Communication par *sockets TCP/IP*

Illustration avec Java

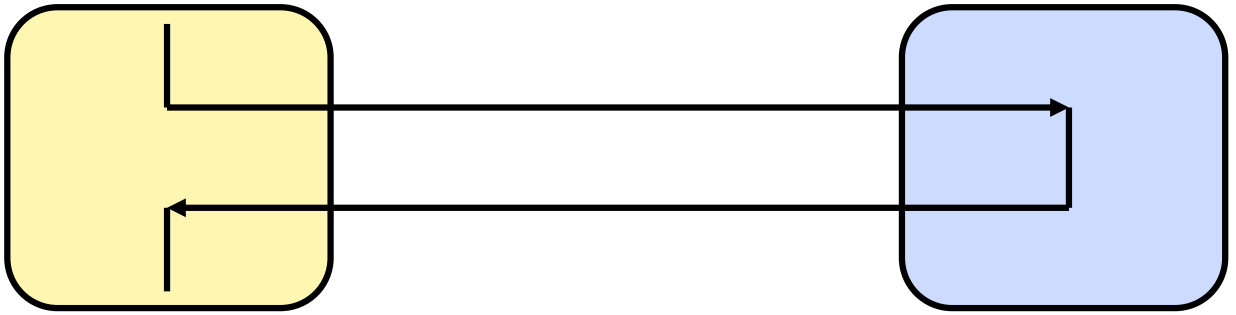
Renaud Lachaize

Université Grenoble Alpes

renaud.lachaize @ imag.fr
Janvier 2020

Ce cours est partiellement basé sur les transparents de Sacha Krakowiak

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

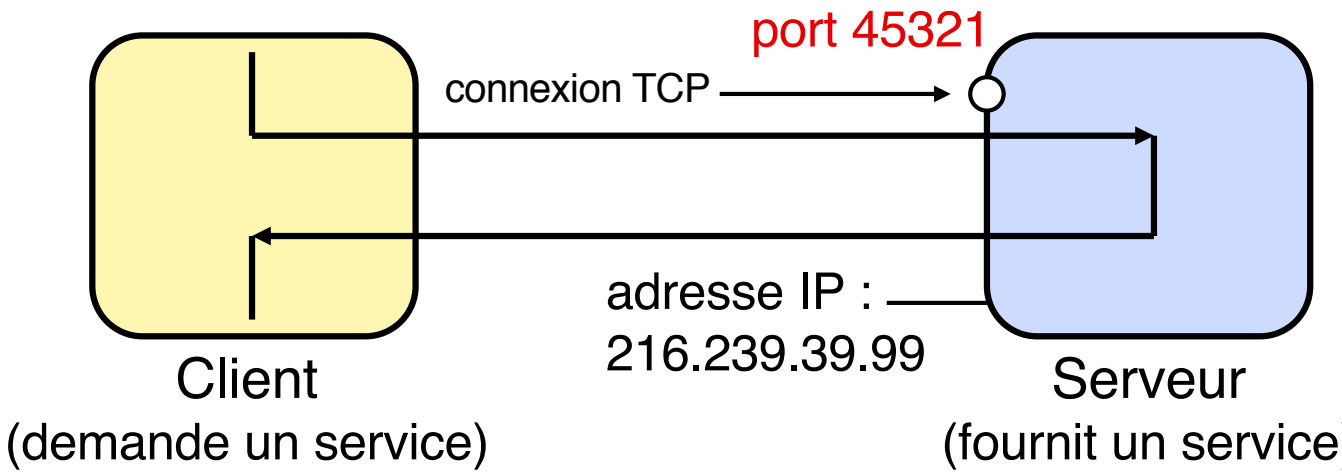


Le schéma client-serveur a été vu en TD pour des processus sur une même machine. Ce schéma se transpose à un réseau, où les processus client et serveur sont sur des machines différentes.

