NETWORK PROGRAMMING WITH PYTHON

Emmanuel Viennet
emmanuel.viennet@univ-paris13.fr

UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI

Université Sorbonne Paris Nord

Contents : day one

1. Brief recalls on IP, TCP, UDP, sockets
2. The Python's socket module
3. Building a UDP service and client (exercice)
4. TCP clients
5. TCP servers: process and threads
   how to serve several clients ?

Contents : day two

1. Standard Internet services: HTTP, SMTP, POP, IMAP
2. Implementing a simple HTTP server in Python
3. Developping a REST API in Python
4. Overview of Python’s Web Frameworks
5. Cybersecurity considerations

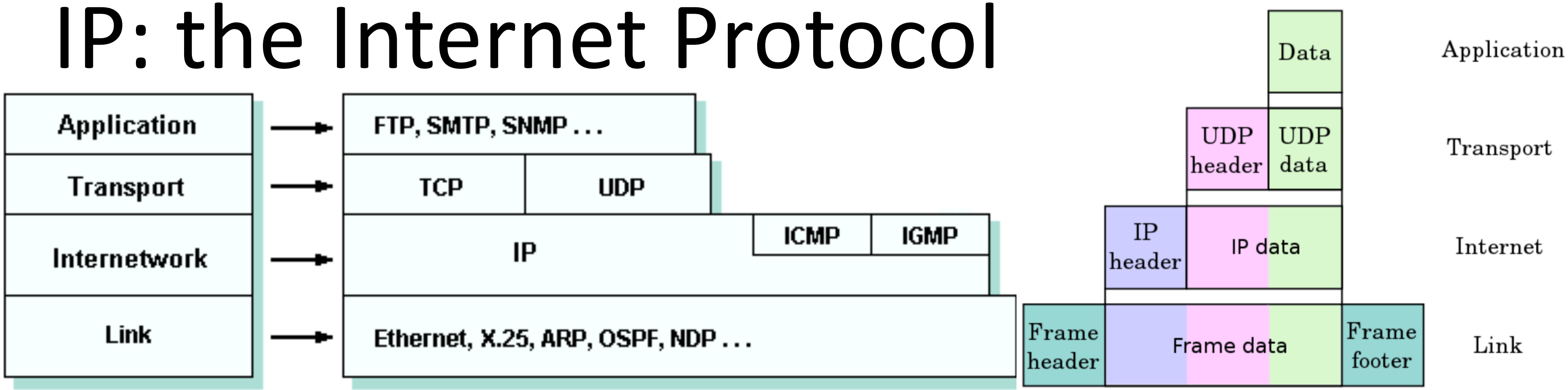
Organization of the course

- A mix of short courses and practical exercices
- You must participate: ask questions, practice on your computer
- You will be asked to present your work to the class

Useful tools

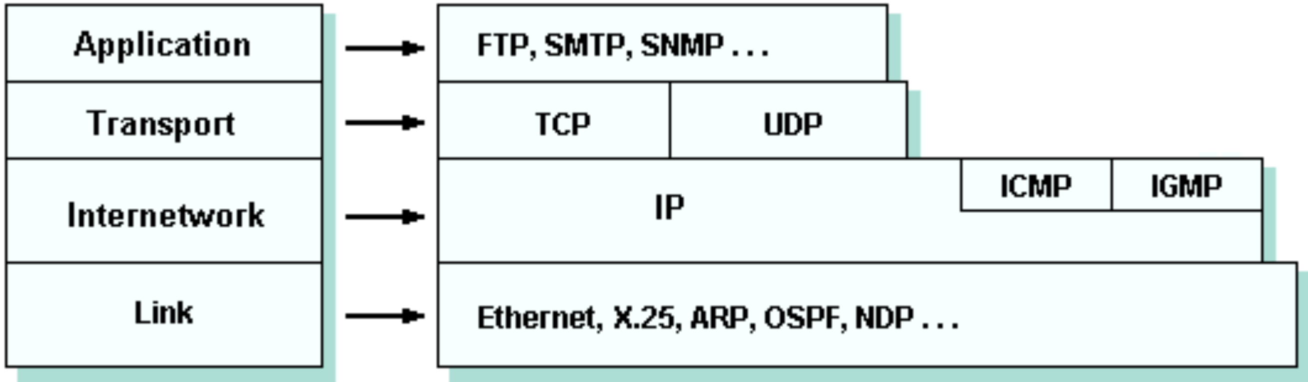
- `netstat`, `ping`, `nmap`
- netcat (`nc`) https://linuxhint.com/nc-command-examples
- `ntop`, ntopng web-based traffic monitoring application https://www.ntop.org
- `wireshark` network protocol analyzer. https://www.wireshark.org
- `wget`, `curl`
- httpie (`http`) command-line API client https://httpie.io
- `jq`, a lightweight and flexible command-line JSON processor https://stedolan.github.io/jq

IP: the Internet Protocol



- IP an interoperable protocol with low overhead
- Based on packet switching
- Logical addressing : “IP address”
- IPv4, IPv6

IP: Transport Protocols



- UDP : low overhead, no connection, fast, unreliable
- TCP : connected, error handling, congestion handling, reliable

IP: the Internet Protocol

To be usable, IP needs some auxiliary services :

- lookup IP addresses from names, such as `usth.edu.vn`
  => `DNS` *Domain Name Service*

- lookup MAC (Ethernet) address for a given IP
  "*who has* `192.168.10.2` ?" => `aa:0f:36:f0:12:34`
  => ARP *Address Resolution Protocol*

Domain Name Service (DNS)

UNIVERSITY OF
SCIENCE AND TECHNOLOGY
OF HANOI

IP: the Domain Name Service (DNS)

```
www.univ-paris13.fr   <->  81.194.43.200
```

- It is impossible to remember all the IP addresses of thousands of servers hosting services on the Internet. There is an easier way to locate servers, which is to associate a **name** and an **IP address**.
- The Domain Name System (DNS) allows hosts to use this name to request the IP address of a given server.
- DNS names are registered and organized on the Internet within specific high-level groups or domains.
- Example of top-level domain names (TLD): .vn, .fr, .com, .net

