Distributed Systems

01r. Sockets Programming Introduction

Paul Krzyzanowski
TA: Yuanzhen Gu
Rutgers University
Fall 2014
Machine vs. transport endpoints

- IP packets address only the machine
  - IP header identifies source IP address, destination IP address
  - IP address is a 32-bit address that refers to a machine
- IP packet delivery is not guaranteed to be reliable or in-order
- Transport-level protocols on top of IP:
  - Allow application-to-application communication
  - TCP/IP
  - UDP/IP
  - Port numbers

```
Process A

machine 192.168.1.5

port 1512

Process B

machine 192.168.1.7

port 25
```
TCP/IP

- Virtual circuit service
- Sends ACK for each received packet
- Checksum to validate data
- Data may be transmitted simultaneously in both directions
- No record markers but data arrives in sequence

<table>
<thead>
<tr>
<th>TCP header</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Src port</td>
<td>Dest port</td>
</tr>
<tr>
<td></td>
<td>Sequence number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Acknowledgement number</td>
<td></td>
</tr>
<tr>
<td>Header length</td>
<td>flags</td>
<td>window</td>
</tr>
<tr>
<td>checksum</td>
<td>Urgent ptr</td>
<td></td>
</tr>
<tr>
<td>Options and pad</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

20 bytes
UDP/IP

- Datagram service
- Packet may be lost
- Data may arrive out of sequence
- Checksum for data but no retransmit

<table>
<thead>
<tr>
<th>Src port</th>
<th>Dest port</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>checksum</td>
</tr>
</tbody>
</table>

8 bytes

UDP header
Sockets

Create a generalized IPC model

• Dominant API for transport layer connectivity
• Operating system interface to the network
• Created at UC Berkeley for 4.2BSD Unix (1983)

• Design goals
  – Communication between processes should not depend on whether they are on the same machine
  – Communication should be efficient
  – Interface should be compatible with files
  – Support different protocols and naming conventions
    • Sockets is not just for the Internet Protocol family
What is a socket?

Abstract object from which messages are sent and received

– Looks like a file descriptor

– Application can select particular style of communication
  • Virtual circuit (connection-oriented), datagram (connectionless), message-based, in-order delivery

– Unrelated processes should be able to locate communication endpoints
  • Sockets can have a name
  • Name should be meaningful in the communications domain
    – E.g., Address & port for IP communications
How are sockets used?

Client: web browser

Send HTTP request message to get a page

Server: web server

Receive HTTP request message

Process HTTP request

Send HTTP response message

Receive HTTP response message

Display a page
Connection-Oriented (TCP) socket operations

Client

1. Create a socket
2. Name the socket (assign local address, port)
3. Connect to the other side
4. read / write byte streams
5. close the socket

Server

1. Create a socket
2. Name the socket (assign local address, port)
3. Set the socket for listening
4. Wait for and accept a connection; get a socket for the connection
5. read / write byte streams
6. close the socket
7. close the listening socket
Connectionless (UDP) socket operations

Client

Create a socket

Name the socket (assign local address, port)

Send a message

Receive a message

close the socket

Server

Create a socket

Name the socket (assign local address, port)

Receive a message

Send a message

close the socket

© 2014 Paul Krzyzanowski
Using sockets in Java

• **java.net** package
  – **Socket** class
    • Deals with sockets used for TCP/IP communication
  – **ServerSocket** class
    • Deals with sockets used for accepting connections
  – **DatagramSocket** class
    • Deals with datagram packets (UDP/IP)

• Both **Socket** and **ServerSocket** rely on the **SocketImpl** class to actually implement sockets
  – But you don’t have to think about that as a programmer
Create a socket for listening: server

Server:

– *create, name, and listen* are combined into one method
– **ServerSocket** constructor

```java
ServerSocket svc = new ServerSocket(80, 5);
```

Several other flavors (see api reference)
1. Server: create a socket for listening

Client: web browser

Server: web server

Server Socket svc = new ServerSocket(80, 5);

