# TCP Programming

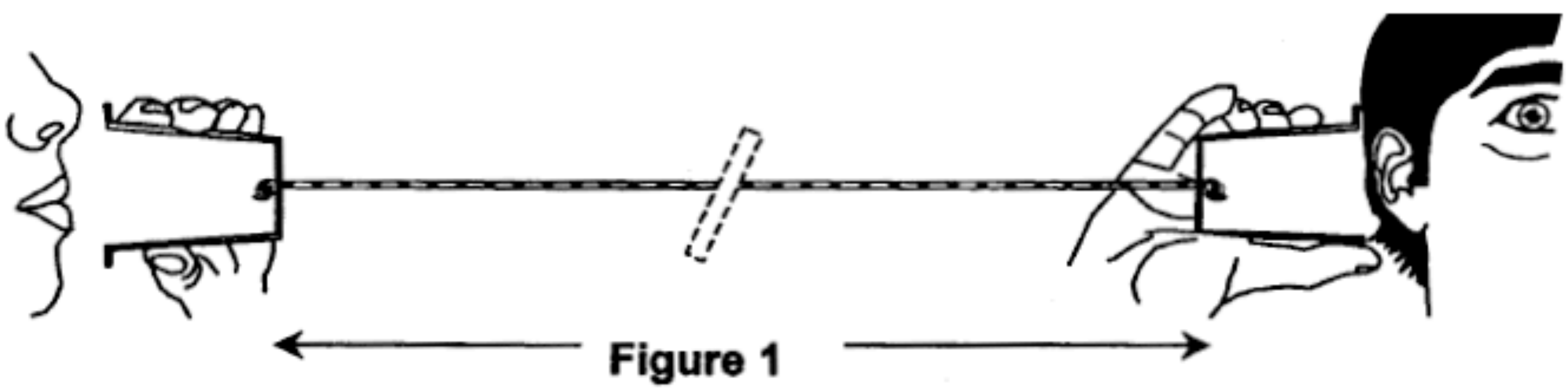
RES, Lecture 2

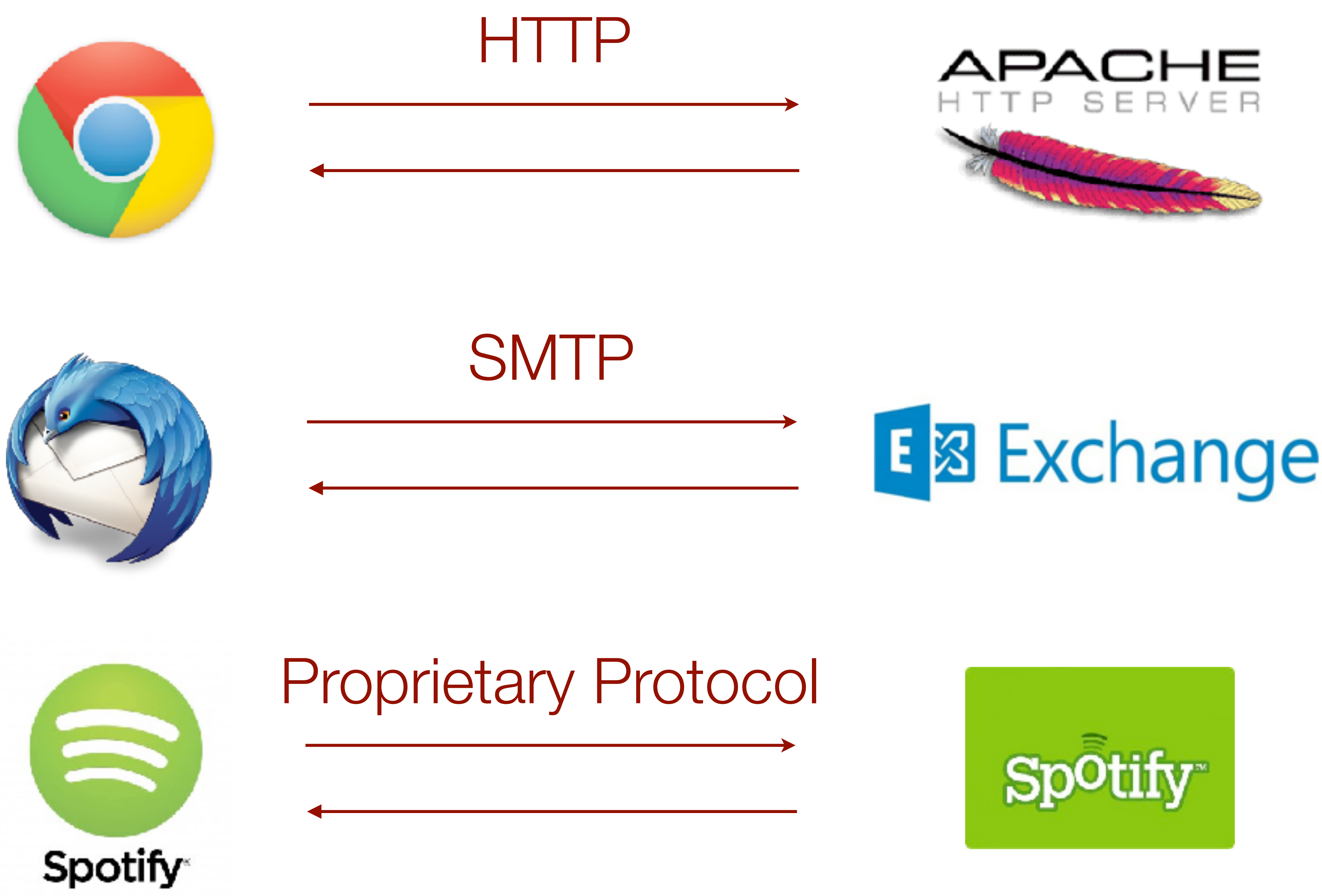
Olivier Liechti

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

www.heig-vd.ch

# Client-Server Programming




HTTP
APACHE
HTTP SERVER
SMTP
Exchange
Proprietary Protocol
Spotify
Spotify

# What is an Application-Level Protocol?



- **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?*

# The TCP Protocol

TCP


UDP

# Transport Protocols

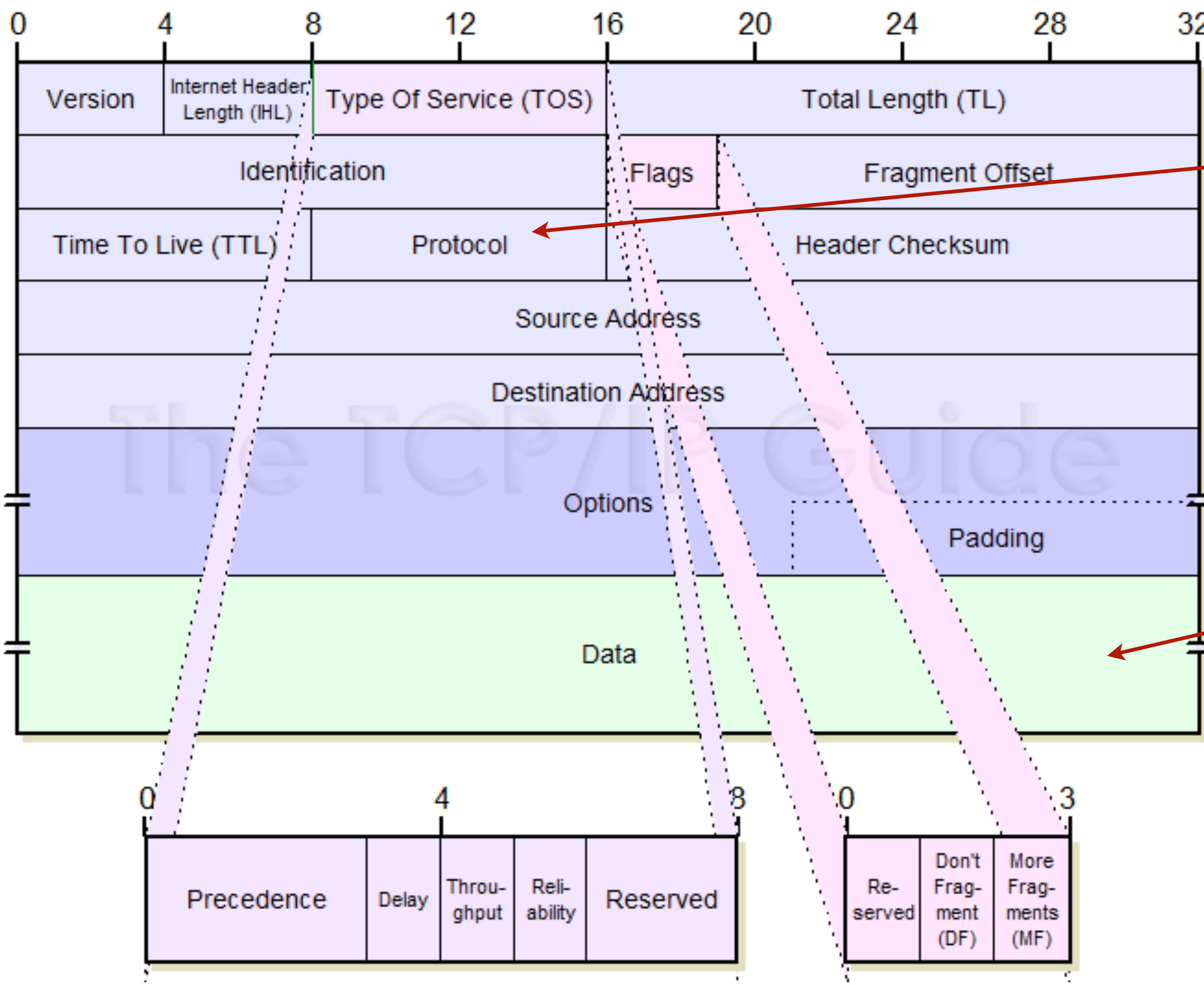


- Both TCP and UDP are **transport protocols**.
