Distributed Systems
01r. Sockets Programming Introduction

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

TCP/IP
- Connection-oriented service
- Packets are acknowledged (sender will retransmit missing data)
- Checksum to validate data
- Data may be transmitted simultaneously in both directions
- No-record markers – data arrives as a stream of bytes (in the correct order)
- TCP also does
  - Flow control: doesn’t send more data than the other side can accept
  - Congestion control slows down rate of transmission if the network appears to be congested (too many lost packets)

UDP/IP
- Datagram service
- Packet may be lost
- Data may arrive out of sequence
- Checksum for data but no retransmit
  - Receiver drops packets with damaged data
- Message-based communication
  - If you send 3 messages of i, j, k bytes the receiver will receive 3 messages of i, j, k bytes
  - With TCP, you may receive one message of i+j+k bytes

What is a socket?
- Abstract object from which messages are sent and received
  - Looks like a file descriptor to programs
  - 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
- Connect to server
- 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

Server: web server
- Receive HTTP request message
- Process HTTP request
- Send HTTP response message
- Display a page
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

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/recvfrom, sendmsg/recvmsg</td>
<td>Exchange data</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

1. Server: create a socket for listening

   Server: web server
   Create a socket for listening
   - ServerSocket constructor
     `ServerSocket svc = new ServerSocket(80, 5);`

Several other flavors (see API reference)

Server: wait for (accept) a connection

- accept method of ServerSocket
  - block until connection arrives
  - return a Socket

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

Client: create a socket

Client: web browser

Server: web server

```java
Server Socket svc = new ServerSocket(80, 5);
Socket req = svc.accept();
```

Block until an incoming connection comes in

Several other flavors (see api reference)

Client: connect to server socket (blocking)

Client: web browser

Server: web server

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

Receive connection request from client

Connection is established

3a. Connection accepted

Client: perform I/O (read, write)

Client: web browser

Server: web server

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

Send HTTP request message to get a page

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());
```

```java
server
DataInputStream in = new BufferedReader(new InputStreamReader(s.getInputStream()));
DataOutputStream out = new DataOutputStream(s.getOutputStream());
```

4. Perform I/O (read, write)

```java
InputStream r_in = req.getInputStream();
OutputStream r_out = req.getOutputStream();
```

Send HTTP request message to get a page

Receive HTTP response message

Display a page
Close the sockets

Close input and output streams first, then the socket

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

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 small 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`
  - `PrintStream`
  - `DataOutputStream`

Handling output

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

BufferedReader allows us to read a line at a time.

The methods readLine() and read() return the basic streams for the socket.

Notice that readLine() removes the terminating newline character from a line.

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

Client: step 1

Read a line of text from the standard input (usually keyboard)

- We use BufferedReader to read the text. For that, we need to use the BufferedReader class on top of the InputStreamReader class on top of the system input stream (System.in)

```java
public class TCPClient {
  public static void main(String[] args) throws Exception {
    BufferedReader userdata = new BufferedReader(new InputStreamReader(System.in));
    String line = userdata.readLine();
    } // send the line to the server
  System.out.println(line + '\n'); // print it
  // read the response from the server
  String result = fromServer.readLine();
  // send the line we read from the user
  toServer.writeBytes(line + '\n');
  // read a line from the user
  line = userdata.readLine();
  // do some processing
  System.out.println(result);
  sock.close();
} // and we're done
```

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
// send the line we read from the user
String result = fromServer.readLine();
// send the line to the server
sock.getOutputStream().writeBytes(line + '\n');
// print it
System.out.println(line);
// read a line from the user
line = fromServer.readLine();
```

Our client – version 1

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

```
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
    }
}
```

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

```
String line = fromClient.readLine(); // read the data
String result = line.length() + "\n" + line.toUpperCase(); // debugging! Let’s see what we got

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");
conn.close(); // debugging message
svc.close(); // stop listening
```
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

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 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) {
    // do work on the data
    System.out.println("closing the connection!");
    conn.close(); // close connection
}
```

We define the server to implement Runnable

- Define a constructor: called for each new 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);
            // do the work
            toClient.writeBytes(line.toUpperCase() + ": " + line.length() + " n");
            // do work
            System.out.println("closing the connection!");
            conn.close(); // close connection
        }
    } catch (IOException e) {
        System.out.println("closing 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!

4. We need to make the server multi-threaded

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";
        BufferedReader user_data = new BufferedReader(new InputStreamReader(System.in));
        if (args.length == 1) {
            System.out.println("usage: java TCPClient server_name");
            System.exit(1);
        } else if (args.length == 2) {
            server = args[1];
            System.out.println("server = " + server);
        }
        Socket sock = new Socket(server, 12345); // connect to localhost port 12345
```
The end