

# Internet Technology

## 02r. Programming with Sockets (Java & TCP)

Paul Krzyzanowski

Rutgers University

Spring 2016

# Sample Client-Server Program

- To illustrate programming with TCP/IP sockets, we'll write a small client-server program:
  - Client:
    - Read a line of text from the user
    - Send it to the server; wait for a response (single line)
    - Print the response
  - Server
    - Wait for a connection from a client
    - Read a line of text
    - Return a response that contains the length of the string and the string converted to uppercase
    - Exit

# Sample Client-Server Program

- We will then embellish this program to:
  - Specify a host & port number on the command line
  - Allow a client to send multiple lines of text
  - Make the server multi-threaded so it can handle concurrent requests

# A brief diversion: input/output

- With Java, you'll often layer different input/output stream classes depending on what you want to do.
- Here are some common ones:
- Input
  - InputStream
  - BufferedReader
  - InputStreamReader
- Output
  - OutputStream
  - DataOutputStream
  - PrintStream
  - DataOutputStream

# A brief diversion: output

OutputStream	The basics – write a byte or a bunch of bytes
DataOutputStream	<p>Allows you to write Unicode (multibyte) characters, booleans, doubles, floats, ints, etc. <i>Watch out if using this because the other side might not be Java and might represent the data differently.</i></p> <p>The two most useful things here are <code>writeBytes(String s)</code>, which writes a string out as a bunch of 1-byte values and <code>write(byte[] b, int off, int len)</code>, which writes a sequence of bytes from a byte array.</p>
PrintStream	Allows you to use <code>print</code> and <code>println</code> to send characters. Useful for line-oriented output.
FilterOutputStream	Needed for <code>PrintStream</code> . On it's own, just gives you the same write capabilities you get with <code>OutputStream</code>

# A brief diversion: input

InputStream	The basics – <code>read</code> a byte or a bunch of bytes
BufferedReader	Buffers input and parses lines. Allows you to read data a line at a time via <code>readLine()</code> . You can also use <code>read(char [] cbuf, int off, int len)</code> to read characters into a portion of an array.
InputStreamReader	You need this to use <code>BufferedReader</code> . It converts bytes (that you'll be sending over the network) to Java characters.

# Client: step 1

- Read a line of text from the standard input (usually keyboard)
  - We use *readLine* to read the text. For that, we need to use the `BufferedReader` class on top of the `InputStreamReader` on top of the system input stream (`System.in`)

```
String line;  
BufferedReader userdata = new BufferedReader(new InputStreamReader(System.in));  
line = userdata.readLine();
```

# Test 1

- Don't hesitate to write tiny programs if you're not 100% sure how something works

```
import java.io.*;

public class line {
    public static void main(String args[]) throws Exception {
        String line;

        BufferedReader userdata = new BufferedReader(new InputStreamReader(System.in));
        line = userdata.readLine();
        System.out.println("got: \"" + line + "\"");
    }
}
```

- Notice that `readLine()` removes the terminating newline character from a line
  - If we want to send line-oriented text, we'll need to suffix a newline (`'\n'`) to the string



# Client: step 2

- Establish a socket to the server, send the line, and get the result
  - Create a socket.
  - For now, we will connect to ourselves – the name “localhost” resolves to our local address.
  - For now, we will hard-code a port number: 12345

```
Socket sock = new Socket("localhost", 12345); // create a socket and connect
```

- Get input and output streams from the socket
  - The methods `getInputStream()` and `getOutputStream()` return the basic streams for the socket
  - Create a `DataOutputStream` for the socket so we can write a string as bytes
  - Create a `BufferedReader` so we can read a line of results from the server

```
DataOutputStream toServer = new DataOutputStream(sock.getOutputStream());  
BufferedReader fromServer = new BufferedReader(  
    new InputStreamReader(sock.getInputStream()));
```

# Client: step 3

- Send the line we read from the user and read the results

```
toServer.writeBytes(line + '\n');    // send the line we read from the user  
  
String result = fromServer.readLine(); // read the response from the server
```

- We're done; print the result and close the socket

```
System.out.println(result);  
  
sock.close();
```

# Our client – version 1

- But we can't test it yet because we don't have the server!

