

Communication par sockets *TCP/IP*

Illustration avec Python 3 et Linux



Renaud Lachaize

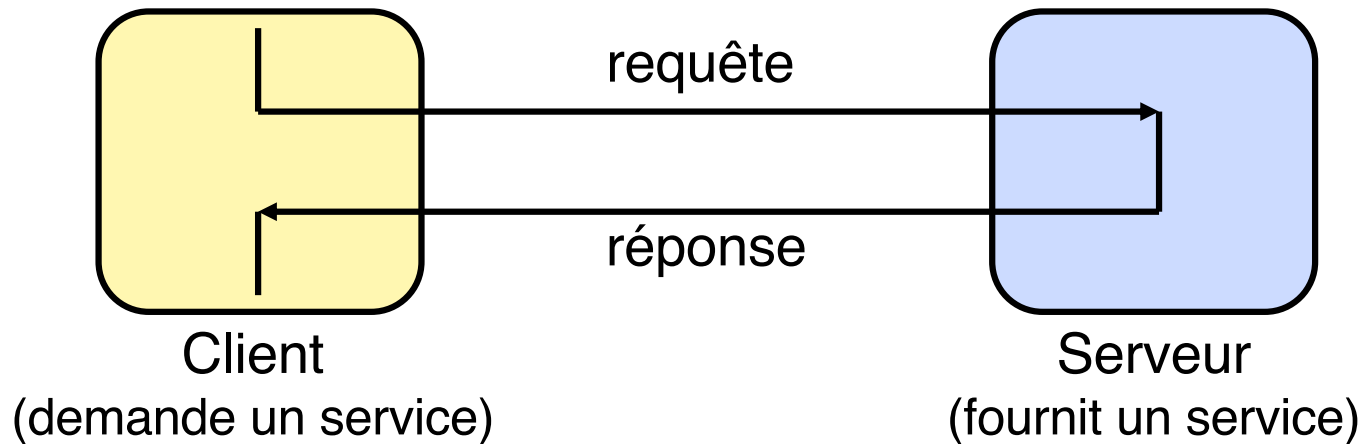
Université Grenoble Alpes

renaud.lachaize @ imag.fr

Avril 2021

Ce cours est partiellement basé sur les diapositives de Sacha Krakowiak

Rappel : le réseau vu de l'utilisateur (1)

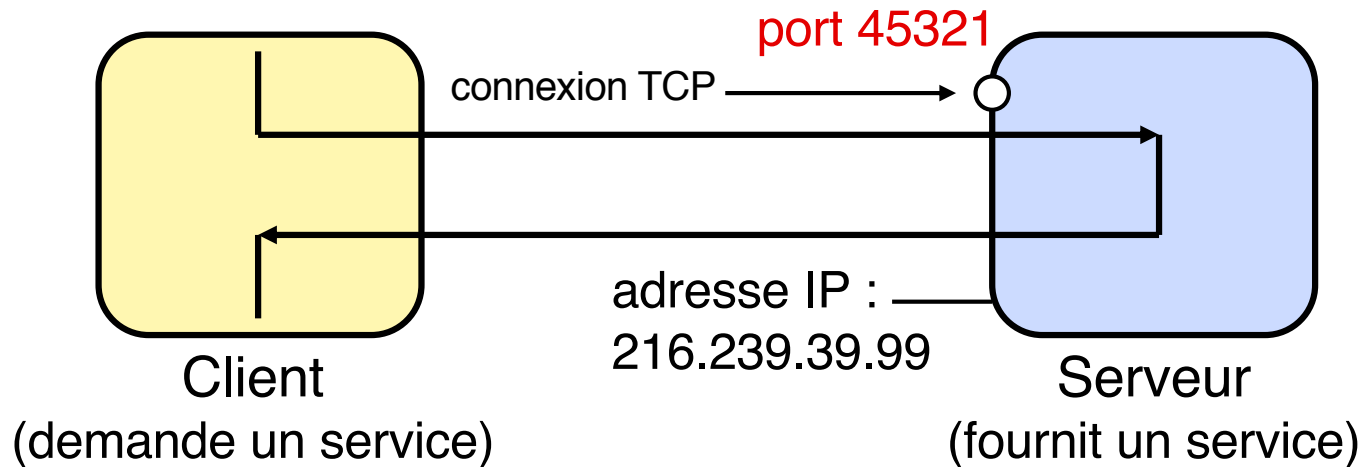


Le schéma **client-serveur** est un concept fondamental pour la programmation d'applications réparties (communication entre deux processus s'exécutant sur des machines distinctes, reliées par un réseau).

Pour le client, un service est souvent désigné par un nom symbolique. Ce nom doit être converti en une adresse interprétable par les protocoles du réseau.

La conversion d'un nom symbolique (par ex. `www.google.com`) en une adresse IP (`216.239.39.99`) est à la charge du service DNS

Le réseau vu de l'utilisateur (2)



En fait, l'adresse IP du serveur ne suffit pas, car le serveur (machine physique) peut comporter différents services; il faut préciser le service demandé au moyen d'un **numéro de port**, qui permet d'atteindre un processus particulier sur la machine serveur.

Un numéro de port comprend 16 bits (0 à 65 535) et est associé à un protocole de transport donné (le port TCP numéro x et le port UDP numéro x désignent des objets distincts).

Les numéros de 0 à 1023 sont réservés, par convention, à des services spécifiques.

