COMP241
Software Engineering Development
Lecture 21: Distributed Computing

Mark Hall

- Chat client and Server—final version
- Remote Method Invocation (RMI)

THE UNIVERSITY OF WAIKATO
DEPARTMENT OF COMPUTER SCIENCE
TARI ROROHIKO

New and improved SimpleChatClient

- Now we can finally put together the complete multithreaded chat client and server program

```
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;


public class SimpleChatClient {

  JTextArea incoming;
  JTextField outgoing;
  BufferedReader reader;
  PrintWriter writer;
  Socket sock;

  public static void main(String[] args) {
    SimpleChatClient client = new SimpleChatClient();
    client.go();
  }
```

```
public void go() {

  // build gui
  JFrame frame = new JFrame("Ludicrously Simple Chat Client");
  JPanel mainPanel = new JPanel();

  incoming = new JTextArea(15,50);
  incoming.setLineWrap(true);
  incoming.setWrapStyleWord(true);
  incoming.setEditable(false);

  JScrollPane qScroller = new JScrollPane(incoming);
  qScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
  qScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

  outgoing = new JTextField(20);

  JButton sendButton = new JButton("Send");
  sendButton.addActionListener(new SendButtonListener());

  mainPanel.add(qScroller);
  mainPanel.add(outgoing);
  mainPanel.add(sendButton);

  setUpNetworking();
  Thread readerThread = new Thread(new IncomingReader());
  readerThread.start();

  frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
  frame.setSize(650,450);
  frame.setVisible(true);
} // close go
```

We're starting a new thread, using a new inner class as the Runnable (job) for the thread. The thread's job is to read from the server's socket stream, displaying any incoming messages in the scrolling text area.

```
private void setUpNetworking() {
  try {
    sock = new Socket("127.0.0.1", 5000);
    InputStreamReader streamReader = new InputStreamReader(sock.getInputStream());
    reader = new BufferedReader(streamReader);

    writer = new PrintWriter(sock.getOutputStream());

    System.out.println("networking established");
  } catch(IOException ex) {
    ex.printStackTrace();
  }
} // close setUpNetworking

public class SendButtonListener implements ActionListener {
  public void actionPerformed(ActionEvent ev) {
    try {
      writer.println(outgoing.getText());
      writer.flush();

    } catch(Exception ex) {
      ex.printStackTrace();
    }
    outgoing.setText("");
    outgoing.requestFocus();
  }
}  // close SendButtonListener inner class
```



```
public class IncomingReader implements Runnable {
  public void run() {
    String message;
    try {

      while ((message = reader.readLine()) != null) {
        System.out.println("read " + message);
        incoming.append(message + "\n");

      } // close while
    } catch(Exception ex) {ex.printStackTrace();}
  } // close run
} // close inner class
} // close outer class (SimpleChatClient)
```

This is what the thread does.

In the run() method, it stays in a loop (as long as what it gets from the server is not null), reading a line at a time and adding each line to the scrolling text area (along with a new line character).

The VerySimpleChatServer

```
import java.io.*;
import java.net.*;
import java.util.*;

public class VerySimpleChatServer {

  ArrayList<PrintWriter> clientOutputStreams;
  public class ClientHandler implements Runnable {

    BufferedReader reader;
    Socket sock;

    public ClientHandler(Socket clientSocket) {
      try {
        sock = clientSocket;
        InputStreamReader isReader = new InputStreamReader(sock.getInputStream());
        reader = new BufferedReader(isReader);
      } catch(Exception ex) {ex.printStackTrace();}
    } // close constructor

    public void run() {
      String message;

      try {
        // keep reading stuff from the server
        while ((message = reader.readLine()) != null) {
          System.out.println("read " + message);
          tellEveryone(message);
        } // close while
      } catch(Exception ex) {ex.printStackTrace();}
    } // close run
  } // close inner class
```

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

public void go() {
  clientOutputStreams = new ArrayList<PrintWriter>();

  try {
    ServerSocket serverSock = new ServerSocket(5000);

    while(true) {
      Socket clientSocket = serverSock.accept();
      PrintWriter writer = new PrintWriter(clientSocket.getOutputStream());
      synchronized(this) {
        clientOutputStreams.add(writer);
      }

      Thread t = new Thread(new ClientHandler(clientSocket));
      t.start();

      System.out.println("got a connection");
    }
    // now if I get here I have a connection
  } catch(Exception ex) {
    ex.printStackTrace();
  }
}
```