- This means that they make it possible for **two programs** (i.e. applications, processes) possibly running on **different machines** to **exchange data**.
- The two protocols also make it possible for several programs to **share the same network interface**. 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



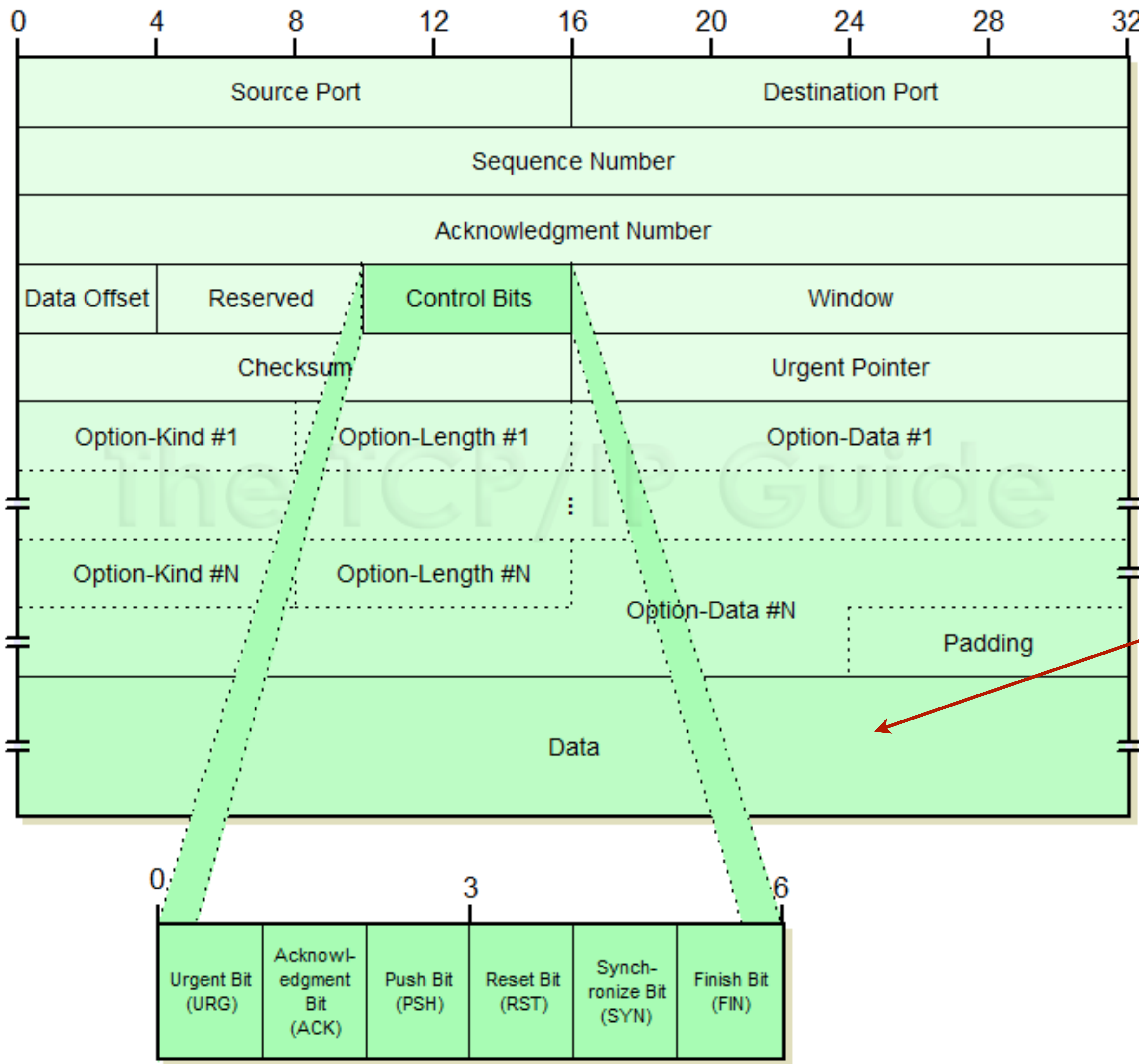
- TCP provides a **connection-oriented service**. The client and the server first have to establish a connection. They can then exchange data through a **bi-directional stream of bytes**.
- TCP provides a **reliable data transfer service.** It makes sure that all bytes sent by one program are received 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...

