UNIVERSITY OF MARYLAND

Lecture 23
Inside Java RMI

Recall

- Java RMI applications consist of three entities
  - Remote object servers
    - Host remote objects
    - Handle communications coming into / going out from these objects
  - Clients
    - Issue calls to methods in remote objects
    - Handle communications going out to / coming in from these objects
  - Object registries
    - Maintain bindings between names, remote objects
    - Have a “bootstrapping” function
      - Provide initial access point for clients needing remote objects
      - Once a client has stub for such an object, no need to consult registry!
- We’ve seen how some of this works
  - Marshaling / unmarshaling via serialization
  - Registration of remote objects in registries
  - Stubs

RMI in More Detail

- "Remote-izing" (aka *exporting*) an object
  - Server for a remote object must set up infrastructure for unmarshaling arguments to remote object, marshaling results
  - Server must also invoke method in remote object
- Registering a remote object
  - Server must provide name, stub for remote object
  - Name / object binding must be made available to potential clients
- Accessing a remote object
  - Client must be able to access remote object
  - Client must be able to marshal arguments to remote object method, unmarshal results

Exporting an Object

- Recall: "exportable" objects must come from a class that implements the `Remote` interface
- What does an exported object need?
  - Incoming method-invocation requests need to be listened for
  - Arguments need to be unmarshaled
  - Methods in the actual object need to be invoked
  - Results need to be marshaled, returned to calling side
- `java.rmi.server.UnicastRemoteObject` provides support for this
  - A class of remote objects
  - Constructor sets up infrastructure for listening for incoming method-invocations, marshaling / unmarshaling, actual method invocation
  - Class also includes

    ```
    static RemoteStub exportObject(Remote obj)
    ```
  - This method exports any object matching the `Remote` interface, returning a stub
    - Stub matches same (sub)interface of Remote that original object does
    - Stub is used on client side
- So: two ways to export an object
  - Make a class that extends `UnicastRemoteObject`; objects created in such a class are automatically exported
  - Make a class that implements `Remote`, then export using `UnicastRemoteObject.exportObject`

Behind the Scenes with `exportObject()`

- What does `exportObject()` do?
  - A new connection (*server socket*) to a port on the host is created
  - A thread is created to listen for connections on the socket
  - Infrastructure is also created to allocate other threads to handle incoming calls (this enables first thread to continue listening)
  - A stub is created for use on the client side
    - Contains information about which socket the remote object listens on
    - Implements argument mashaling, result unmarshaling
  - A *skeleton* is created for use on the host side

    Handles argument umarshaling, result marshaling of results, actual call of method
- Note: multiple threads can access a remote object!
  - If there are multiple method invocations in progress, more than one thread on the host will be accessing the object
  - You must ensure remote objects are thread-safe!

Stubs and Skeletons

- Both are objects in classes that are (as of Java 5) created automatically from the class of the object being exported
  - If the class of the remote object is Foo …
  - … then the class for stubs is Foo_Stub and the class for skeletons is Foo_Skel
  - Before Java 5 the programmer had to create these using a command rmic
- Both objects handle marshaling, unmarshaling
- Stub class also implements same `Remote` (sub)interface that original class does

Object Registration

- Purpose of registration is to make it easier for clients to use remote objects
- RMI uses *object registries* to handle this task
  - Object registry is a separate thread from server
  - Often, it even runs in a different JVM from the server, but it must run on the same host as server
  - This last fact implies that there can be several different registries running in a given application
- The registry binds names (strings) to stubs for remote objects
- Clients looking for remote objects must first locate a registry

Creating Object Registries

- Approach 1: rmiregistry
  - A command distributed with Java that can be run at the OS command line to start a registry
  - Takes an option argument: the port on which to listen for requests (default is 1099)
  - The registry created in this fashion continues to run even if the server JVM terminates
    - Such a registry can be used by several servers
    - If a server terminates, any references to remote objects on the server in such a registry will continue to exist
    - If a client tries to access such a nonexistent remote object, an exception is thrown
- Approach 2: `LocateRegistry.createRegistry()`
  - `LocateRegistry` is a class in `java.rmi.registry`
  - `static Registry createRegistry (int port)`
    - (From Java 7 documentation): "Creates and exports a `Registry` instance on the local host that accepts requests on the specified port"
    - "Exports" here means "makes the object remote"
    - What is returned is a stub for the remote `Registry` instance!

