TCP Programming

RES, Lecture 2 (first part)

Olivier Liechti
Juergen Ehrensberger

heig-vd

HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD

www.heig-vd.ch

Client-Server Programming

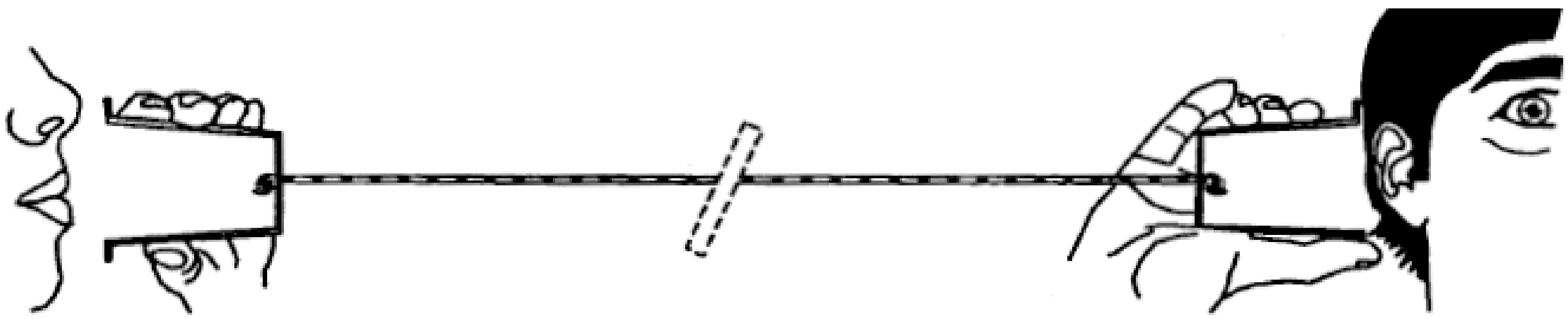


Figure 1

HTTP

APACHE
HTTP SERVER

SMTP

Exchange

Proprietary Protocol

Spotify

Spotify

What is an Application-Level Protocol?

HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch

- **A set of rules** that specify how the application components (e.g. clients and servers) **communicate with each other**. Typically, a protocol defines at least:
  - **Which transport-layer protocol** is used to exchange application-level messages. (e.g. TCP for HTTP)
  - **Which port number(s)** to use (e.g. 80 for HTTP)
  - **What kind of messages** are exchanged by the application components and the **structure** of these messages.
  - The **actions** that need to be taken when these messages are received and the **effect** that is expected.
  - Whether the protocol is **stateful** or **stateless**. In other words, whether the protocol requires the server to manage a session for every connected client.

Network Programming

Given a application-level protocol,

how can we implement a client and server in a particular programming language?

What abstractions, APIs, libraries are available to help us do that?

We know about TCP, UDP and IP. But how can we benefit from these protocols in our code?

Network Programming

Given a application-level protocol,

how can we implement a client and server in a particular programming language?

What abstractions, APIs, libraries are available to help us do that?

We know about TCP, UDP and IP. But how can we benefit from these protocols in our code?

The TCP Protocol

TCP

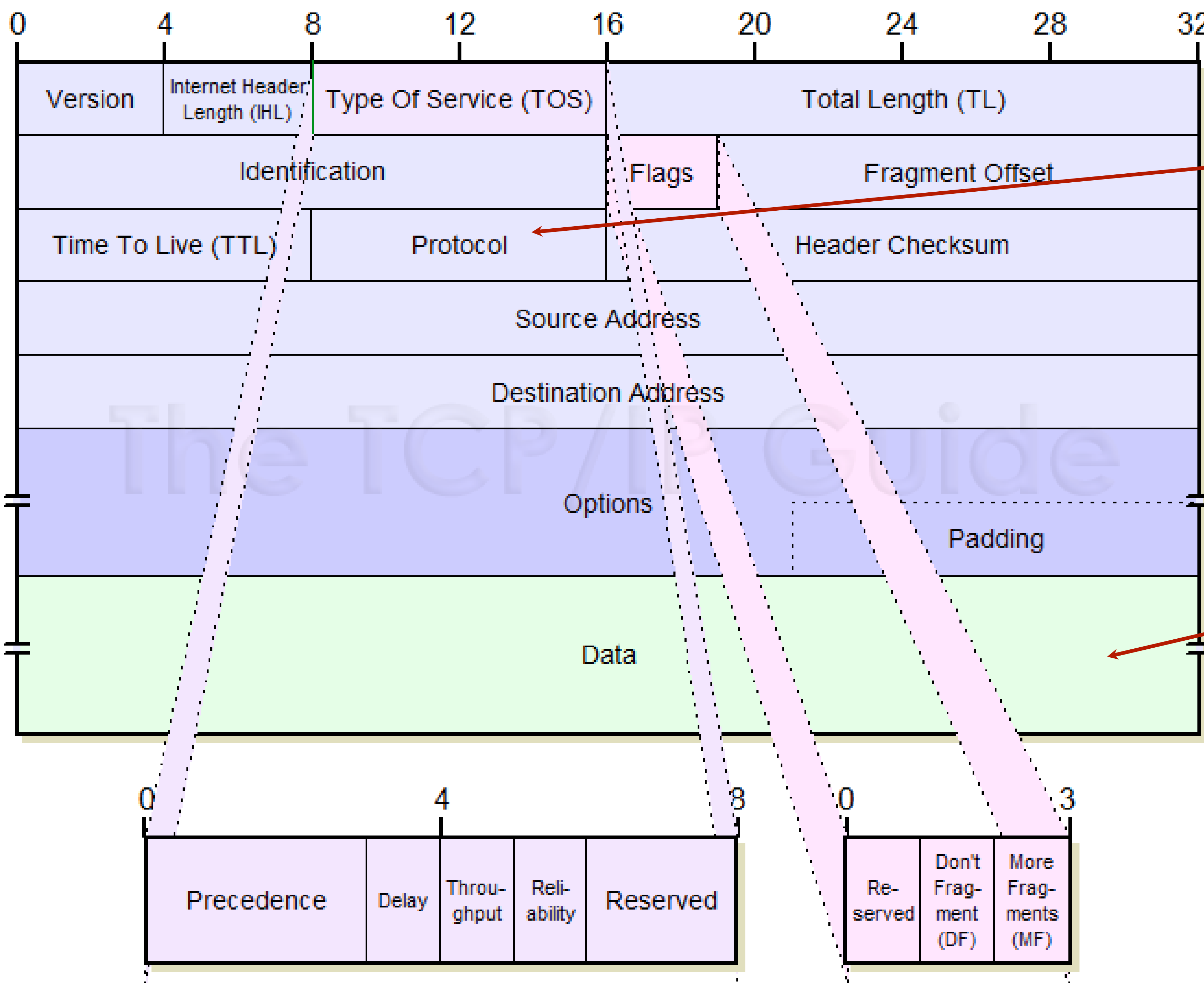
UDP

Transport Protocols

- Both TCP and UDP are **transport protocols**.
- This means that they make it possible for **two programs** (i.e. applications, processes) possibly running on **different machines** to **exchange data**.
- The two protocols also make it possible for several programs to **share the same network interface**. They use the notion of **port** for this purpose.
- TCP and UDP define the **structure of messages**. With TCP, messages are called **segments**. With UDP, messages are used **datagrams**.
- The structure of TCP segments (**number and size of headers**) is more complex than the structure of UDP datagrams.
- Both TCP segments and UDP datagrams can be **encapsulated in IP packets**. In that case, we say that the **payload** of the IP packet is a TCP segment, respectively a UDP datagram.