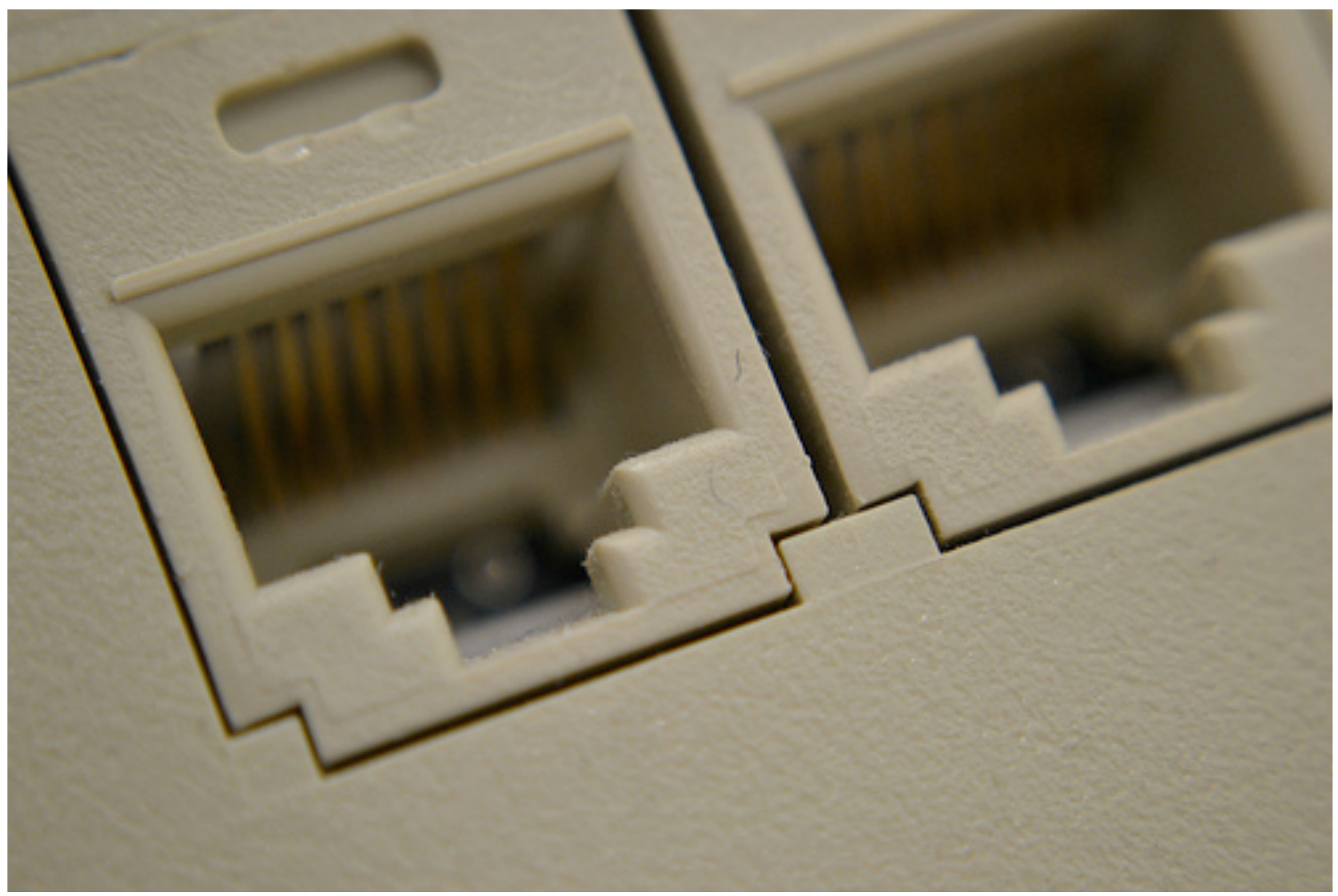
http://www.tcpipguide.com/free/t_TCPMessageSegmentFormat-3.htm

