

# The API Socket (TCP)

TEchnologies Internet (TEI)

---

Olivier Liechti

heig-vd

Haute Ecole d'Ingénierie et de Gestion  
du Canton de Vaud

**W1**

## **Introduction to Java IO**

Byte and character streams, dealing with files, the decorator pattern, custom reader/writers classes, buffered IOs

**W2**

## **Socket API - TCP**

Client and server programming, sockets and streams, multi-threaded servers

**W3**

## **Application-level Protocol**

How to specify your own communication protocol? Implement the client and the server.

**W4**

## **Socket API - UDP**

Client and server programming, broadcast/multicast, service discovery protocols

# This Week

---

- **Monday**

- **08:30 - 09:00** : Review of the IO Benchmark exercise
- **09:00 - 09:20** : Introduction to the Socket API (client and server)
- **09:20 - 10:00** : The Calculator Exercise (Phase 1)
- **10:30 - 12:00** : The Calculator Exercise (Phase 2)

- **Wednesday**

- **08:30 - 08:50** : Using Threads for network programming in Java
- **08:45 - 09:00** : The Multi-Threaded Calculator Exercise

# Benchmarking buffered IOs

## **Review**

# What Do We Need?

---

- We need a way to **record measures** we take during the experiment:
  - Creating a **Measure class** to **represent a single measure**, with one field for every experiment parameter and one field for the measured value is certainly useful.
  - Creating a **MeasuresCollector class** to **record all measures** when we run the experiments is also useful.
  - The Measure and MeasuresCollector should also allow us to **export results in a CSV stream**.