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)



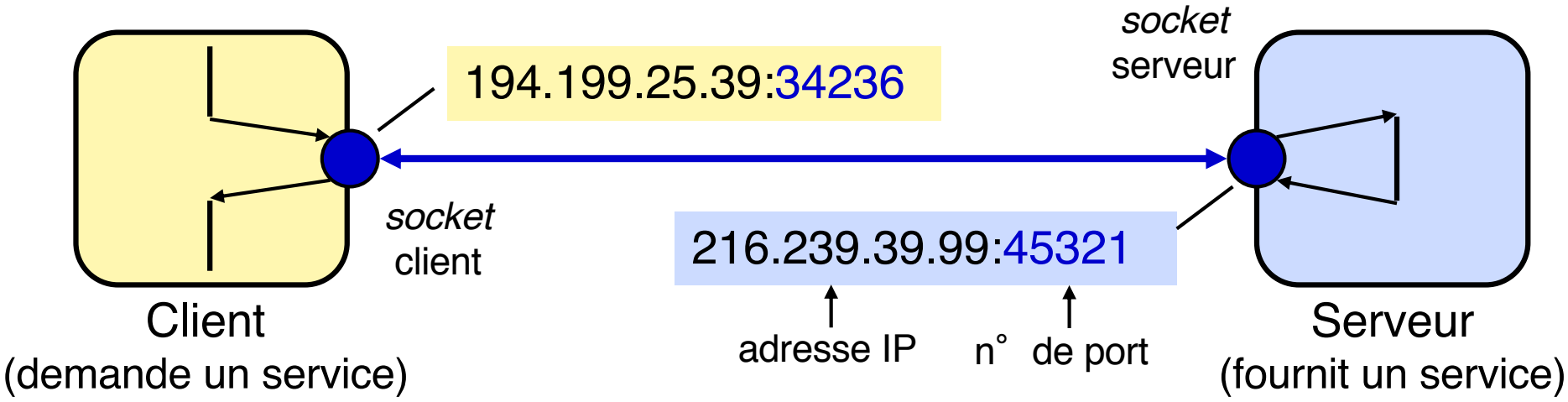
En fait, l'adresse IP du serveur ne suffit pas, car le serveur (machine physique) peut comporter différents services; il faut préciser le service demandé au moyen d'un numéro de port, qui permet d'atteindre un processus particulier sur la machine serveur.

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

Les numéros de 0 à 1023 sont réservés, par convention, à des services spécifiques.
Exemples (avec TCP) :

7 : echo	25 : SMTP (acheminement mail)	443 : HTTPS (HTTP sécurisé)
22 : SSH	80 : HTTP (serveur web)	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.

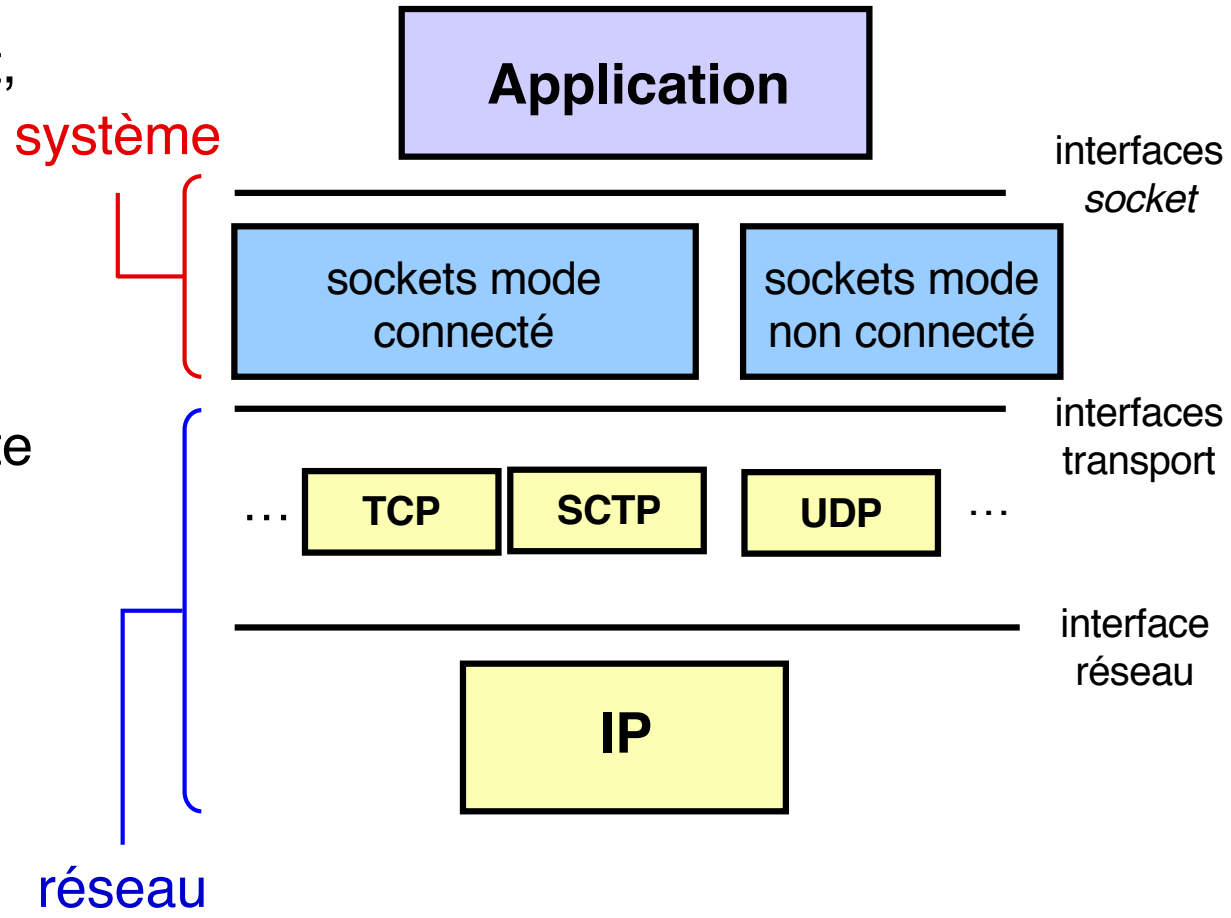
Une fois le canal de communication établi entre processus client et serveur, ceux-ci peuvent communiquer en utilisant les mêmes primitives que pour l'accès aux fichiers/tubes.