Example: **telnet www.heig-vd.ch 80**

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

Example (client): **nc localhost 2019**

# 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

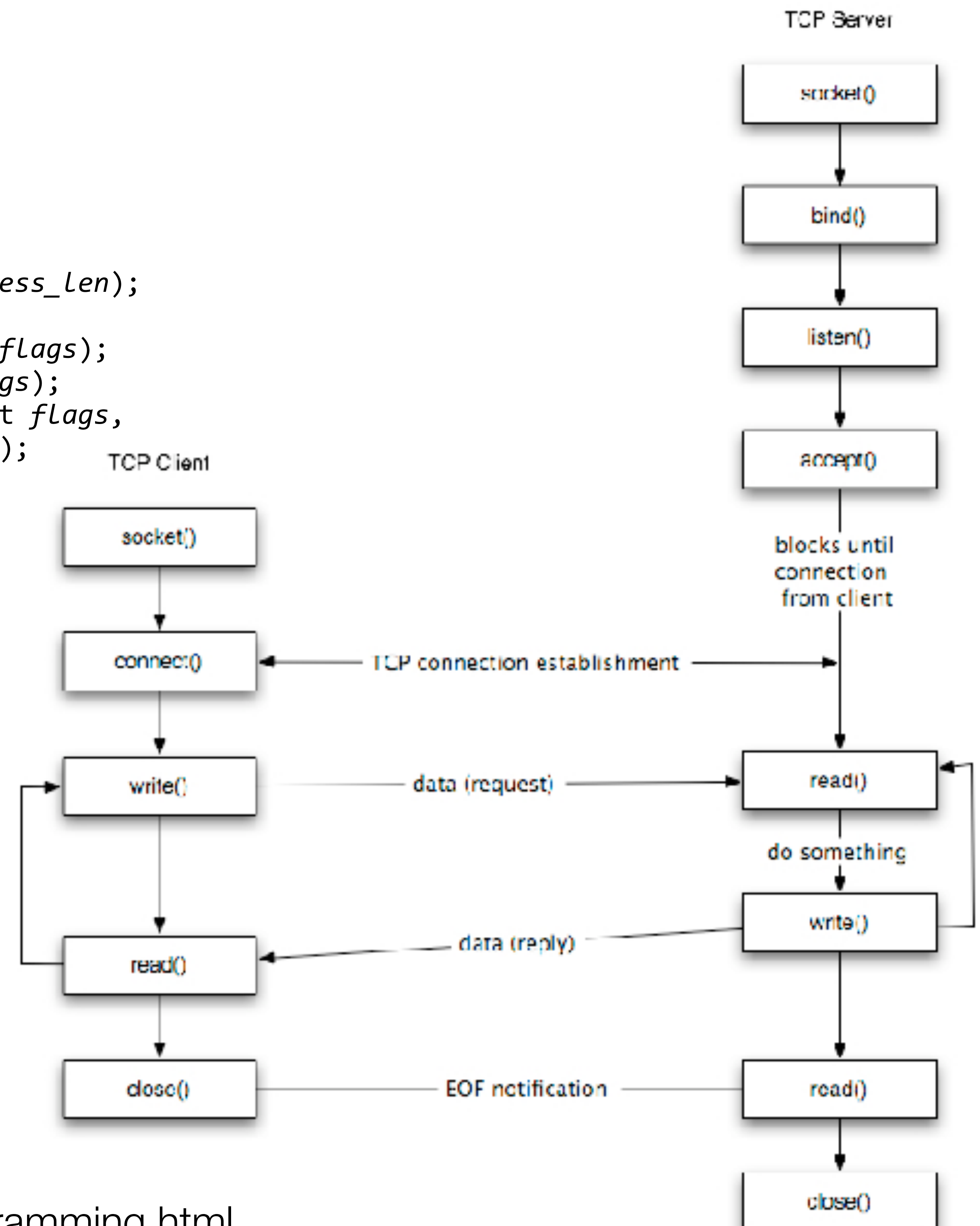


- The Socket API is a **standard interface**, which defines **data structures** and **functions** for writing client-server applications.
- It has originally been developed in the context of the Unix operating system and specified as a C API.
- It is now available **across nearly all operating systems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

```
<sys/socket.h>
```

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

```
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]);
```