# What Do We Need?

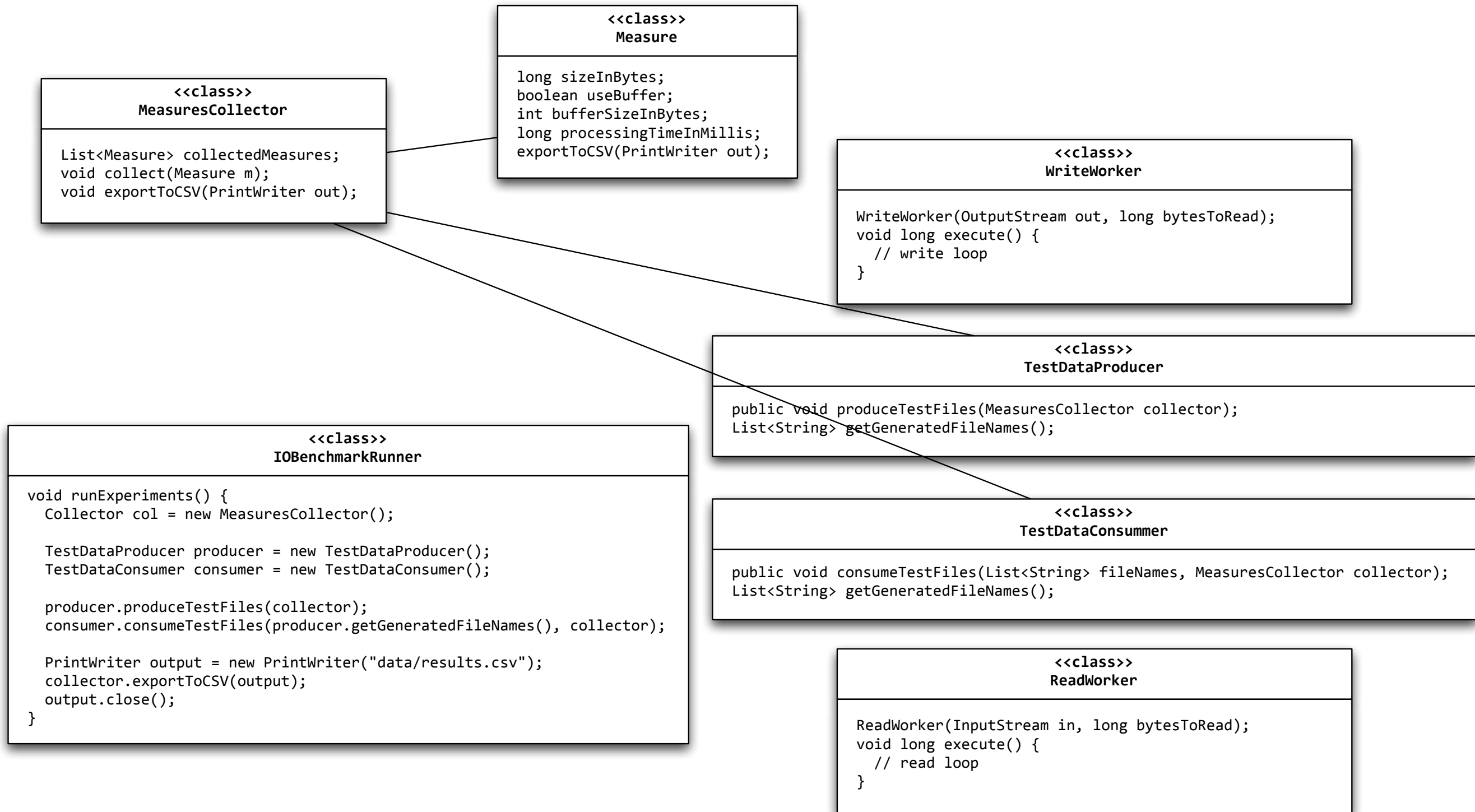
---

- We want to **produce test data**:
  - We need the generated data files for measuring read speed. We can also **measure write speed when generating the test files**.
  - We can define a **TestDataProducer** to run the experiment several times, one time for each **combination of the parameters**.
- We want to **consume test data**:
  - We can **measure the read speed** by reading every generated test file.
  - When we generate a test data file, we can **encode the parameter values in the file name** (if the number of parameters is not too large).

# What Do We Need?

---

- We can **implement the read and write loops** in dedicated classes:
  - **ReadWorker** has a **constructor**, which allows TestDataConsumer to **pass it an InputStream instance** (with a particular buffering behavior).
  - **ReadWorker** has an **execute** method, that will read all bytes on the InputStream and return the **execution time**.
  - **WriteWorker** has a **constructor**, which allows TestDataProducer to pass it an OutputStream instance (with a particular buffering behavior) and a **number of bytes to write**.
  - **WriteWorker** has an **execute** method, that will write the specified number of bytes on the OutputStream and return the **execution time**.





How would you **describe** and **compare**  
**TCP** and **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.

