequence of bits)
  - Serializable objects can be transmitted from one computer to another
- It probably isn't a good idea for an object to be both remote and serializable

Conditions for serializability

- If an object is to be serialized:
  - The class must be declared as public
  - The class must implement Serializable
  - The class must have a no-argument constructor
  - All fields of the class must be serializable: either primitive types or serializable objects

Remote interfaces and class

- A Remote class has two parts:
  - The interface (used by both client and server):
    - Must be public
    - Must extend the interface java.rmi.Remote
    - Every method in the interface must declare that it throws java.rmi.RemoteException (other exceptions may also be thrown)
  - The class itself (used only by the server):
    - Must implement a Remote interface
    - Should extend java.rmi.server.UnicastRemoteObject
    - May have locally accessible methods that are not in its Remote interface

Remote vs. Serializable

- A Remote object lives on another computer (such as the Server)
  - You can send messages to a Remote object and get responses back from the object
  - All you need to know about the Remote object is its interface
  - Remote objects don't pose much of a security issue
- You can transmit a *copy* of a Serializable object between computers
  - The receiving object needs to know how the object is implemented; it needs the class as well as the interface
  - There is a way to transmit the class definition
  - Accepting classes *does* pose a security issue

Security

- It isn’t safe for the client to use somebody else’s code on some random server
  - Your client program should use a more conservative security manager than the default
  - `System.setSecurityManager(new RMISecurityManager());`
- Most discussions of RMI assume you should do this on both the client and the server
  - Unless your server also acts as a client, it isn’t really necessary on the server

The server class

- The class that defines the server object should extend `UnicastRemoteObject`
  - This makes a connection with exactly one other computer
  - If you must extend some other class, you can use `exportObject()` instead
  - Sun does *not* provide a `MulticastRemoteObject` class
- The server class needs to register its server object:
  - `String url = "rmi://" +` ***host*** `+ ":" +` ***port*** `+ "/" +` ***objectName***`;`
    - The default port is 1099
  - `Naming.rebind(url,` ***object***`);`
- Every remotely available method must throw a `RemoteException` (because connections can fail)
- Every remotely available method should be `synchronized`

Hello world server: interface

```
import java.rmi.*;

public interface HelloInterface extends Remote {
   public String say() throws RemoteException;
}
```

Hello world server: class

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

public class Hello extends UnicastRemoteObject
                          implements HelloInterface {
   private String message; // Strings are serializable

   public Hello (String msg) throws RemoteException {
      message = msg;
   }

   public String say() throws RemoteException {
      return message;
   }
}
```

Registering the hello world server

```
class HelloServer {
   public static void main (String[] argv) {
      try {
         Naming.rebind("rmi://localhost/HelloServer",
                                      new Hello("Hello, world!"));
         System.out.println("Hello Server is ready.");
      }
      catch (Exception e) {
         System.out.println("Hello Server failed: " + e);
      }
   }
}
```

The hello world client program

-
  ```
  class HelloClient {
     public static void main (String[] args) {
        HelloInterface hello;
        String name = "rmi://localhost/HelloServer";
        try {
           hello = (HelloInterface)Naming.lookup(name);
           System.out.println(hello.say());
         }
         catch (Exception e) {
           System.out.println("HelloClient exception: " + e);
         }
      }
  }
  ```

rmic

- The class that implements the remote object should be compiled as usual
- Then, it should be compiled with rmic:
  - rmic Hello
- This will generate files Hello_Stub.class and Hello_Skel.class
- These classes do the actual communication
  - The “Stub” class must be *copied* to the client area
  - The “Skel” was needed in SDK 1.1 but is no longer necessary

Trying RMI

- In three different terminal windows:
  1. Run the registry program:
     - `rmiregistry`
  2. Run the server program:
     - `java HelloServer`
  3. Run the client program:
     - `java HelloClient`

- If all goes well, you should get the “Hello, World!” message

Summary

1. Start the registry server, rmiregistry
2. Start the object server
    1. The object server registers an object, with a name, with the registry server
3. Start the client
    1. The client looks up the object in the registry server
4. The client makes a request
    1. The request actually goes to the Stub class
    2. The Stub classes on client and server talk to each other
    3. The client’s Stub class returns the result

References

- **Trail: RMI**
  by Ann Wollrath and Jim Waldo
  - http://java.sun.com/docs/books/tutorial/rmi/index.html

- **Fundamentals of RMI Short Course**
  by jGuru
  - http://developer.java.sun.com/developer/onlineTraining/rmi/RMI.html

- **Java RMI Tutorial**
  by Ken Baclawski
  - http://www.ccs.neu.edu/home/kenb/com3337/rmi_tut.html