

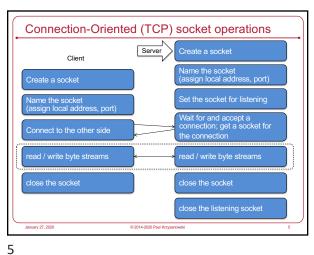
Machine vs. transport endpoints • IP is a network layer protocol: packets address only the machine - IP header identifies source IP address, destination IP address · IP packet delivery is not guaranteed to be reliable or in-order Transport-level protocols on top of IP: TCP & UDP - Allow application-to-application communication - Port numbers: identify communication "channel" at each host machine 192.168.1.5 machine 192.168.1.7 port 1512 port 25 (Process A) Process B

1

3

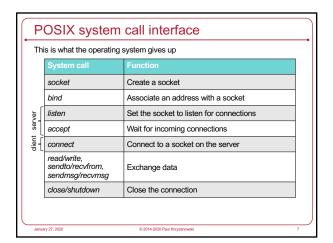
What is a socket? Abstract object from which messages are sent and received - Looks like a file descriptor to programs - Provides a communication channel for applications - Application can select particular style of communication Stream (connection-oriented) or datagram (connectionless) - Unrelated processes 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 Send HTTP request message to get a page Receive HTTP request message Process HTTP request Send HTTP response message Receive HTTP response message Close the connection Display a page



Connectionless (UDP) socket operations Client Server Name the socket (assign local address, port) Name the socket assign local address, port)

Paul Krzyzanowski 1



7

Create a socket for listening: server

Server:

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

port backlog

ServerSocket svc = new ServerSocket (80, 5);

Several other flavors (see API reference)

9

Server: wait for (accept) a connection

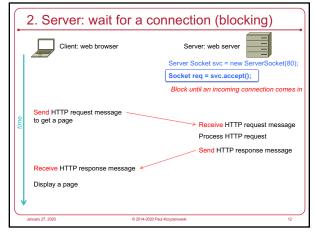
accept method of ServerSocket

- block until connection arrives

- return a Socket

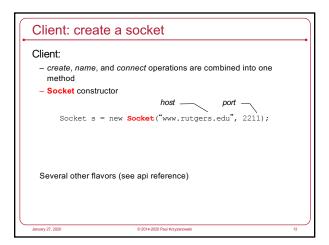
ServerSocket svc = new ServerSocket(80, 5);
Socket req = svc.accept();

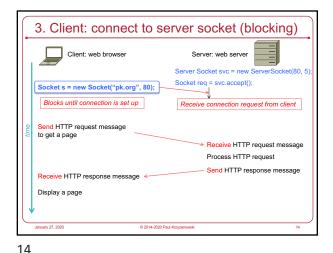
10



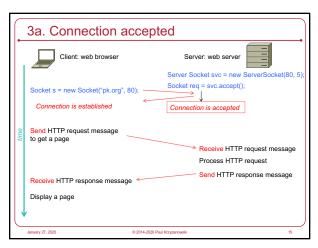
11 12

Paul Krzyzanowski 2



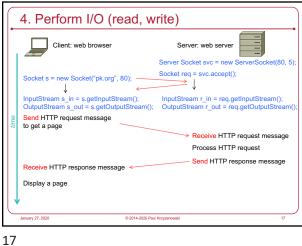


13



Exchange data Obtain InputStream and OutputStream from Socket - Layer whatever you need on top of them - e.g. DataInputStream, PrintStream, BufferedReader,  $\dots$ Example: client DataInputStream in = new DataInputStream(s.getInputStream()); PrintStream out = new PrintStream(s.getOutputStream()); DataInputStream in = new BufferedReader( new InputStreamReader(req.getInputStream())); String line = in.readLine(); 

15



Close the sockets Close input and output streams first, then the socket out.close(); in.close(); s.close();
} catch (IOException e) {} server: out.close(); in.close(); // close connection socket // close ServerSocket req.close(); svc.close(); } catch (IOException e) {}

Paul Krzyzanowski 3

18

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

19

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

1

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

Output
• OutputStream

- InputStream
   PufferedDeep
- BufferedReader
- DataOutputStream
- InputStreamReader
- PrintStream
- DataOutputStream

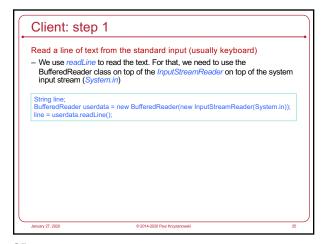
22

20

21

23 24

Paul Krzyzanowski 4



25

 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();

27

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

29 30

Paul Krzyzanowski 5

28

32

34



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 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 lanuary 27, 2020

31

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 // get the input/output streams for the socket ufferedReader 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 // stop listening

Our server – version 1 blic 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 SufferedReader from Client = new BufferedReader(
new InputStreamReader(conn.getInputStream())):
DataOutputStream toClient = new DataOutputStream(conn.getOutputStream()); 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

35 36

Paul Krzyzanowski 6

## 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

37

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)

40

38

```
Client — Version 3

We create a while loop to read lines of text

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

Import java.io.*;
Import ja
```

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

41 42

Paul Krzyzanowski 7

44

46

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 January 27, 2020

Version 4 – add multi-threading to the server We define the server to implement Runnable - Define a constructor: called for each new thread ublic 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) {

\*\*bis conn = sock; // store the socket for the connection lanuary 27, 2020

43

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 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 When run returns, the thread

Version 4 – add multi-threading to the server The per-connection work is done in the thread {
BufferedReader fromClient = new BufferedReader(new InputStreamReader(conn.getInputStream()));
DataOutputStream toClient = new DataOutputStream(conn.getOutputStream());  $\label{eq:while (fline = fromClient.readLine()) != null) { // while there's data from the client System.out.println("got line \"" + line + "\""); // while there's data from the client System.out.println("got line \"" + line + "\""); // while there's data from the client System.out.println("got line \"" + line + "\""); // while there's data from the client System.out.println("got line \"" + line + "\"");$ 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);

45

Version 5 Allow the client to specify the server name on the command line - If it's missing, use "localhost" c class TCP-clent (
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)); Socket sock = new Socket(server, 12345); // connect to localhost port 12345

The end

47 48

Paul Krzyzanowski 8