COMP241

Software Engineering Development

Lecture 19: Networking and Threads

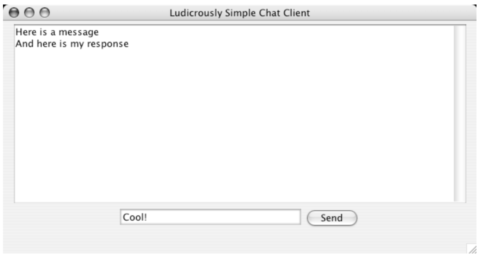
Mark Hall

- Client/Server
- Socket
  - Reading/writing
- Simple example
- ServerSocket
- Chat client
- Multithreading

THE UNIVERSITY OF WAIKATO

DEPARTMENT OF COMPUTER SCIENCE
TARI ROROHIKO

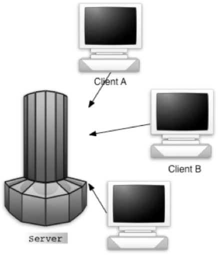
Chat Program



Here is a message
And here is my response

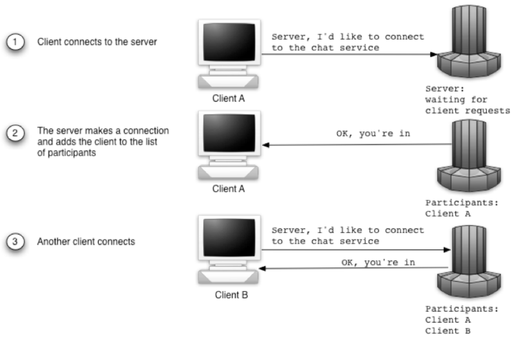
Chat Program Overview

- The client has to know about the server
- The server has to know about ALL the clients



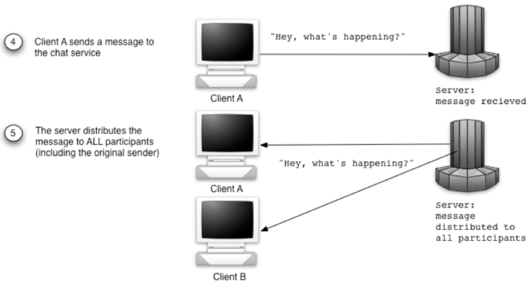
Chat Program Overview

- How it works:


1 Client connects to the server
Client A
Server, I'd like to connect to the chat service
Server: waiting for client requests
2 The server makes a connection and adds the client to the list of participants
Client A
OK, you're in
Participants: Client A
3 Another client connects
Client B
Server, I'd like to connect to the chat service
OK, you're in
Participants: Client A Client B

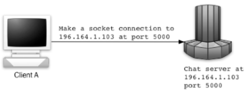
Chat Program Overview

- How it works:

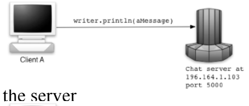

4 Client A sends a message to the chat service
Client A
"Hey, what's happening?"
Server: message recieved
5 The server distributes the message to ALL participants (including the original sender)
Client A
"Hey, what's happening?"
Client B
Server: message distributed to all participants

Connecting, Sending and Receiving

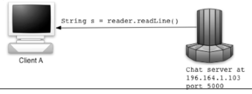
- Connect
  - Client connects to the server by establishing a **`Socket`** connection


Make a socket connection to 196.164.1.103 at port 5000
Client A
Chat server at 196.164.1.103 port 5000

- Send
  - Client **sends** a message to the server



- Receive
  - Client **gets** a message from the server


String s = reader.readLine()
Client A
Chat server at 196.164.1.103 port 5000

Make a network Socket connection

```
Socket chSock = new Socket(“196.164.1.103”, 5000);
```

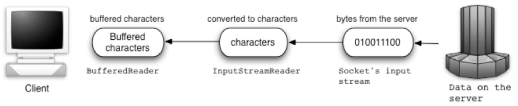
- A Socket connection means the *two* machines have information about each other
  - Network location (IP address)
  - TCP port
- TCP port
  - 16 bit number that identifies a specific program (service) on the server
  - 0 - 1023 are reserved for **well-known** services
    - 20 (FTP), 23 (Telnet), 25 (SMTP), 80 (HTTP), 443 (HTTPS) etc.

Reading from a Socket

- Socket provides an `InputStream` for reading (and an `OutputStream` for writing) from the network
- Last time we saw `InputStreamReader`—a bridge between byte-level and text input
  - We can use this to read text from the `Socket`
  - For writing text to a Socket we can use a `PrintWriter`

Reading from a Socket

```
Socket chSock = new Socket(“196.164.1.103”, 5000);
InputStreamReader strm =
  new InputStreamReader(chSock.getInputStream());
BufferedReader buffR = new BufferedReader(strm);
String message = reader.readLine();
```



