Lecture 5
JAVA (46-935)
Somesh Jha
What is a socket?

- A *socket* is a two-way communication link between two programs running on the network.

- A socket is bound to a port or an address so that the network layer knows where to send the data.
Client-Server Concepts

- A server runs on a specific computer or a host and has a socket that is bound to a specific port number.

- A port is like the local address of the socket on the host.

- A server just waits (listening on the socket) for a client to make a connection request.
A simple application

Figure 1: A simple Client-Server system
package threadRelated;

import java.io.*;
import java.net.*;

public class Client {
    public static final int DEFAULT_PORT = 6789;
    public static void usage() {
        System.out.println("Usage: java Client <hostname>[<port>] ");
        System.exit(0);
    }

    public static void main(String[] args) {
        int port = DEFAULT_PORT;
        Socket s = null;
        PrintWriter out = null;
        BufferedReader in = null;

        // Parse the port specification
        if ((args.length != 1) && (args.length != 2)) usage();
        if (args.length == 1) port = DEFAULT_PORT;
        else {
            try { port = Integer.parseInt(args[1]); } 
            catch (NumberFormatException e) { usage();}
        }

        try {
            // Create a socket to communicate to the specified host and port
            s = new Socket(args[0], port);
            // Create streams for reading and writing lines of text
            // from and to this socket
InputStreamReader tempReader =
    new InputStreamReader(s.getInputStream());

in = new BufferedReader(tempReader);

out = new PrintWriter(s.getOutputStream(),true);

BufferedReader bSystemIn =
    new BufferedReader(new InputStreamReader(System.in));

    //Tell the user that we've connected
    System.out.println("Connected to "+s.getInetAddress()
    +"."+s.getPort());

    String line;
    while(true) {
        //print a prompt
        System.out.print(">");
        System.out.flush();
        // read a line from the console; check for EOF
        line = bSystemIn.readLine();
        System.out.println("(Client) Read line : "+line);
        if (line == null) break;
        //Send it to the server
        out.println(line);
        out.flush();
        //Read a line from the server
        line = in.readLine();
        //Check if connection is closed (i.e. for EOF)
        if (line == null) {
            System.out.println("Connection closed by server.");
            break;
        }
        //And write the line to the console
        System.out.println("Server says: "+line);
        System.out.flush();
    }

    catch (UnknownHostException e) {
System.err.println("Couldn't find host "+args[0]);
}
catch (IOException e) {
    System.err.println("Error in performing I/O: "+e.getMessage());
    System.exit(i);
}
//Always be sure to close the sockets and the streams
finally {
    try {
        if(s != null) {
            out.close();
            in.close();
            s.close();
        }
    }
    catch (IOException e2) {
        System.err.println("Error in closing streams and sockets ");
    }
}
} //end of class Client
Client Program

- Pass the host name (on which the server is running) and the port number to the main program.

- If the port number is not passed, a default port number is used.

- Try to create a connection with the server.

```java
// Create a socket to communicate to the specified host and port
s = new Socket(args[0], port);
```
Client Program (contd)

- If the connection succeeds, we have socket (a two way connection) between the client and the server.

- Variable \( s \) contains the connection or the socket between the client and the server.

- We will convert the connection or the socket into input or output streams. The fragment of code that does that is:

```java
//Create streams for reading and writing lines of text
// from and to this socket
InputStreamReader tempReader =
    new InputStreamReader(s.getInputStream());

in = new BufferedReader(tempReader);

out = new PrintWriter(s.getOutputStream(),true);
```
Client side I/O

- Whenever we read in we are actually reading from the socket or the connection.

- If there is nothing to read, the statement blocks.

- Whenever we write to out we actually write to the socket or the connection and hence the data will reach the server.
While loop

- The client sits in a loop reading lines from the screen.

- Each line that is read is sent to the server and the answer (supposed to be the reversed line) is read from the server response.

- The reversed line is output on the screen.

- If the client receives a null input, it exits.
finally clause

- Notice the finally clause:

```java
finally {
    try {
        if(s != null) {
            out.close();
            in.close();
            s.close();
        }
    }
    catch (IOException e2) {
        System.err.println("Error in closing streams and sockets ");
    }
}
```

- Recall that the finally clause is called when we are exiting the try block (either because of an exception or normally).