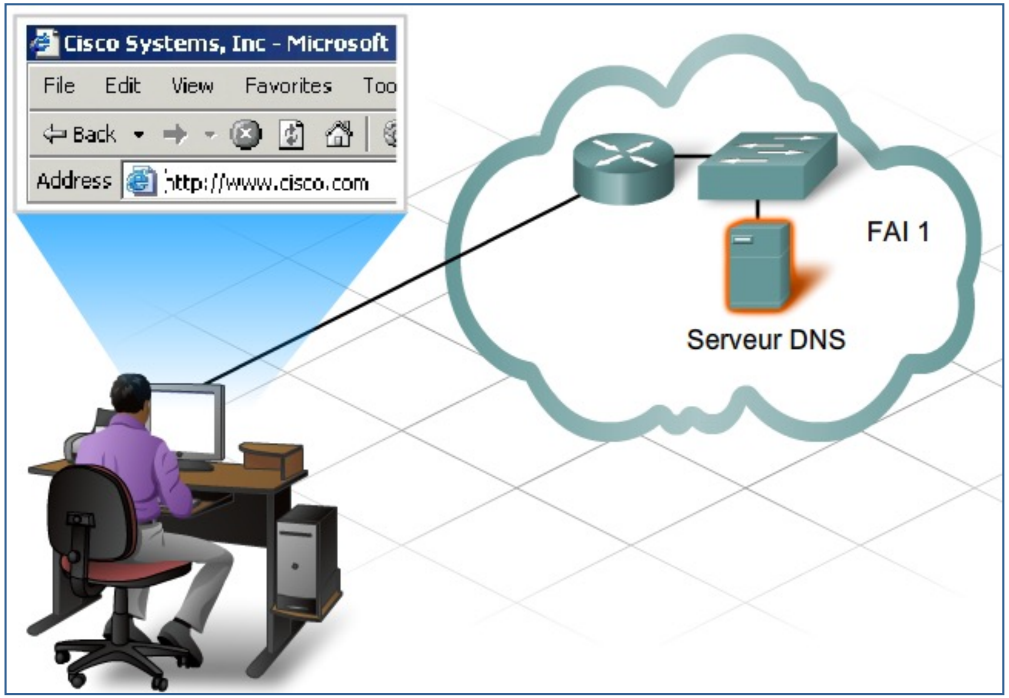
IP: the Domain Name Service (DNS)

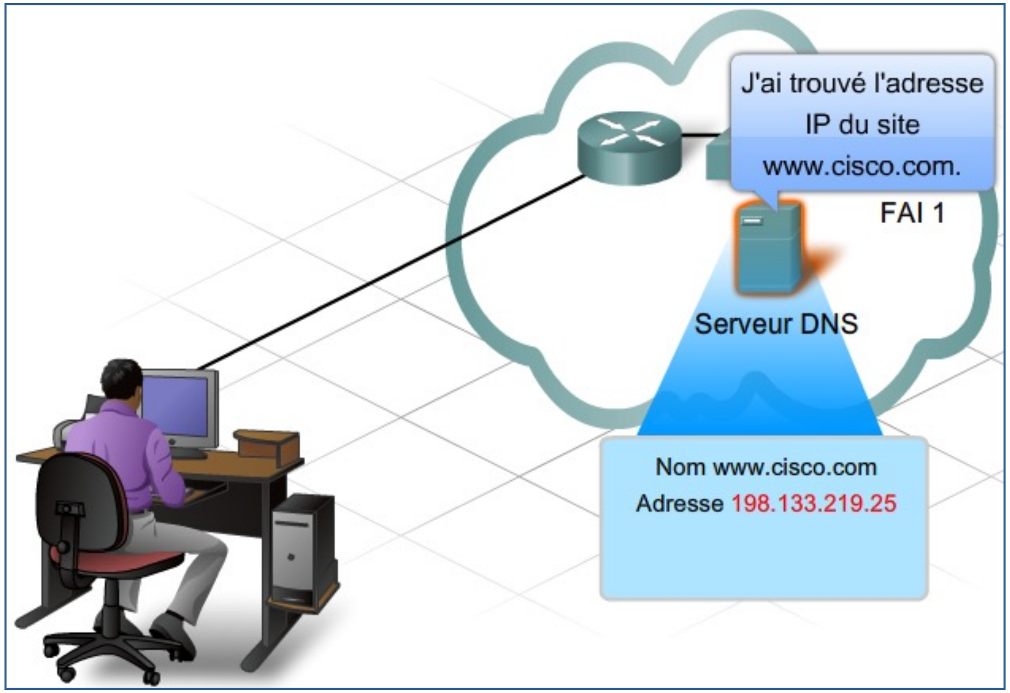
A domain name server contains a table that associates host names with the corresponding IP addresses.

- When a client wants to see a page on a web server:
  - It sends a request to the DNS server to find the web server name match <=> IP address of the web server
  - The DNS server consults its name table to determine the IP address associated with this Web server
  - If it knows its IP address, it communicates it to the client host
  - If it does not know it, it transfers this request to another higher level DNS server (that of its Internet operator for example)
  - If no name server can determine the IP address, the timeout for the request is exceeded, and the client can not communicate with the Web server

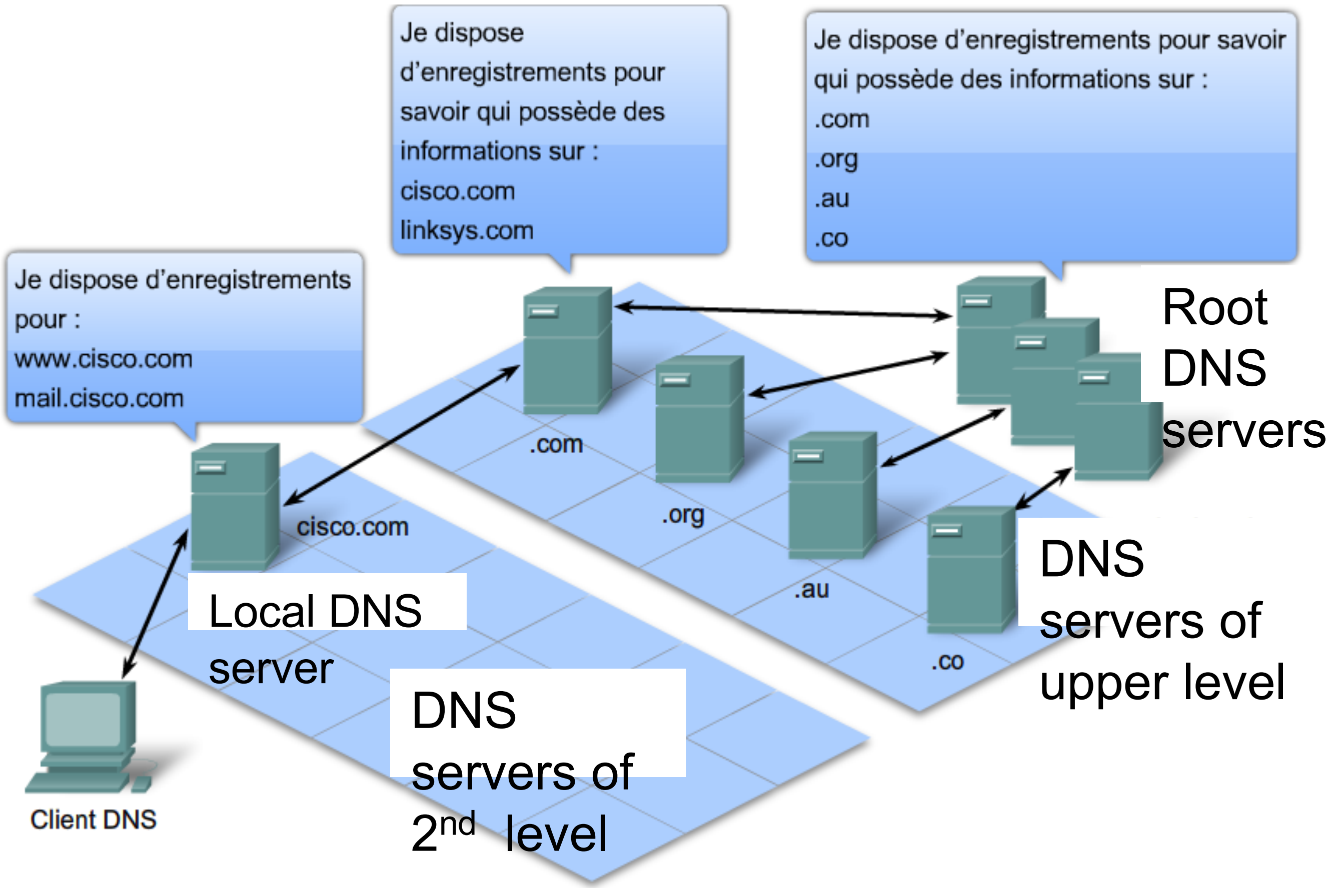
IP: the Domain Name Service (DNS)

