Java RMI
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 marshallled for transport
  – Value is unmarshallled 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
// runs on one mach.
class ChatServerImpl implements ChatServer ... {
    public void say(String s) {
        System.out.println(s);
    }
    ...
}
class Chatter { // runs on another mach.
    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
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

• Don’t need to use rmic anymore
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
• 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
• Solution #2: Provide a code base where stub code for objects can be downloaded
  
  java -Djava.rmi.server.codebase=<url> ...

  – Specifies location of classes originating from this server
  – URL can be, e.g., http:// or file://
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 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
interface Server extends Remote {

    Connection logon(String name, Client c)
    throws RemoteException;

}
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;
}
interface Client extends Remote {

    void said(String who, String msg) throws RemoteException;

    void whoChanged(String[] who) throws RemoteException;

}
Server’s Remote Object creation

Server

Server s = new ServerImpl();

Hosted Remote Objects

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

```java
Naming.rebind("ChatServer", s);
```

Server

```
ServerImpl
```

Hosted Remote Objects

```
Stub
```

RMI Registry

```
ChatServer
```

```
ServerImpl
```
Client’s Remote Object creation

Client object also implements extension of Remote interface

Client c = new ClientImpl();
Client looks up Server

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

ServerImpl Stub

lookup

ChatServer

ServerImpl Stub

RMI Registry

Hosted Remote Objects

ServerImpl

Server

returns stub
After lookup finished

Client

ServerImpl Stub

Hosted Remote Objects

ServerImpl

Hosted Remote Objects

Server
Invokes remote Server method

Connection

```
conn = s.logon("Bill", c);
```

Client

```
ClientImpl
```

```
ServerImpl
```

```
Stub code for remote logon call
```

to server process

```
Method: logon
Stub for c
String "Bill"
```

logon
Receives remote call

(Skeleton) code for remote logon call

Method: logon
Stub for c
String “Bill”

... from client process

“Bill”
ClientImpl
Stub c

unmarshalled arguments

Hosted Remote Objects

ServerImpl
Executes the call

... create new Connection object

ConnectionImpl

"Bill"

ClientImpl Stub c

call logon ...

ServerImpl

Hosted Remote Objects
Returns the result

Server

... return this as the result

ConnectionImpl

(Skeleton) code for remote logon call

Hosted Remote Objects

ServerImpl

Return value:
Stub for conn

... to client process
Receives the result

`ServerImpl Stub` receives the result of the `logon` call and passes it to the `Conn Stub`. The return value is unmarshalled.

Client side diagram:
- `conn` is passed to `ServerImpl Stub`.
- `logon` is called on `ServerImpl Stub`.
- The result is marshalled and passed to `Conn Stub`.
- `Conn Stub` receives the result and passes it back to the client.

Arrow labels:
- From `ServerImpl Stub` to `Conn Stub`: `... from server process`
- From `Conn Stub` to client: `unmarshalled return value`
Security Manager

• When using a code base, we must download stub code from a remote site. This 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

  System.setSecurityManager(new RMISecurityManager());
Policy Files

• In addition to security manager, need to specify a security policy

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>