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
```

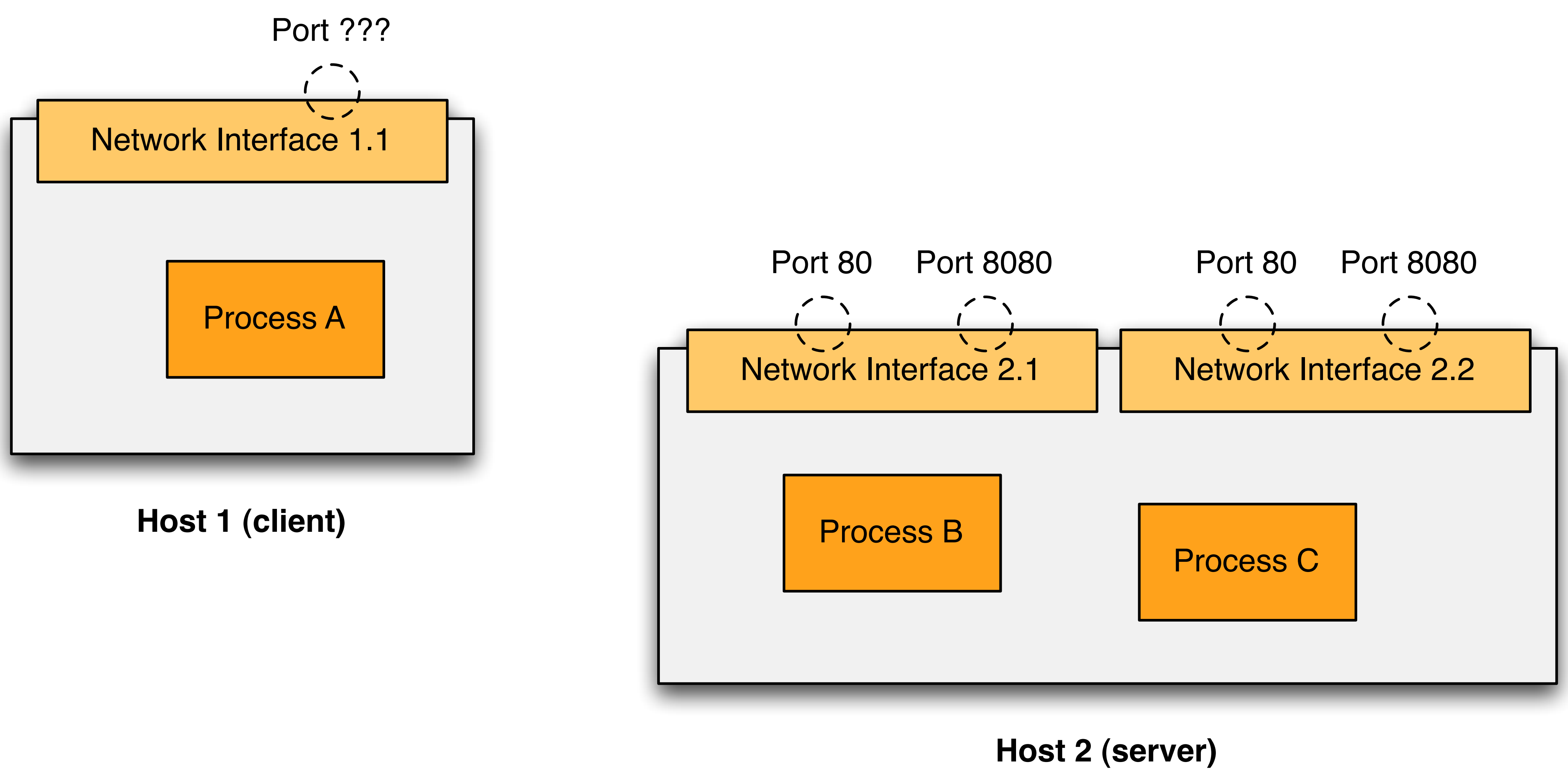
# Using the Socket API for a TCP **Client**



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

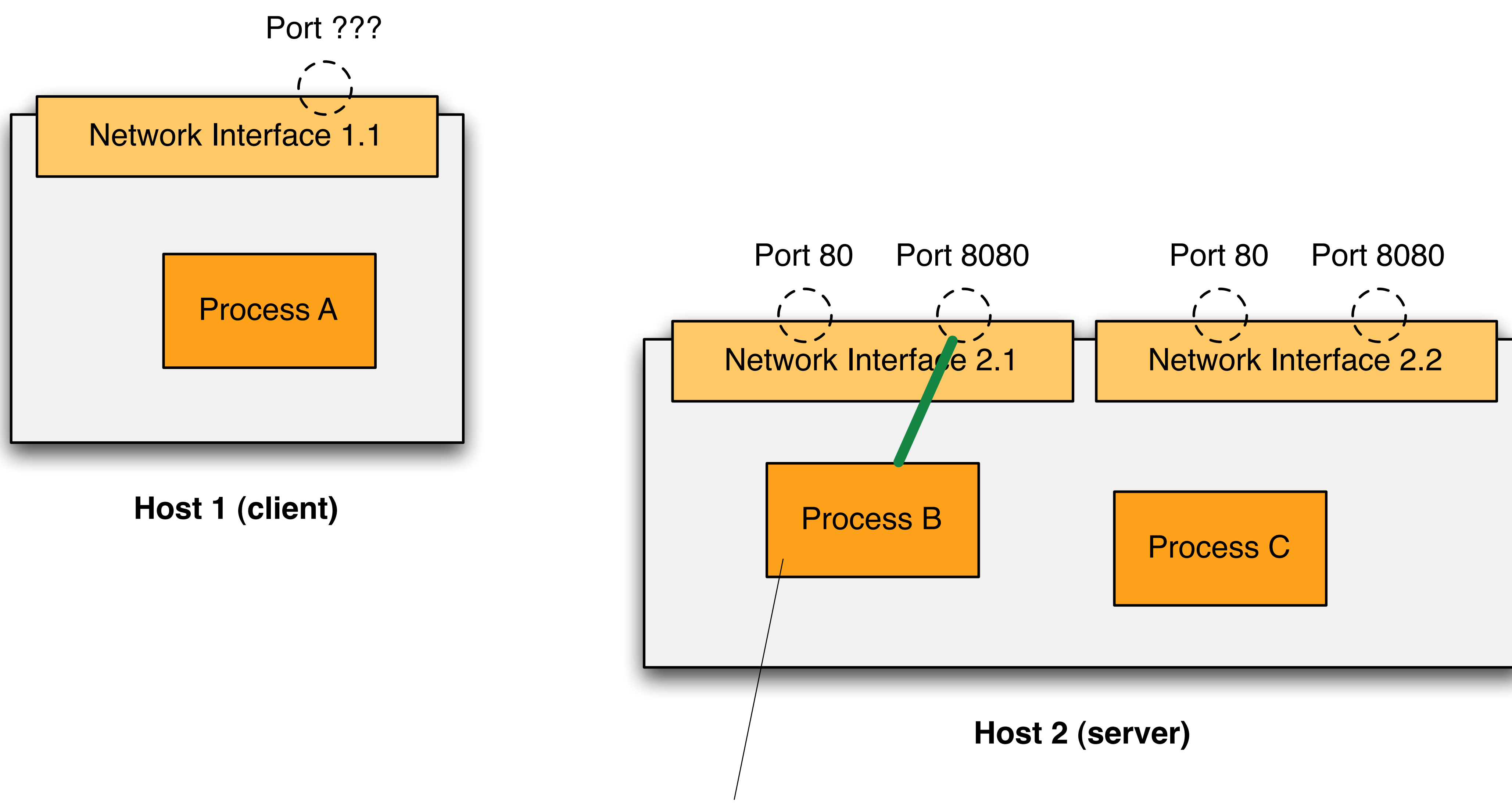
# Using the Socket API





# Using the Socket API in Java

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

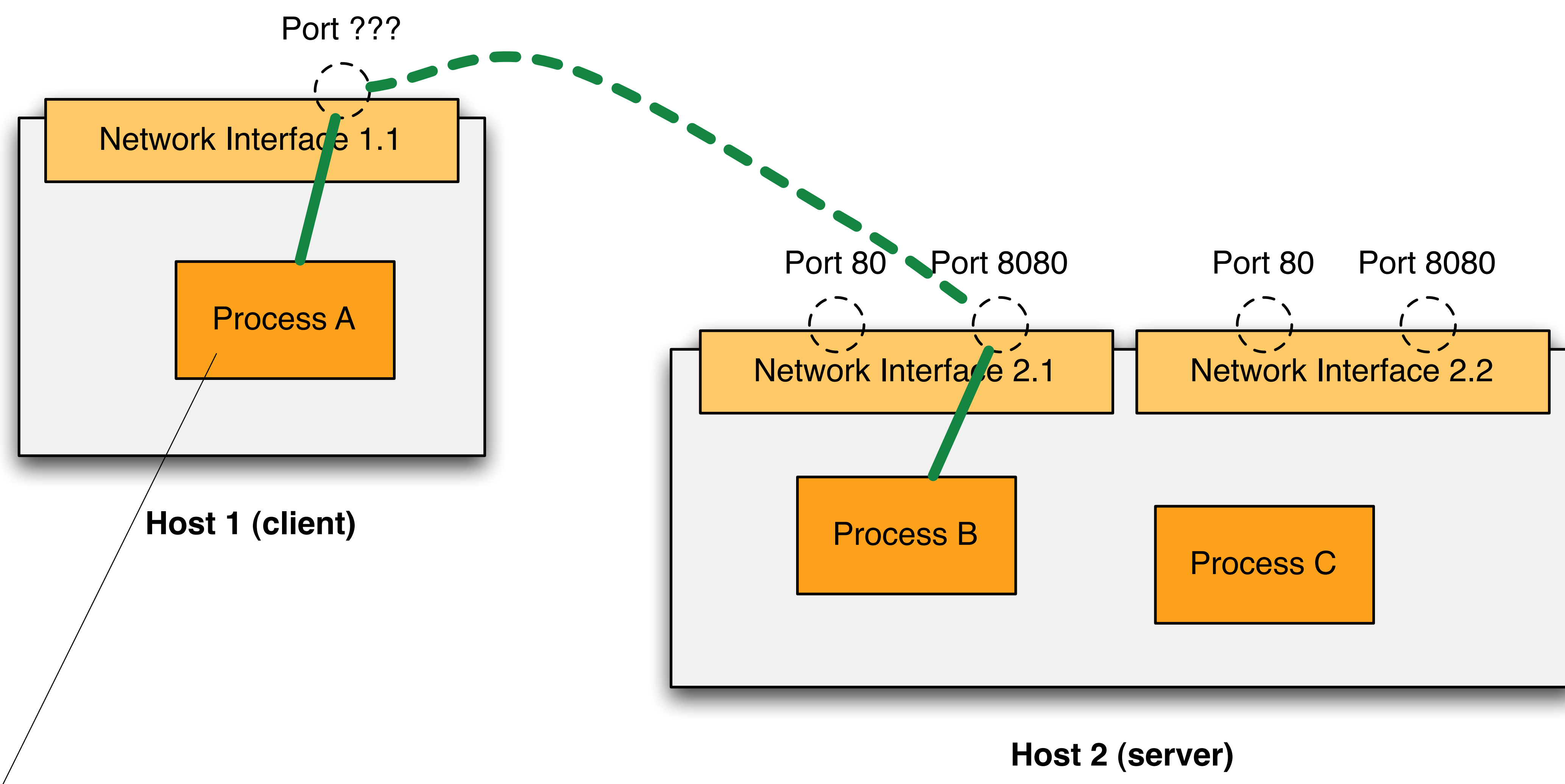