```
  public synchronized void tellEveryone(String message) {
    Iterator<PrintWriter> it = clientOutputStreams.iterator();
    while(it.hasNext()) {
      try {
        PrintWriter writer = it.next();
        writer.println(message);
        writer.flush();
      } catch(Exception ex) {
        ex.printStackTrace();
      }
    } // end while
  } // close tellEveryone
} // close VerySimpleChatServer
```


Distributed Computing

- So far, we've seen that every method we've invoked has been on an object running in the same VM as the caller
- In a distributed computing scenario we have multiple VM's on multiple machines
  - Typically one or more servers providing heavy computing power for one or more clients
  - Client wants to be able to call a method on an object running on the server

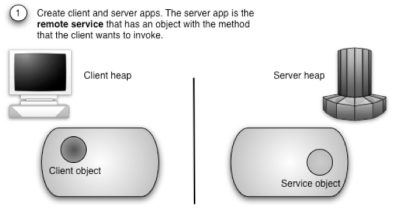
```
double doCalcUsingDatabase(CalcNumbers numbers)
```


Distributed Computing

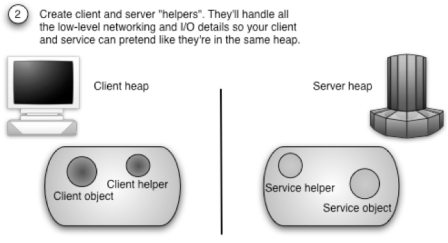
- How can object A on the client get a reference to object B on the server?
  - Two different heaps/JVMs involved
- Can't do it *directly*

A design for remote method calls

- Create four things: server, client, server helper and client helper

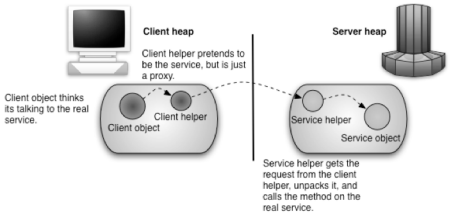


A design for remote method calls



The role of the 'helpers'

- The 'helpers' are the objects that actually do the communication
  - Client **acts** as though its calling a method on a local object
    - Actually it is—the client calls a method on the client helper
  - The client helper is a *proxy* for the real service



How the method call happens

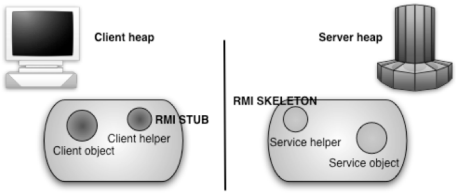
1. Client object calls `doBigThing()` on the client helper object
2. Client helper packages up information about the call (arguments, method name, etc.) and ships it over the network to the service helper
3. Service helper unpacks the information from the client helper, finds out which method to call (and on which object) and invokes the **real** method on the **real** service object

Java RMI provides the helper objects

- RMI does the hard work
  - Makes the client helper look like the real service (i.e. gives the client helper the same methods as the remote service)
  - Provides the runtime infrastructure
    - Lookup service to allow the client to find and get the client helper
    - Provides all of the networking and I/O code
- Lots of potential exceptions
  - Because ultimately, a call on the proxy involves sockets and streams
- Choice of *protocols*
  - JRMP: RMI's native protocol for Java-to-Java remote calls
  - IIOP: CORBA's protocol for Java-to-non-Java

Java RMI provides the helper objects

- In RMI, the client helper is a 'stub' and the server helper is a 'skeleton'



Overview of making a remote service

- Step one:
  - Make a **Remote Interface** (e.g. `MyService.java`)
    - Defines the method that a client can call remotely
    - Client will use this as the *polymorphic* class type for your service
    - Both the Stub and the actual service will implement this
- Step two:
  - Make a **Remote Implementation** (e.g. `MyServiceImpl.java`)
    - This is the actual class that does the real work
    - Has the actual implementation of the remote methods defined in the **remote interface**

Overview of making a remote service

- Step three:
  - Generate the **stubs** and the **skeletons** using **rmic**
  - These are the client and server 'helpers'
  - You don't have to create them or ever look at the source code that generates them
  - `rmic` (part of the JDK) does it for you
- Step four:
  - Start the **RMI registry** (`rmiregistry`)
- Step five:
  - Start the **remote service**
    - Your service implementation class instantiates an instance of the service and registers it with the RMI registry
    - Registering the service makes it available for clients

Make a Remote Interface

- Extend `java.rmi.Remote`
  - Remote is a 'marker' interface that declares no methods

```
Public interface MyRemote extends Remote {
```

- Declare that all methods throw a **`RemoteException`**
  - The remote interface is used by the client as the polymorphic type for the service
    - The stub implements this and uses sockets and I/O, hence the exception

```
import java.rmi.*;
public interface MyRemote extends Remote {
  public String sayHello() throws RemoteException;
}
```


Make a Remote Interface

- Be sure arguments and return values are primitives or `Serializable`
  - Transmission across the network requires that objects be converted to byte streams

Make a Remote Implementation

- Implement the Remote interface
  - Your service has to implement the remote interface

```
public class MyRemoteImpl extends UnicastRemoteObject
  implements MyRemote {
  public String sayHello() {
    return "Server says, 'Hey'";
  }
}
```

- Extend **`UnicastRemoteObject`**
  - Inherit some functionality related to 'being remote'

Make a Remote Implementation

- Write a no-arg constructor that declares a `RemoteException`
  - `UnicastRemoteObject`'s constructor throws a `RemoteException`, so our constructor has to throw it too

```
public MyRemoteImpl() throws RemoteException { }
```

- Register the service with the RMI registry
  - When you register the *implementation object*, the RMI system actually puts the *stub* in the registry

```
try {
  MyRemote service = new MyRemoteImpl();
  Naming.rebind("RemoteHello", service);
} catch (Exception ex) { ... }
```


Generate stubs and skeletons

- Run rmic on the remote implementation class (**not** the remote interface)

```
rmic MyRemoteImpl
```

  - Notice that you don't say ".class" on the end
  - Produces two new classes for the helper objects: `MyRemoteImpl_Stub.class` and `MyRemoteImpl_Skel.class`
- Run `rmiregistry` in a terminal
  - Make sure you start it from a directory that has access to your classes

Start the service

- From another terminal start your service
  - This will probably be from the `main()` method in your remote implementation class
  - Note, it is also possible to bootstrap the `rmiregistry` from your service implementation class (could put the following in your `main()`):

```
try {
 System.err.println("Attempting to start rmi registry...");
 java.rmi.registry.LocateRegistry.createRegistry(1099);
} catch (Exception ex) {…}
```

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!