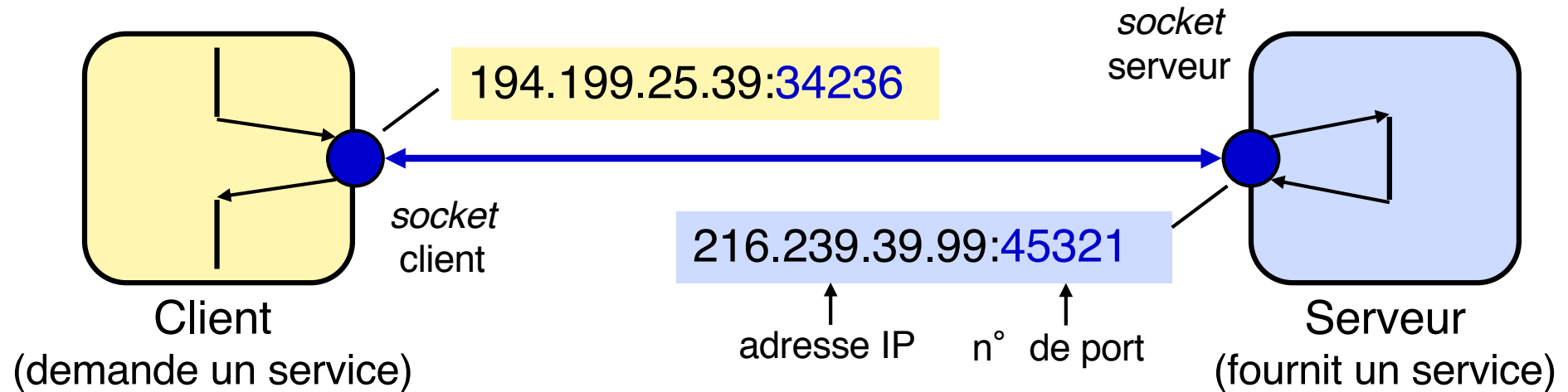
Exemples (avec TCP) :

7 : echo
22 : SSH

25 : SMTP (acheminement mail)
80 : HTTP (serveur web)

443 : HTTPS (HTTP sécurisé)
465 : SMTPS (SMTP sécurisé)

Le réseau vu de l'utilisateur (3)



Pour programmer une application client-serveur, il est commode d'utiliser les *sockets*. Les *sockets* fournissent une interface qui permet d'utiliser facilement les protocoles de transport tels que TCP et UDP.

Une *socket* est simplement un moyen de désigner l'extrémité d'un canal de communication bidirectionnel, côté client ou serveur, en l'associant à un port.

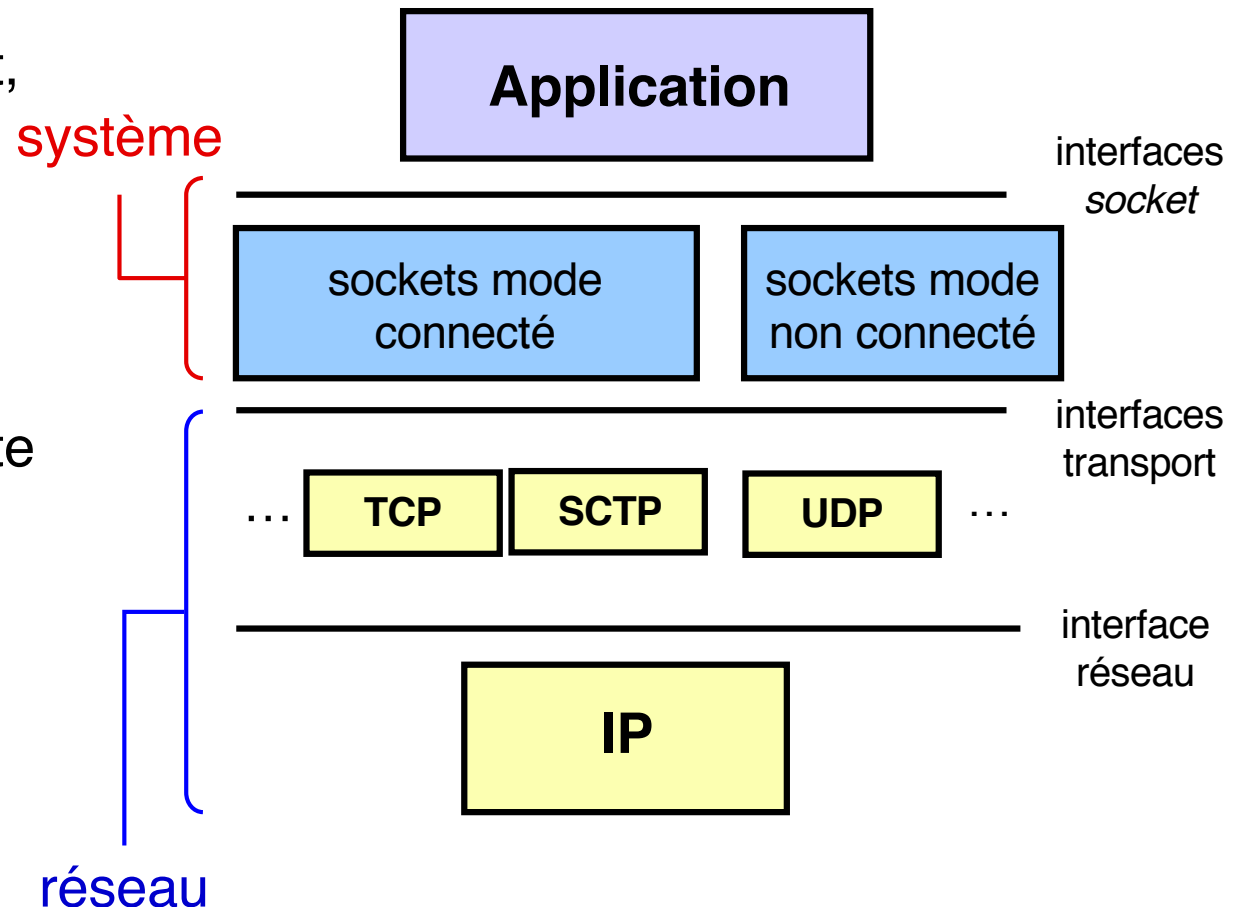
Une fois le canal de communication établi entre processus client et serveur, ceux-ci peuvent communiquer en utilisant les mêmes primitives que pour l'accès à des fichiers (ou à des tubes Unix) : `read`, `write`, ...

Place des *sockets*

Les *sockets* fournissent une interface d'accès, à partir d'un hôte, aux interfaces de transport, notamment TCP et UDP

TCP (mode connecté) : une liaison est établie au préalable entre deux hôtes, et ensuite un flot d'octets est échangé sur cette liaison

UDP (mode non connecté) : aucune liaison n'est établie. Les messages sont échangés individuellement



Dans la suite de ce cours, ne considérons que des *sockets* TCP.