Example : request to website « www.cisco.com » :





IP: the Domain Name Service (DNS)



Une hiérarchie de serveurs DNS contient les enregistrements de ressources qui associent les noms aux adresses.

IP: the Domain Name Service (DNS)

Local or **cache** resolution

- On the client side, each machine has at least the address of a DNS server (primary server) and possibly the address of a second (secondary server).
- When an application (browser, FTP client, mail client ...) needs to resolve a symbolic name to a network address, it sends a request to the **local resolver** (process on the client machine)
  NB: if the client DNS cache contains the IP address of the site being searched, there will be no "DNS query".
- The local resolver passes the DNS request from the client to the name server of the local zone (primary server)

IP: the Domain Name Service (DNS)

Client configuration

- DNS servers are usually given by the DHCP configuration server;
- On Unix/Linux, they can be manually specified in `/etc/resolv.conf`

```
nameserver 192.168.64.1
```

IP: the Domain Name Service (DNS)

Some useful commands

- `host`
- `nslookup`
- `dig`
- `whois`

Hint: to get a quick help, you can use https://tldr.ostera.io

dig

4

dig makes DNS queries!

$ dig google.com
answers have 5 parts:
- query: google.com
- TTL: 22
- class: IN (for "internet") ignore this.
- record type: A
- record value: 172.217.13.110

dig TYPE domain.com

this lets you choose which DNS record to query for!

types to try: NS, MX, TXT, CNAME, A (default)

dig @8.8.8.8 domain

Google DNS server

dig @server lets you pick which DNS server to query! Useful when your system DNS is misbehaving

dig +trace domain

traces how the domain gets resolved, starting at the root nameservers

if you just updated DNS, dig +trace should show the new record

dig -x 172.217.13.174

makes a reverse DNS query - find which domain resolves to an IP! Same as

dig ptr 174.13.217.172.in-addr.arpa

dig +short domain

Usually dig prints lots of output! With +short it just prints the DNS record

Quizz

1. What is the IP address of your computer ?
   Is it IPv4 or IPv6 ?

2. What is the IP address of your smartphone ?

3. What is the IP address of `iutv.univ-paris13.fr` ?

4. What are the address of your DNS servers ?

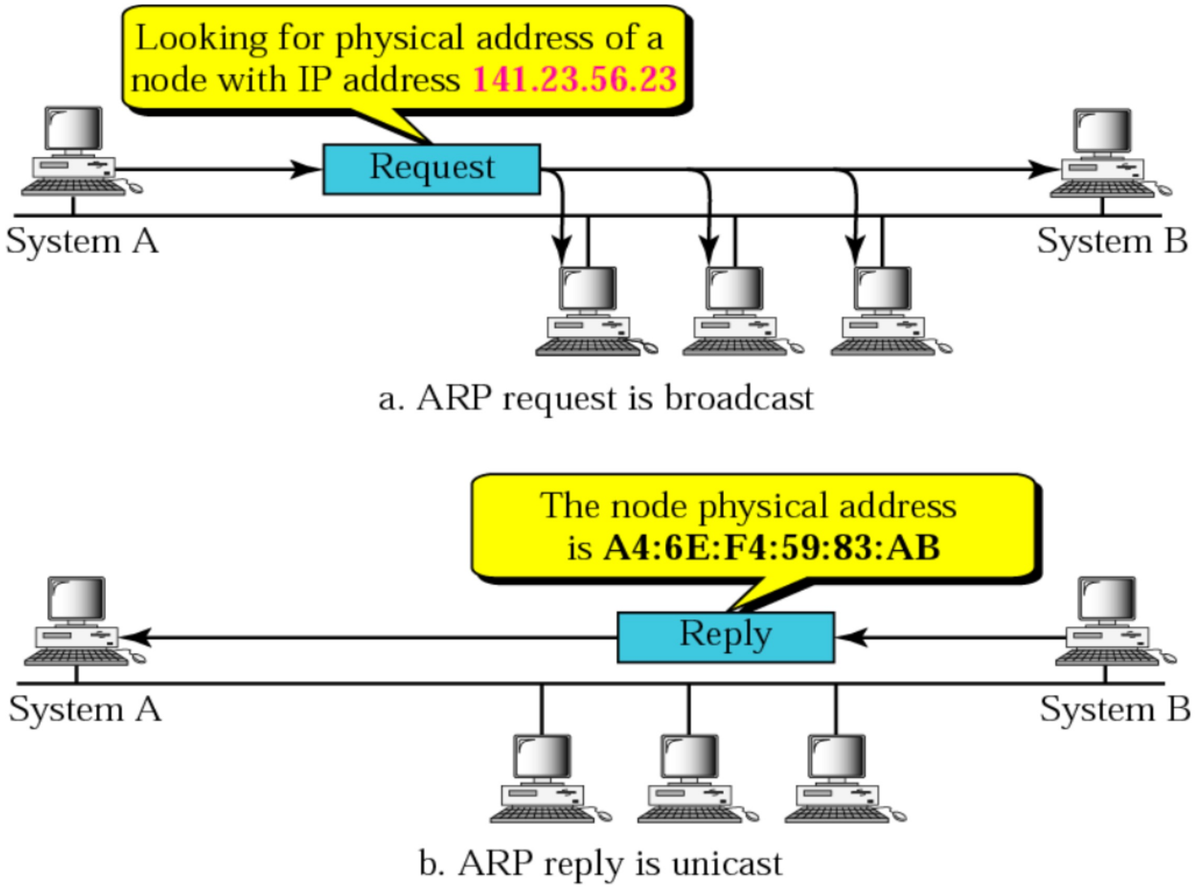
Address Resolution Protocol (ARP)

IP: the Address Resolution Protocol (ARP)

On a local network (LAN), Ethernet frames encapsulate IP packets.