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 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

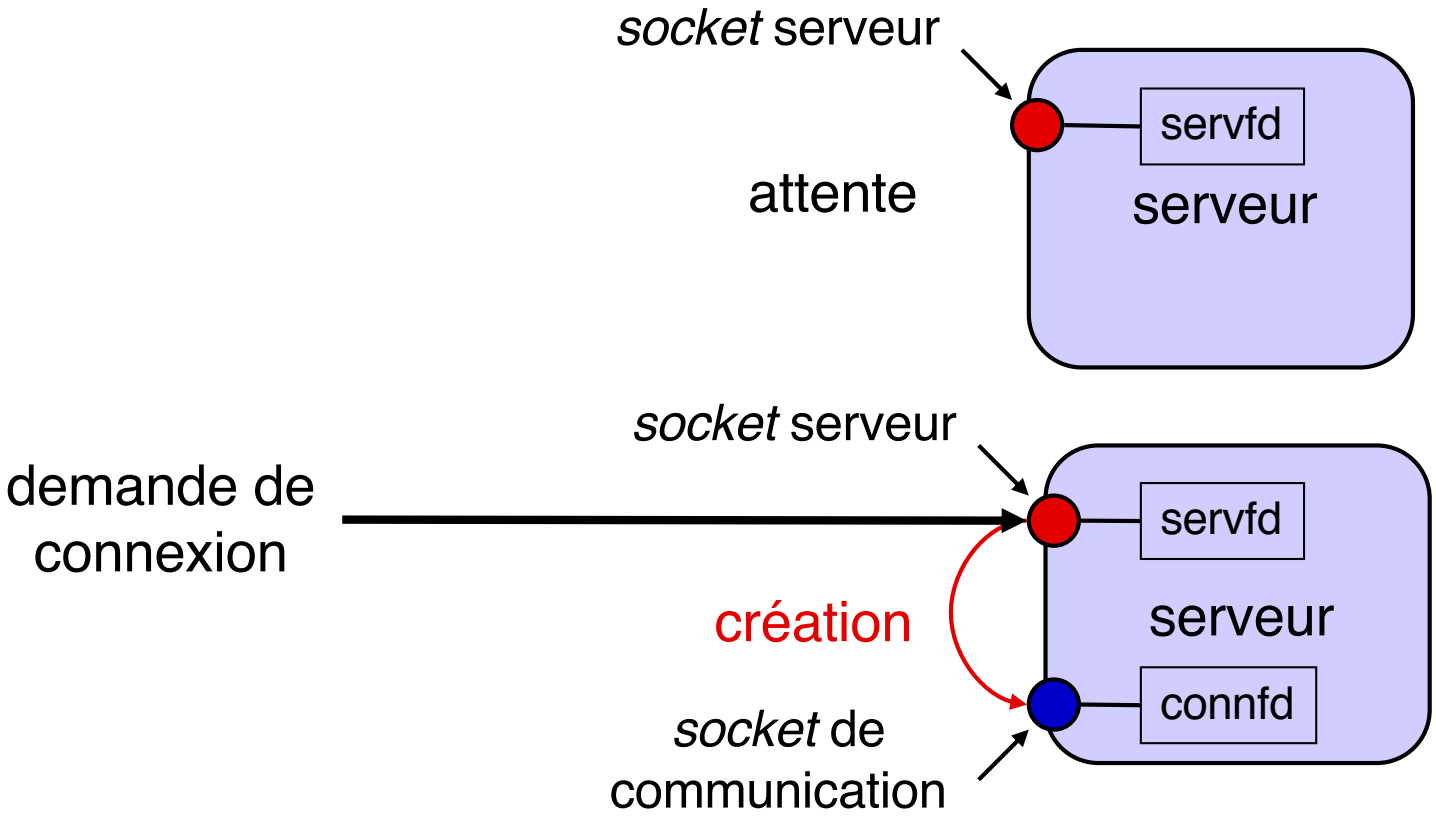
Permet d'agir sur le débit d'émission pour éviter la surcharge du réseau