Send HTTP request message to get a page

Receive HTTP request message

Process HTTP request

Send HTTP response message

Receive HTTP response message

Display a page
Server: wait for (accept) a connection

• **accept** method of `ServerSocket`
  – block until connection arrives
  – return a `Socket`

```java
ServerSocket svc = new ServerSocket(80, 5);
Socket req = svc.accept();
```
2. Server: wait for a connection (blocking)

Client: web browser

Server: web server

Server Socket svc = new ServerSocket(80);

Socket req = svc.accept();

Block until an incoming connection comes in

Send HTTP request message to get a page

Receive HTTP request message

Process HTTP request

Send HTTP response message

Receive HTTP response message

Display a page
Create a socket: client

Client:

- *create*, *name*, and *connect* operations are combined into one method
- **Socket** constructor

```java
Socket s = new Socket("www.rutgers.edu", 2211);
```

Several other flavors (see api reference)
3. Client: connect to server socket (blocking)

Client: web browser

Socket s = new Socket("pk.org", 80);

Blocks until connection is set up

Send HTTP request message to get a page

Receive HTTP response message

Display a page

Server: web server

Server Socket svc = new ServerSocket(80, 5);

Socket req = svc.accept();

Receive connection request from client

Receive HTTP request message

Process HTTP request

Send HTTP response message

September 12, 2014
© 2014 Paul Krzyzanowski
Client: web browser

Server: web server

Socket s = new Socket("pk.org", 80);

Connection is established

Send HTTP request message to get a page

Receive HTTP request message

Process HTTP request

Send HTTP response message

Receive HTTP response message

Display a page
Exchange data

- Obtain InputStream and OutputStream from Socket
  - layer whatever you need on top of them
    - e.g. DataInputStream, PrintStream, BufferedReader, ...

Example:

**client**
```java
dataInputStream in = new DataInputStream(s.getInputStream());
PrintStream out = new PrintStream(s.getOutputStream());
```

**server**
```java
dataInputStream in = new BufferedReader(new InputStreamReader(req.getInputStream()));
String line = in.readLine();
dataOutputStream out = new DataOutputStream(req.getOutputStream());
out.writeBytes(mystring + '\n')
```
4. Perform I/O (read, write)

Client: web browser

Server: web server

Server Socket svc = new ServerSocket(80, 5);

Socket s = new Socket("pk.org", 80);

InputStream s_in = s.getInputStream();
OutputStream s_out = s.getOutputStream();

Send HTTP request message to get a page

Receive HTTP request message

Process HTTP request

Receive HTTP response message

Send HTTP response message

Display a page

InputStream r_in = req.getInputStream();
OutputStream r_out = req.getOutputStream();
Close the sockets

Close input and output streams first, then the socket

**client:**

```java
try {
    out.close();
    in.close();
    s.close();
} catch (IOException e) {} 
```

**server:**

```java
try {
    out.close();
    in.close();
    req.close();    // close connection socket
    svc.close();    // close ServerSocket
} catch (IOException e) {} 
```
TCP vs. UDP sockets

• TCP (“stream sockets”)
  – Requires a connection (connection-oriented)
  – Dedicated socket for accepting connections
  – Communication socket provides a bi-directional link
  – Byte-stream: no message boundaries

• UDP (“datagram sockets”)
  – Connectionless: you can just send a message
  – Data send in discrete packets (messages)
UDP workflow

Client

Send request packet

Wait for request packet

Process request

Send response packet

Receive response packet

Server

September 12, 2014

© 2014 Paul Krzyzanowski
/* read a line from the user */
BufferedReader user_input = new BufferedReader(new InputStreamReader(System.in));
String line = user_input.readLine();

/* convert it to an array of bytes */
byte[] out_data = line.getBytes();

/* create a datagram socket */
DatagramSocket s = new DatagramSocket();

InetAddress addr = InetAddress.getByName("test.pk.org"); /* look up IP address */
int port = 1234; /* port number */

/* construct the packet */
DatagramPacket out_packet = new DatagramPacket(out_data, out_data.length, addr, port);