Ethernet frames are using *physical addresses*.

ARP is a decentralized protocol used to match IP and physical addresses.



IP: the Address Resolution Protocol (ARP)

ARP does not need specific configuration: it's working « out of the box ».

On linux, the `arp` command can be used to display the ARP cache and other informations.
See for instance
https://www.computerhope.com/unix/arp.htm

Sockets

IP Sockets

Sockets are Operating System object representing a network ”connection”.

The offer an API (system calls) to open a connection, wait for incoming clients, send and receive packets.
Sockets allow to use various transport protocols: UDP, TCP, or even “raw” packets.

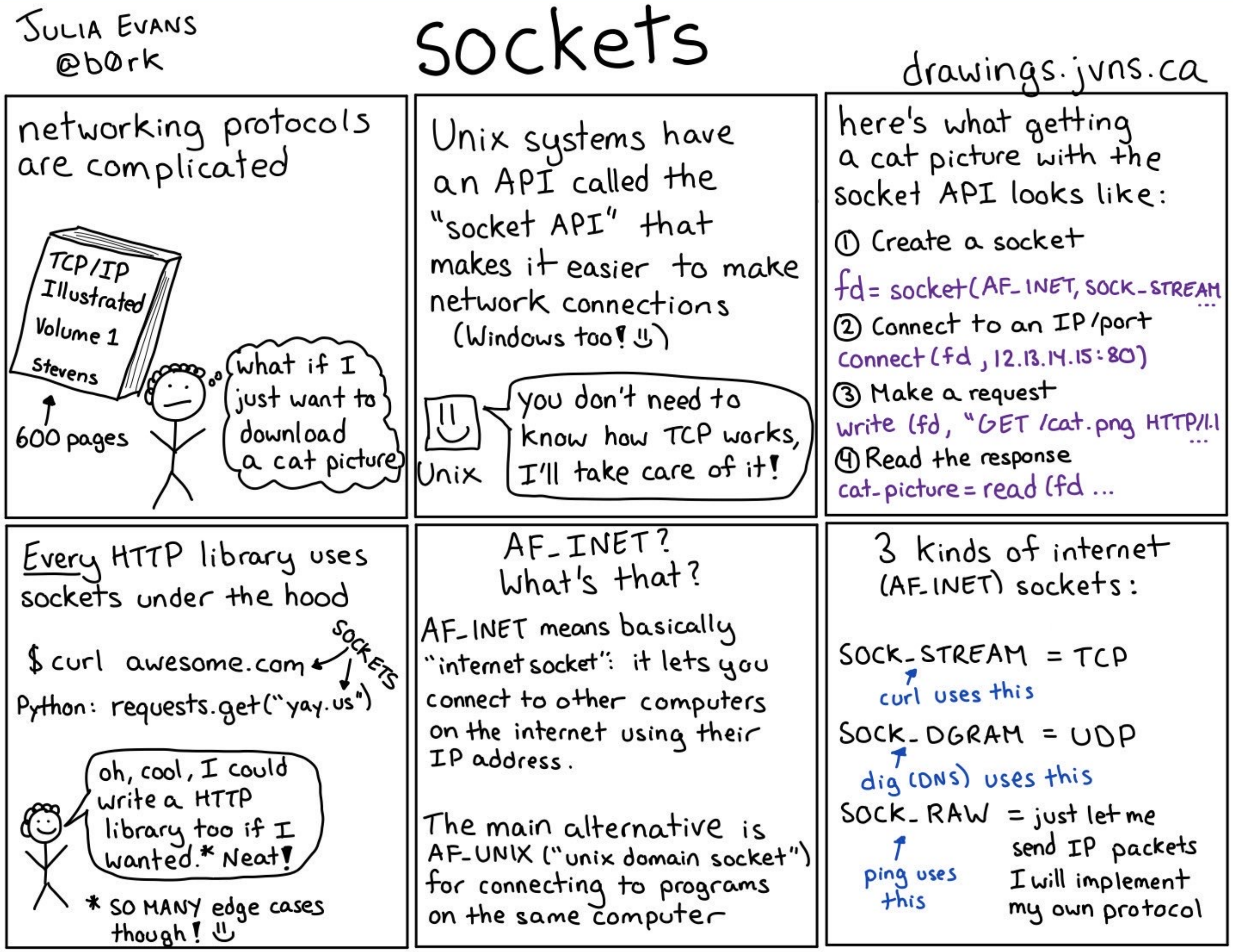
Note: sockets are mainly used with IP, but can also be used locally, on the same system, for interprocess communication (“unix sockets”).

IP Sockets

TCP and UDP sockets allows **inter-process** communication between **distant machines**.

To do this, one needs:

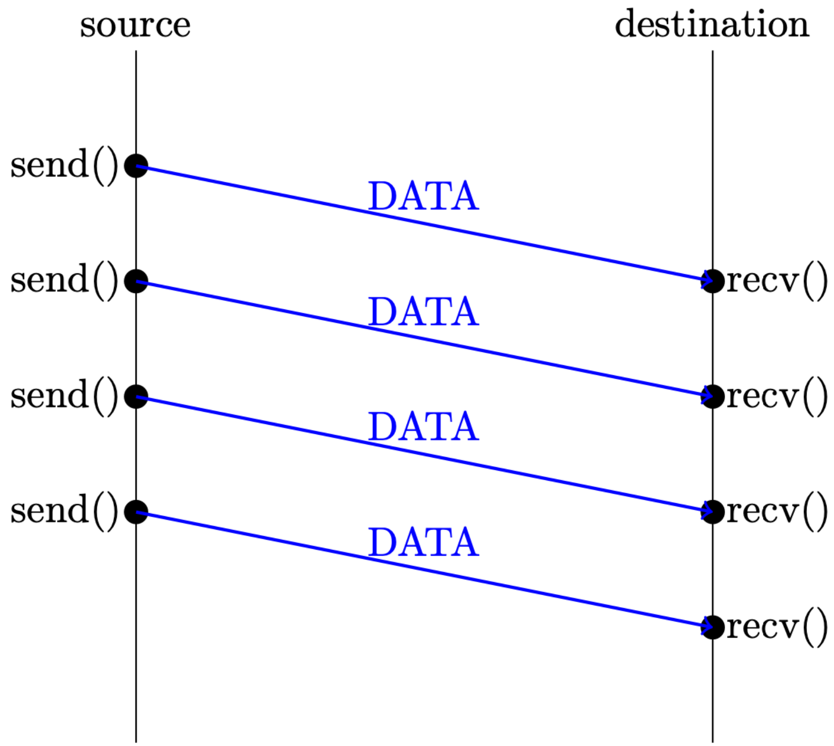
- a way to identify the remote machines: the **IP address**
- a way to identify a “service” on the remote machine: the **port** (an integer number)