Transport Protocols

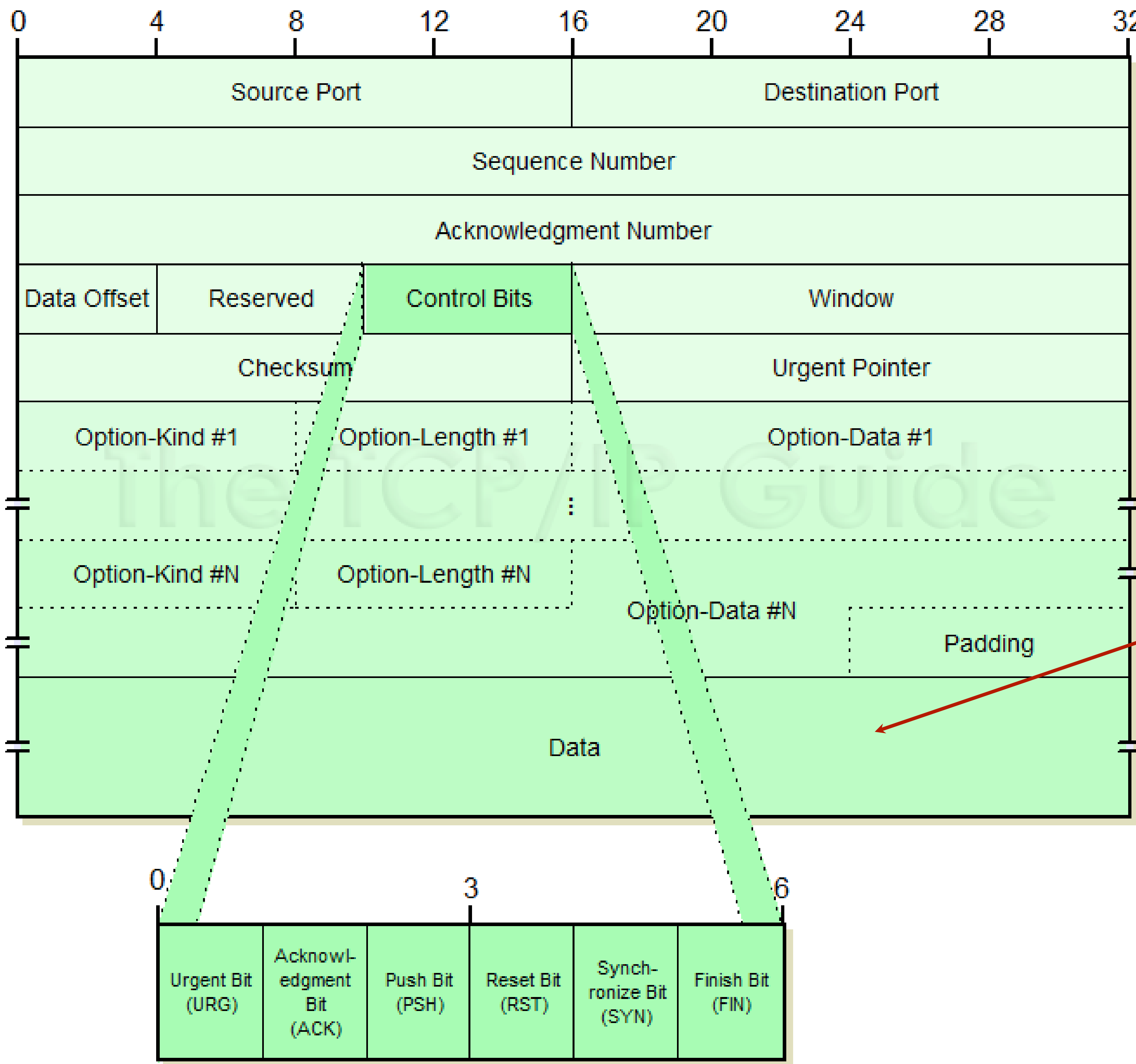
- TCP provides a **connection-oriented service**. The client and the server first have to establish a connection. They can then exchange data through a **bi-directional stream of bytes**.
- TCP provides a **reliable data transfer service**. It makes sure that all bytes sent by one program are received by the other. It also preserves the **ordering** of the exchanged bytes.
- UDP provides a **connectionless service**. The client can send information to the server at any time, **even if there is no server listening**. In that case, the information will simply be lost.
- UDP **does not guarantee the delivery** of datagrams. It is possible that a datagram sent by one client will never reach its destination. The ordering is not guaranteed either.
- TCP supports **unicast** communication. UDP supports **unicast**, **broadcast** and **multicast** communication. This is useful for **service discovery**.



If "Data" is a TCP segment, this field has the decimal value "6". If it is a UDP datagram, this field has the decimal value "17".

This can contain a TCP segment, a UDP datagram, or something else.

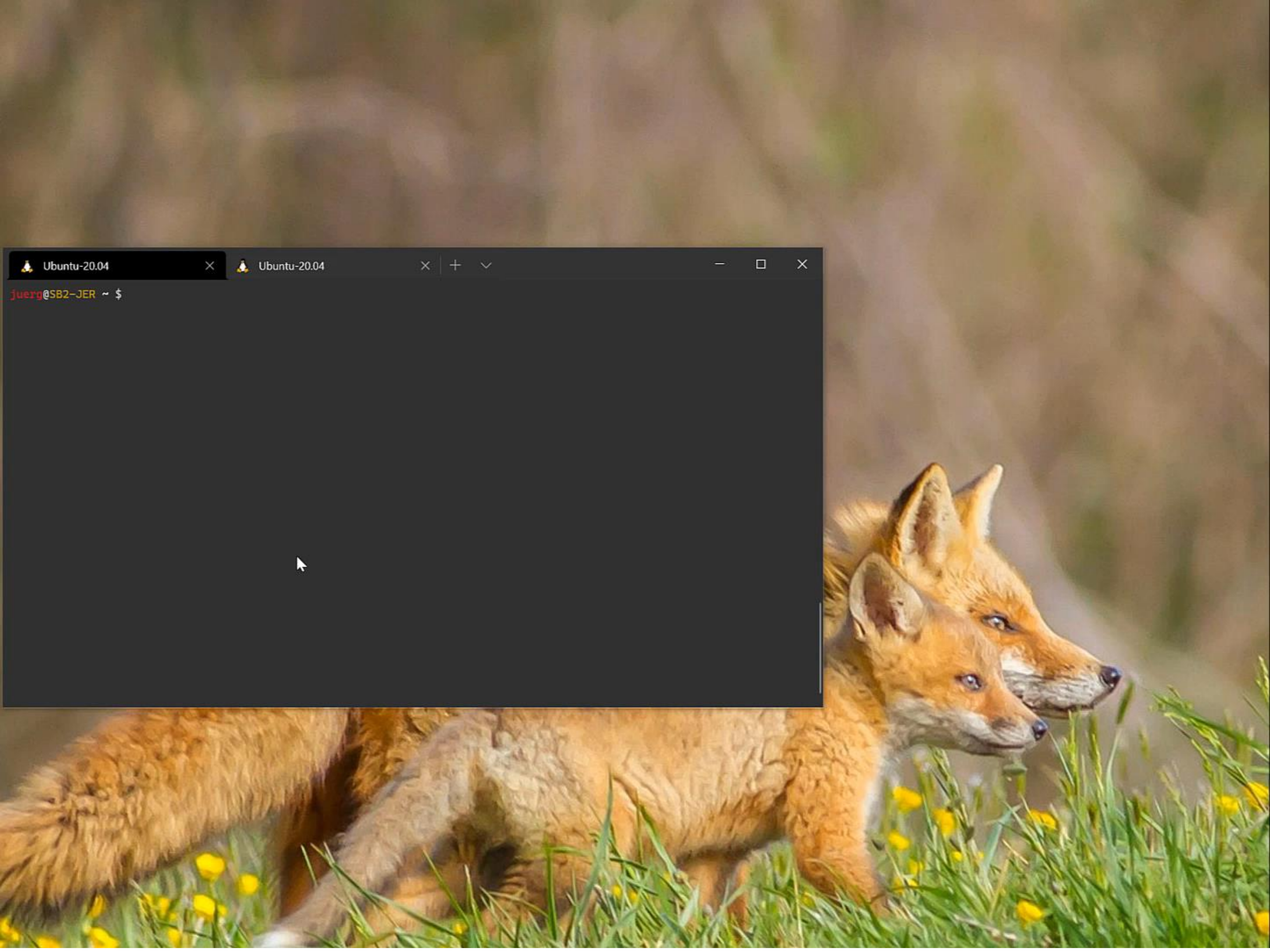
http://www.tcpipguide.com/free/t_IPDatagramGeneralFormat.htm