```java
// Listen on port 8080
ServerSocket serverSocket = new ServerSocket(8080);

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

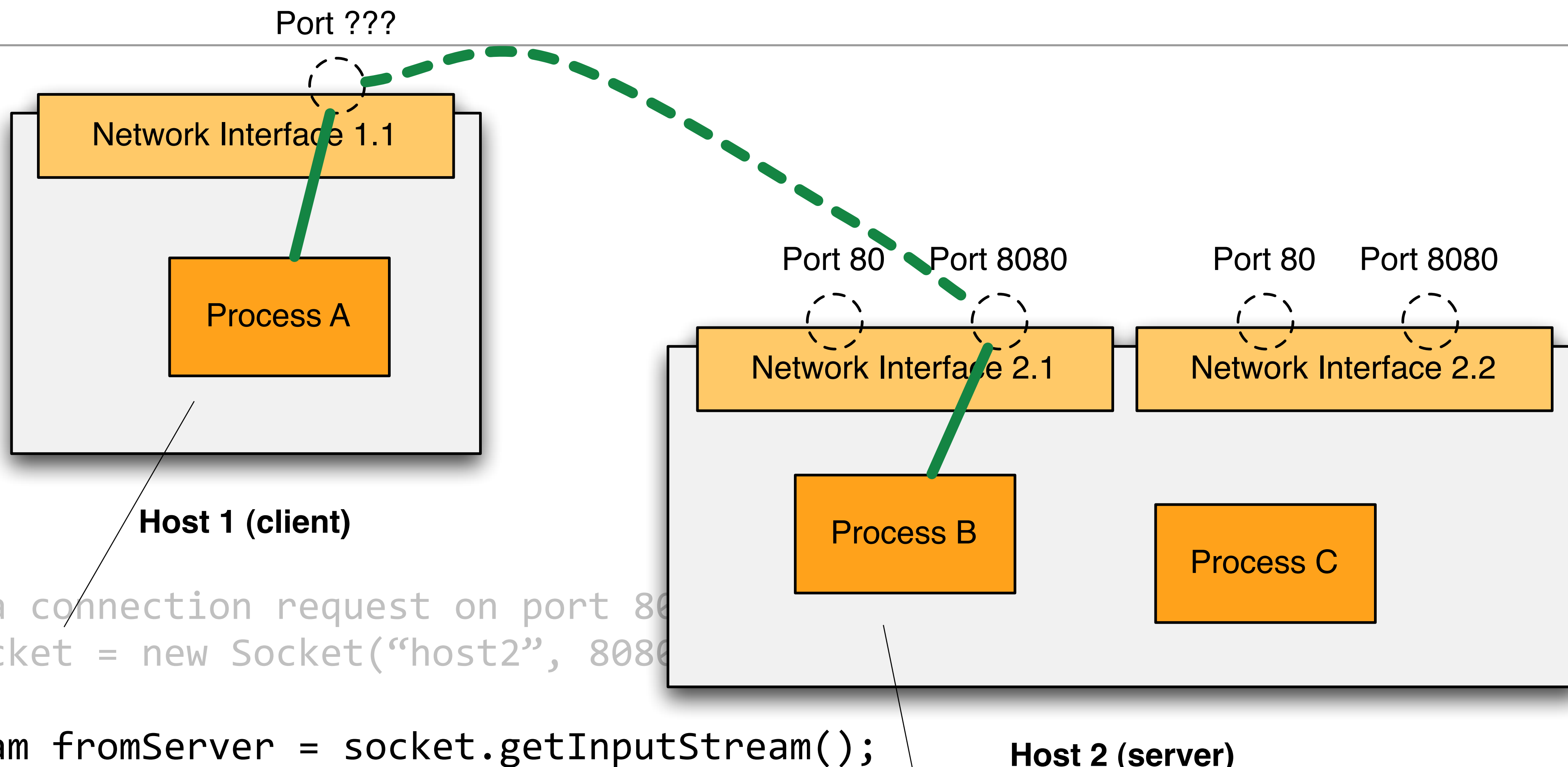
# Using the Socket API in Java

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




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

# Using the Socket API in Java




```
// Makes a connection request on port 80
Socket socket = new Socket("host2", 8080

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**

# Code walkthrough



**establish a connection with server**

```
public void sendWrongHttpRequest() {
     Socket clientSocket = null;
     OutputStream os = null;
     InputStream is = null;

     try {
          clientSocket = new Socket("www.lematin.ch", 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, 0, newBytes);
          }