Le protocole TCP

Principales caractéristiques de TCP

Communication bidirectionnelle par flots d'octets

Transmission fiable

Fiabilité garantie dès lors que la liaison physique existe

Transmission ordonnée

Ordre de réception identique à l'ordre d'émission

Contrôle de flux

Permet au récepteur de limiter le débit d'émission en fonction de ses capacités de réception

Contrôle de congestion

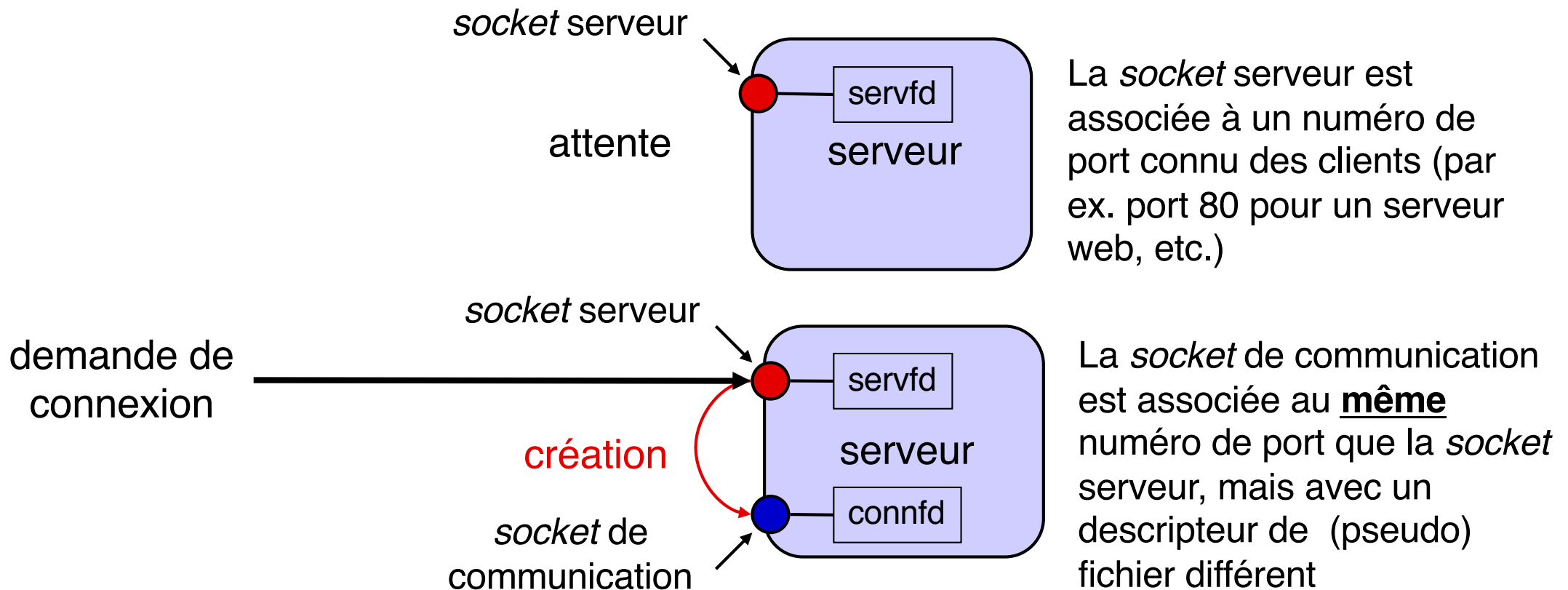
Permet d'agir sur le débit d'émission pour éviter la surcharge du réseau

Ne pas confondre contrôle de flux (entre récepteur et émetteur) et contrôle de congestion (entre réseau et émetteur)

Sockets TCP côté serveur (1)

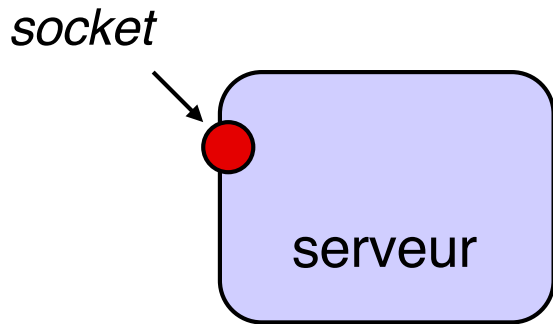
Un serveur en mode connecté doit **attendre** une nouvelle demande de connexion de la part d'un client, puis **traiter** la (ou les requêtes) envoyée(s) sur cette connexion par le client.

Les fonctions d'attente et de traitement sont séparées, pour permettre au serveur d'attendre de nouvelles demandes de connexion pendant qu'il traite des requêtes en cours.



Sockets côté serveur (2)

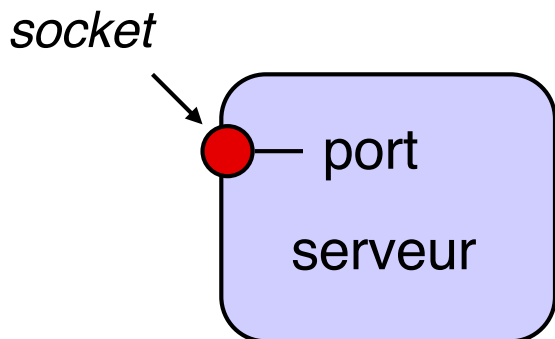
On procède en 4 étapes, décrites schématiquement ci-après (détails plus tard)



Étape 1 : créer une *socket*

```
server_sock = socket.socket(addr_family, socket.SOCK_STREAM)
```

↑
socket.AF_INET (pour IPv4) ou TCP
socket.AF_INET6 (pour IPv6)



Étape 2 : associer la *socket* à une adresse IP et un numéro de port TCP locaux :

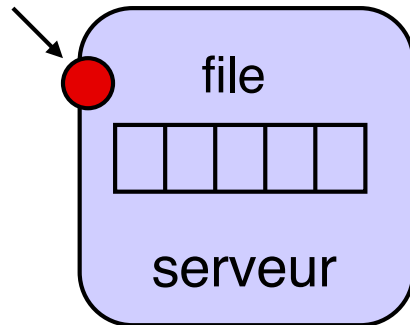
```
server_sock.bind((my_addr, my_port))
```