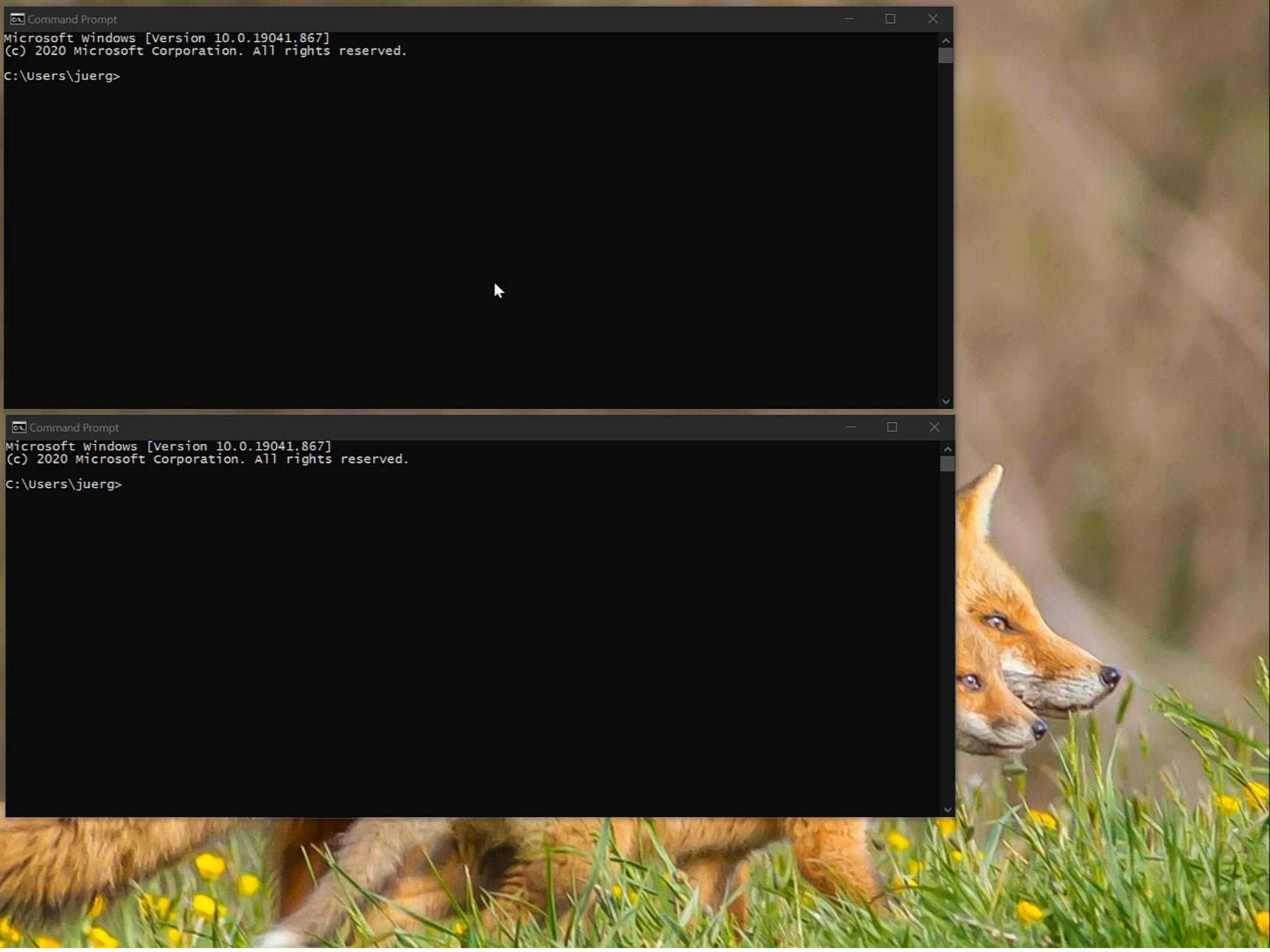
The bytes that you write in your java program will be here...

http://www.tcpipguide.com/free/t_TCPMessageSegmentFormat-3.htm

Example: **telnet www.google.ch 80**


Ubuntu-20.04
Ubuntu-20.04
juerg@SB2-JER ~ $

Example (server): **nc -kl 2019**
Example (client): **nc localhost 2019**


Command Prompt
Microsoft Windows [Version 10.0.19041.867]
(c) 2020 Microsoft Corporation. All rights reserved.
C:\Users\juerg>
Command Prompt
Microsoft Windows [Version 10.0.19041.867]
(c) 2020 Microsoft Corporation. All rights reserved.
C:\Users\juerg>

Command Prompt - ncat -lk 2019

```
Microsoft Windows [Version 10.0.19041.867]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\juerg>ncat -lk 2019
asdfasdf
fsdfgsdfg
dfgsdfg
Hello World
Salut
sdfasdf
Hello
ADD 10 20
RESULT 30
```

```
[RST, ACK] Seq=1 Ack=1 Win=0 Len=0
[SYN] Seq=0 Win=65535 Len=0 MSS=65495 W
[SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0
[ACK] Seq=1 Ack=1 Win=2619648 Len=0
[PSH, ACK] Seq=1 Ack=1 Win=2619648 Len=
[ACK] Seq=1 Ack=9 Win=2619648 Len=0
[PSH, ACK] Seq=1 Ack=9 Win=2619648 Len=
[ACK] Seq=9 Ack=7 Win=2619648 Len=0
[PSH, ACK] Seq=9 Ack=7 Win=2619648 Len=
[ACK] Seq=7 Ack=19 Win=2619648 Len=0
[PSH, ACK] Seq=7 Ack=19 Win=2619648 Len
[ACK] Seq=19 Ack=17 Win=2619648 Len=0
```

