CMSC 433 – Programming Language Technologies and Paradigms
Fall 2004

Java RMI
Nov. 30, 2004

Distributed Computing
• Programs that cooperate and communicate over a network
  – E-mail
  – Web server and web client
  – SETI @Home

Key Features of Distrib. Comp.
• Machines are not all the same
  – But all adhere to same communication protocol
• Network is “slow”
  – Sending a message takes a lot of time
• Network is unreliable
  – Machines may join and leave with no warning
  – Part of the network may fail

Different Approaches to Distributed Computation
• Connecting via sockets
  – E.g., project 1
  – Custom protocols for each application
• RPC/DCOM/CORBA/RMI
  – Make what looks like a normal function call
  – Function actually invoked on another machine
  – Arguments are marshalled for transport
  – Value is unmarshalled on return
Remote Method Invocation

- Easy way to get distributed computation
- Have stub for remote object
  - Calls to stub get translated into network call
  - Implemented on top of sockets
- Arguments and return values are passed over network
  - Java takes care of the details

A Simple Example

```java
class ChatServerImpl ...
    public void say(String s) {
        System.out.println(s);
    }
...
class Chatter {
    public static void main(String... args) {
        ChatServer c = // get remote object;
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        while (true) {
            System.out.print(">");
            c.say(br.readLine());
        }
    }
```

Remote Objects

- Object should
  - Extend `java.rmi.server.UnicastRemoteObject`
    - Constructor declared to throw `RemoteException`
  - Implement a remote interface
    - A remote interface extends `java.rmi.Remote`
    - All methods in a remote interface throw `RemoteException`
      - “Something bad happened on the network”
  - Side note: actually, don’t need to extend `UnicastRemoteObject`, but it’s much easier

Remote Interfaces
Stubs

- Client only sees the RemoteInterface
  - ConcreteObject can have other methods

- Remote objects represented using stub
  - Stub sends arguments over network
  - Stub receives result back from network

Compiling Stubs with rmic

- Generates stub code for a class
  - For 1.1, also generates skeleton class
    - Stub on client side communicates with skeleton on remote side
  - Skeleton not needed for 1.2+
    - And 1.2+ generates position-independent code
    - Use -v1.2 if you want

- Generates stubs for all methods declared in the class’ Remote interface
  - Other methods don’t get a stub

Passing Arguments

- To pass an argument to a remote method
  - (Or return a result from a remote method)
  - It must be either
    - A primitive type (int, double, etc.),
    - Serializable (e.g., String), or
    - Remote (i.e., implement a sub-interface of Remote)
  - Primitives passed as you’d expect

Passing Serializable vs. Remote

- Serializable objects passed by value
  - Same Serializable in different calls materializes different objects at receiver

- Remote objects passed by reference
  - Same Remote object in different calls yields same stub object, which passes arguments back to same remote object
### Stub Code

- Objects contain both data and code
  - When you receive a remote object, you need the stub for that remote object

- Solution #1: All clients have stub code on their classpath
  - Or stub code for another class with same remote interface

### Downloading Code

- Solution #2: Provide a code base where stub code for objects can be downloaded
  ```java
  java -Djava.rmi.server.codebase=<url> ...
  ```
  - Specifies location of classes originating from this server
  - `url` can be, e.g., `http://` or `file://`

### Security Manager

- Downloading code (even stub code) from the internet is potentially risky
  - Need to limit what downloaded code could do
  - Must install a Security Manager before you download any code from RMI code bases

- Can use
  ```java
  System.setSecurityManager(
    new RMISecurityManager());
  ```

### Policy Files

- In addition to security manager, need to specify a security policy
  ```java
  grant {
    permission java.net.SocketPermission "*:1024-65535", "connect,accept";
    permission java.net.SocketPermission "+:80", "connect";
  };
  ```

- Set security policy when JVM started
  - `java -Djava.security.policy=<file name>`
Getting the First Remote Object

- Can make objects available in RMI registry
  - Each object has a name (that you specify)
  - Registry listens on a port (1099 default)

- Naming.lookup(url) gets object from reg.
  - E.g., Naming.lookup("rmi://localhost/Chat");
  - Use to get first reference to remote object
  - Don’t need to lookup for objects returned by remote methods

Starting an RMI Registry

- Method 1: Separate RMI registry process
  - Command rmiregistry
    - Run with stubs in classpath, or specify codebase
    - Listens on port 1099 by default

- Method 2: Start in same JVM
  - LocateRegistry.createRegistry(int port)
  - Advantage: dies when your program dies
    - No registries lying around on machine

Advertising Remote Objects

- Call Naming.{bind/unbind/rebind} to place objects in registry
  - E.g., Naming.bind("rmi://localhost/Chat");

- Can bind/unbind/rebind name on localhost
- Can lookup name on any host

Example: RMI Chat Server

- Server
  - Runs the chat room

- Client
  - Participant in chat room
  - Receives messages from others in room

- Connection
  - Uniquely identifies a client
  - Used to speak in chat room
Server

interface Server extends Remote {
    Connection logon(String name, Client c)
        throws RemoteException;
}

Connection

interface Connection extends Remote {
    /** Say to everyone */
    void say(String msg)
        throws RemoteException;
    /** Say to one person */
    void say(String who, String msg)
        throws RemoteException;
    String [] who()
        throws RemoteException;
    void logoff()
        throws RemoteException;
}

Client

interface Client extends Remote {
    void said(String who, String msg)
        throws RemoteException;
    void whoChanged(String [] who)
        throws RemoteException;
}

Server’s Remote Object creation

Server s = new ServerImpl();

Object added to table because it implements extension of Remote interface
Remote Object registry

Naming.rebind("ChatServer", s).

ServerImpl

Hosting Remote Objects

Server

RMI Registry

ChatServer

ServerImpl Stub

Client’s Remote Object creation

Client c = new ClientImpl();

ClientImpl

Hosting Remote Objects

Client

Hosted Remote Objects

Client object also implements extension of Remote interface

Server looks up Server

Server s = (Server) Naming.lookup("/host/ChatServer");

ServerImpl Stub

RMI Registry

ServerImpl

Server

After lookup finished

ClientImpl

Hosting Remote Objects

Client

Hosted Remote Objects

ServerImpl

ServerImpl Stub

Server

Client

Hosted Remote Objects

ServerImpl

Server

ClientImpl
Invokes remote Server method

Connection conn = s.logon(“Bill”, c);

Executes the call

... create new Connection object

Returns the result

... return this as the result
Receives the result

Stub code for remote logon call

From server process

ServerImpl Stub

Conn Stub

Return value: Stub for conn

unmarshalled return value

Client