Distributed Systems
01r. Sockets Programming Introduction

Paul Krzyzanowski
TA: David Domingo
Rutgers University
Fall 2018

Machine vs. transport endpoints
• IP is a network layer protocol; packets address only the machine
  – IP header identifies source IP address, destination IP address
• IP packet delivery is not guaranteed to be reliable or in-order
• Transport-level protocols on top of IP: TCP & UDP
  – Allow application-to-application communication
  – Port numbers: identify communication “channel” at each host

What is a socket?
Abstract object from which messages are sent and received
– Looks like a file descriptor to programs
– Provides a communication channel for applications
– Application can select particular style of communication
  • Stream (connection-oriented) or datagram (connectionless)
– Unrelated processes need to locate communication endpoints
  • Sockets have a name
  • Name is meaningful in the communications domain
  – For IP networking, name = { address & port number }

How are sockets used?
Client: web browser
Server: web server
Connect to server
Send HTTP request message to get a page
Accept the connection
Receive HTTP request message
Process HTTP request
Send HTTP response message
Receive HTTP response message
Display a page
Disconnect from server (close)
Close the connection

Connection-Oriented (TCP) socket operations
Client
Create a socket
Name the socket (assign local address, port)
Connect to the other side
read / write byte streams
close the socket
Server
Create a socket
Name the socket (assign local address, port)
Set the socket for listening
Wait for and accept a connection; get a socket for the connection
read / write byte streams
close the 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)
Send a message
Receive a message
Close the socket
POSIX system call interface

This is what the operating system gives up

<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</td>
<td>Exchange data</td>
</tr>
<tr>
<td>sendto/recvfrom</td>
<td></td>
</tr>
<tr>
<td>sendmsg/recvmsg</td>
<td></td>
</tr>
<tr>
<td>close/shutdown</td>
<td>Close the connection</td>
</tr>
</tbody>
</table>

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

```java
Socket req = svc.accept();
```

Block until an incoming connection comes in

2. Server: wait for a connection (blocking)

```java
Socket req = svc.accept();
```
Client: create a socket

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

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: web server

Socket svc = new ServerSocket(80, 5);

Receive connection request from client

Send HTTP request message

Receive HTTP response message

Display a page

Blocks until connection is set up

Connections are established

4. Perform I/O (read, write)

Client: web browser

Server: web server

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

- InputStream s_in = s.getInputStream();
- OutputStream s_out = s.getOutputStream();
- InputStream r_in = req.getInputStream();
- OutputStream r_out = req.getOutputStream();

Send HTTP request message

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:

```java
client
DataInputStream in = new DataInputStream(s.getInputStream());
PrintStream out = new PrintStream(s.getOutputStream());
String line = in.readLine();
out.writeBytes(mystring + "\n")
```

```java
server
DataInputStream in = new BufferedReader(new InputStreamReader(req.getInputStream()));
String line = in.readLine();
DataOutputStream out = new DataOutputStream(req.getOutputStream());
out.writeBytes(mysting + "\n")
```

Close the sockets

Close input and output streams first, then the socket

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

```java
server:
try {
    out.close();
in.close();
req.close(); // close connection socket
svc.close(); // close ServerSocket
} catch (IOException e) {}
```
Programming with sockets: Sample program

Sample Client-Server Program

To illustrate programming with TCP/IP sockets, we'll write a tiny client-server program:

Client:
1. Read a line of text from the user
2. Send it to the server; wait for a response (single line)
3. Print the response

Server:
1. Wait for a connection from a client
2. Read a line of text
3. Return a response that contains the length of the string and the string converted to uppercase
4. 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.</td>
</tr>
<tr>
<td></td>
<td>Watch out if using this because the other side might not be Java and might represent the data differently.</td>
</tr>
<tr>
<td></td>
<td>The two most useful things here are writeBytes(String s), which writes a string out as a bunch of 1-byte values and write(byte b, int off, int len), which writes a sequence of bytes from a byte array.</td>
</tr>
<tr>
<td>PrintStream</td>
<td>Allows you to use print and println to send characters. Useful for line-oriented output.</td>
</tr>
<tr>
<td>FilterOutputStream</td>
<td>Needed for PrintStream. On its 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>
<tr>
<td>BufferedReader</td>
<td>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.</td>
</tr>
<tr>
<td>InputStreamReader</td>
<td>You need this to use BufferedReader. It converts bytes (that you'll be sending over the network) to Java characters.</td>
</tr>
</tbody>
</table>
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
public class line {
    String line;
    BufferedReader userdata = new BufferedReader(new InputStreamReader(System.in));
    line = userdata.readLine();
}
```

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

```java
Socket sock = new Socket("localhost", 12345); // create a socket and connect
BufferedReader fromServer = new BufferedReader(new InputStreamReader(sock.getInputStream()));
```

Send the line we read from the user and read the results
- Use `DataOutputStream` stream to send strings as bytes
- Use `BufferedReader` to read the results from the server
- Print to the console

```java
DataOutputStream toServer = new DataOutputStream(sock.getOutputStream());
DataOutputStream fromServer = new DataOutputStream(sock.getInputStream());
String result = fromServer.readLine(); // read a line from the server
```

Create a TCP client
- This socket's purpose is 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

```java
public static void main(String[] args) throws Exception {
    ServerSocket svc = new ServerSocket(12345, 5); // listen on port 12345
    Socket sock = new Socket("localhost", 12345);
    DataInputStream toServer = new DataInputStream(sock.getInputStream());
    DataOutputStream fromServer = new DataOutputStream(sock.getOutputStream());
    System.out.println(result); // print the result and close the socket
    sock.close();
    sock.close();
}
```
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
```

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

```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
        // wait for a connection
        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
    }
}
```

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

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 ending\n"); // debugging message
svc.close(); // stop listening
```

Our server – version 1

```java
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 ending\n");
    }
}
```
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

  September 11, 2018 © 2014 - 2018 Paul Krzyzanowski

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

  September 11, 2018 © 2014 - 2018 Paul Krzyzanowski

Version 3

Our server – version 2

```java
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 \\
");
            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)

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

Client – Version 3

We create a while loop to read lines of text

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

```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);
        DataOutputStream toServer = new DataOutputStream(sock.getOutputStream());
        BufferedReader fromServer = new BufferedReader(new InputStreamReader(sock.getInputStream()));
        while ((line = userdata.readLine()) != null) {  // while there's data from the client
            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
    }
}
```
The server handles only one connection

1. Run the server in one window
2. Run the client in another window
   - Type a bunch of text
   - Each line produces a response from the server
3. 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.
4. 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
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
    public TCPServer(Socket sock) {
        this.conn = sock;
    }
}
```

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 + "");
            String result = line.length() + " : " + line.toUpperCase() + '
'; // do the work
            toClient.writeBytes(result); // send the result
        }
        System.out.println("closing the connection");
        conn.close(); // close connection and exit the thread
    }
    catch (IOException e) {
        System.out.println(e);
    }
}
```

Version 5

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];
        } else if (args.length == 0) {
            System.out.println("server = " + server);
        }
        Socket sock = new Socket(server, 12345); // connect to localhost port 12345
    }
}
```

The end