/* send it out on the socket */
s.send(out_packet);
Receive a packet

```java
byte in_buf[] new byte[1500];
int port = 4321;    /* port number on which we want to receive data */

/* create a datagram socket */
DatagramSocket s = new DatagramSocket(port);

/* create the packet for receiving the data*/
DatagramPacket in_packet = new DatagramPacket(in_buf, in_buf.length);

/* get the packet from the socket*/
s.receive(in_packet);

System.out.println(
    "received data [" + new String(in_packet.getData(), 0, in_packet.getLength()) + "] +
    " from address: " + in_packet.getAddress() +
    " port: " + in_packet.getPort();
```

September 12, 2014 © 2014 Paul Krzyzanowski
The sockets system call interface
(What the operating system gives us)
# POSIX system call interface

<table>
<thead>
<tr>
<th>System call</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>socket</td>
<td>Create a socket</td>
</tr>
<tr>
<td>bind</td>
<td>Associate an address with a socket</td>
</tr>
<tr>
<td>listen</td>
<td>Set the socket to listen for connections</td>
</tr>
<tr>
<td>accept</td>
<td>Wait for incoming connections</td>
</tr>
<tr>
<td>connect</td>
<td>Connect to a socket on the server</td>
</tr>
<tr>
<td>read/write, sendto/rerecvfrom, sendmsg/rerecvmsg</td>
<td>Exchange data</td>
</tr>
<tr>
<td>close/shutdown</td>
<td>Close the connection</td>
</tr>
</tbody>
</table>
Step 1 (client & server)

Create a socket

```c
int s = socket(domain, type, protocol)
```

**AF_INET**: Address Family: group of protocols for communication. 
- **AF_INET** is for IPv4
- **AF_INET6** is IPv6
- **AF_BTH** is Bluetooth

**SOCK_STREAM**: Type of protocol within the family.
- **SOCK_STREAM**: reliable, in-order, 2-way. TCP/IP

**SOCK_DGRAM**: datagrams (UDP/IP)

**SOCK_RAW**: “raw” – allows app to modify the network layer header

useful if some families have more than one protocol to support a given service. 0: unspecified

Conceptually similar to `open` BUT
- `open` creates a new reference to a possibly existing object
- `socket` creates a new instance of an object

September 12, 2014

© 2014 Paul Krzyzanowski
Step 2 (client & server)

Name the socket (assign address, port)

\[
\text{int error} = \text{bind}(s, \text{addr}, \text{addrlen})
\]

- **socket**: The socket from the `socket` system call.
- **Address structure**: `struct sockaddr*`
- **length of address structure**: This is a data structure that makes sense for whatever address family you selected.

Naming for an IP socket is the process of assigning our address to the socket. The address is the full transport address: the IP address of the network interface as well as the UDP or TCP port number.
Step 3a (server)

Set socket to be able to accept connections

```c
int error = listen(s, backlog)
```

The socket from the `socket` system call.

Number of connections you’ll allow between `accept` system calls

The socket that the server created with `socket` is now configured to accept new connections. This socket will only be used for accepting connections. Data will flow onto another socket.
Step 3 (client)

Connect to server

```c
int error = connect(s, svraddr, svraddrlen)
```

- `socket`: The socket from which we’re connecting.
- `address structure`: Full transport address of the destination: address and port number of the service.
- `length of address structure`

The client can send a connection request to the server once the server did a `listen` and is waiting for `accept`. 
Step 3b (server)

Wait for a connection from client

```c
int snew = accept(s, clntaddr, &clntalen)
```

- `socket`
- `pointer to address structure`
- `length of address structure`

This tells you where the socket came from: full transport address.

This is the listening socket

new socket for this communication session

Block the process until an incoming connection comes in.
Step 4. Exchange data

read/write system calls (same as for file systems)

send/recv system calls

int send(int s, void *msg, int len, uint flags);
int recv(int s, void *buf, int len, uint flags);

sendto/recvfrom system calls

int sendto(int s, void *msg, int len, uint flags, struct sockaddr *to, int tolen);
int recvfrom(int s, void *buf, int len, uint flags, struct sockaddr *from, int *fromlen);

sendmsg/recvmsg system calls

int sendmsg(int s, struct msghdr *msg, uint flags);
int recvmsg(int s, struct msghdr *msg, uint flags);

Like read and write but these support extra flags, such as bypassing routing or processing out of band data. Not all sockets support these.