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.

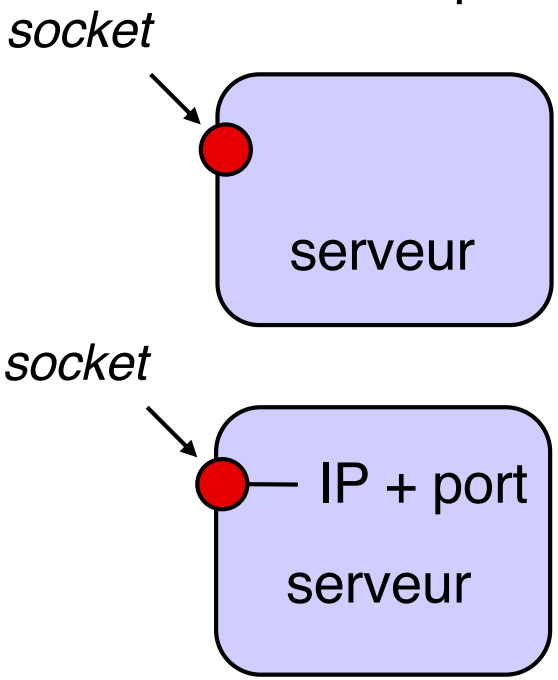


La *socket* serveur est associée à un numéro de port connu des clients (par ex. port 80 pour un serveur web, etc.)

La *socket* de communication est associée au **même** numéro de port que la *socket* serveur, mais avec un descripteur différent

Sockets TCP côté serveur (2)

On procède en plusieurs étapes, décrites schématiquement ci-après :



Étape 1 : créer une *socket* serveur TCP:

```
ServerSocket servSock = new ServerSocket();
```

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

```
int myPort = …
InetSocketAddress ipAddrAndPort;

ipAddrAndPort = new InetSocketAddress(myPort);

servSock.bind(ipAddrAndPort, …);
```

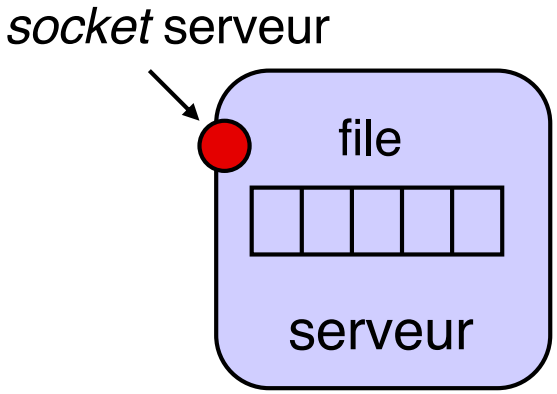
Numéro de port sur lequel le serveur doit écouter

Cette classe décrit un couple (addresse IP, n° de port)

Cette version du constructeur indique implicitement qu'il faut considérer l'ensemble des adresses IP associées à la machine locale (*"wildcard"*), plutôt qu'une adresse particulière

Voir page suivante (backlog)

Sockets TCP côté serveur (3)



Étape 2.b : définir la taille de la file d'attente pour les nouvelles connexions

```
int backlogSize = …;
…
servSock.bind(ipAddrAndPort, backlogSize);
```

Taille de la file d'attente (nombre de nouvelles connexions)

Voir page précédente

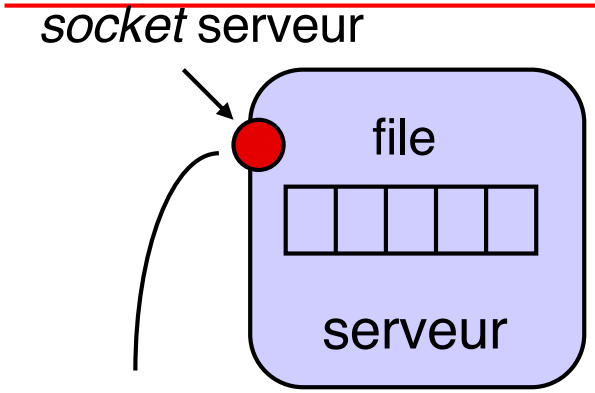
Après les étapes décrites précédemment, la *socket* serveur est fonctionnelle :

Le système d'exploitation de la machine serveur peut commencer à recevoir des demandes de connexion vers cette socket et à établir une connexion TCP avec les clients qui le contactent.