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

public class TCPClient {
    public static void main(String args []) throws Exception {
        String line;          // user input
        BufferedReader userdata = new BufferedReader(new InputStreamReader(System.in));

        Socket sock = new Socket("localhost", 12345);          // connect to localhost port 12345
        DataOutputStream toServer = new DataOutputStream(sock.getOutputStream());
        BufferedReader fromServer = new BufferedReader(
            new InputStreamReader(sock.getInputStream()));

        line = userdata.readLine();          // read a line from the user
        toServer.writeBytes(line + '\n');    // send the line to the server
        String result = fromServer.readLine(); // read a one-line result
        System.out.println(result);         // print it
        sock.close();                       // and we're done
    }
}
```

# Server: step 1

- Create a socket for listening
  - This socket's purpose is only to accept connections
  - Java calls this a `ServerSocket`
  - For now, we'll use a hard-coded port: 12345
    - If the port number is 0, the operating system will assign a port.
  - The backlog is the maximum queue length for unserviced arriving connections
    - The backlog is missing or 0, a default backlog will be used

```
ServerSocket svc = new ServerSocket(12345, 5); // listen on port 12345
```



port

max  
backlog

# Server: step 2

- Wait for a connection
  - This method will block until a connection comes in
  - When a client connects to port 12345 on this machine, the `accept()` method will return a new socket that is dedicated to communicating to that specific client

```
Socket conn = svc.accept(); // get a connection
```

# Test 2

- We can now test that a client can connect to the server
- Let's write a tiny server that just waits for a connection and then exits

```
import java.net.*;

public class wait {
    public static void main(String args[]) throws Exception {
        ServerSocket svc = new ServerSocket(12345, 5); // listen on port 12345

        Socket conn = svc.accept(); // get a connection
    }
}
```

- Now run the client in another window
  - As soon as the client starts, it will establish a connection and the server will exit

# Server: step 3

- Get input/output streams for the socket
  - We will create a *BufferedReader* for the input stream so we can use `readLine` to read data a line at a time
  - We will create a *DataOutputStream* for the output stream so we can write bytes.

```
// get the input/output streams for the socket
BufferedReader fromClient = new BufferedReader(
    new InputStreamReader(conn.getInputStream()));
DataOutputStream toClient = new DataOutputStream(conn.getOutputStream());
```

# Server: step 4

- Read a line of data from the client (via *fromClient*)

```
String line = fromClient.readLine();           // read the data
System.out.println("got line \"" + line + "\""); // debugging! Let's see what we got
```

- Create the result

```
// do the work
String result = line.length() + ": " + line.toUpperCase() + "\n";
```

- Write the result to the client (via *writeBytes*)

```
toClient.writeBytes(result); // send the result
```



# Server: step 5

- Done! Close the socket
  - Close the socket to the client to stop all communication with that client
  - Close the listening socket to disallow any more incoming connections. Servers often run forever and therefore we often will not do this.

```
System.out.println("server exiting\n"); // debugging message
conn.close(); // close connection
svc.close(); // stop listening
```

# Our server – version 1

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

public class TCPServer {
    public static void main(String args[]) throws Exception {
        ServerSocket svc = new ServerSocket(12345, 5); // listen on port 12345

        Socket conn = svc.accept(); // wait for a connection

        // get the input/output streams for the socket
        BufferedReader fromClient = new BufferedReader(
            new InputStreamReader(conn.getInputStream()));
        DataOutputStream toClient = new DataOutputStream(conn.getOutputStream());

        String line = fromClient.readLine(); // read the data from the client
        System.out.println("got line \"" + line + "\""); // show what we got

        String result = line.length() + ": " + line.toUpperCase() + '\n'; // do the work

        toClient.writeBytes(result); // send the result

        System.out.println("server exiting\n");
        conn.close(); // close connection
        svc.close(); // stop listening
    }
}
```

# Let's test it – Test 3

- Compile TCPServer.java and TCPClient.java

```
javac *.java
```

- In one window, run

```
java TCPServer
```

- In another window, run

```
java TCPClient
```

- The client will wait for input. Type something

```
Hello
```

- It will respond with the server's output:

```
5: HELLO
```

# Version 2

---

- We don't want the server to exit
- Instead, have it wait for another connection
- Simple:
  - Create the ServerSocket
  - Then put everything else in a forever loop ( `for(;;)` )
  - Never close the ServerSocket
- Now we can keep the server running and try running the client multiple times

# Our server – version 2

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

public class TCPServer {
    public static void main(String args[]) throws Exception {
        ServerSocket svc = new ServerSocket(12345, 5); // listen on port 12345

        for (;;) {
            Socket conn = svc.accept(); // get a connection from a client

            BufferedReader fromClient = new BufferedReader(
                new InputStreamReader(conn.getInputStream()));
            DataOutputStream toClient = new DataOutputStream(conn.getOutputStream());

            String line = fromClient.readLine(); // read the data from the client
            System.out.println("got line \"" + line + "\"");