heig-vd

Haute Ecole d'Ingénierie et de Gestion  
du Canton de Vaud



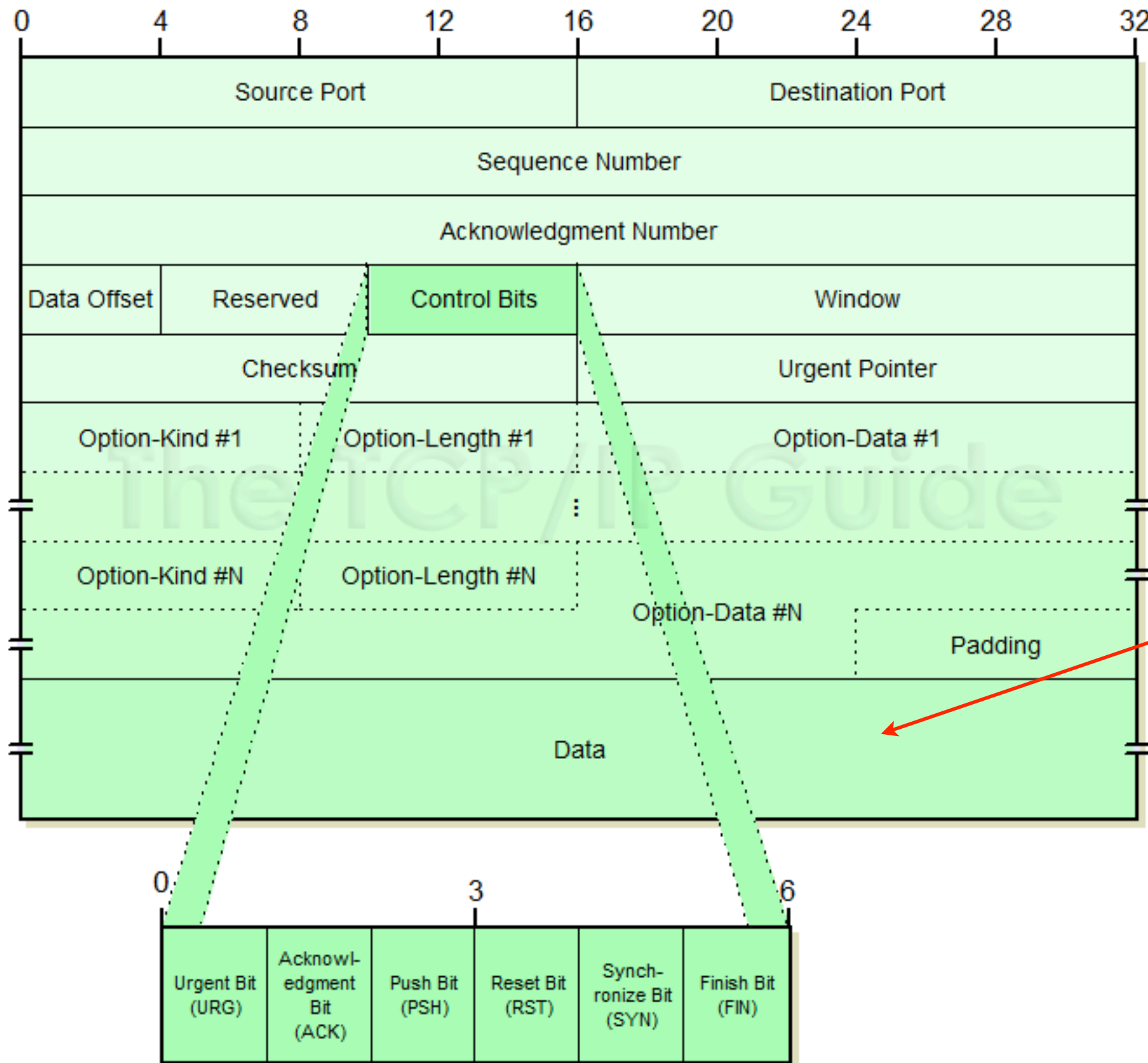
TCP



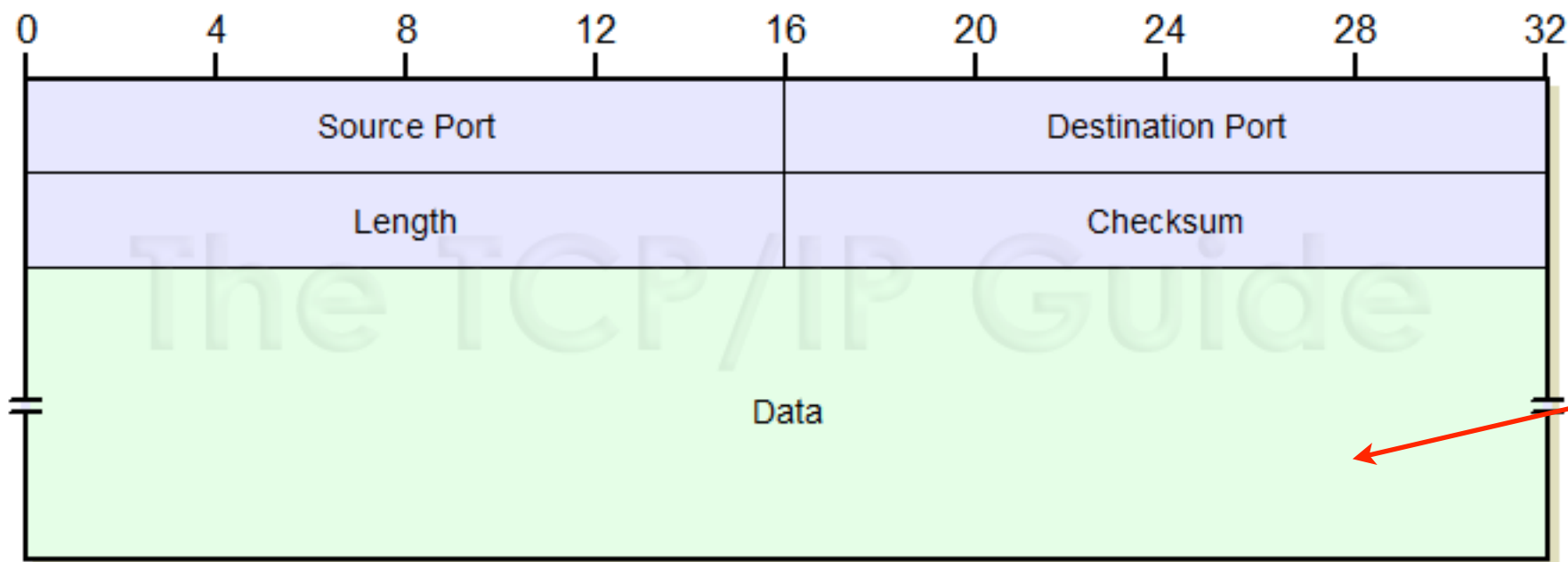
UDP



- 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**.