On peut éventuellement utiliser une chaîne de caractères vide ("") pour indiquer qu'il faut considérer l'ensemble des adresses IP associées à la machine locale ("*wildcard*"), plutôt qu'une adresse particulière

Numéro de port TCP sur lequel le serveur doit écouter
Par exemple : 8542

Sockets côté serveur (3)

socket serveur



Étape 3 : indiquer que c'est une socket serveur

```
LISTEN_BACKLOG_SIZE = 5 # par exemple  
server_sock.listen(LISTEN_BACKLOG_SIZE)
```

Taille max de la file d'attente des demandes

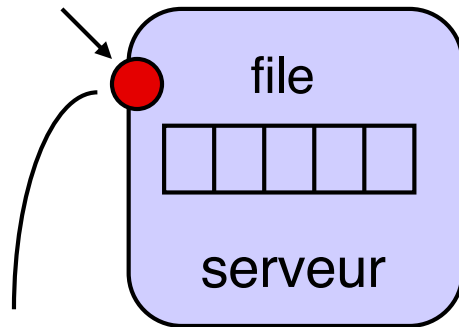
Une *socket* serveur est en attente de demandes de connexion.

Après un appel réussi à *listen*, le système d'exploitation de la machine serveur peut commencer à recevoir des demandes de connexion

Si une demande de connexion arrive pendant qu'une autre est en cours de traitement, elle est placée dans une file d'attente. Si une demande arrive alors que la file est pleine, elle est rejetée (pourra être refaite plus tard) ; voir primitive *connect* plus loin.

Sockets TCP côté serveur (4)

socket serveur



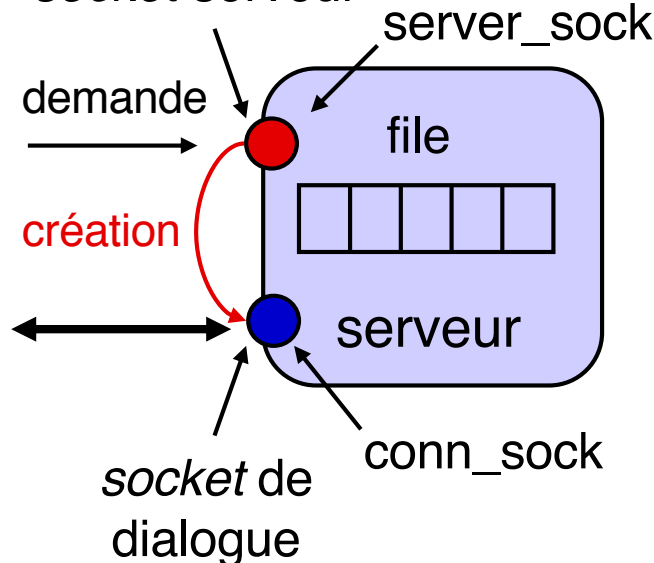
prête à accepter les demandes de connexion

Étape 4a : permettre à l'application (côté serveur) de prendre connaissance d'une nouvelle connexion

```
(conn_sock, client_info) = server_sock.accept()
```

la primitive accept est **bloquante** (si la file est vide)

socket serveur



Étape 4b : obtention d'un canal de communication/dialogue par l'application (côté serveur)

```
(conn_sock, client_info) = server_sock.accept()
```

Au retour de l'appel à *accept* (qu'il soit bloquant ou non),

- cette variable contient une référence vers un objet socket permettant de dialoguer avec un client
- la connexion est retirée de la file d'attente

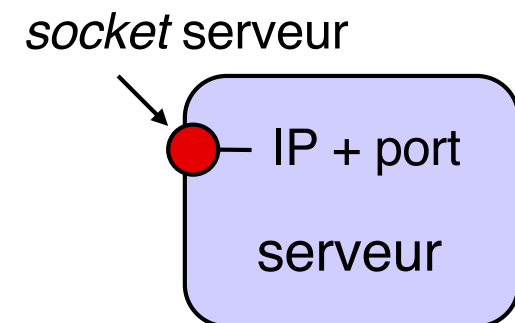
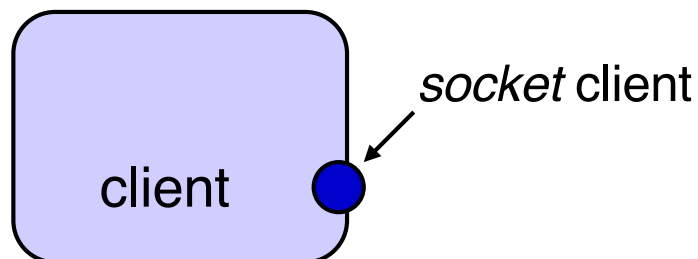
Sockets TCP côté client (1)

On procède en plusieurs étapes, décrites schématiquement ci-après

On suppose que l'on connaît l'adresse d'un serveur et le numéro de port (TCP) d'une *socket* serveur sur celui-ci
(un processus serveur est en attente sur ce port)

Étape 1 : créer une *socket* :

```
client_sock = socket.socket(addr_family, socket.SOCK_STREAM)
```



Le serveur est en attente sur la *socket* (*accept*)

Sockets TCP côté client (2)

Étape 1 bis :

associer la socket à une adresse et/ou un numéro de port (locaux) particuliers:

```
client_sock.bind((my_client_addr, my_client_port))
```



Cette étape n'est en général pas nécessaire et elle est souvent omise.