Chaque nouvelle connexion établie par le système d'exploitation du serveur est placée dans la file d'attente.
Comment la file d'attente est-elle vidée ? Voir étape suivante.

Si une demande de connexion arrive alors que la file est pleine, elle est rejetée (pourra être réessayée plus tard) ; voir primitive *connect* plus loin.

Sockets TCP côté serveur (4)

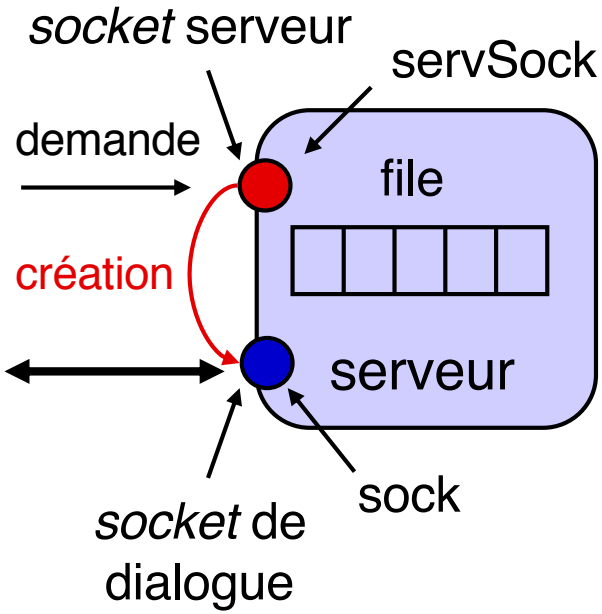


prête à accepter les demandes de connexion

Étape 3a : permettre **à l'application** (côté serveur) de prendre connaissance d'une nouvelle connexion

```
Socket sock;
…
sock = servSock.accept();
```

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



Étape 3b : obtention d'un canal de communication/dialogue **par l'application** (côté serveur)

```
sock = servSock.accept();
```

Au retour de l'appel à *accept*,

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

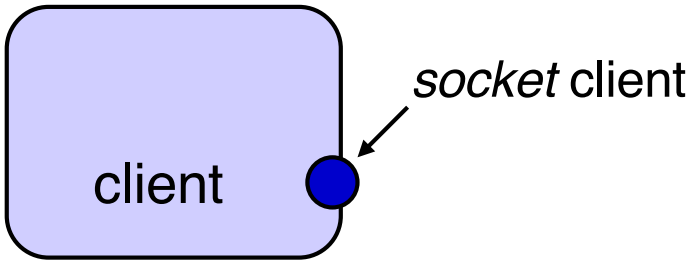
Sockets TCP côté client (1)

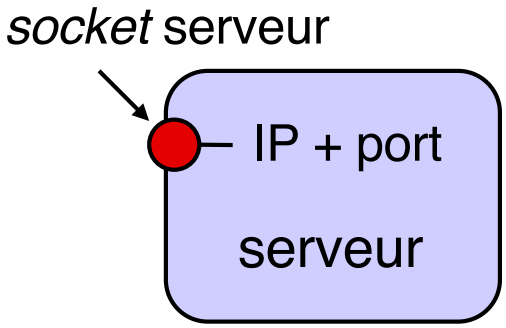
On procède en plusieurs étapes, décrites schématiquement ci-après

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* :

```
Socket sock = new Socket();
```





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:

```
InetSocketAddress localIpAddrAndPort = new InetSocketAddress(…);
sock.bind(localIpAddrAndPort);
```

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* »)
- Si la machine possède plusieurs adresses IP, associe à la socket *l'adresse IP par défaut* de la machine