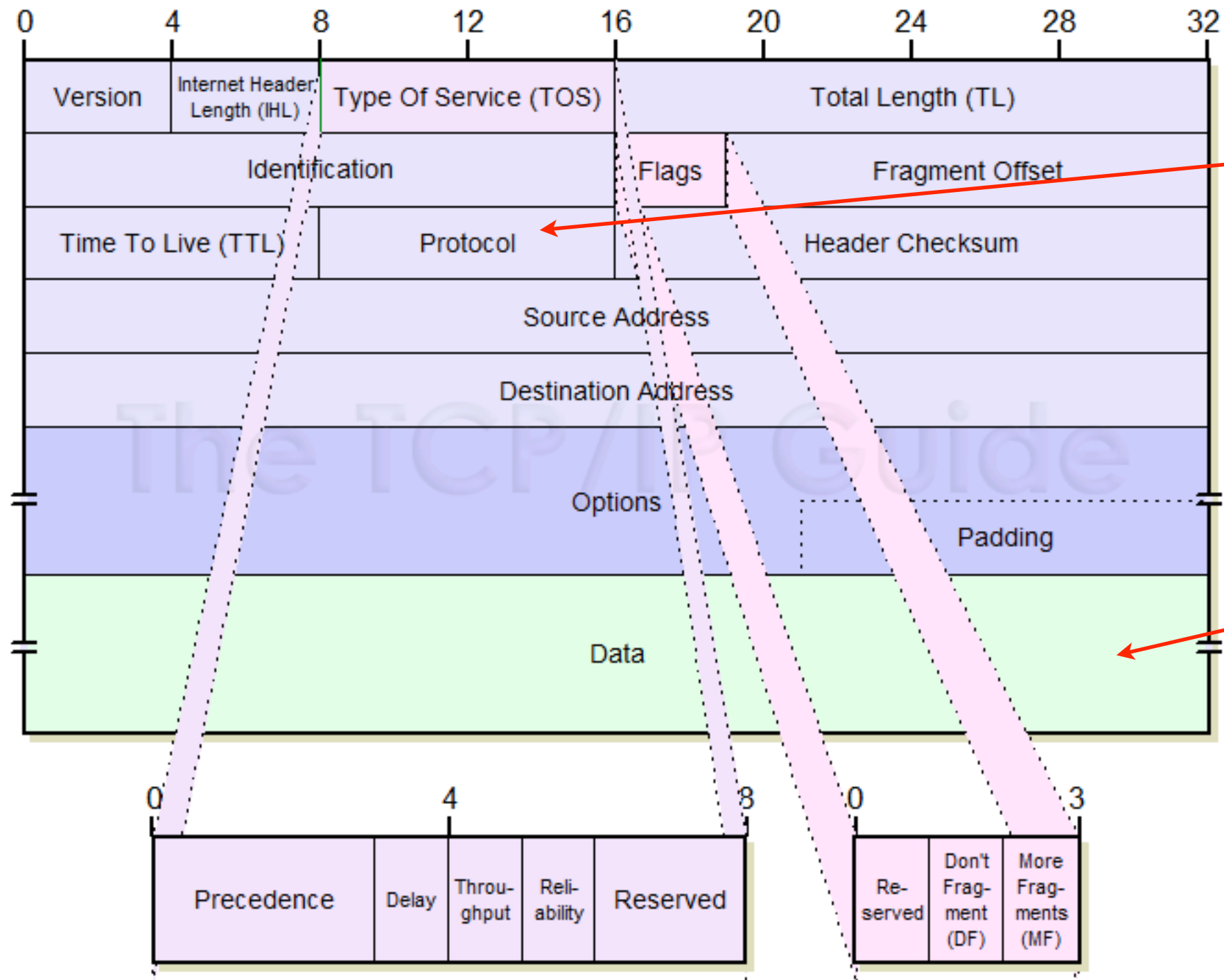


The bytes that you write in your java program will be here...



