CMSC433, Spring 2001 Programming Language Technology and Paradigms Java RMI

Alan Sussman

October 25, 2001

Administrivia

- Read Chapter 15, pages 973-979, on RMI
- Project 3 due tomorrow
- Project 2 grades out, but will need some regrading
- Project 4 due November 7
  - distributed chat server, using RMI
  - posted by tomorrow, central server running soon

Last time

- Multithreaded programming
  - and the Java memory model
  - Motto is: Be careful!

Different Approaches to Distributed Computation

- High-performance, parallel scientific apps
- Connecting via sockets
  - custom protocols for each application
- RPC/DCOM/CORBA/RMI
  - make what looks like a normal function call
  - function is actually invoked on another machine
  - Arguments are *marshalled* for transport
  - return value is marshalled as well

Remote Method Invocation

- Easy way to get distributed computation
- Have stub for remote object
  - calls to stub get translated into network call
- Arguments and return values can be passed over network

Remote Objects and Interfaces

- Remote Objects are those that can be referenced remotely
  - extend **java.rmi.UnicastRemoteObject**
  - constructor throws **java.rmi.RemoteException**
- Remote interfaces describe services that can be provided remotely
  - extend **java.rmi.Remote** interface
  - all methods throw **java.rmi.RemoteException**

RMIC - RMI Compiler

- Generates stub code for a class
  - For 1.1, also generates skeleton class
  - skeleton not needed for 1.2+
- Generates stubs for all methods declared in remote interfaces
  - other methods don't get a stub

Passing arguments

- Can pass arbitrary values as arguments
- Can return arbitrary values as results
- To pass a value, it must either be
  - **Serializable**, or
  - **Remote**
- Passing the same Serializable object in different calls
  - will materialize different objects at receiver

Downloading code

- When you pass a reference to a remote class
  - receiver (the method being called) needs stub class code (class file generated by *rmic*)
- When you pass a ref to serializable class
  - receiver needs class (the class file from *javac*)
- References to remote and serializable classes annotated with *RMI codebase*
  - where code can be loaded from

SecurityManager

- Must install some Security Manager to allow download of classes from RMI codebase
- Can use **RMISecurityManager**
  System.setSecurityManager(new RMISecurityManager());
- Modify policy file to grant permissions
  - example of that on Lectures web page

Naming.lookup

- Naming.lookup is used to bootstrap RMI communication
  - Get your first reference to a remote object
- Someone runs an RMIRegistry
  - a separate Java VM (or can run inside a JVM also running something else)
  - that listens to a particular port (default 1099)
- Can bind/unbind/rebind name on localhost (the one running the registry)
- Can lookup name on any host

RMI Chat server example

- 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

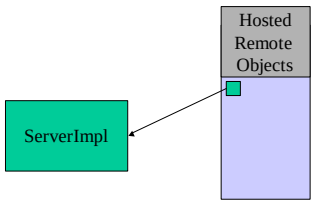
```
interface Connection extends Remote {
        /** Say to everyone */
        void say(String msg)
                throws RemoteException;
        / ** Say to one person */
        void say(String who, String msg)
                throws RemoteException;
        String [] who()
                throws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


Client

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

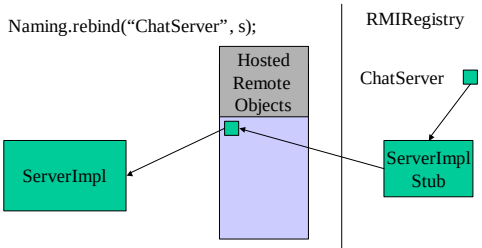

Remote Object creation on Server host

Server s = new ServerImpl();



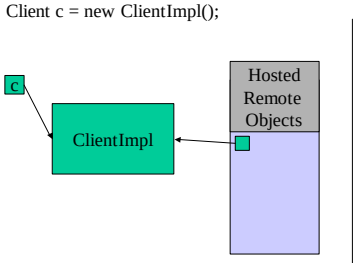
RMI Registry on Server host

Naming.rebind("ChatServer", s);



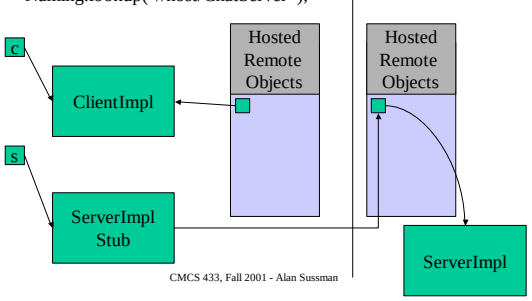
Client creation on Client host

Client c = new ClientImpl();



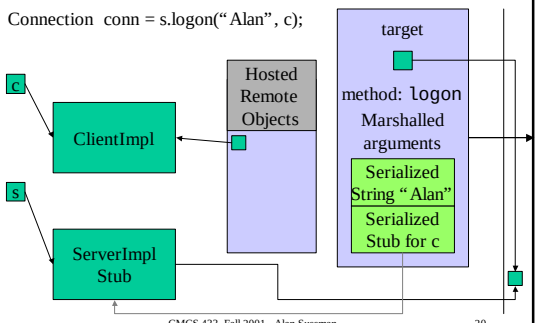
Server Lookup on Client host

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

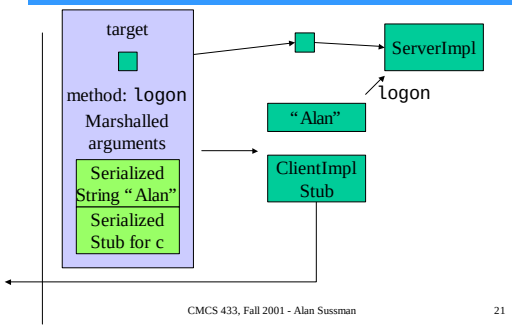


Invoke message on Server from Client host
prepare call on client

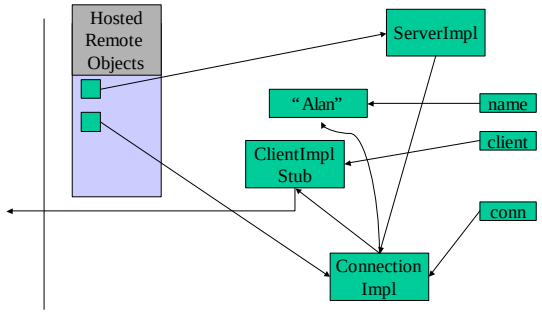
Connection conn = s.logon("Alan", c);



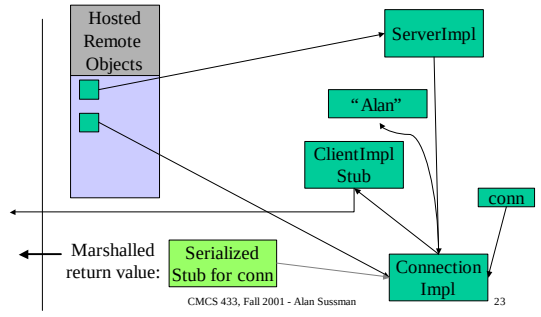
Unmarshalling arguments on Server host



Execution on Server host



Marshalling return value on Server host



Unmarshalling return value on Client host