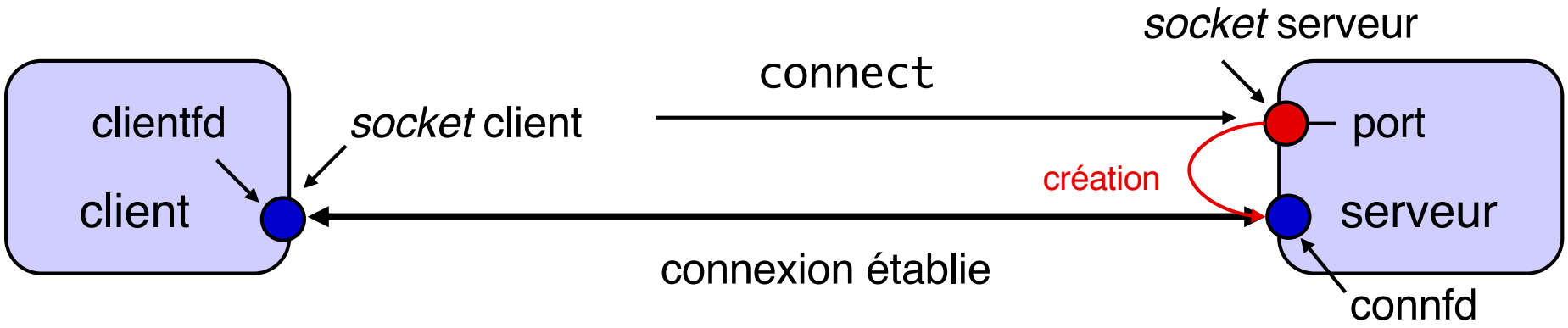
Sockets TCP côté client (3)

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

```
InetSocketAddress serverIpAddrAndPort = new InetSocketAddress(…);
sock.connect(serverIpAddrAndPort);
```

connect envoie une demande de connexion vers la *socket* serveur

(une exception *IOException* est levée en cas d'échec de la 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 (*« 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 Java

- Java fournit différents types de flux d'entrée/sorties
- À partir d'un objet Socket, on peut obtenir deux flux d'octets bruts (binaires)
  - `InputStream` (flux d'entrée) et `OutputStream` (flux de sortie)
- Au dessus de ces flux bruts, il est possible de construire des flux de plus haut niveau d'abstraction. Par exemple:
  - Des flux de caractères
  - Des flux de types Java de base (types primitifs et String)
  - Des flux d'objets Java arbitraires
- Pour plus de détails sur les flux en Java, voir la documentation du paquetage `java.io` et/ou l'un des tutoriels suivants:
  - https://docs.oracle.com/javase/tutorial/essential/io/streams.html
  - http://tutorials.jenkov.com/java-io/streams.html

Dans le contexte d'une communication par sockets, le type de flux (classes `java.io` à utiliser) dépend de la spécification du protocole applicatif (format choisi pour l'échange des données).

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 ?
  - Mais plusieurs clients peuvent envoyer des paquets à destination d'un même couple [@ 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 :
  - **[@ IP source , port source , @ 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
- Solutions
  - Messages de taille fixe
  - Messages de taille variable
    - En-tête de taille fixe indiquant le type/la taille du message complet
    - Marqueurs de fin de champs, 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 déployés sur des plateformes hétérogènes
  - 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 « boutismes » (*endianness*) différents

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 ambigüe 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.

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

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

- Il existe par convention un boutisme « réseau » standard 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.
- Remarques :
  - Cette précaution s'applique aussi aux adresses IP et numéros de port.
  - 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 » (par exemple, une image/vidéo)
  - En Java: la machine virtuelle Java est un processeur virtuel *big endian* donc les étapes de conversion entre boutisme local et boutisme réseau ne sont pas nécessaires.

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 :

```
servSock = new ServerSocket();
servSock.bind(ipAddrAndPort, backlogSize);

while (true) {
        sock = servSock.accept();

        // le code « métier » du serveur proprement dit,
        // qui implémente le dialogue avec un client
        this.handleDialogWithClient(sock);

        // lorsque ce dialogue se termine, on ferme la connexion
        // (si pas déjà fait dans la méthode précédente)
        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 :

```
clientSock= new Socket();

clientSock.connect(serverIpAddrAndPort);

// le code « métier » du client proprement dit,
// qui implémente le dialogue avec le serveur
This.handleDialogWithServer(clientSock);

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

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

Pour exécuter l'application :