JULIA EVANS
@b0rk
sockets
drawings.jvns.ca
networking protocols are complicated
TCP/IP Illustrated Volume 1 Stevens
600 pages
what if I just want to download a cat picture
Unix systems have an API called the "socket API" that makes it easier to make network connections (Windows too!)
Unix
you don't need to know how TCP works, I'll take care of it!
here's what getting a cat picture with the socket API looks like:
① Create a socket
fd = socket(AF_INET, SOCK_STREAM ...
② Connect to an IP/port
connect(fd, 12.13.14.15:80)
③ Make a request
write (fd, "GET /cat.png HTTP/1.1 ...
④ Read the response
cat_picture = read (fd ...
Every HTTP library uses sockets under the hood
$ curl awesome.com
SOCKETS
Python: requests.get("yay.us")
oh, cool, I could write a HTTP library too if I wanted.* Neat!
* SO MANY edge cases though!
AF_INET? What's that?
AF_INET means basically "internet socket": it lets you connect to other computers on the internet using their IP address.
The main alternative is AF_UNIX ("unix domain socket") for connecting to programs on the same computer
3 kinds of internet (AF_INET) sockets:
SOCK_STREAM = TCP
curl uses this
SOCK_DGRAM = UDP
dig (DNS) uses this
SOCK_RAW = just let me send IP packets I will implement my own protocol
ping uses this

Sockets / UDP protocol

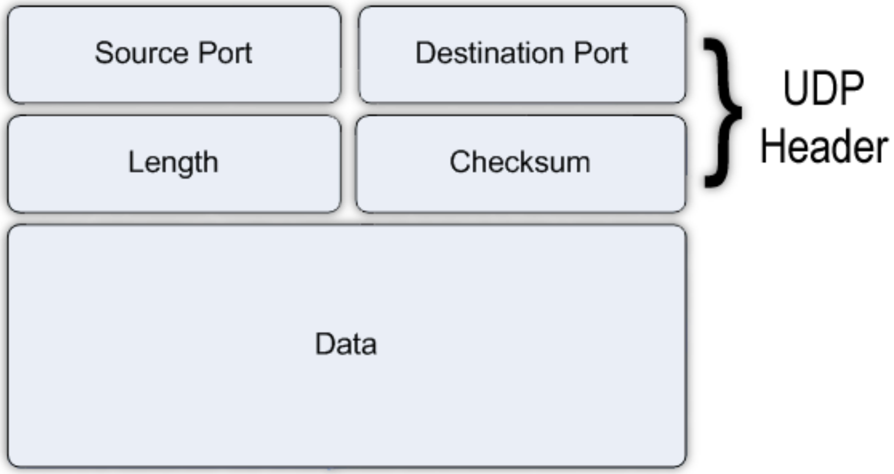
UDP does not use “connection”: the source can send packets (datagrams) to the destination

- `sendto()` sends a datagram
- `recv()` receives (read) an incoming datagram.



Sockets / UDP protocol

UDP is very similar to raw IP, but adds ports numbers, in order to be able to address different services (process).
The source port is used to be able to send a reply.



Sockets / UDP protocol

Programming UDP sockets in Python

- the **server** will
  1. create a **socket**, specifying the kind (IP: `AF_INET`) and the transport protocol (`SOCK_DGRAM` for UDP)
  2. ***bind*** the socket to an address: the addresses are specified by the pair `(IP, port)` pair
  3. wait to receive incoming data

Sockets / UDP protocol

Programming UDP sockets in Python

```
import socket

PORT = 8765       # the port you will use
BUFSIZE = 1024  # the max message size you’re expecting
```

Details on https://docs.python.org/3/library/socket.html

Sockets / UDP protocol

Programming UDP sockets in Python

- the **server** will
  1. create a **socket:**
     `s = socket.socket( socket.AF_INET, socket.SOCK_DGRAM )`
  2. ***bind*** the socket to an address:
     `s.bind( ( "", PORT ) )`
  3. wait to receive incoming data:
     `data, addr = s.recvfrom( BUFSIZE )`

Details on https://docs.python.org/3/library/socket.html

Sockets / UDP protocol

Programming UDP sockets in Python

- the **client** will
  1. create a **socket:**
     ```
     s = socket.socket( socket.AF_INET, socket.SOCK_DGRAM )
     ```
  2. ***send a datagram*** to an address:
     ```
     s.sendto( message, ( host, PORT ) )
     ```
  3. Repeat, or close the socket.

Python strings and bytes

In Python 3, we have two type to handle "strings":

- string (`str`) are UNICODE strings
- they must be **encoded** as **bytes** before being sent on the network, or stored in a disk file.

For more details, see e.g. https://betterprogramming.pub/strings-unicode-and-bytes-in-python-3-everything-you-always-wanted-to-know-27dc02ff2686

Python strings and bytes

- `s.encode("utf8")` encode the string `s` as bytes using UTF-8
- `data.decode("utf8")` decode data bytes and gets a Python string.

```
>>> len("các bạn")
7
>>> "các bạn".encode("utf8")
b'c\xc3\xa1c b\xe1\xba\xa1n'
>>> len("các bạn".encode("utf8"))
10
>>> b'c\xc3\xa1c b\xe1\xba\xa1n'.decode("utf8")
'các bạn'
```

Exercice: UDP client/server

1. Write an UDP server, printing all received messages and never ending. Exemple:

```
$./udp-server.py 1234
Started server on UDP/1234
[2021-11-22 16:36:33,167] client 192.168.0.13 sent "hello"
```

2. Write an UDP client, sending a message:
   Usage:

```
$./udp-client.py 192.168.0.13 1234 "hello”
```

   should send the message"hello” to the specified server on UDP/1234

Exercice: UDP client/server (cont.)

3. Using your `udpclient`, try to communicate with the `udpserver` of some other students
4. Use `wireshark` to show the network traffic from/to your computer
   - what is the ARP traffic when you try to send a message to a new machine?
   - locate some UDP message. What is the Ethernet frame size ? Compare it with the size of the text message sent.
   - can `wireshark` read the text of the messages sent by `udpclient` ?

Sockets / TCP protocol

Transport Control Protocol (TCP) is a *reliable* protocol.

To ensure that the packets are not lost in the way, and will be read in the intended order, TCP needs to:

- manage (open, close) a connection
- number the packets sent
- send ACKnowledgements, the receiver telling "*OK, I received that packets*"

Sockets / TCP protocol

Transport Control Protocol (TCP) is a *reliable* protocol.