Other Useful Methods in `java.rmi.registry.LocateRegistry`

- `LocateRegistry` also includes methods for finding existing registries
- Samples:
  - `public static Registry getRegistry()throws RemoteException`

    (From the Java 7 documentation) "Returns a reference to the remote object Registry for the local host on the default registry port of 1099"
  - `public static Registry getRegistry(int port) throws RemoteException`

    Like above version, except that the given port is used
  - `public static Registry getRegistry(String host, int port) throws RemoteException`

    Live previous versions, except that given host name is used rather than local host

Useful Methods in `java.rmi.registry.Registry`

- `Registry` is an interface that extends `Remote`
- Registry objects allow remote objects to be bound to / unbound from names, names to be looked up, etc.
- Useful methods (quoted from Java 7 documentation)

  `void bind(String name, Remote obj)`

  Binds a remote reference (i.e. stub) to the specified name in this registry

  `String[] list()`

  Returns an array of the names bound in this registry

  `Remote lookup(String name)`

  Returns the remote reference (i.e. stub) bound to the specified name in this registry

  `void rebind(String name, Remote obj)`

  Replaces the binding for the specified name in this registry with the supplied remote reference

  `void unbind(String name)`

  Removes the binding for the specified name in this registry

rmi.Naming

- Registry objects allow names to be looked up, bound, etc.
  - These are instance methods, so the registry object must first be retrieved, e.g.
    ```
    Registry registry = LocateRegistry.getRegistry();
    registry.rebind(name, object);
    ```
  - Remote registries often accessed via URLs and port numbers, e.g.
    ```
    Registry registry = LocateRegistry.getRegistry("www.cs.umd.edu",1099);
    ```
- The class `Naming` includes "shortcuts" for these registry manipulations
  - The versions of bind / rebind / etc. assume the string argument is a URL including a host name, port and object name
  - The appropriate registry is also obtained automatically
  - So, instead of
    ```
    Registry registry = LocateRegistry.getRegistry("www.cs.umd.edu",1099);
    registry.rebind("foo", object);
    ```
    You can do this:
    ```
    Naming.rebind("//www.cs.umd.edu:1099/foo", object);
    ```
  - If you leave off the "`//www.cs.umd.edu:1099`" part, `Naming.rebind` will look for the registry running on the local host on the default port (1099)

CLASSPATH Considerations

- Registries associate stubs to names
- Stubs are objects created from classes that are automatically constructed from the remote object's class
- For a registry to store an object, it needs access to the object's class!
  - Objects store fields
  - Classes store methods
- How to do this?
  - If the registry and the server are on the same host, make sure the CLASSPATH of the registry includes the directories used by the server
  - If the registry and server are on different hosts, use the `java.rmi.server.codebase` property of the JVM that is running the server

Property??
`java.rmi.server.codebase`??

- Properties are name/value pairs that define information about a JVM and its environment
  - E.g. library paths, OS, etc.
  - Inside Java, to get current properties, can execute `System.getProperties()`
- Codebase property of JVM gives (space-separated) list of URLs from which class files published by JVM can be downloaded
- Code base properties can be set in special profile files, inside Java program, or at command line

  E.g.
  - Following command-line entry specifies the given URL as a code base

    java - Djava.rmi.server.codebase=http://webline/public/mystuff.jar foo
  - Following command in Java does the same:

    System.setProperty ("java.rmi.server.codebase", "http://webline/public/mystuff.jar");
- How does this help?
  - When a JVM publishes a stub to a registry, the codebase of the object's class is also provided as an annotation
  - To find the class of the stub object the registry JVM will first try to consult its own CLASSPATH
  - If it cannot find the class in its CLASSPATH, it next looks in the object's codepath annotation (provided by the server)!
  - BE CAREFUL! IF THE REGISTRY JVM'S CLASSPATH CONTAINS A CLASS OF THE SAME NAME AS THE STUB, THE REGISTRY WILL GET ITS LOCAL CLASS RATHER THAN THE (CORRECT) REMOTE ONE!!

Security

- Downloading remote code poses security risks
  - The code may be buggy
  - The code may be malicious
- Java enables the definition of security managers and policies to limit the access downloaded code has to system resources
- These can be defined in system properties like `java.security.manager` and `java.security.policy`
- Your JVM may have these set already, and they may interfere with remote downloading
- You can also set a security manager in your Java program using `System.setSecurityManager()`

Accessing Remote Objects

- To use a remote object, a client must:
  - Get a registry that has a name assigned to the object
  - Perform a lookup on the registry using the given object's name
    - Note: `registry.lookup()` returns objects of type `Remote`
    - You must cast the object to the particular subclass of `Remote` that the object's class implements
- If successful, the registry will return a stub to the given object
- To invoke the remote object's method, invoke the method of the same name on the stub!
- E.g. (from TestStringClient.java; recall `TestString` extends `Remote`)

```
Registry registry = LocateRegistry.getRegistry(host);
TestString stub1 = (TestString) registry.lookup("Test1");
…
String response = stub1.getTestString();
```

- Same CLASSPATH considerations as for registries apply. Also
  - If client uses local classes to create objects to pass (via marshaling / unmarshaling) to server …
  - … then client's class files must be accessible to server (either via CLASSPATH or codepath property of client)