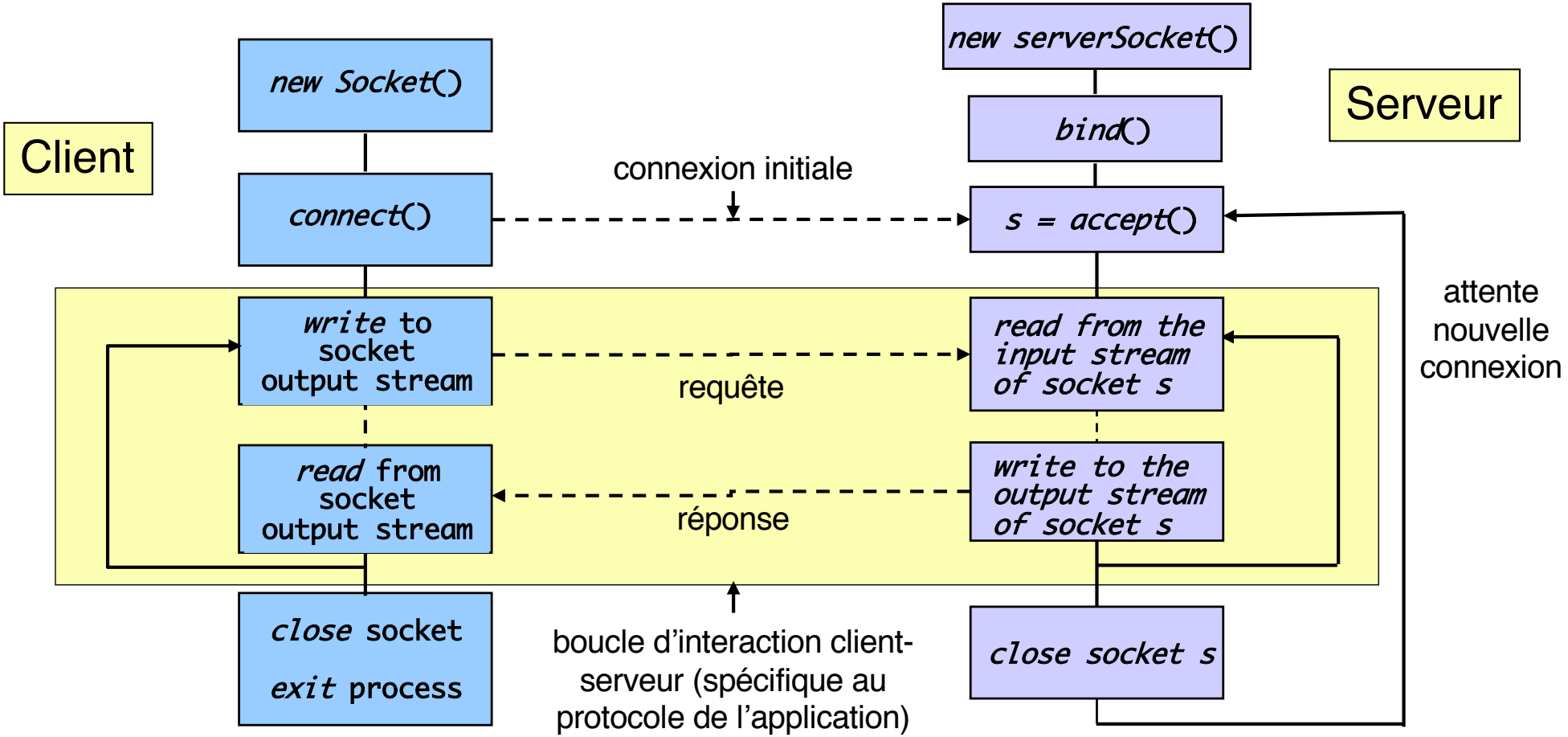
Lancer le programme serveur sur une machine, en indiquant un numéro de port >1023
(les numéros ≤1023 nécessitent les privilèges de l'administrateur (super-utilisateur) de la machine)

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 d'une application Java, l'approche la plus naturelle (utilisée dans les pages suivantes) consiste à utiliser des threads.

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.

Client-serveur en mode concurrent (2)

Il existe d'autres solutions pour concevoir un serveur concurrent :