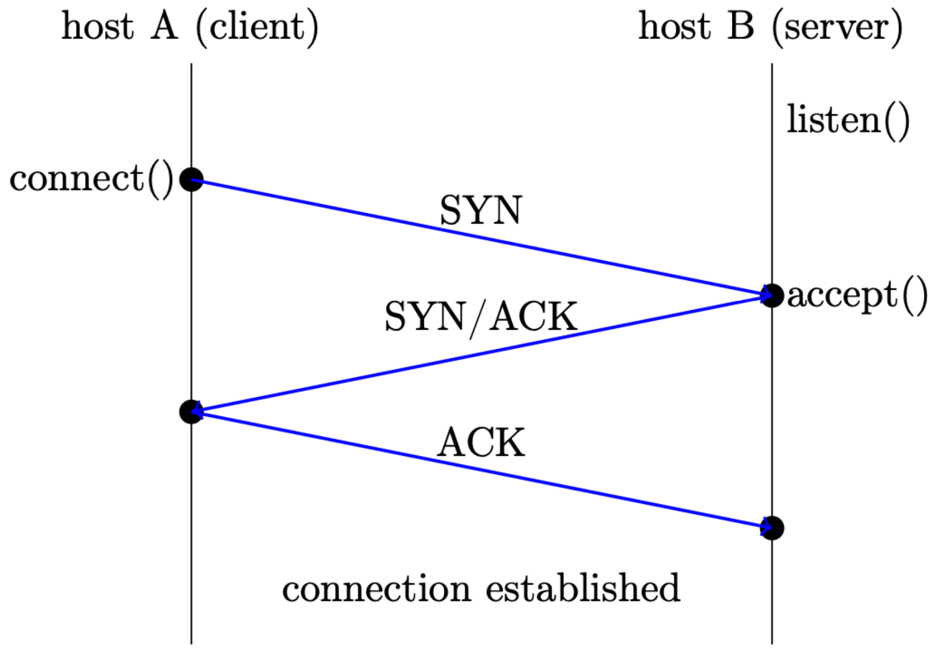
To ensure that the packets are not lost in the way, and will be read in the intended order, TCP needs to:

- manage (open, close) a connection
- number the packets sent
- send ACKnowledgements, the receiver telling "*OK, I received that packets*"

Sockets / TCP protocol

TCP connection opening

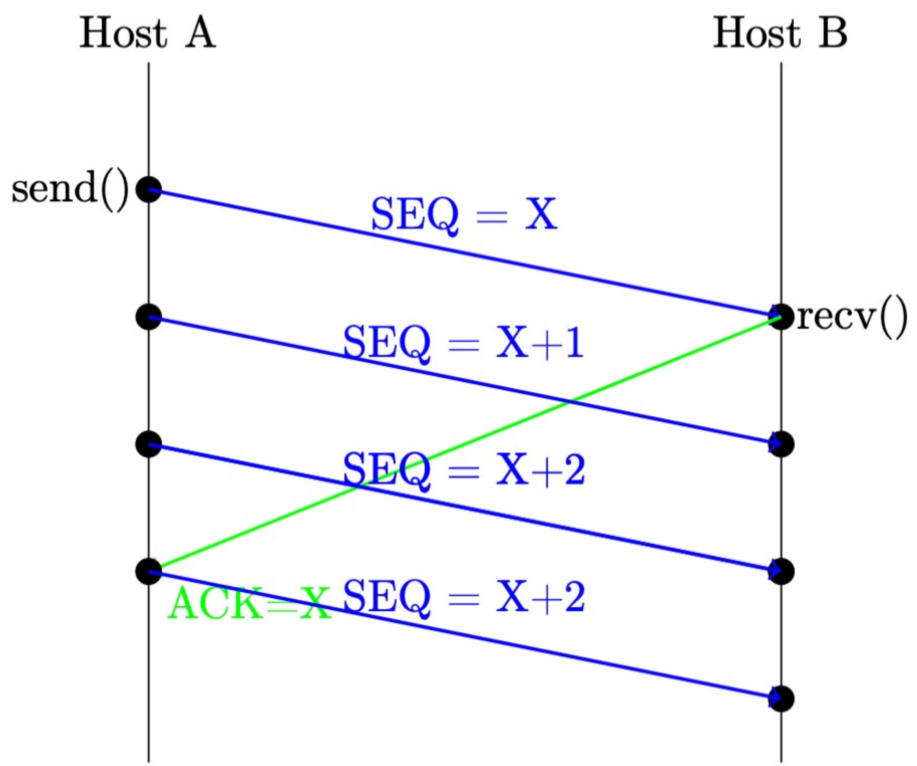
“triple handshake”



Sockets / TCP protocol

TCP packets ACK

(using a sliding window)

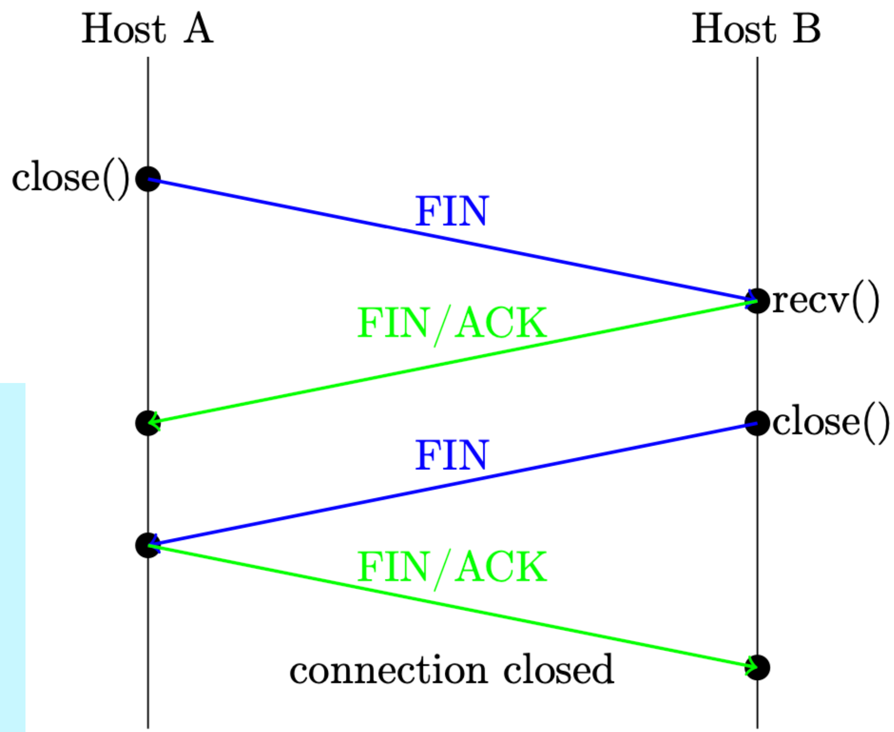


Sockets / TCP protocol

Closing a TCP connection

bidirectional handshake

Note: *when using TCP sockets, the programmer don't have to take care of TCP protocol details: the OS will automaticaly send the required packets.*