Dans ce cas, le système d'exploitation de la machine cliente :

- Choisit un numéro de port libre dans une plage prédéfinie (ports dit « *éphémères* » ou « *dynamiques* »). Cette plage prédéfinie varie selon les systèmes d'exploitation et leur configuration.
- Si la machine possède plusieurs adresses IP, associe à la socket *l'adresse IP par défaut* de la machine

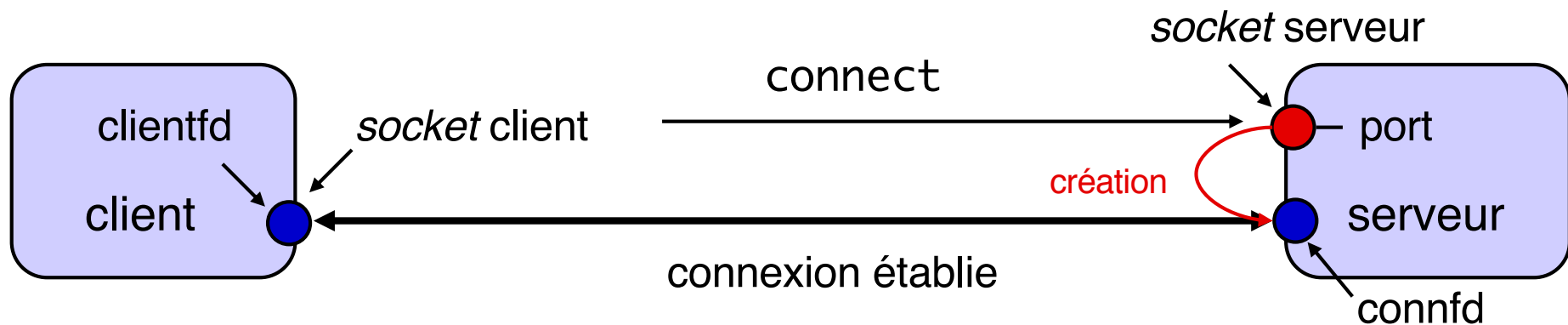
Sockets TCP côté client (3)

Étape 2 : établir une connexion entre la socket client et le serveur

```
# Obtention d'une structure sock_addr décrivant l'application serveur
# Pour cela, le client doit connaître :
# - le nom ou l'adresse IP de la machine serveur (cf. host ci-dessous)
# - le numero de port TCP de l'application serveur (cf. port ci-dessous)
sock_addr = socket.getaddrinfo(host, port, family=addr_family,
                                type=socket.SOCK_STREAM, proto=socket.IPPROTO_TCP,
                                flags=socket.AI_CANONNAME)

# Lancement de la tentative de connexion :
client_sock.connect(sock_addr)
```

connect envoie une demande de connexion vers la *socket* serveur
(une exception *OSError* est levée en cas d'échec de la tentative de connexion)



Le client et le serveur peuvent maintenant dialoguer sur la connexion

Échanges sur une connexion entre *sockets*

Une fois la connexion établie, le client et le serveur disposent chacun d'un descripteur (pseudo-fichier) vers l'extrémité correspondante de la connexion.

À partir de chaque extrémité, on peut créer 2 flux pour interagir avec l'interlocuteur distant:

- Un **flux d'entrée** (en **lecture**) : réception/consommation des données émises par l'interlocuteur
- Un **flux de sortie** (en **écriture**) : émission/production de données vers l'interlocuteur

Ces flux associés à une connexion sont grosso modo analogues à des flux d'accès à un fichier classique ... mais avec quelques différences importantes.

En particulier, la production et la consommation sont nécessairement en mode FIFO (*first in, first out*). Il n'est pas possible de « rembobiner » un flux lié à une connexion réseau. Caractéristique analogue aux flux associés à un tube Unix (« *Unix pipe* »).

Une lecture sur un flux d'entrée peut être bloquante (si le tampon de réception ne contient pas de nouvelles données) ... jusqu'à l'arrivée de nouvelles données envoyées par l'interlocuteur (ou jusqu'à la rupture de la connexion réseau).

Manipulation de flux en Python

- Python fournit un ensemble de primitives pour manipuler les deux flux de communication intégrés au sein d'une socket.
 - ◆ Voir la documentation officielle :
 - ❖ <https://docs.python.org/3/library/socket.html>
 - ❖ <https://docs.python.org/3/howto/sockets.html>

- Il est également possible de manipuler les flux associés à une socket de la même façon que des flux d'accès à un fichier.
 - ◆ Pour obtenir un objet d'interface « fichier » à partir d'une socket, utiliser la fonction `socket.makefile()` (cf. documentation ci-dessus)
 - ◆ Ensuite, à partir de cet objet, il est possible d'utiliser les interfaces de manipulations de flux d'entrées/sorties (on parle d'*Input/Output streams* ou encore d'*I/O streams*). Voir la documentation : <https://docs.python.org/3/library/io.html>
 - ◆ Ces flux peuvent être de différents types :
 - ❖ Flux de caractères (*text I/O stream*) ou flux d'octets arbitraires (*binary I/O stream*)
 - ❖ Avec utilisation d'un tampon mémoire intermédiaire (*buffered I/O*) ou sans (*raw I/O*)

Dans le contexte d'une communication par sockets, le choix du type de flux le plus propice dépend de la spécification du protocole applicatif (format choisi pour l'échange des données) et des caractéristiques de l'application.

Identification des connexions

- **Comment une machine peut-elle identifier la connexion réseau (et donc la socket concernée) lorsqu'elle reçoit un paquet ?**

- En considérant l'adresse IP et le numéro de port (TCP) de destination indiqués par l'émetteur ?
 - ◆ Oui ... mais plusieurs clients peuvent envoyer des paquets à destination d'un même couple [adresse IP serveur , port serveur]
 - ❖ ... car, sur une machine serveur, les différentes sockets de dialogue créées à partir de la même socket d'écoute sont toutes associées au même numéro de port
 - ◆ Donc ce n'est pas suffisant !