- It is good practice to close all the streams and sockets before you exit the program or a method (if they are not going to be used anywhere else).
package threadRelated;

import java.io.*;
import java.net.*;

public class Server extends Thread {

    public final static int DEFAULT_PORT = 6789;
    static final boolean DEBUG=true;

    protected int port;
    protected ServerSocket listen_socket;

    //Exit with an error message, when an exception occurs
    public static void fail(Exception e, String msg) {
        System.err.println(msg+" ".e);
        System.exit(1);
    }

    //Create a ServerSocket to listen for connections on; start the thread
    public Server(int port) {
        if (port == 0) port = DEFAULT_PORT;
        this.port = port;
        try {
            listen_socket = new ServerSocket(port);
        } catch (IOException e) {
            fail(e,"Exception creating server socket");
        }
        System.out.println("Server: listening on port "+port);
        this.start();
    }
}
// The body of the server thread. Loop forever, listening for and // accepting connections from clients. For each connection, // create a Connection object to handle communication through the // new Socket
public void run() {
    try {
        while(true) {
            Socket client_socket = listen_socket.accept();
            Connection c = new Connection(client_socket);
        }
    } catch (IOException e) { fail(e,"Exception while listening for connections"); }
}

// Start the server up, listening on an optionally specified port
public static void main(String[] args) {
    int port = 0;
    if (args.length == 1) {
        try { port = Integer.parseInt(args[0]); }
        catch (NumberFormatException e) { port=0; }
    }
    new Server(port);
}

// This class is the thread that handles all communication with a client
class Connection extends Thread {
    static final boolean DEBUG=true;
    protected Socket client;
    protected PrintWriter out;
    protected BufferedReader in;

//Initialize the streams and start the thread
public Connection(Socket client_socket) {
    client = client_socket;
    try {
        in = new BufferedReader(new InputStreamReader(client.getInputStream()));
        out = new PrintWriter(client.getOutputStream(),true);
    }
    catch (IOException e) {
        try { client.close(); }
        catch (IOException e2) { }
        System.err.println("Exception while getting socket streams: "+e);
        return;
    }
    this.start();
}

//Provide the service
//Read a line, reverse it send it back
public void run(){
    String line;
    try {
        for (;;) {
            if (DEBUG) {
                System.out.println("Server ready to read ");
            }

            //read in a line
            line = in.readLine();
            if (DEBUG) {
                System.out.println("Line read "+line);
            }
            if (line.equals("bye") ||
            line == null) break;
            StringBuffer bufferedLine = new StringBuffer(line);
            String reversedLine = (bufferedLine.reverse()).toString();
            if (DEBUG) {
                System.out.println(" Reversed Line "+reversedLine);
            }
        }
    }
}
out.println(reversedLine);
out.flush();
}//end of for
}// end of try
catch (IOException e) {

finally { try { client.close(); } catch(IOException e2) {;
 }
 }
}//end of Connection
Server loop

• After initialization, the server sits in an infinite loop listening for connections on the ServerSocket `listen_socket`.

```java
while(true) {
    Socket client_socket = listen_socket.accept();
    Connection c = new Connection(client_socket);
}
```

• If a client is requesting a connection, the call `accept` succeeds and returns a `Socket`.

• `Socket client_socket` represents the connection between the client and the server.

• A thread (`Connection` is thread) is spawned to handle the connection between the client and the server.
Server Loop (Contd)

- Notice that several clients could be connected to the Server at the same time. Each connection has a dedicated thread handling it.

- Notice that this is a classic application of multi-threading. A server could be handling multiple connections concurrently.

- By assigning priority to different threads, the server can assign priorities to different clients.
Connection **thread**

- This thread makes streams out of the socket just like in the case of the client.

- Each time in the while loop, server reads a line from `in`, reverses it, and sends the reversed line to the client by writing on the output stream `out`.

- The fragment of code reversing a line is shown below:

```java
StringBuffer bufferedLine = new StringBuffer(line);
String reversedLine = (bufferedLine.reverse()).toString();
```
Connecting to the Web

- The `URL` class and the related classes (`URLConnection` and `URLEncoder`) are more appropriate than socket if one is connecting to a web-site.

- In fact `URLs` are high-level connection to the `Web` which uses sockets in its implementation.