Command Prompt - ncat 10.193.156.177 2019

```
   IPv4 Address. . . . . . . . . . . : 192.168.91.100
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.91.1

Wireless LAN adapter Wi-Fi:

   Connection-specific DNS Suffix  . : einet.ad.eivd.ch
   Link-local IPv6 Address . . . . . : fe80::7960:38ce:8662:3fe1%25
   IPv4 Address. . . . . . . . . . . : 10.193.156.177
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 10.193.156.1

Ethernet adapter Bluetooth Network Connection:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

C:\Users\juerg>ncat 10.193.156.177 2019
asdfasdf
fsdfgsdfg
dfgsdfg
Hello World
Salut
^C
C:\Users\juerg>ncat 10.193.156.177 2019
sdfasdf
Hello
ADD 10 20
RESULT 30
```

```
id 0
```

```
0000  02 00 00 00 45 00 00 28  85 ba 40 00 80 06 00 00   ····E··( ··@·····
0010  0a c1 9c b1 0a c1 9c b1  ff f7 07 e3 5f 11 80 22   ········ ····_··"
0020  d2 1f 97 e5 50 10 27 f9  e7 e2 00 00               ····P·'· ····
```

Adapter for loopback traffic capture: <live capture in progress> | Packets: 12 · Displayed: 12 (100.0%) | Profile: Default

The Socket API

Network Programming

Given a application-level protocol,

how can we implement a client and server in a particular programming language?

What abstractions, APIs, libraries are available to help us do that?

We know about TCP, UDP and IP. But how can we benefit from these protocols in our code?

The Socket API

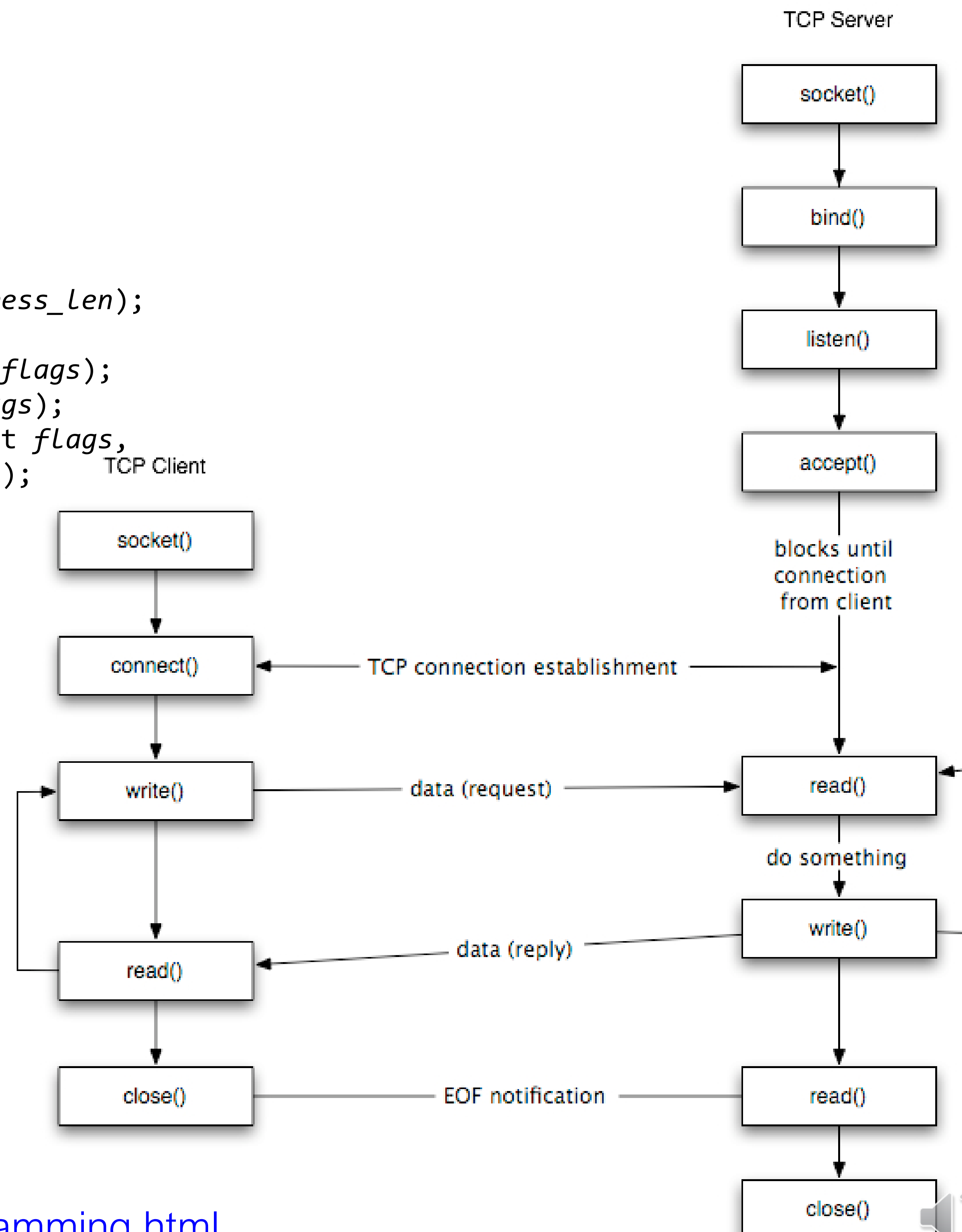
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

- The Socket API is a **standard interface**, which defines **data structures** and **functions** for writing client-server applications.
- It has originally been developed in the context of the Unix operating system and specified as a C API.
- It is now available **across nearly all operating systems and programming environments**.

<sys/socket.h>

```
int     accept(int socket, struct sockaddr *address,
             socklen_t *address_len);
int     bind(int socket, const struct sockaddr *address,
             socklen_t address_len);
int     connect(int socket, const struct sockaddr *address,
             socklen_t address_len);
int     getpeername(int socket, struct sockaddr *address,
             socklen_t *address_len);
int     getsockname(int socket, struct sockaddr *address,
             socklen_t *address_len);
int     getsockopt(int socket, int level, int option_name,
             void *option_value, socklen_t *option_len);
int     listen(int socket, int backlog);
ssize_t recv(int socket, void *buffer, size_t length, int flags);
ssize_t recvfrom(int socket, void *buffer, size_t length,
             int flags, struct sockaddr *address, socklen_t *address_len);
ssize_t recvmsg(int socket, struct msghdr *message, int flags);
ssize_t send(int socket, const void *message, size_t length, int flags);
ssize_t sendmsg(int socket, const struct msghdr *message, int flags);
ssize_t sendto(int socket, const void *message, size_t length, int flags,
             const struct sockaddr *dest_addr, socklen_t dest_len);
int     setsockopt(int socket, int level, int option_name,
             const void *option_value, socklen_t option_len);
int     shutdown(int socket, int how);
int     socket(int domain, int type, int protocol);
int     socketpair(int domain, int type, int protocol,
             int socket_vector[2]);
```

HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch



http://www.cs.dartmouth.edu/~campbell/cs50/socketprogramming.html

Using the Socket API for a TCP **Server**

HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

```
1. Create a "receptionist" socket

2. Bind the socket to an IP address / port

3. Loop
   3.1. Accept an incoming connection (block until a client arrives)
   3.2. Receive a new socket when a client has arrived
   3.3. Read and write bytes through this socket, communicating
        with the client
   3.4. Close the client socket (and go back to listening)

4. Close the "receptionist" socket
```

Using the Socket API for a TCP **Client**

HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

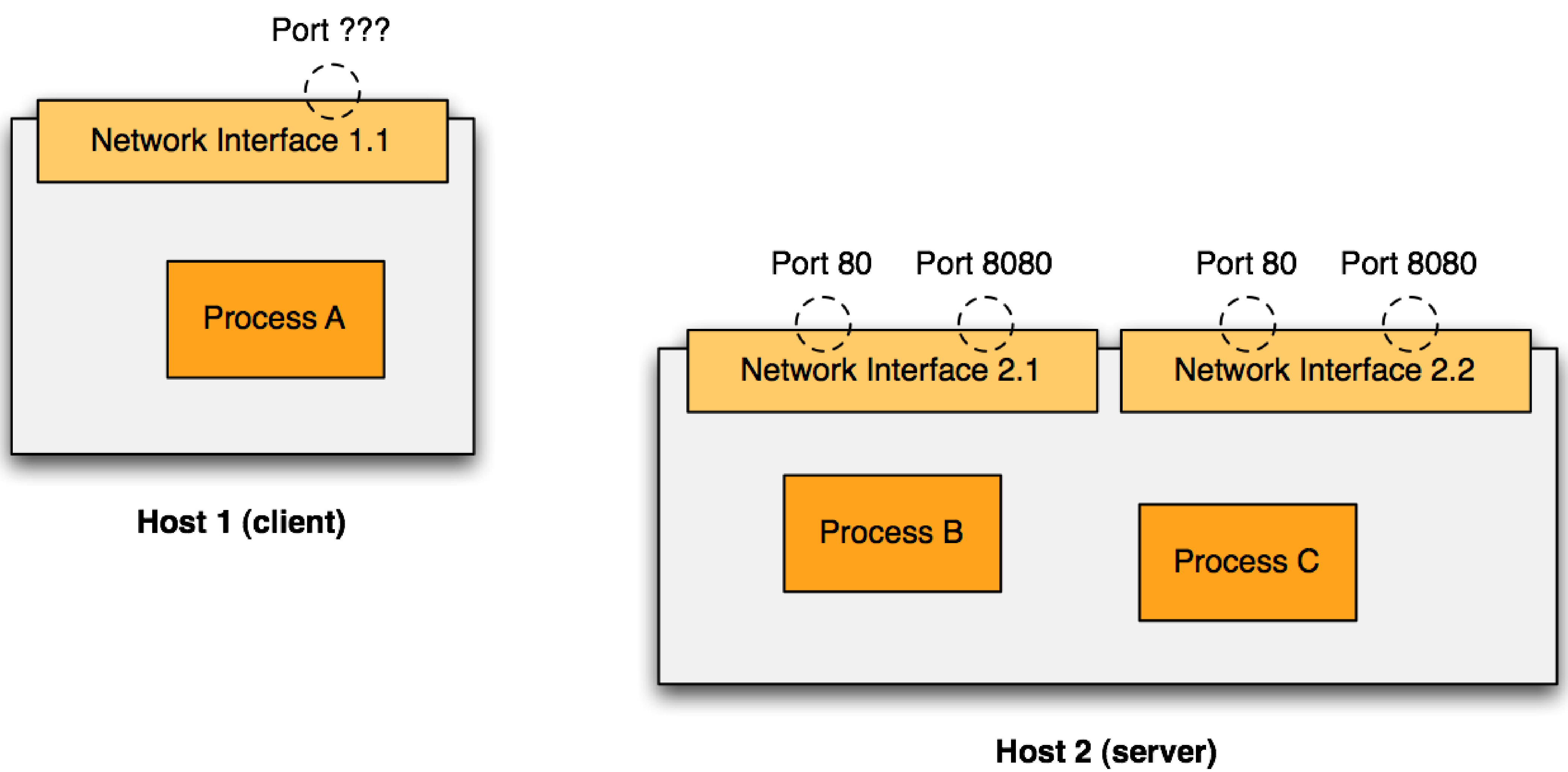
1. Create a **socket**

2. Make a **connection request** on an IP address / port

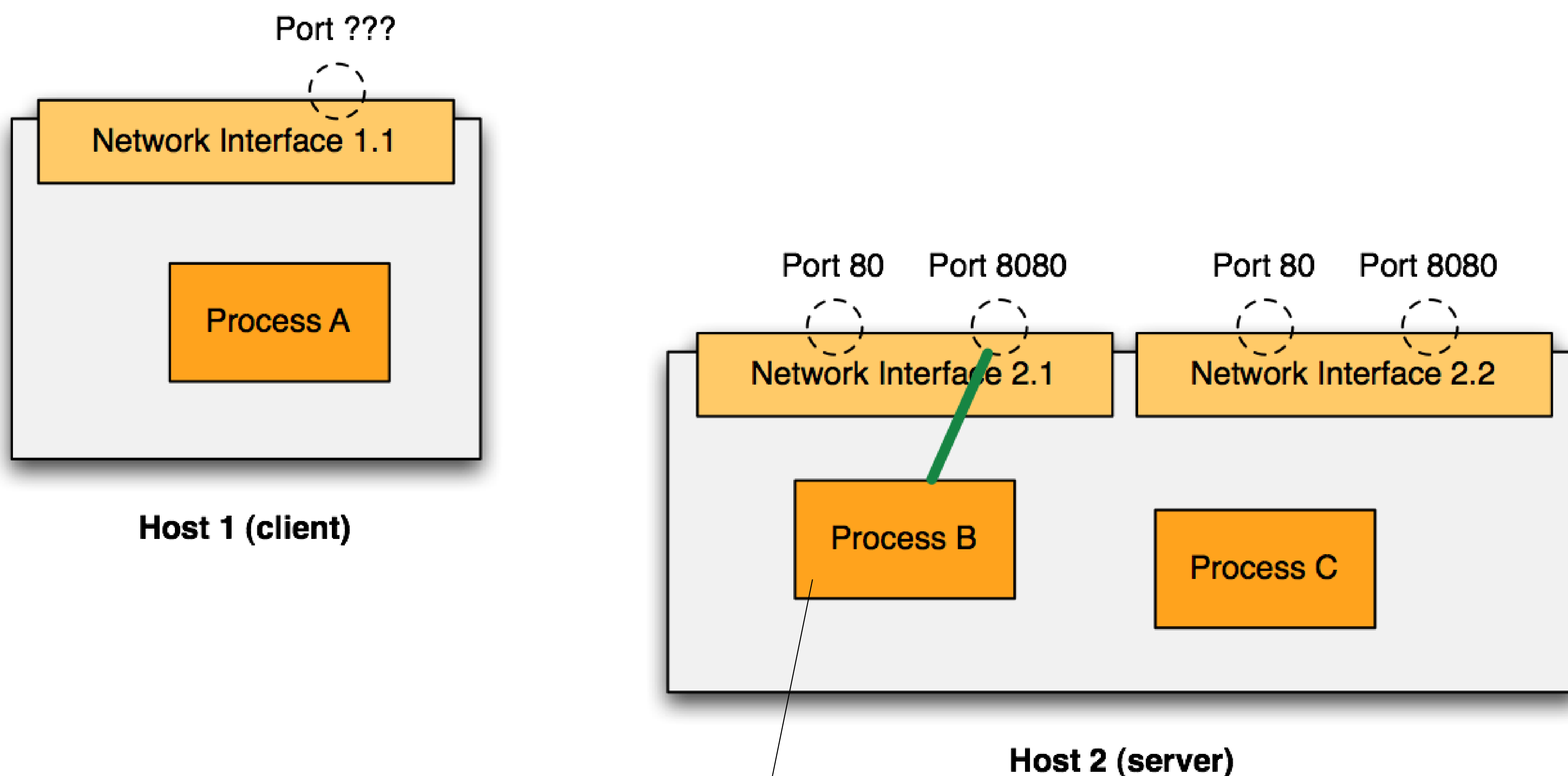
3. **Read** and **write** bytes through this socket, communicating with the client