- **On peut identifier (de manière unique) la connexion associée à un paquet à partir du quadruplet suivant :**
 - ◆ [**addr. IP source , port source , addr. IP destination , port destination**]
 - ◆ Cette méthode est utilisée par le système d'exploitation pour aiguiller un paquet reçu vers la socket concernée

Echange de données à travers un réseau

Protocole applicatif

- Un processus qui obtient des données à partir des sockets TCP manipule un flux d'octets
- C'est à l'application de découper ce flux de réception en messages
- Comment faire (au niveau applicatif) ? Plusieurs approches :
 - ◆ Messages de taille fixe
 - ◆ Messages de taille variable → nécessité d'utiliser :
 - ❖ Soit un en-tête de taille fixe indiquant le type/la taille du message complet
 - ❖ Soit des marqueurs de fin de champs et de fin de message

Echange de données à travers un réseau

Interopérabilité (1/3)

- **Un protocole applicatif doit pouvoir fonctionner correctement entre plateformes hétérogènes**
 - ◆ (« *Plateformes* » ici au sens large : processeur, système d'exploitation, langage de programmation ...)
 - ◆ Nécessité de permettre la communication entre un processus client et un processus serveur mis en œuvre sur des plateformes hétérogènes
 - ❖ Par exemple, une application cliente écrite en langage Java et s'exécutant sur le système macOS doit pouvoir communiquer avec une application écrite en langage C et s'exécutant sur le système Linux, dès lors qu'elles sont conformes à la même spécification de protocole applicatif.
 - ◆ Nécessité de pouvoir utiliser/exécuter un même programme (client ou serveur) sur différentes plateformes (portabilité du code)

- **Différentes sources d'hétérogénéité**
 - ◆ Plateformes avec des tailles/formats de mots différentes
 - ◆ et/ou avec des conventions de « boutisme » différentes (en anglais, “*endianness*” : convention “*big endian*” vs. “*little endian*”).

Echange de données à travers un réseau

Interopérabilité (2/3)

Implication n° 1 : La spécification du protocole doit fournir une définition précise de la taille des types de base (et de l'encodage) utilisés pour les champs d'un message

- Taille : 32 bits / 64 bits / ...

- Encodage :
 - ◆ Entier signé ou non
 - ◆ Représentation binaire (par exemple : complément à 2 pour un entier, notation IEEE 754 pour un nombre flottant, encodage des chaînes de caractères ...)

- Attention :
 - ◆ Différents langages n'associent pas nécessairement le même format à un type du même nom (comme par exemple *integer*)
 - ◆ Pour certains langages, un même type n'a pas nécessairement le même format selon les plateformes (exemple : le type *int* du langage C)
 - ◆ En Java : il existe une spécification universelle (valable pour toutes les plateformes) et non ambiguë des types de base (*types primitifs*). Par exemple, le type *int* correspond toujours à un entier signé encodé en complément à 2 sur 32 bits.
 - ◆ En Python, les types de base sont également définis de manière portable

Echange de données à travers un réseau

Interopérabilité (3/3)

Implication n° 2 : La spécification du protocole doit fournir une définition précise du « boutisme » pour les types de bases utilisés dans les champs d'un message.

Exemple avec un entier stocké sur 4 octets :

« Ai-je reçu le nombre 0x1234abcd ou bien 0xcdab3412 ? »

- Il existe par convention un boutisme « réseau » standard (*network byte order*) à utiliser pour les champs d'un message à envoyer sur le réseau. Il s'agit du boutisme *big endian*.
 - ◆ Lors de la fabrication d'un message (avant émission) : effectuer conversion entre boutisme local et boutisme réseau
 - ◆ Après la réception d'un message : opération inverse
 - ◆ En Python, voir les fonctions commençant par `socket.hton` et `socket.ntoh` (*hton* = *host to network* et *ntoh* = *network to host*). Cf. documentation: <https://docs.python.org/3/library/socket.html>
- Remarques :
 - ◆ Cette précaution s'applique aussi aux champs de type « adresses IP » et « numéro de port » inclus dans les messages envoyés sur le réseau
 - ◆ Cette précaution **n'est pas nécessaire** pour les contenus basés sur des séquences arbitraires d'octets :
 - ❖ Chaînes de caractères (si la taille d'un caractère ne dépasse pas un octet)
 - ❖ Fichiers binaires « opaques » (c'est-à-dire manipulés comme un flux d'octets). Par exemple, une image/vidéo).
 - ❖ C'est le cas pour les exemples de code fournis dans le TP associé à ce cours.

Un application client-serveur avec *sockets* (1)

Principes de la programmation d'une application avec *sockets*
(les déclarations et initialisations de variables sont omises).

Côté serveur :

```
server_sock = socket.socket(addr_family, socket.SOCK_STREAM)
server_sock.bind((my_addr, my_port))
server_sock.listen(LISTEN_BACKLOG_SIZE)

while True:
    (conn_sock, client_info) = server_sock.accept()

    # le code « métier » du serveur proprement dit,
    # qui implémente le dialogue avec un client
    handle_dialog_with_client(conn_sock)

    # lorsque ce dialogue se termine, on ferme la connexion
    # (si pas déjà fait dans la méthode précédente)
    conn_sock.close();

    # maintenant on peut accepter la prochaine connexion
```

Un application client-serveur avec *sockets* (2)

Principes de la programmation d'une application avec *sockets*
(les déclarations et initialisations de variables sont omises).

Côté client :

```
client_sock = socket.socket(addr_family, socket.SOCK_STREAM)
sock_addr = socket.getaddrinfo(...)
client_sock.connect(sock_addr)

# le code « métier » du client proprement dit,
# qui implémente le dialogue avec le serveur
handle_dialog_with_server(client_sock)

# lorsque ce dialogue se termine, on ferme la connexion
# (si pas déjà fait dans la méthode précédente)
client_sock.close()
```

Un application client-serveur avec *sockets* (3)

Voir les exemples fournis en TP.

Pour exécuter l'application :

Lancer le programme serveur sur une machine, en indiquant un numéro de port >1023

Les numéros ≤ 1023 nécessitent les privilèges de l'administrateur (super-utilisateur) de la machine