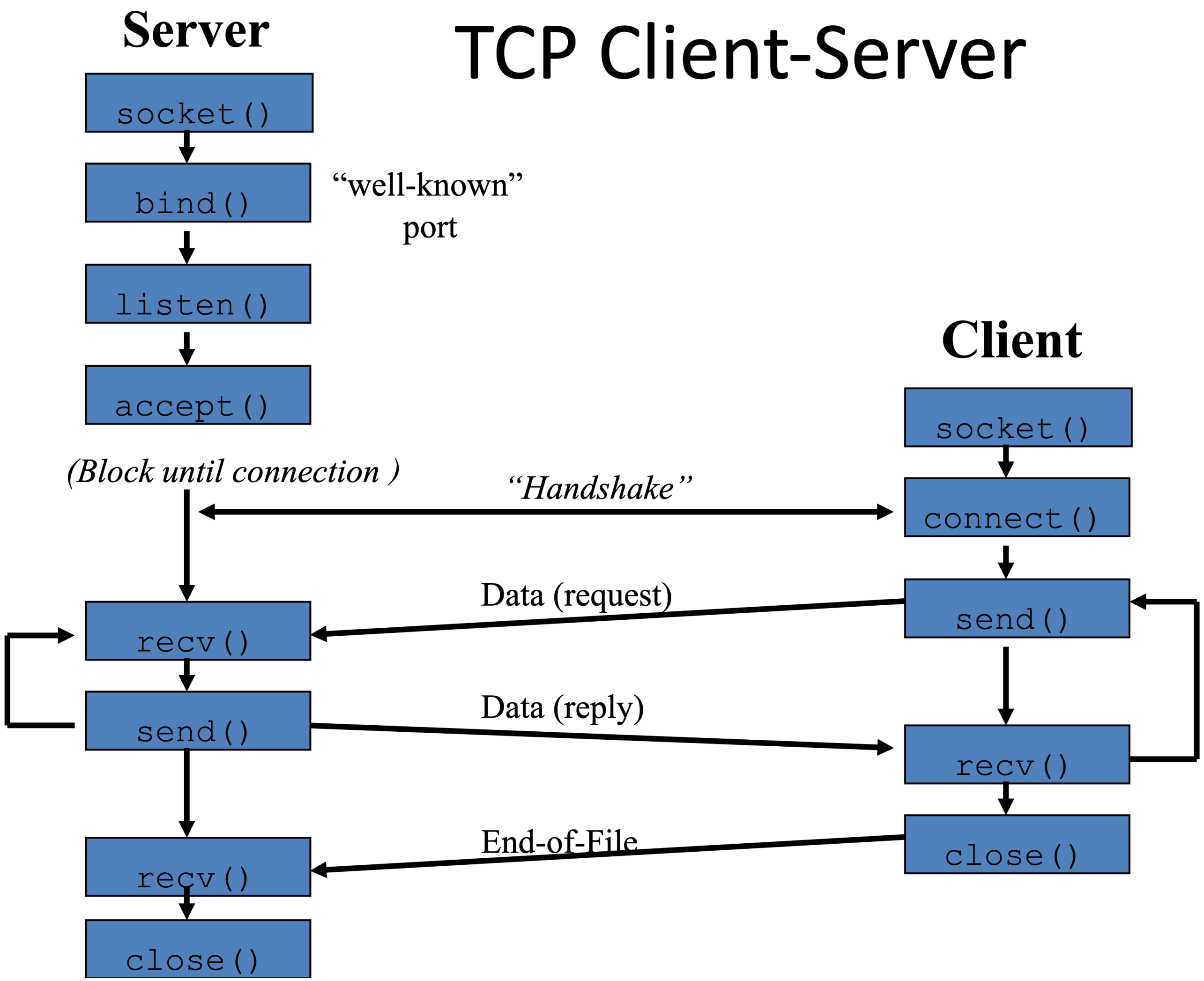


Sockets / TCP protocol

Programming TCP sockets in Python

- the **server** will
  1. create a **socket** (default to TCP)
  2. ***bind*** the socket to an address
  3. **listen** to the connection (prepare the OS to accept connections)
  4. **accept** new incoming connections
  5. wait to receive incoming data: **recv**

TCP Client-Server



Sockets / TCP protocol

Programming a TCP **server** in Python https://docs.python.org/3/library/socket.html

```
s = socket.socket()
s.bind( ('', PORT) )
s.listen( 0 )

# Waiting for a connection
cnx, addr = s.accept()

# reading data from this connection
msg = cnx.recv( BUFSIZE )
```

Sockets / TCP protocol

Programming a TCP **client** in Python https://docs.python.org/3/library/socket.html

```
s = socket.socket()
s.connect( ('', PORT) )

# Sending a string
s.send( message )

# Receiving a string:
s.recv( BUFSIZE)
```

```
# Closing the connection
s.close( message )
```

Exercice (TCP 1)

1. What are the types (classes) of **cnx** and **addr** ?

2. Open 2 terminal windows on a linux system
    1. in the first one, launch a Python interpreter and start a TCP server on port 6161 (socket, bind, listen, accept)
       What are the "blocking" calls ?
    2. In the other terminal, use netstat and locate your server.
    3. The launch python and **connect()** to the server. What do you observe ?

Exercice (TCP 2)

3. How can we know when the connection is closed ?
4. Write a server, **tcp-server.py**, same usage as UDP version.
   The server should :
   - print the received message
   - send back a text message to the client

   When the connection with the client is closed, the server should wait (accept) for another client.

Exercice (TCP 3)

5. Use the command `nc` to connect to your tcp-server
6. Write **tcp-client.py**, same usage as UDP version.
    - Send a message (string) to the server
    - Read and display the response
    - close the connection.

netcat

nc

lets you create TCP (or UDP) connections from the command line

I hand wrote this HTTP request for you

nc -l PORT

start a server! this listens on PORT and prints everything received

network connection → nc → stdout

nc IP PORT

be a client! opens a TCP connection to IP:PORT.
(to send UDP use -u)

stdin → nc → network connection

make HTTP requests by hand

```
printf 'GET / HTTP/1.1\r\nHost:
example.com\r\n\r\n'
  | nc example.com 80
```

all one line

type in any weird HTTP request you want!

send files

want to send a 100 GB file to someone on the same wifi network? easy!