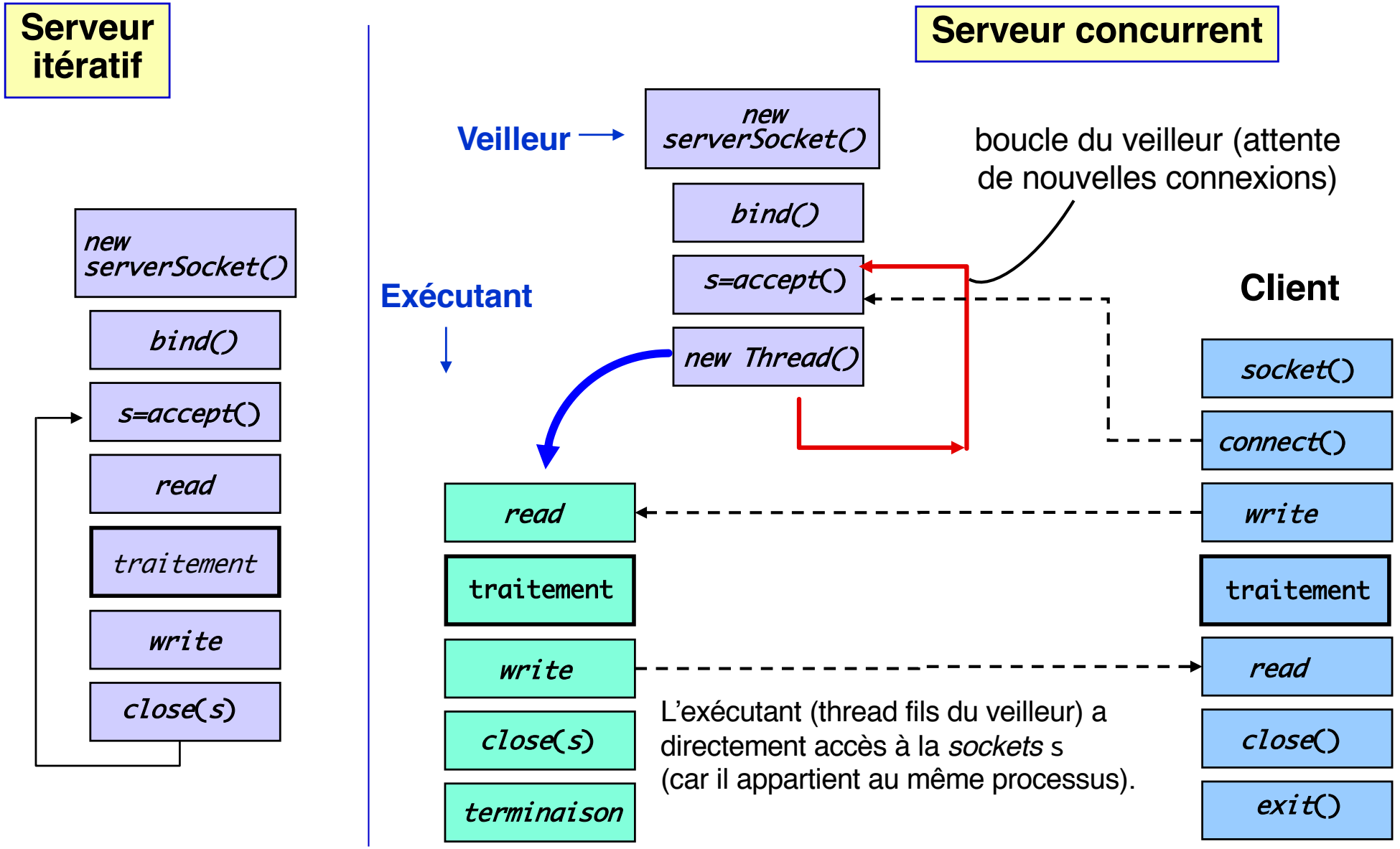
- Utilisation d'un ensemble pré-alloué de flots *exécutants* afin d'éviter les créations systématiques (cf. exercices de TP)

- 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*).

  - Ces notions ne seront pas étudiées dans ce cours. [voir par exemple la bibliothèque Java NIO]

Client-serveur en mode concurrent (3)



Client-serveur avec *sockets* : lancement

Mode d'emploi

sur le serveur

```
{imablade04$} server 4321 &
```

lance le serveur sur le port 4321

sur le client

```
{mandelbrot$} client imablade04.e.ujf-grenoble.fr 4321
```

appelle le serveur distant

Les programmes client et serveur sont indépendants, et compilés 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. 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 socketsTCP 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 Java

- Tutoriel Oracle : All about sockets
  http://docs.oracle.com/javase/tutorial/networking/sockets/index.html

- Tutoriel IBM developperWorks : Java sockets 101
  http://jmvidal.cse.sc.edu/csce590/spring02/j-sockets-ltr.pdf

- Kenneth Calvert, Michael Donahoo.TCP/IP Sockets in Java: Practical Guide for Programmers. Second edition. Morgan Kaufmann, 2008.

- Elliotte Rusty Harold. Java Network Programming. Fourth edition. O'Reilly, 2013.