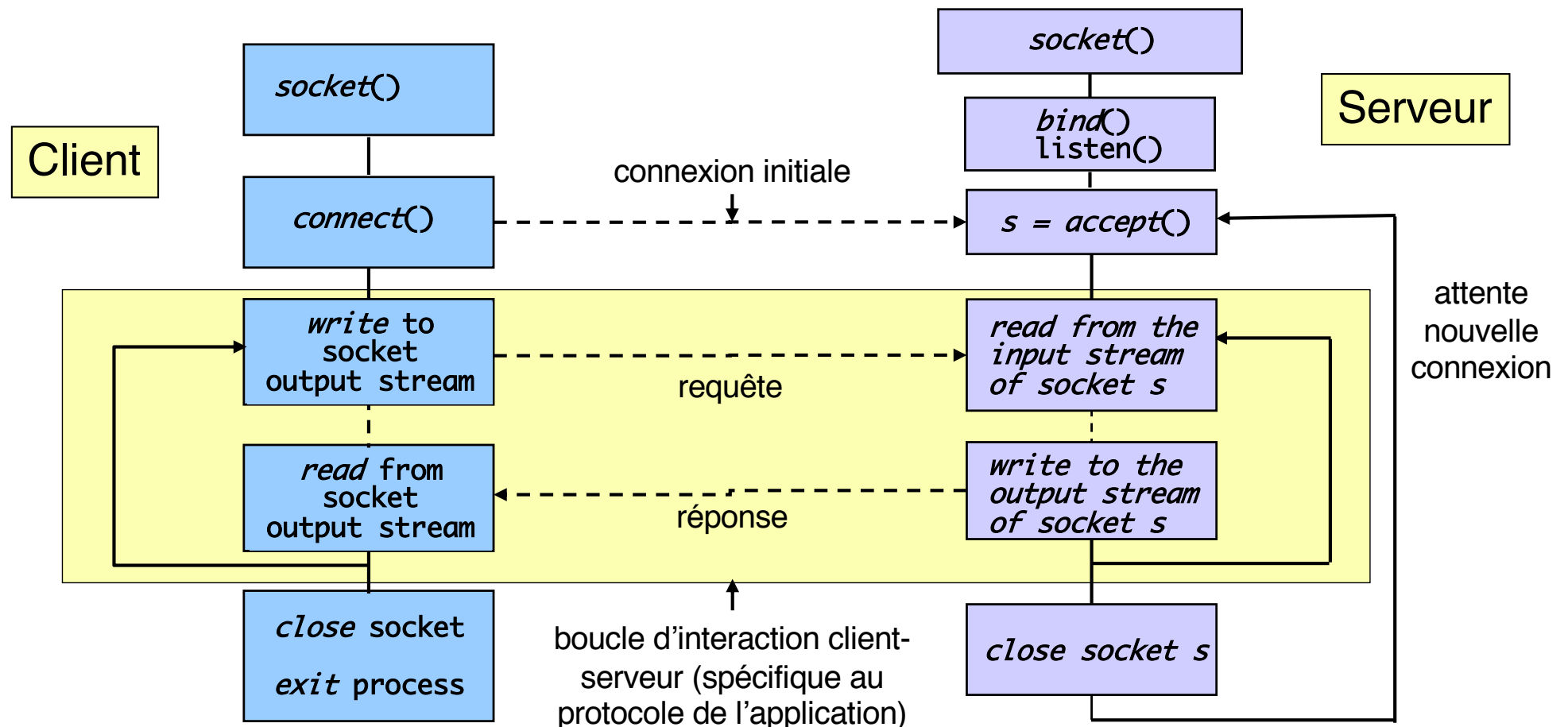
Lancer le programme client sur une autre machine (ou dans un autre processus de la même machine), en spécifiant adresse du serveur et numéro de port

Remarque : On n'a pas prévu d'arrêter le serveur (il faut tuer le processus qui l'exécute). Dans une application réelle, il faut prévoir un mécanisme pour arrêter proprement le serveur.

Client-serveur en mode itératif

Les programmes précédents réalisent un serveur en **mode itératif** : un seul client est servi à la fois.

Schéma :



Client-serveur en mode concurrent (1)

Pour réaliser un serveur en **mode concurrent**, une solution consiste à **créer un nouveau flot d'exécution** pour servir chaque demande de connexion.

Pour gérer de multiples flots d'exécution, on peut utiliser **plusieurs threads** (au sein d'un même processus) ou bien **plusieurs processus**.

Dans le cas de nos exemples en Python étudiés en TP, **nous utiliserons une architecture multiprocessus** pour construire un serveur concurrent.

Architecture du serveur concurrent :

- Il y a un flot d'exécution principal (appelé *veilleur*) qui attend sur *accept()*.
- Lorsqu'il reçoit une demande de connexion, le veilleur crée un nouveau flot (appelé *exécutant*) qui va interagir avec le nouveau client.
- Après la création de l'exécutant, le veilleur revient se mettre en attente sur *accept()*.
- Plusieurs exécutants peuvent co-exister simultanément.
- Lorsqu'un exécutant a fini de dialoguer avec un client, il se termine.