4. **Close** the client socket

Using the Socket API



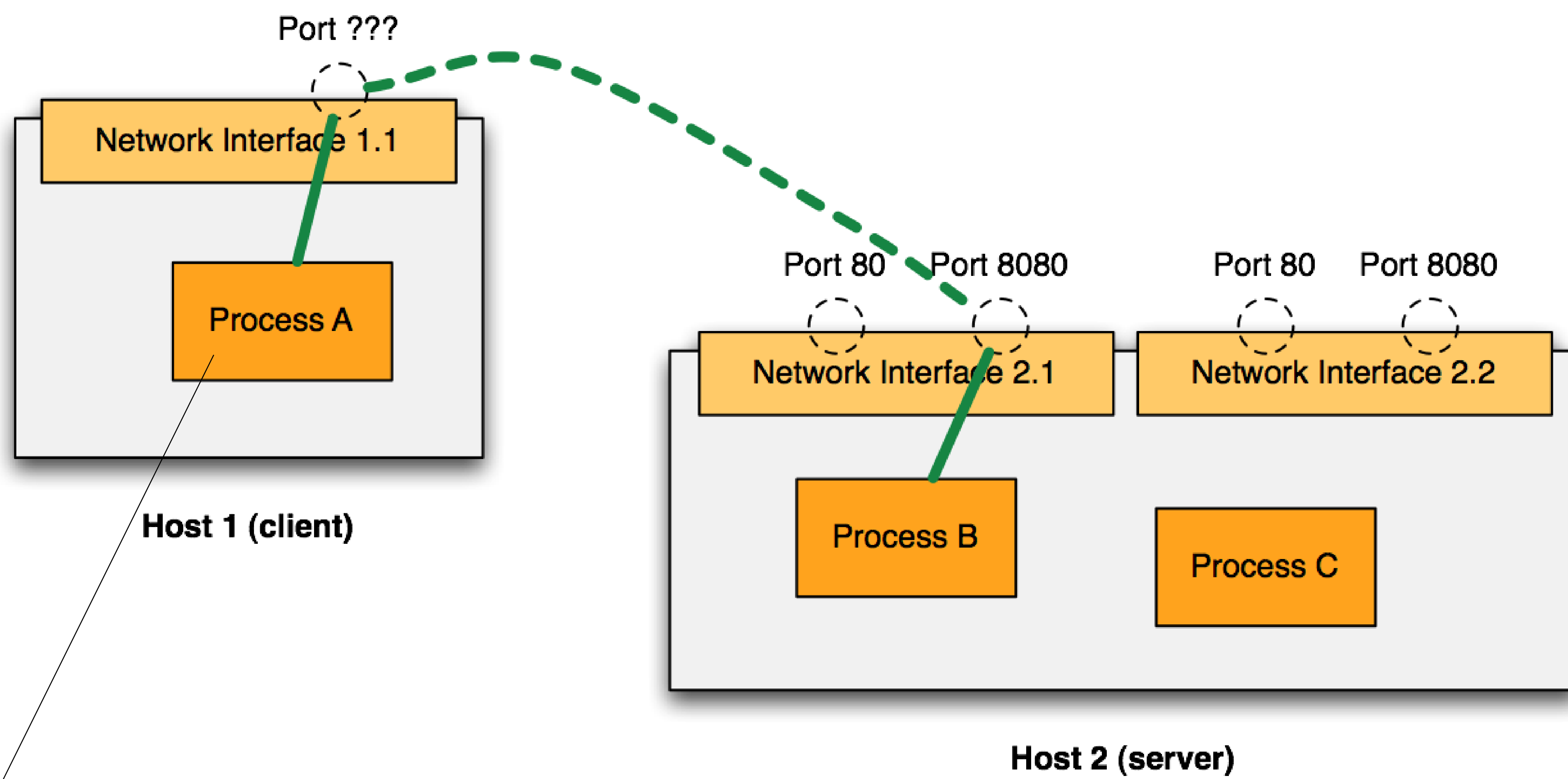
Using the Socket API in Java




```java
// Listen on port 8080
ServerSocket serverSocket = new ServerSocket(8080);

// Wait (block) until a client makes a connection request...
Socket commSocket = serverSocket.accept();
```