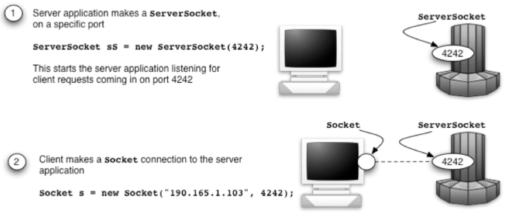
Simple Example

- The DailyAdvice server
  - A program that offers up practical, inspirational tips to get you through the day :-)
    - E.g. “Treat yourself to a cold one! You deserve it!”, “Tell your boss the report will have to wait. There’s powder at Aspen!”, “That shade of green isn’t really workin’ for you…” etc.
- DailyAdviceClient
  - Pulls a message from the server each time it connects

```
import java.io.*;
import java.net.*;
public class DailyAdviceClient {
  public void go() {
    try {
      Socket s = new Socket("127.0.0.1", 4242);
      InputStreamReader sR =
        new InputStreamReader(s.getInputStream());
      BufferedReader bR = new BufferedReader(sR);

      String advice = bR.readLine();
      System.out.println("Today you should: " + advice);

      bR.close();
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }

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

Writing a simple server

- How it works:


1 Server application makes a **ServerSocket**, on a specific port

```
ServerSocket sS = new ServerSocket(4242);
```

This starts the server application listening for client requests coming in on port 4242

2 Client makes a **Socket** connection to the server application

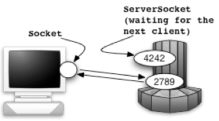
```
Socket s = new Socket("190.165.1.103", 4242);
```

Writing a simple server

3 Server makes a new **Socket** to communicate with this client

```
Socket s = sS.accept();
```

The `accept()` method blocks while its waiting for a client `Socket` connection. When a client finally tries to connect, the method returns a plain old `Socket` (on a *different* port) that knows how to communicate with the client (i.e. knows the *client's* IP address and port number). The `Socket` is on a different port than the `ServerSocket`, so that the `ServerSocket` can go back to waiting for other clients.



DailyAdviceServer code

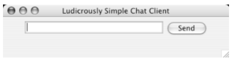
```
import java.io.*;
import java.net.*;
public class DailyAdviceServer {
  // daily advice comes from this array
  String [] adviceList = {"Take smaller bites", "Treat
yourself to a cold one!", Tell your boss what you *really*
think"};

  public void go() {
    try {
      ServerSocket sS = ServerSocket(4242);
      // The server goes into a permanent loop, waiting for
      // (and servicing) client requests
      while(true) {
        Socket s = sS.accept();
        PrintWriter w = new PrintWriter(s.getOutputStream());
        String advice = getAdvice();
        writer.println(advice);
        writer.close();
        System.out.println(advice);
      }
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
```

The `accept()` method blocks (just sits there) until a request comes in, and then the method returns a `Socket` (on some anonymous port) for communicating with the client.

Now we use the `Socket` connection to the client to make a `PrintWriter` and send it (`println()`) a String advice message. Then we close the `Socket` because we are done with this client.

DailyAdviceServer code

```
  private String getAdvice() {
    int random = (int) (Math.random() * adviceList.length);
    return adviceList[random];
  }

  public static void main(String [] args) {
    DailyAdviceServer server = new DailyAdviceServer();
    server.go();
  }
} // end class
```



Writing a Chat Client

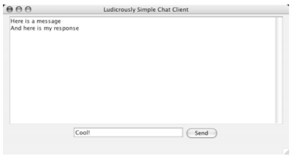
```
public class SimpleChatClientA {
  JTextField outgoing;
  PrintWriter writer;
  Socket sock;

  public go() {
    // make gui and register a listener with the send button
    // call setUpNetworking() method
  }

  public void setUpNetworking() {
    // make a Socket, then make a PrintWriter
    // assign the PrintWriter to writer instance variable
  }

  public class SendButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      // get text from the text field and send it to
      // the server using the writer (a PrintWriter)
    }
  } // close inner class
} // close outer class
```

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

public class SimpleChatClientA {
  JTextField outgoing;
  PrintWriter writer;
  Socket sock;

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

   public void go() {
     JFrame frame = new JFrame("Ludicrously Simple Chat Client");
     JPanel mainPanel = new JPanel();
     outgoing = new JTextField(20);
     JButton sendButton = new JButton("Send");
     sendButton.addActionListener(new SendButtonListener());
     mainPanel.add(outgoing);
     mainPanel.add(sendButton);
     setUpNetworking();
     frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
     frame.setSize(400,500);
     frame.setVisible(true);
   } // close go
```

```
    private void setUpNetworking() {
      try {
        sock = new Socket("127.0.0.1", 5000);

         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
  }
```

Writing a Chat Client

- Version Two: send and receive



Here is a message
And here is my response

Cool!

The server sends a message to all client participants, as soon as the message is received by the server. When a client sends a message, it doesn't appear in the display area until the server sends it to everyone.

- When do you get messages from the server?
  1. Option One: Poll the server every 20 seconds
  2. Option Two: Read something in from the server each time the user sends a message
  3. Option Three: Read messages as soon as they're sent from the server

Multithreading in Java

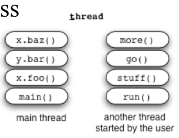
- Java has multiple threading built right into the fabric of the language

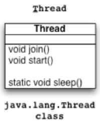
```
Thread t = new Thread();
t.start();
```

- By creating a new Thread *object*, you've launched a separate *thread of execution*, with its own call stack; except…
- The thread above doesn't actually *do* anything
  - The thread "dies" virtually the instant it's born
- Need a *job* for the thread to do

Multithreading

- Java has multiple threads but only one `Thread` class





- A thread is a separate thread of execution
- Every Java app starts up a main thread—the thread that puts the main() method on the bottom of the stack
  - The JVM is responsible for starting the main thread (and other threads, as it chooses, e.g. garbage collection thread

- **`Thread`** is a class that represents a thread of execution
  - Methods (amongst others) for starting, joining one thread with another and putting a thread to sleep