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

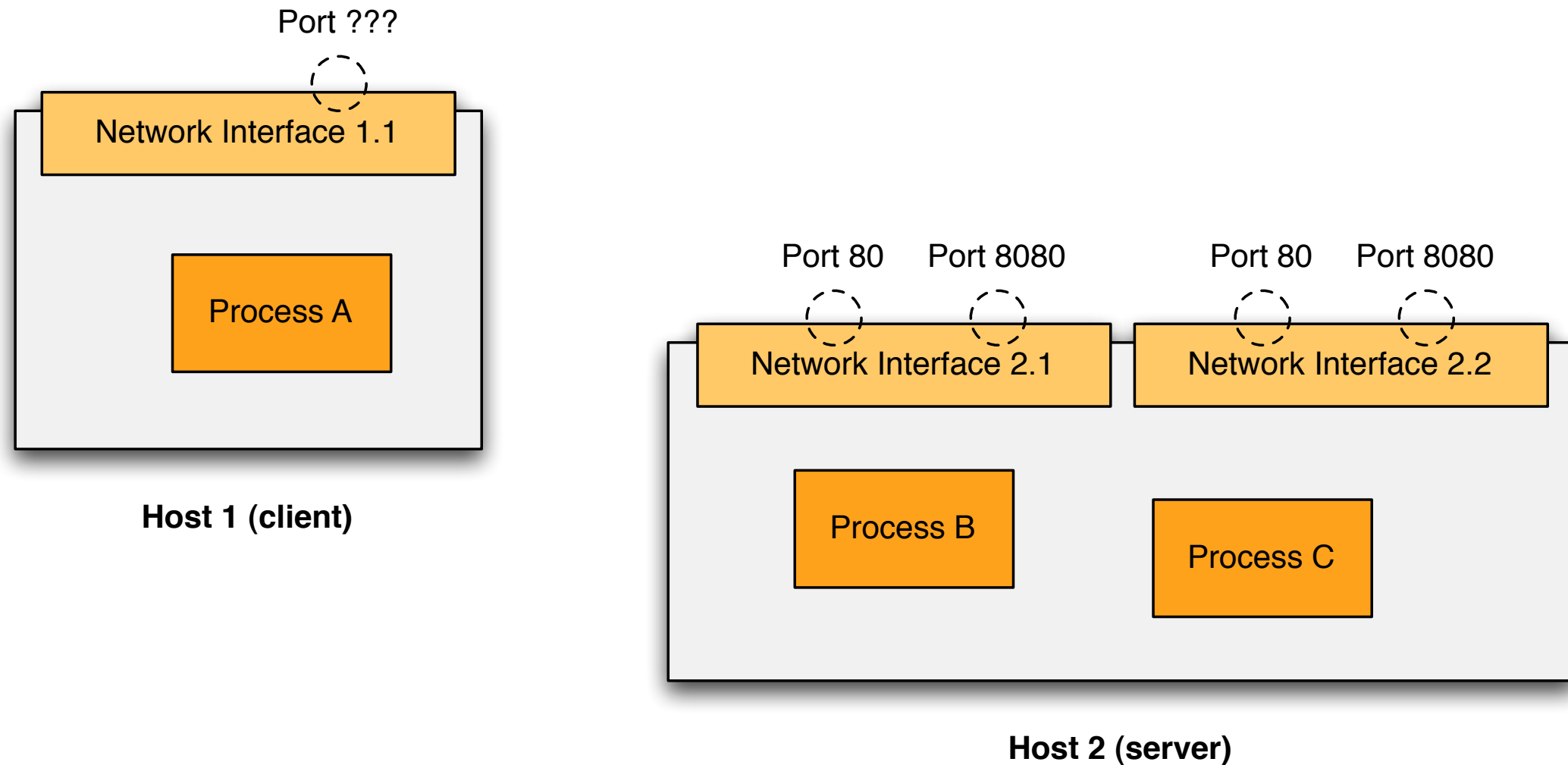
[http://www.tcpguide.com/free/t\\_UDPMessageFormat.htm](http://www.tcpguide.com/free/t_UDPMessageFormat.htm)



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".