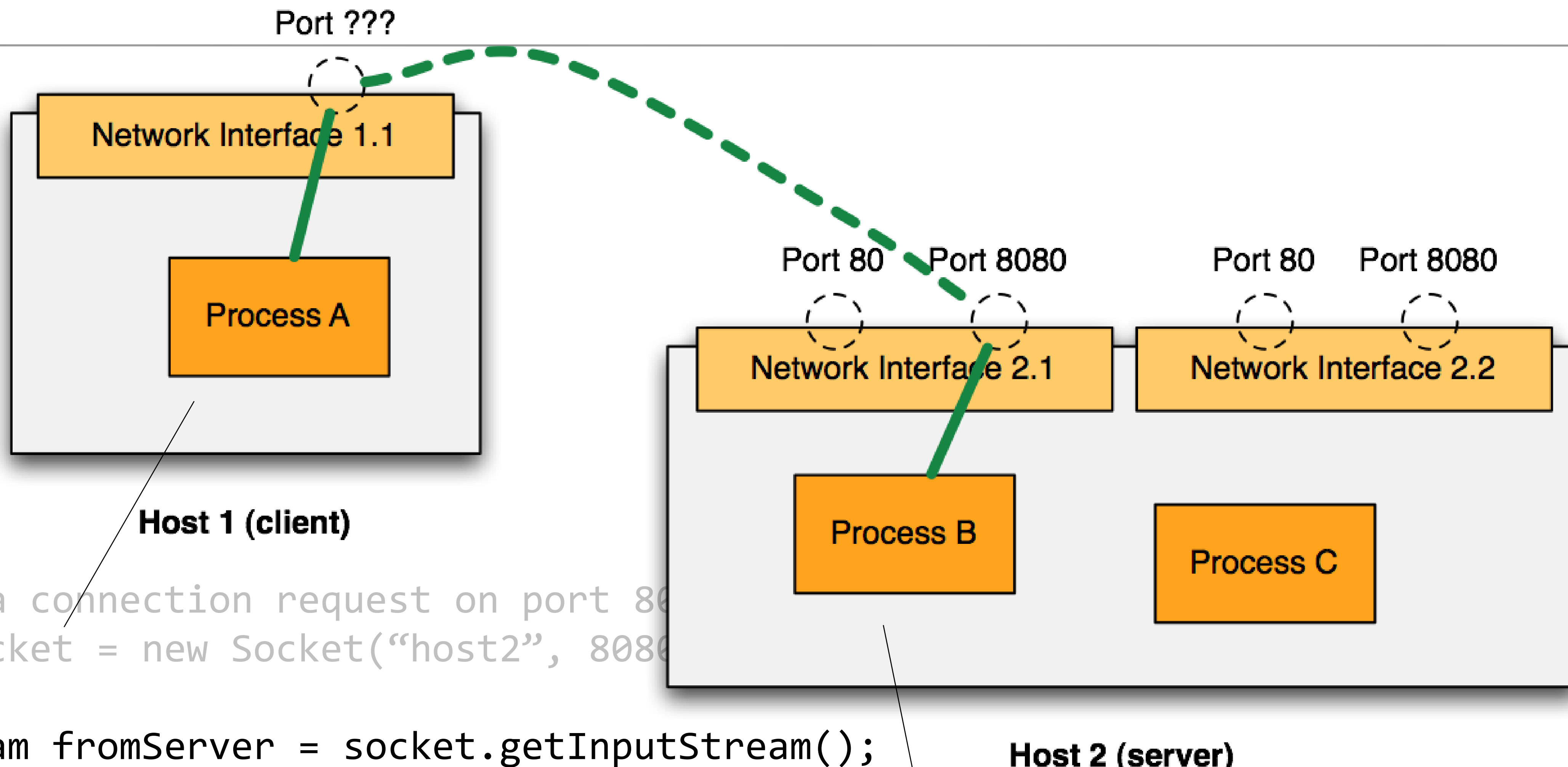
Using the Socket API in Java

HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch




```
// Makes a connection request on port 8080
Socket socket = new Socket("host2", 8080);
```

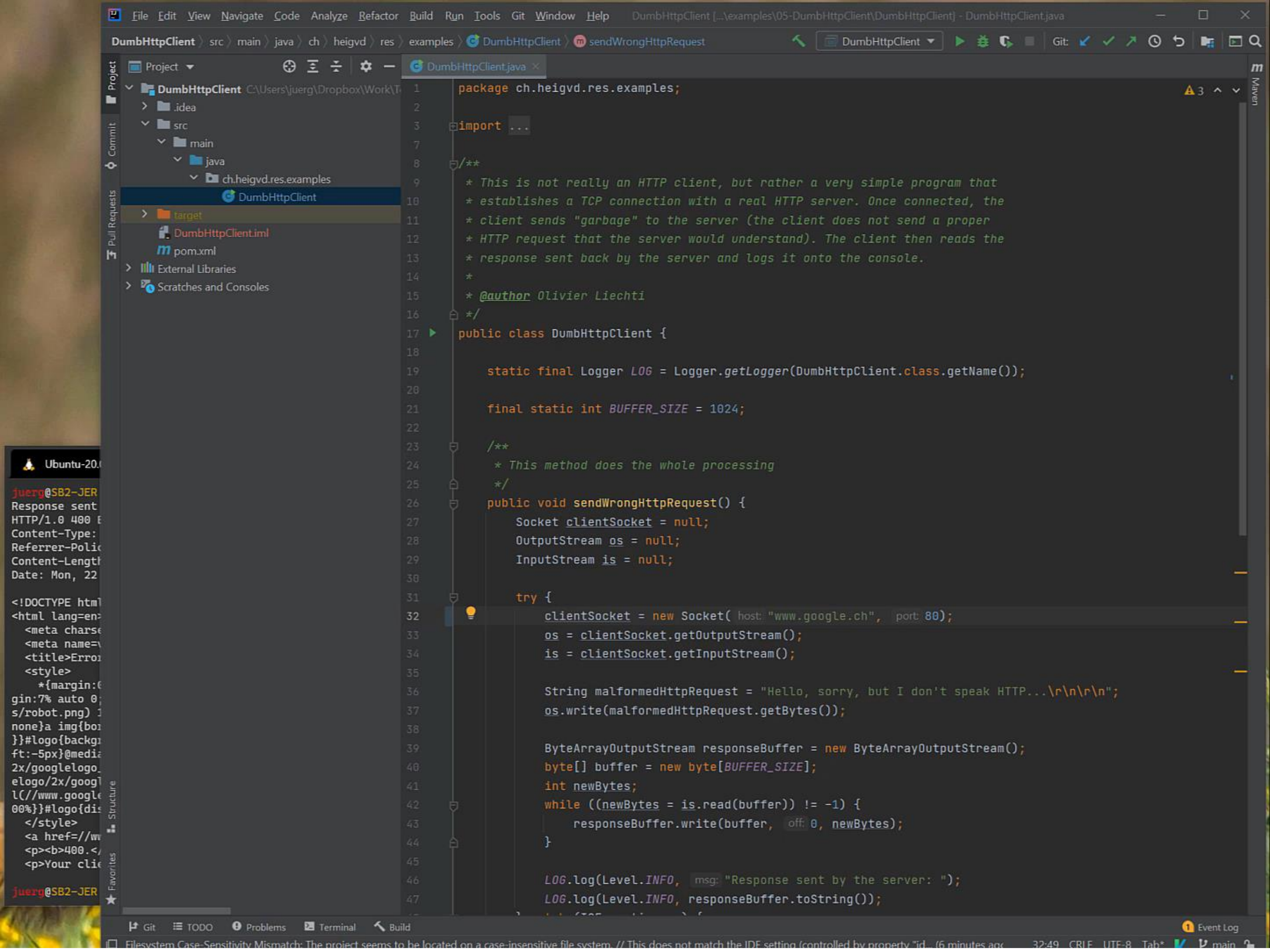
Using the Socket API in Java




```
// Makes a connection request on port 8
Socket socket = new Socket(“host2”, 808

InputStream fromServer = socket.getInputStream();
OutputStream toServer = socket.getOutputStream();
```

