COMP241
Software Engineering Development
Lecture 22: Distributed Computing 2

Mark Hall

- Remote Method Invocation (RMI)
  - Dynamic class loading

THE UNIVERSITY OF WAIKATO
DEPARTMENT OF COMPUTER SCIENCE
TARI ROROHIKO

Last time…

- Looked at Java's RMI:
  - Helper objects
  - `Remote` interface
  - Remote implementation
  - `rmic`
  - `rmiregistry`

Complete code for the server side

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

public class MyRemoteImpl
  extends UnicastRemoteObject
  implements MyRemote

  public String sayHello() {
    return "Server says, 'Hey'";
  }

  public MyRemoteImpl() throws RemoteException { }

  public static void main (String[] args) {
    try {
      MyRemote service = new MyRemoteImpl();
      Naming.rebind("RemoteHello", service);
    } catch (Exception ex) {
      // Try and bootstrap the rmiregistry...
      try {
        System.err.println("Attempting to start rmi registry...");
        java.rmi.registry.LocateRegistry.createRegistry(1099);
        MyRemoteImpl service = new MyRemoteImpl();
        Naming.rebind("RemoteHello", service);
        System.out.println("MyRemote bound in RMI registry");
      } catch (Exception ex2) {
        // not sure what is wrong now
        ex2.printStackTrace();
      }
    }
  }
}
```

Extending UnicastRemoteObject is the easiest way to make a remote object

You MUST implement your remote interface!

How does the client get the stub object?

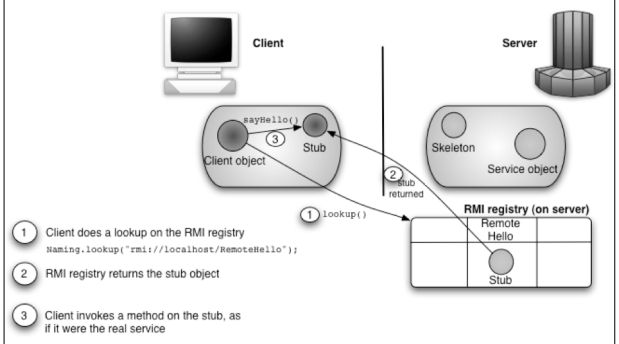
- The client needs to get the stub object
  - Client will call server methods on this
  - Client does a "lookup" in the RMI registry

```
MyRemote service = (MyRemote) Naming.lookup("rmi://localhost/RemoteHello");
```

Client always uses the remote interface as the type of service. In fact, the client never needs to know the actual class name of your remote service.

You have to cast it to the interface, since the lookup method returns type Object.

This has to be the name that the service was registered under.



Complete client code

```
import java.rmi.*;

public class MyRemoteClient {

  public static void main(String [] args) {
    new MyRemoteClient().go();
  }

  public void go() {
    try {
      MyRemote service = (MyRemote) Naming.lookup("rmi://localhost/RemoteHello");

      String s = service.sayHello();

      System.out.println(s);
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Class Files

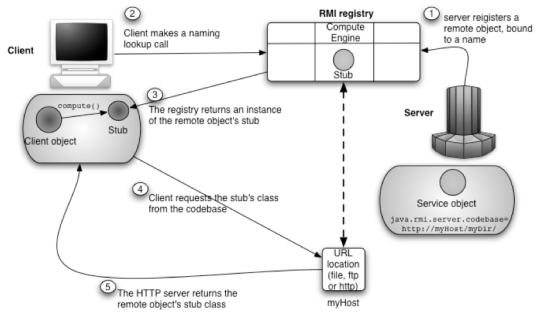
- Be sure each machine has the class files it needs
- Top three mistakes with RMI
  - Forgetting to start the rmiregistry before starting the remote service
    - Bootstrapping the registry eliminates this
  - Forgetting to make arguments and return types serializable
  - Forgetting to give the stub class to the client

RMI dynamic class loading

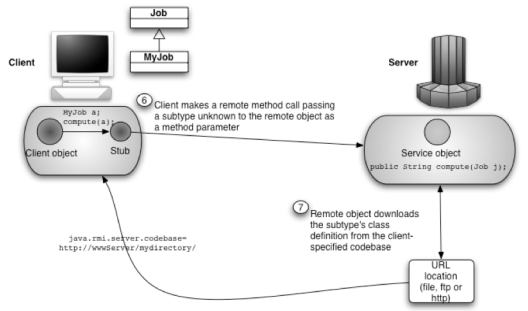
- RMI can download classes across the network to the service's VM
  - Eliminate the need to package the **stub** class with the client application
  - Essential if requests to the server involve custom classes that are not part of the JRE
    - I.e. client sends an object as a parameter to a method on the remote server where that objects class is not in the classpath of the server
    - The class of the object sent by the client will be a ***subtype*** of the declared parameter type
      - Implementation of an **interface** or a **subclass** of the method parameter

`java.rmi.server.codebase=<URI>`

Getting the stub class dynamically



Getting classes from the client dynamically



Specifying the codebase

- Set the "codebase" property from the command line when starting the client/server

```
java -Djava.rmi.server.codebase=http://webvector/export/

Java -Djava.rmi.server.codebase=http://web/pub/mystuff.jar
```


Test 2

- Covers everything from lecture 12 onwards
  - GUIs, event handling, swing, text I/O, sockets, threads and RMI
- Similar layout to test 1
- 5 questions
  - 1 "fill in the blanks" question
  - 1 "find the errors in this code" question
  - 2 "write code from scratch" questions
  - 1 "draw the output of these code snippets" question