          LOG.log(Level.INFO, "Response sent by the server: ");
          LOG.log(Level.INFO, responseBuffer.toString());
     } catch (IOException ex) {
          LOG.log(Level.SEVERE, null, ex);
     } finally {
...
     }
}
```

**get streams to send and receive bytes**

**read bytes sent by the server until the connection is closed**

Example: **04-StreamingTimeServer**

# Code walkthrough

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

bind on TCP port

```
ServerSocket serverSocket = null;
Socket clientSocket = null;
BufferedReader reader = null;
PrintWriter writer = null;

try {
  serverSocket = new ServerSocket(listenPort);
  logServerSocketAddress(serverSocket);
  clientSocket = serverSocket.accept();

  logSocketAddress(clientSocket);
  reader = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
  writer = new PrintWriter(clientSocket.getOutputStream());

 for (int i = 0; i < numberOfIterations; i++) {
    writer.println(String.format("{'time' : '%s'}", new Date()));
    writer.flush();
    LOG.log(Level.INFO, "Sent data to client, doing a pause...");
    Thread.sleep(pauseDuration);
  }
} catch (IOException | InterruptedException ex) {
  LOG.log(Level.SEVERE, ex.getMessage());
} finally {
   reader.close();
   writer.close();
   clientSocket.close();
   serverSocket.close();
}
```

block until a client makes a connection request

we want to exchange characters with the clients (we should specify the encoding!)

we make sure to flush the buffer, so that characters are actually sent!

# Handling Concurrency

# Concurrency in Network Programming

***You don't want your server to talk to only one client  at the time, do you?***

*Even for* ***stateless*** *protocols…*


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

**blocking IO (synchronous)**

n employee = n threads

employees are expensive

limited space for employees in the truck

few employees => long queue

**non-blocking IO (asynchronous)**

there is only 1 employee (1 thread)

customers are called back when the request is fulfilled

no queue

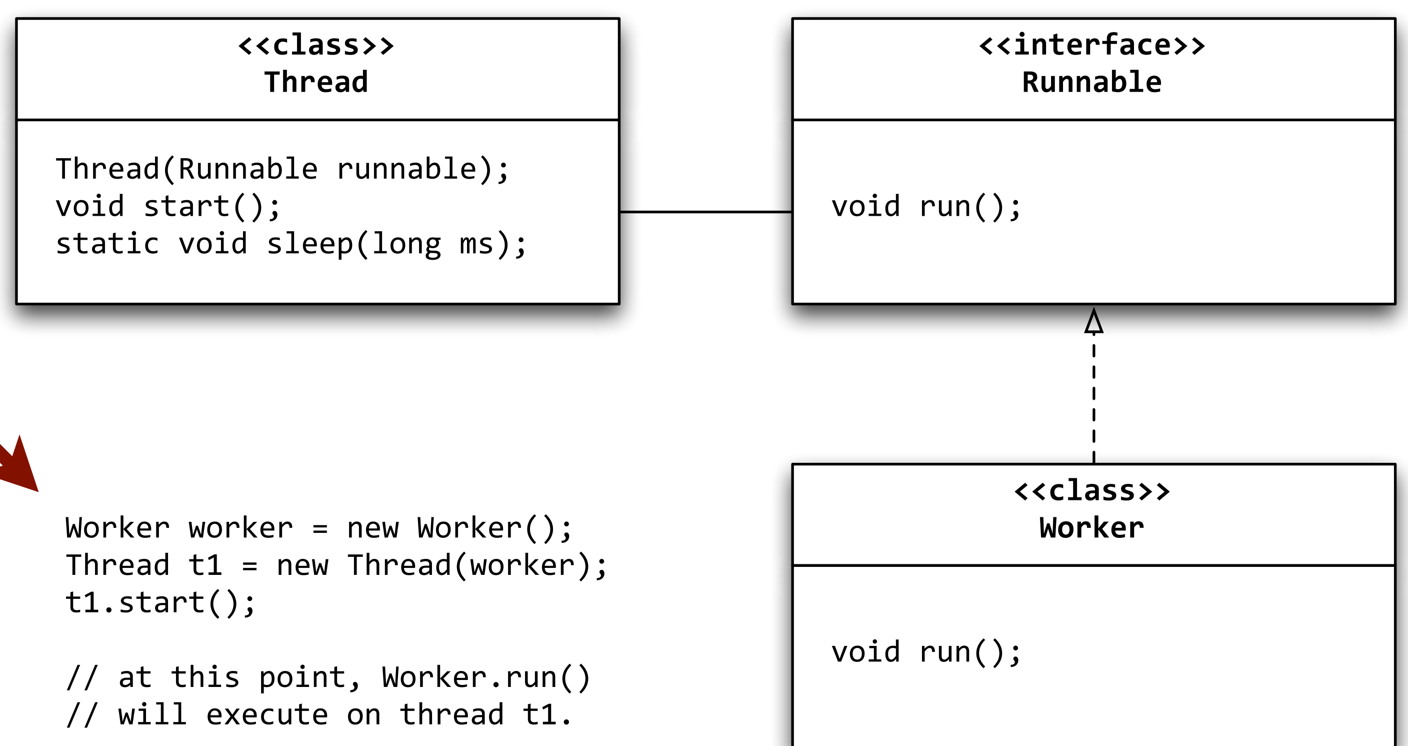
# Concurrent Programming



- On top of the **operating system**, it is possible launch the Java Virtual Machine (**JVM**) several times (by invoking the java command). In this scenario, there is **one process (program) for every JVM instance**.
- If you don't do anything special, there is a **single execution thread** within each JVM. This means that all instructions in your code are executed **sequentially**.
- Very often, you write software where you want to **perform several tasks at the same time** (concurrently). For instance:
  - Manage a UI **while** fetching data from the network,
  - Talking to one HTTP client **while** talking to another HTTP client,
  - Have a worker do complex calculations on a subset of the data, **while** having another worker do the same calculations on another subset.
- You can use **threads** (also called **lightweight processes**) for this purpose.

# Concurrent Programming in Java

- In Java, there are two main types
  - The **Thread class**, which *could be extended* to implement the behaviour you want to run in parallel.
  - The **Runnable interface**, which *is implemented* for the same purpose and is passed as an argument to the Thread constructor.