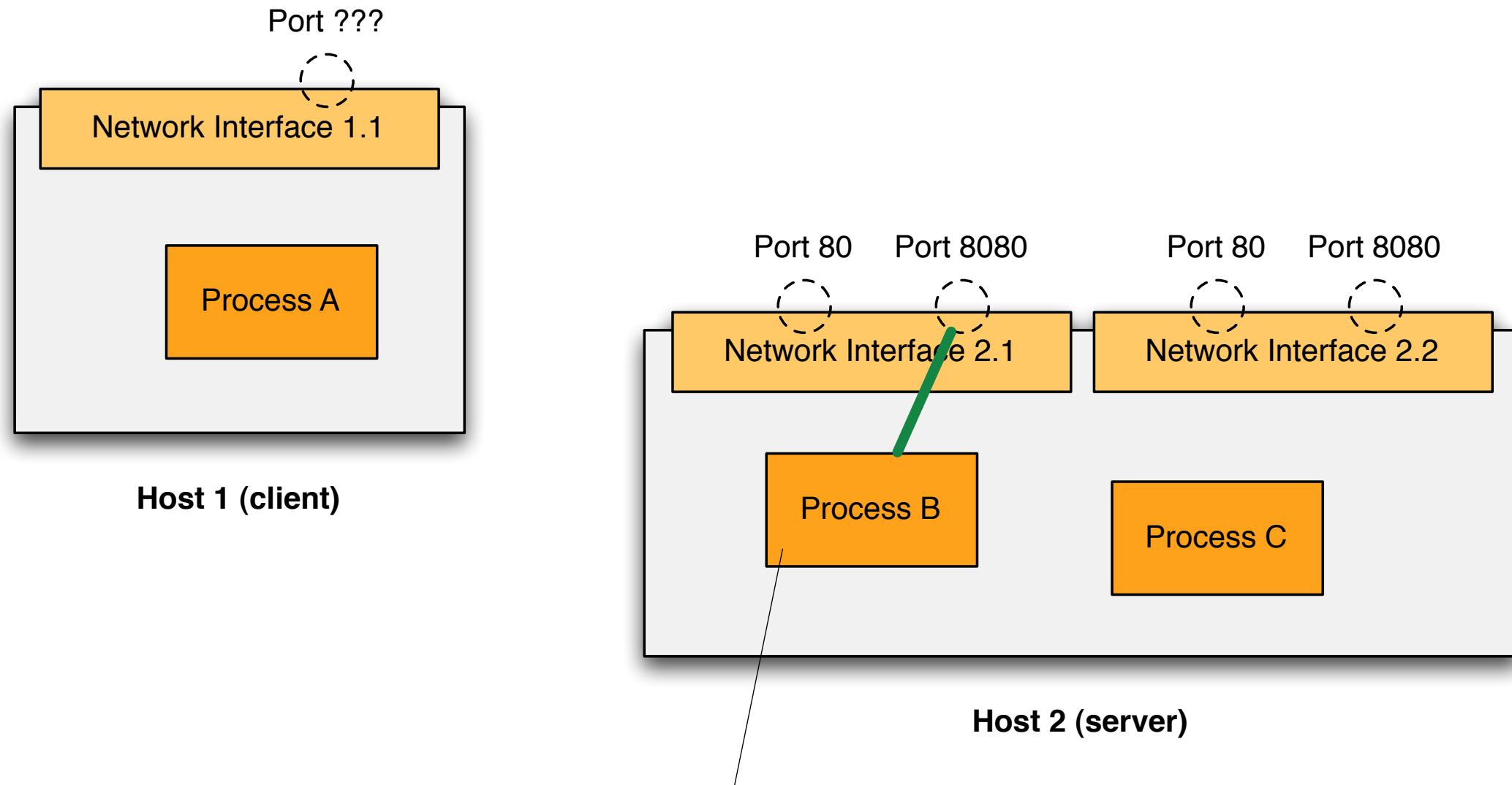
This can contain a TCP segment, a UDP datagram, or something else.