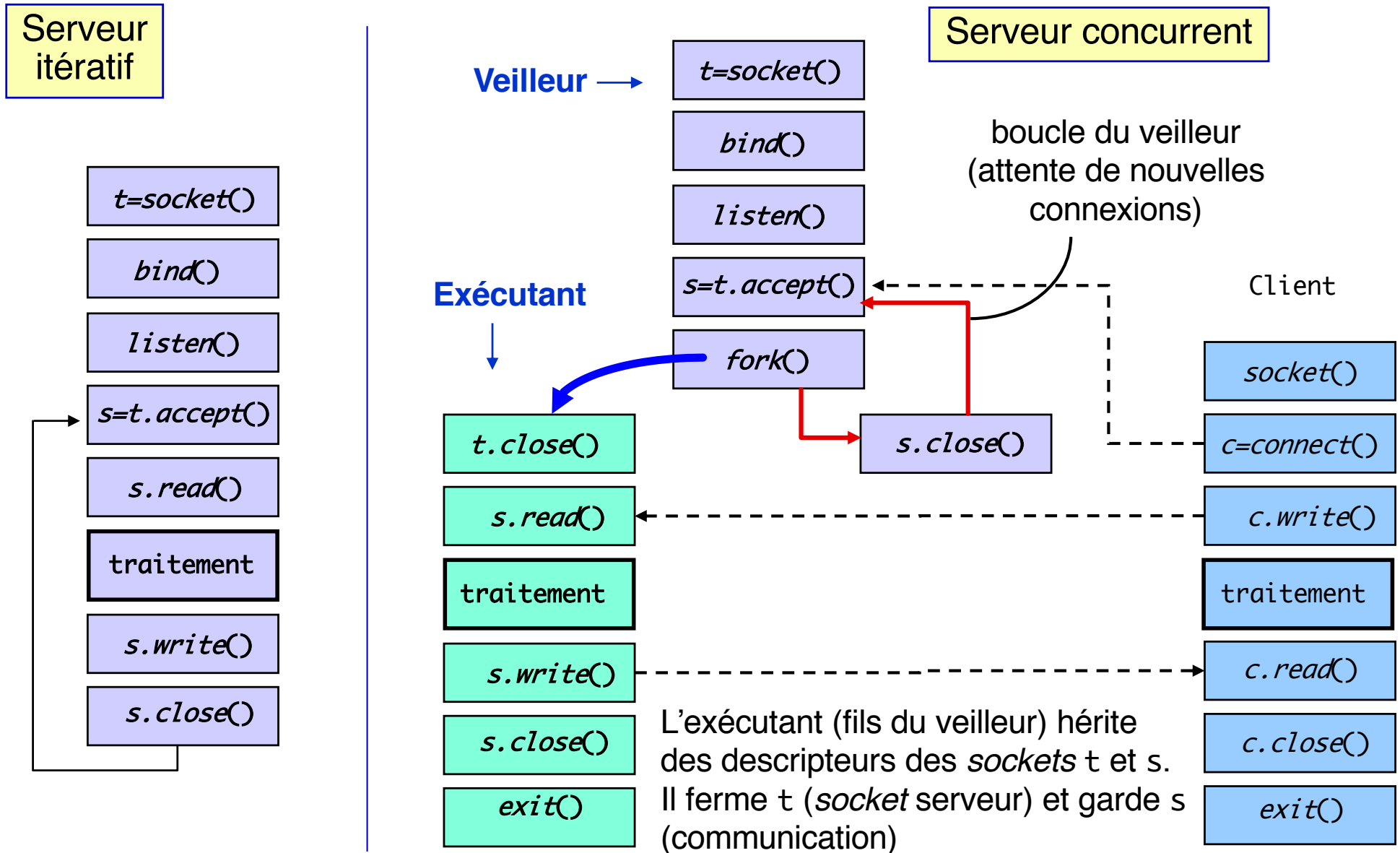
Client-serveur en mode concurrent (2)

Il existe d'autres solutions pour concevoir un serveur concurrent (que nous n'étudierons pas par manque de temps) :

- Utilisation d'un ensemble pré-alloué de flots *exécutants* afin d'éviter les créations systématiques.
 - ◆ Un exécutant peut être « ré-utilisé » pour traiter successivement plusieurs clients.

- Utilisation de mécanismes qui permettent à un même flot d'exécution de gérer de manière concurrente plusieurs canaux de communication.
 - ◆ On parle de « programmation événementielle » (*event-driven programming*).

Client-serveur en mode concurrent (3)



Client-serveur avec *sockets* : lancement

Mode d'emploi

sur le serveur

```
mandelbrot$ server 4321
```

lance le serveur
sur le port 4321

sur le client

```
imapc204-03$ client mandelbrot.e.ujf-grenoble.fr 4321
```

appelle le
serveur distant

Les programmes client et serveur sont indépendants, et lancés séparément. Le lien entre les deux est la connaissance par le client du **nom du serveur** et du **numéro de port** du service (et du protocole de transport utilisé).

Client et serveur peuvent s'exécuter sur deux machines différentes, ou sur la même machine (dans ce dernier cas, utiliser le nom `localhost`). Le serveur doit être lancé **avant** le client.

Observer la liste des sockets TCP sur une machine (Linux)

■ Commande netstat

- ◆ `netstat -t` (équivalent à : `netstat -A inet --tcp`)

■ Options utiles

- ◆ `-a` ou `--all` : permet d'afficher toutes les sockets existantes sur la machine (par défaut, seules les sockets connectées sont listées)
- ◆ `-l` ou `--listen` : affiche uniquement les sockets serveurs
- ◆ `-p` : affiche le pid du processus propriétaire d'une socket
- ◆ `-e` : permet de connaître l'utilisateur associé au processus propriétaire d'une socket
- ◆ `--numeric-hosts` : désactiver la résolution des noms de machines (affichage des adresses IP)
- ◆ `--numeric-ports` : désactiver la résolution des numéros de ports (par défaut, les numéros de ports utilisés par les services usuels sont remplacés par le nom du service correspondant, à partir des informations disponibles dans le fichier `/etc/services`)

Observer la liste des sockets TCP sur une machine

Exemple

■ Configuration du test

- ◆ Serveur lancé sur la machine imablade04 (port 7777)
- ◆ Client lancé sur la machine mandelbrot

```
mandelbrot$ netstat -t -a --numeric-ports --numeric-hosts
Active Internet connections (servers and established)
Proto    Recv-Q  Send-Q  Local Address           Foreign Address         State
...
tcp      0        0      195.220.82.165:43103    195.220.82.164:7777    ESTABLISHED
...
```

```
imablade04$ netstat -t -a --numeric-ports --numeric-hosts
Active Internet connections (servers and established)
Proto    Recv-Q  Send-Q  Local Address           Foreign Address         State
...
tcp      0        0      0.0.0.0:7777             0.0.0.0:*              LISTEN
tcp      0        0      195.220.82.164:7777    195.220.82.165:43103    ESTABLISHED
...
```

Un autre outil pour l'étude des sockets : ss

■ Commande

- ◆ `ss -t`

■ Options utiles

- ◆ `-a` ou `--all` : permet d'afficher toutes les sockets existantes sur la machine (par défaut, seules les sockets connectées sont listées)
- ◆ `-l` ou `--listening` : affiche uniquement les sockets serveurs
- ◆ `-p` : affiche le pid du processus propriétaire d'une socket
- ◆ `-e` : permet de connaître l'utilisateur associé au processus propriétaire d'une socket
- ◆ `-n` ou `--numeric` : désactiver la résolution des noms de services
- ◆ `-r` ou `--resolve` : activer la résolution des noms de machines et de services

Bibliographie : Programmation réseau en Python

- Les documentations citées dans les diapositives précédentes

- Tutoriels :
 - ◆ <https://realpython.com/python-sockets/>
 - ◆ https://python.developpez.com/cours/apprendre-python3/?page=page_20