receiver:

```
nc -l 8080 > file
```

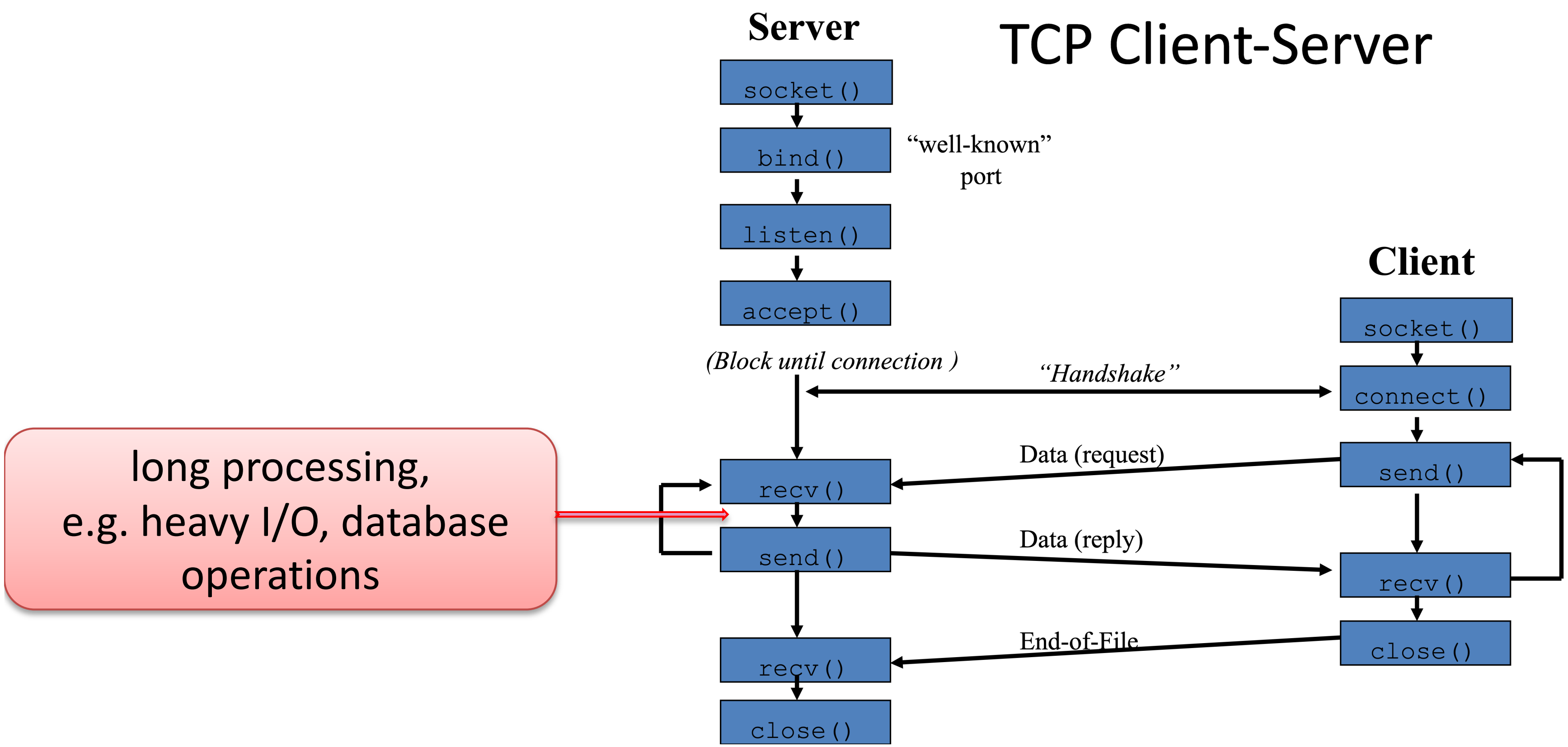
sender:

```
cat file | nc YOUR_IP 8080
```

I ♥ this trick! It works even if you're disconnected from the internet!

TCP server / solution & discussion

TCP servers: handling several clients



TCP servers: handling several clients

Two approaches:

- Forking : **multi-process** server, one process per client session

- Threading: **multi-thread** serve, one thread per client

Python's **socketserver** module simplifies the task of writing network servers. https://docs.python.org/3/library/socketserver.html

TCP server: example

```python
import socketserver

class MyTCPSocketHandler(socketserver.BaseRequestHandler):
    """Class instantiated once per connection to the server, and must
    override the handle() method to implement communication to the client.
    """
    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(BUFSIZE).strip()
        print(f"{self.client_address[0]} wrote:\n {self.data}")
        # Just send back the same data, but upper-cased
        self.request.sendall(self.data.upper())

if __name__ == "__main__":
    port = int(sys.argv[1])
    # instantiate the server, and bind to localhost on port 9999
    server = socketserver.TCPServer(("", port), MyTCPSocketHandler)

    # activate the server: will keep running until Ctrl-C
    server.serve_forever()
```

TCP server: sequential by default

```python
    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(BUFSIZE).strip()

        delay = len(self.data)
        print(f"{self.client_address[0]} waiting for {delay} seconds")
        time.sleep(delay)
        print(f"{self.client_address[0]} answered")
        # just send back the same data, but upper-cased
        self.request.sendall(self.data.upper())
```

TERMINAL   PROBLEMS   OUTPUT   DEBUG CONSOLE

```
viennet@valpo:Codes$ nc localhost 9998
Alllllo
ALLLLLOviennet@valpo:Codes$
```

```
viennet@valpo:Codes$ nc localhost 9998
Ali
ALIviennet@valpo:Codes$
```

```
viennet@valpo:Codes$ ./tcp-socketserver-01-waiting.py 9998
127.0.0.1 waiting for 7 seconds
127.0.0.1 answered
127.0.0.1 waiting for 3 seconds
127.0.0.1 answered
```

Wait until response to client 1 is completed before starting processing client 2.

TCP server: multi-process (forking)

```
if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: tcp-server port")
        exit(1)
    port = int(sys.argv[1])

    # instantiate the server, and bind to localhost on given port
    server = socketserver.ForkingTCPServer(("", port), MyTCPSocketHandler)

    # activate the server
    # this will keep running until Ctrl-C
    server.serve_forever()
```

Using multiple terminals, `nc` and `ps`, show that this server is really launching several processes.

TCP server: multi-thread

```
if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: tcp-server port")
        exit(1)
    port = int(sys.argv[1])

    # instantiate the server, and bind to localhost on given port
    server = socketserver.ThreadingTCPServer(("", port), MyTCPSocketHandler)

    # activate the server
    # this will keep running until Ctrl-C
    server.serve_forever()
```

Using multiple terminals, `nc` and `ps`, show that this server uses only one process.

Thread vs process : pros and cons

- **multi-process** (forking) server:
  - isolation (virtual memory)
  - harder to share data structure between clients

- **Multi-threading** :
  - faster to launch a thread than a process
  - share memory (same address space)
  - prone to hard to spot bugs (race conditions, concurrent access)

(end of part 1)