# Using the Socket API in Java





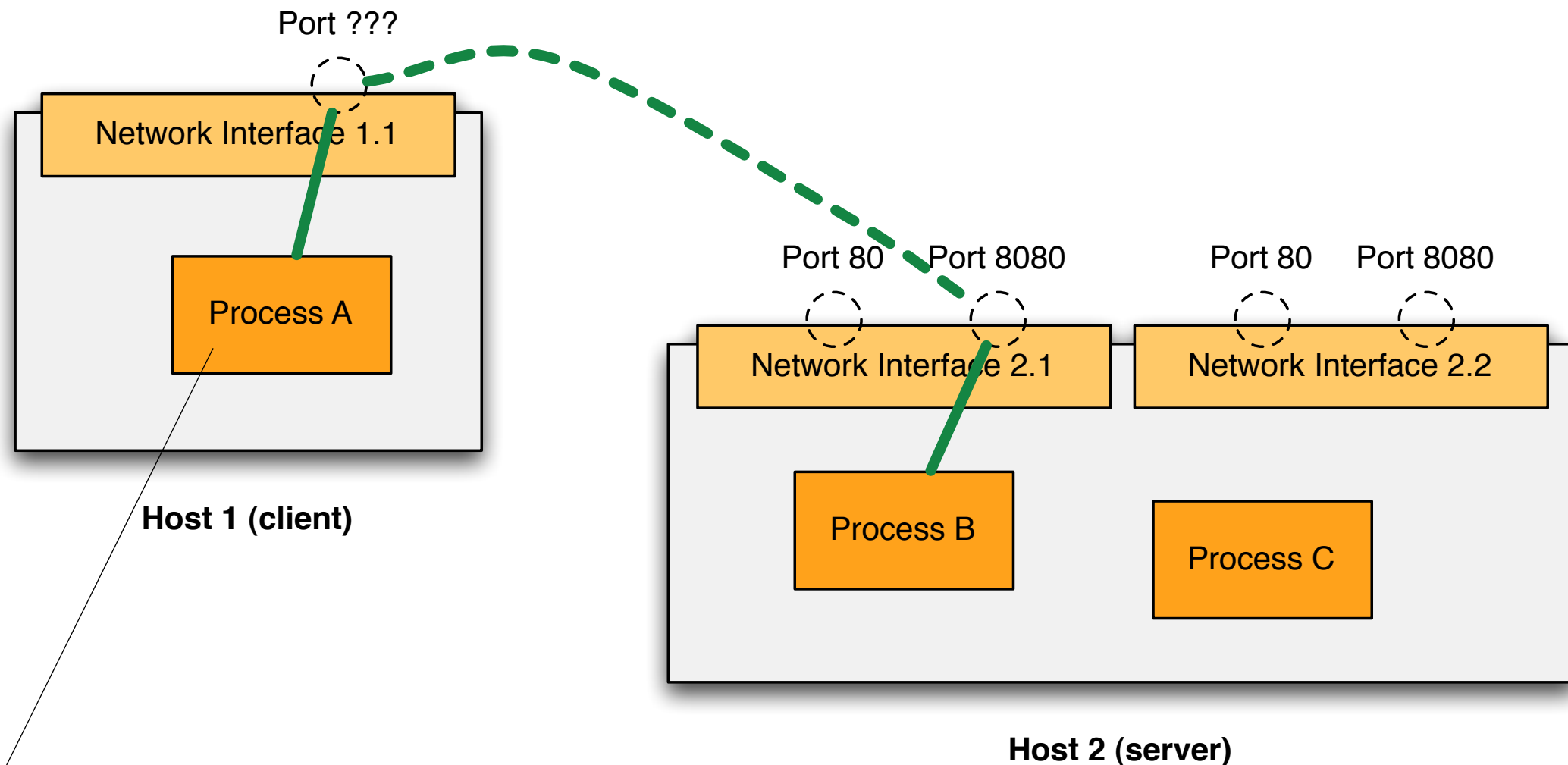
# Using the Socket API in Java



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

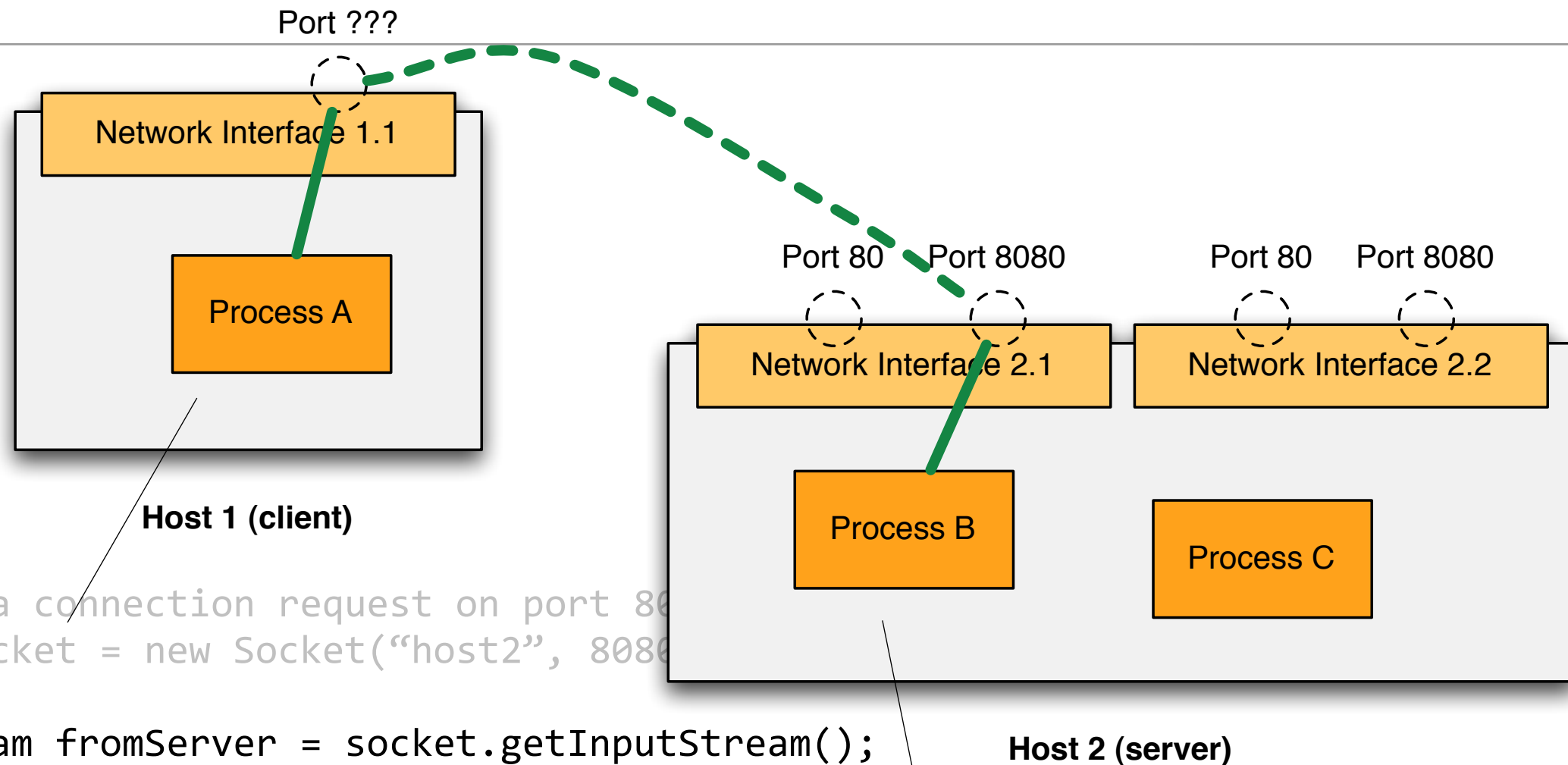
// Wait until a client makes a connection request...
Socket commSocket = serverSocket.accept();
```

# Using the Socket API in Java



```
// 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 8080
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();
```

# The Calculator Exercise

---

- **Phase 1 : Implement a simple TCP server and a simple TCP client**
  - The server must **accept connections** on a specified port.
  - The server implements a single **ABOUT** command that returns basic information about the program (e.g. “This program was written at HEIG-VD, etc.”).
  - To use this command, the client can make a connection request to the server. As soon as the connection is established, the client sends the following bytes: [**ABOUT\n**] through the socket.
  - **When the server receives this sequence of bytes**, it sends the command output to the client and closes the connection.
  - If the client sends **any other sequence of bytes**, the server **sends an error** message.

# Read Sections of the Java Tutorial

---

- **Lesson: All About Sockets**

[\*http://docs.oracle.com/javase/tutorial/networking/sockets/index.html\*](http://docs.oracle.com/javase/tutorial/networking/sockets/index.html)