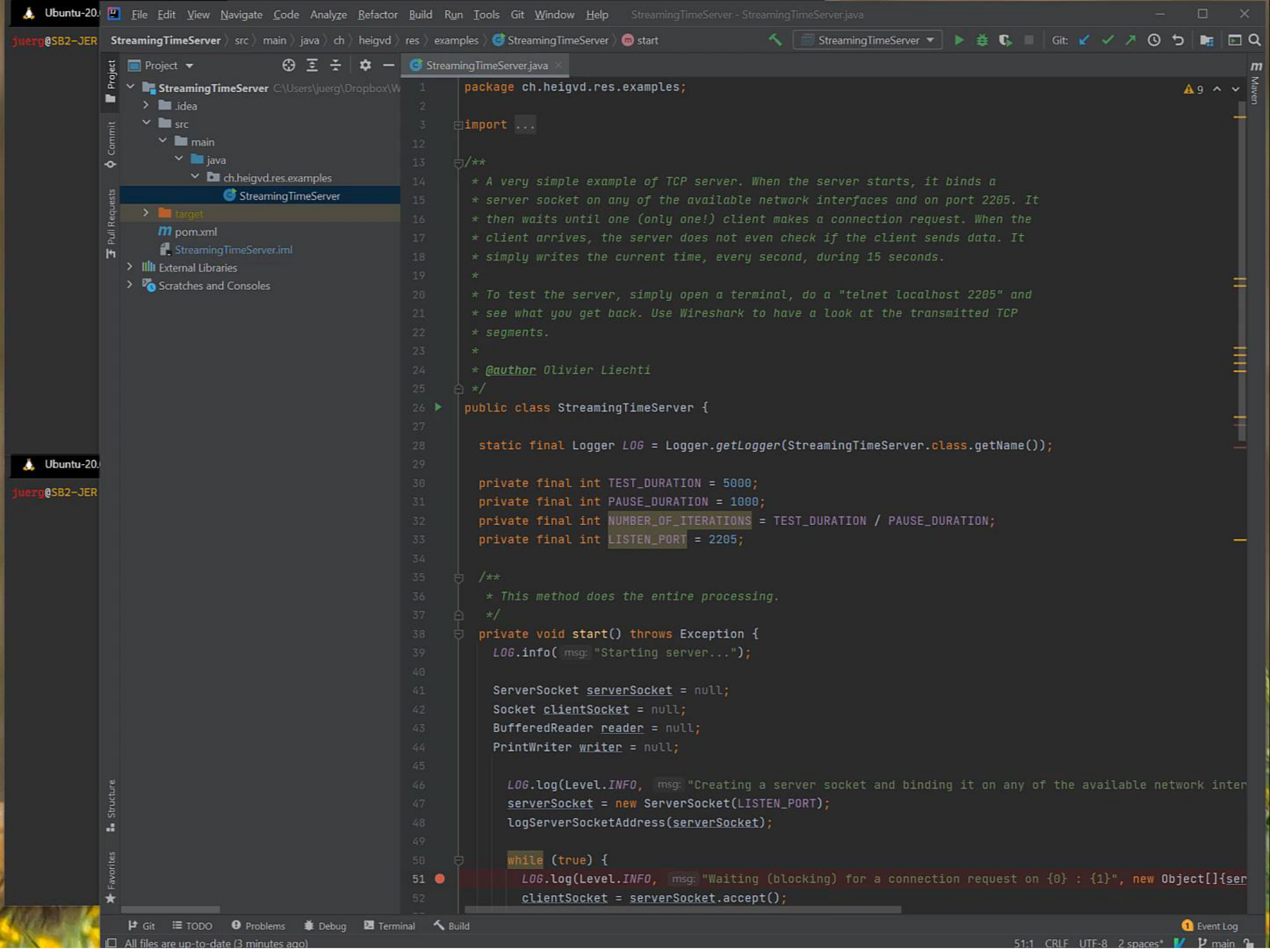
```
// Listen on port 8080
ServerSocket serverSocket = new ServerSocket(8080);

// Wait until a client makes a connection request...
Socket commSocket = serverSocket.accept();

InputStream fromClient = commSocket.getInputStream();
OutputStream toClient = commSocket.getOutputStream();
```

Example: **05-DumbHttpClient**



DumbHttpClient.java
package ch.heigvd.res.examples;
import ...
/**
 * This is not really an HTTP client, but rather a very simple program that
 * establishes a TCP connection with a real HTTP server. Once connected, the
 * client sends "garbage" to the server (the client does not send a proper
 * HTTP request that the server would understand). The client then reads the
 * response sent back by the server and logs it onto the console.
 *
 * @author Olivier Liechti
 */
public class DumbHttpClient {
    static final Logger LOG = Logger.getLogger(DumbHttpClient.class.getName());
    final static int BUFFER_SIZE = 1024;
    /**
     * This method does the whole processing
     */
    public void sendWrongHttpRequest() {
        Socket clientSocket = null;
        OutputStream os = null;
        InputStream is = null;
        try {
            clientSocket = new Socket(host: "www.google.ch", port: 80);
            os = clientSocket.getOutputStream();
            is = clientSocket.getInputStream();
            String malformedHttpRequest = "Hello, sorry, but I don't speak HTTP...\r\n\r\n";
            os.write(malformedHttpRequest.getBytes());
            ByteArrayOutputStream responseBuffer = new ByteArrayOutputStream();
            byte[] buffer = new byte[BUFFER_SIZE];
            int newBytes;
            while ((newBytes = is.read(buffer)) != -1) {
                responseBuffer.write(buffer, off: 0, newBytes);
            }
            LOG.log(Level.INFO, msg: "Response sent by the server: ");
            LOG.log(Level.INFO, responseBuffer.toString());

Example: **04-StreamingTimeServer**



Ubuntu-20.
juerg@SB2-JER
File Edit View Navigate Code Analyze Refactor Build Run Tools Git Window Help
StreamingTimeServer - StreamingTimeServer.java
StreamingTimeServer src main java ch heigvd res examples StreamingTimeServer start
Project
StreamingTimeServer C:\Users\juerg\Dropbox\W
.idea
src
main
java
ch.heigvd.res.examples
StreamingTimeServer
target
pom.xml
StreamingTimeServer.iml
External Libraries
Scratches and Consoles
StreamingTimeServer.java
package ch.heigvd.res.examples;
import ...
/**
 * A very simple example of TCP server. When the server starts, it binds a
 * server socket on any of the available network interfaces and on port 2205. It
 * then waits until one (only one!) client makes a connection request. When the
 * client arrives, the server does not even check if the client sends data. It
 * simply writes the current time, every second, during 15 seconds.
 *
 * To test the server, simply open a terminal, do a "telnet localhost 2205" and
 * see what you get back. Use Wireshark to have a look at the transmitted TCP
 * segments.
 *
 * @author Olivier Liechti
 */
public class StreamingTimeServer {
static final Logger LOG = Logger.getLogger(StreamingTimeServer.class.getName());
private final int TEST_DURATION = 5000;
private final int PAUSE_DURATION = 1000;
private final int NUMBER_OF_ITERATIONS = TEST_DURATION / PAUSE_DURATION;
private final int LISTEN_PORT = 2205;
/**
 * This method does the entire processing.
 */
private void start() throws Exception {
LOG.info(msg: "Starting server...");
ServerSocket serverSocket = null;
Socket clientSocket = null;
BufferedReader reader = null;
PrintWriter writer = null;
LOG.log(Level.INFO, msg: "Creating a server socket and binding it on any of the available network inter
serverSocket = new ServerSocket(LISTEN_PORT);
logServerSocketAddress(serverSocket);
while (true) {
LOG.log(Level.INFO, msg: "Waiting (blocking) for a connection request on {0} : {1}", new Object[]{ser
clientSocket = serverSocket.accept();
Git TODO Problems Debug Terminal Build
Event Log
All files are up-to-date (3 minutes ago)
51:1 CRLF UTF-8 2 spaces* main

End of part 1