If we’re using UDP (connectionless), we don’t need to do connect, listen, accept. These calls allows you to specify the destination address (sendto, sendmsg) to send a message and get the source address (recvfrom, recvmsg) when receiving a message.
Step 5

Close connection

\[ \text{shutdown}(s, \text{ how}) \]

\textit{how:}
- SHUT_RD (0): can send but not receive
- SHUT_WR (1): cannot send more data
- SHUT_RDWR (2): cannot send or receive (=0+1)

You can use the regular \textit{close} system call too, which does a complete shutdown, the same as \textit{shutdown}(s, SHUT_RDWR).
Java provides shortcuts that combine calls

Example

```java
Socket s = new Socket("www.rutgers.edu", 2211);```

**POSIX / C**

```c
int s = socket(AF_INET, SOCK_STREAM, 0);

struct sockaddr_in myaddr; /* initialize address structure */
myaddr.sin_family = AF_INET;
myaddr.sin_addr.s_addr = htonl(INADDR_ANY);
myaddr.sin_port = htons(0);
bind(s, (struct sockaddr *)&myaddr, sizeof(myaddr));

/* look up the server's address
struct hostent *hp;    /* host information */
struct sockaddr_in servaddr;    /* server address */
memset((char*)&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(2211);
hp = gethostbyname("www.rutgers.edu");

if (connect(fd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0) {
    /* connect failed */
}
```
Programming with sockets: Sample program
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:
  – Have a continuously-running server
  – Allow a client to send multiple lines of text
  – Make the server multi-threaded so it can handle concurrent requests
  – Specify a host on the command line
Classes for 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
## Handling output

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputStream</td>
<td>The basics – write a byte or a bunch of bytes</td>
</tr>
<tr>
<td>DataOutputStream</td>
<td>Allows you to write Unicode (multibyte) characters, booleans, doubles, floats, ints, etc. <em>Watch out if using this because the other side might not be Java and might represent the data differently.</em> 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.</td>
</tr>
<tr>
<td>PrintStream</td>
<td>Allows you to use <code>print</code> and <code>println</code> to send characters. Useful for line-oriented output.</td>
</tr>
<tr>
<td>FilterOutputStream</td>
<td>Needed for PrintStream. On it’s own, just gives you the same write capabilities you get with OutputStream</td>
</tr>
</tbody>
</table>
## Handling input

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputStream</td>
<td>The basics – read a byte or a bunch of bytes</td>
</tr>
</tbody>
</table>
| BufferedReader   | Buffers input and parses lines. Allows you to read data a line at a time via `readLine()`.
You can also use `read(char [] cbuf, int off, int len)` to read characters into a portion of an array. |
| InputStreamReader| You need this to use [BufferedReader](#). 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`)

```java
String line;
BufferedReader userdata = new BufferedReader(new InputStreamReader(System.in));
line = userdata.readLine();
```
• Don’t hesitate to write tiny programs if you’re not 100% sure how something works

```java
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

  ```java
  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

  ```java
  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

```java
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

```java
System.out.println(result);
sock.close();
```
Our client – version 1

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

```java
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
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

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

```java
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.

```java
// 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`)

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

• Create the result

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

• Write the result to the client (via `writeBytes`)

```java
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.

```java
System.out.println("server exiting\n"); // debugging message
cconn.close(); // close connection
svsvc.close(); // stop listening
```
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() + 
            ": "); // do the work
        toClient.writeBytes(result); // send the result

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

            String result = line.length() + ": " + line.toUpperCase() + '
';       // 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)
We create a while loop to read lines of text

When `readLine()` returns null, that means there’s no more.

```java
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

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

    // do work on the data

} // close connection
```

`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

```java
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

```java
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.
The per-connection work is done in the thread

```java
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 + \\
"\n");

            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);
    }
}
```
• Allow the client to specify the server name on the command line
  – If it’s missing, use “localhost”

```java
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