- **What Is a Socket?**

[\*http://docs.oracle.com/javase/tutorial/networking/sockets/definition.html\*](http://docs.oracle.com/javase/tutorial/networking/sockets/definition.html)

- **Reading from and Writing to a Socket**

[\*http://docs.oracle.com/javase/tutorial/networking/sockets/readingWriting.html\*](http://docs.oracle.com/javase/tutorial/networking/sockets/readingWriting.html)

- **Writing the Server Side of a Socket**

[\*http://docs.oracle.com/javase/tutorial/networking/sockets/clientServer.html\*](http://docs.oracle.com/javase/tutorial/networking/sockets/clientServer.html)

# The Calculator Exercise

---

- **Phase 2 : Add a **COMPUTE** command**
  - In addition to the ABOUT command, the client can send a COMPUTE command with the following syntax: [**COMPUTE** **OPERAND1** **OPERATOR** **OPERAND2**\n], where
    - **OPERAND1** and **OPERAND2** are numeric values (double)
    - **OPERATOR** is either \* or +
  - **When the server receives a command from the client**, it first needs to determine whether it is a ABOUT or a COMPUTE command. If it is a COMPUTE command, then it needs to parse the command, compute the result (by multiplying, respectively adding the operands).
  - The server has to **do proper error handling** (invalid command, syntax error, etc.).