```
Worker worker = new Worker();
Thread t1 = new Thread(worker);
t1.start();

// at this point, Worker.run()
// will execute on thread t1.
```

# Concurrent Programming in Java



- There are other classes related to threads, in the java.util.concurrent package. An important one is the **ExecutorService**, which makes it possible to use **thread pools**.
- A thread pool gives you a way to limit the number of threads spawned (by your server), so that you will not consume all resources. Others are queued.

```
package de.vogella.concurrency.threadpools;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class Main {
  private static final int NTHREDS = 10;

  public static void main(String[] args) {
    ExecutorService executor = Executors.newFixedThreadPool(NTHREDS);
    for (int i = 0; i < 500; i++) {
      Runnable worker = new MyRunnable(10000000L + i);
      executor.execute(worker);
    }
    // This will make the executor accept no new threads
    // and finish all existing threads in the queue
    executor.shutdown();
    // Wait until all threads are finish
    executor.awaitTermination();
    System.out.println("Finished all threads");
  }
}
```

http://www.vogella.com/tutorials/JavaConcurrency/article.html

## Single Threaded
## Single Process
## Blocking

**Not really an option...**

The server implements a loop.

It waits for a client to arrive.
Then services the client until done.

Then only goes back to accept the next client.

Can only talk to 1 client at the time

It is only when we reach this line that a new client can connect

```
serverSocket = new ServerSocket(port);
while (true) {

   clientSocket = serverSocket.accept();

   in = new BufferedReader(new
   InputStreamReader(clientSocket.getInputStream()));
   out = new PrintWriter(clientSocket.getOutputStream());
   String line;
   boolean shouldRun = true;

   LOG.info("Reading until client sends BYE");

   while ( (shouldRun) && (line = in.readLine()) != null ) {
      if (line.equalsIgnoreCase("bye")) {
         shouldRun = false;
      }
      out.println("> " + line.toUpperCase());
      out.flush();
   }
   clientSocket.close();
   in.close();
   out.close();
}
```

It takes a long time to serve each client

## Single Threaded
## Multi Process
## Blocking

**How apache httpd did it**
(with pre-fork, kind of...)

The server implements a loop.
It waits for a client to arrive.
When the client arrives, the server forks a new process.

The child process serves the client while the server is immediately ready to serve the next client.

Forking a process is kind of heavy... and resource hungry

While the child process serves the client...

... the parent can immediately welcome the next client.

```
while(1) {  // main accept() loop
    sin_size = sizeof their_addr;
    new_fd = accept(sockfd, (struct sockaddr *)&their_addr,
&sin_size);
    if (new_fd == -1) {
        perror("accept");
        continue;
    }

    inet_ntop(their_addr.ss_family,
        get_in_addr((struct sockaddr *)&their_addr),
        s, sizeof s);
    printf("server: got connection from %s\n", s);

    if (!fork()) { // this is the child process
        close(sockfd); // child doesn't need the listener
        if (send(new_fd, "Hello, world!", 13, 0) == -1)
            perror("send");
        close(new_fd);
        exit(0);
    }
    close(new_fd);  // parent doesn't need this
}
```

http://www.beej.us/guide/bgnet/output/html/multipage/clientserver.html#simpleserver

Multi Threaded
Single Process
Blocking

**The 'old' Java way**

The server uses a first thread to wait for connection requests from clients.

Each time a client arrives, a new thread is created and used to serve the client.

Millions of clients, millions of threads?

Resource hungry.
Not scalable.

The ReceptionistWorker implements a run() method that will execute on its own thread.

```
private class ReceptionistWorker implements Runnable {

   @Override
   public void run() {
      ServerSocket serverSocket;

      try {
         serverSocket = new ServerSocket(port);
      } catch (IOException ex) {
         LOG.log(Level.SEVERE, null, ex);
         return;
      }

      while (true) {
         LOG.log(Level.INFO, "Waiting for a new client");
         try {
            Socket clientSocket = serverSocket.accept();
            LOG.info("A new client has arrived...");
            new Thread(new ServantWorker(clientSocket)).start();
         } catch (IOException ex) {
            LOG.log(Level.SEVERE, ex.getMessage(), ex);
         }
      }
   }
}
```

As soon as a client is connected, a new thread is created.
The code that manages the interaction with the client executes on this thread.

**2 types of workers, n+1 threads**

Example: **07-TcpServers**

Single Thread
Single Process
Asynchronous Programming

**The 'à la Node.js' way**

The server uses a single thread, but in a non-blocking, asynchronous way.

Callback functions have to be written, so that they can be invoked when clients arrive, when data is received, etc.

Different programming logic.
Scalable.

We are registering callback functions on the various types of events that can be notified by the server...

```
// let's create a TCP server
const server = net.createServer();

// it reacts to events: 'listening', 'connection', 'close', etc.
// register callback functions, to be invoked when the events
// occur (everything happens on the same thread)

server.on('listening', callbackFunctionToCallWhenSocketIsBound);
server.on('connection',
callbackFunctionToCallWhenNewClientHasArrived);

//Start listening on port 9907
server.listen(9907);

// This callback is called when the socket is bound and is in
// listening mode. We don't need to do anything special.
function callbackFunctionToCallWhenSocketIsBound() {
   console.log("The socket is bound and listening");
   console.log("Socket value: %j", server.address());
}

// This callback is called after a client has connected.
function callbackFunctionToCallWhenNewClientHasArrived(socket) {
  ...
}
```

... and we code these functions, implementing the behavior that is expected when the events occur.

Select is a blocking operation (with a possible timeout). It blocks until something has happened on one of the provided sets of file descriptors.

Single Thread
Single Process
IO Multiplexing

## The 'select' way

Sockets are set in a non-blocking state, which means that read(), write() and other functions do not block.

System calls such as select() or poll() block, but work on multiple sockets. They return if data has arrived on at least one of the sockets.

Watch out for performance.

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
    fd_set rfds;
    struct timeval tv;
    int retval;

    /* Watch stdin (fd 0) to see when it has input. */
    FD_ZERO(&rfds);
    FD_SET(0, &rfds);
    /* Wait up to five seconds. */
    tv.tv_sec = 5;
    tv.tv_usec = 0;

    retval = select(1, &rfds, NULL, NULL, &tv);
    /* Don't rely on the value of tv now! */

    if (retval == -1)
        perror("select()");
    else if (retval)
        printf("Data is available now.\n");
        /* FD_ISSET(0, &rfds) will be true. */
    else
        printf("No data within five seconds.\n");

    return 0;
}
```

Here, we know that something has happened on one of the sockets. We can iterate over the set of file descriptors and get the data.

Example: **06-PresenceApplication**