            String result = line.length() + ": " + line.toUpperCase() + '\n'; // do the work

            toClient.writeBytes(result); // send the result

            System.out.println("closing the connection\n");
            conn.close(); // close connection
        }
    }
}
```

# Version 3: let's support multiple lines

- Instead of having the server close the connection when a single line of text is received, allow the client to read multiple lines of text
  - Each line is sent to the server; the response is read & printed
  - An end of file from the user signals the end of user input
    - This is typically control-D on Mac/Linux/Unix systems (see the *stty* command)

Note: You only care what the end of file character is as a user typing into a terminal window. As a programmer, you will never see the character and will rely on the driver to tell you that an end of file has been detected.

# Client – Version 3

- We create a while loop to read lines of text
- When `readLine()` returns null, that means there's no more.

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

public class TCPClient {
    public static void main(String argv[]) throws Exception {
        String line; // user input
        BufferedReader userdata = new BufferedReader(new InputStreamReader(System.in));

        Socket sock = new Socket("localhost", 12345); // connect to localhost port 12345
        DataOutputStream toServer = new DataOutputStream(sock.getOutputStream());
        BufferedReader fromServer = new BufferedReader(
            new InputStreamReader(sock.getInputStream()));

        while ((line = userdata.readLine()) != null) { // read a line at a time
            toServer.writeBytes(line + '\n'); // send the line to the server
            String result = fromServer.readLine(); // read a one-line result
            System.out.println(result); // print it
        }
        sock.close(); // we're done with the connection
    }
}
```

# Version 3 – server changes

- We need to change the server too
  - Read lines from a socket until there are no more
  - When the client closes a socket and the server tries to read, it will get an end-of-file: *readline()* will return a null
  - A simple loop lets us iterate over the lines coming in from one client

```
while ((line = fromClient.readLine()) != null) { // while there's data from the client

    // do work on the data

}
System.out.println("closing the connection\n");
conn.close(); // close connection
```



# The server handles only one connection

- Run the server in one window
- Run the client in another window
  - Type a bunch of text
  - Each line produces a response from the server
- Run the client again in yet another window
  - Type a bunch of text
  - Nothing happens. There's no connection to the server!
  - You have to exit the first client before this one can connect.
- We need to make the server multi-threaded

# Version 4 – add multi-threading to the server

- We define the server to implement Runnable
- Define a constructor: called for each new thread

```
public class TCPServer implements Runnable {
    Socket conn;      // this is a per-thread copy of the client socket
                    // if we defined this static, then it would be shared among threads

    TCPServer(Socket sock) {
        this.conn = sock;      // store the socket for the connection
    }
}
```

# Version 4 – add multi-threading to the server

- The main function just gets connections and creates threads

```
public static void main(String args[]) throws Exception {  
    ServerSocket svc = new ServerSocket(12345, 5); // listen on port 12345  
  
    for (;;) {  
        Socket conn = svc.accept(); // get a connection from a client  
        System.out.println("got a new connection");  
  
        new Thread(new TCPServer(conn)).start();  
    }  
}
```

This creates the thread's state  
and calls the constructor

This creates the thread of  
execution and calls *run()* in the  
thread.  
When *run* returns, the thread  
exits.

# Version 4 – add multi-threading to the server

- The per-connection work is done in the thread

```
public void run() {
    try {
        BufferedReader fromClient = new BufferedReader(new InputStreamReader(conn.getInputStream()));
        DataOutputStream toClient = new DataOutputStream(conn.getOutputStream());
        String line;

        while ((line = fromClient.readLine()) != null) { // while there's data from the client
            System.out.println("got line \"" + line + "\"");

            String result = line.length() + ": " + line.toUpperCase() + '\n'; // do the work

            toClient.writeBytes(result); // send the result
        }

        System.out.println("closing the connection\n");
        conn.close(); // close connection and exit the thread
    } catch (IOException e) {
        System.out.println(e);
    }
}
```

# Version 6

- Allow the client to specify the server name on the command line
  - If it's missing, use "localhost"

```
public class TCPClient {
    public static void main(String args[]) throws Exception {
        String line; // user input
        String server = "localhost"; // default server
        BufferedReader userdata = new BufferedReader(new InputStreamReader(System.in));

        if (args.length > 1) {
            System.err.println("usage: java TCPClient server_name");
            System.exit(1);
        } else if (args.length == 1) {
            server = args[0];
            System.out.println("server = " + server);
        }

        Socket sock = new Socket(server, 12345); // connect to localhost port 12345
    }
}
```

The end