Les sockets

Que se passe-t-il quand je consulte http ://www.ssi.gouv.fr dans mon navigateur Internet ?

Pourquoi les sockets ?

- Nécessité d'une abstraction par l'OS pour l'utilisateur d'un moyen de communiquer depuis la couche utilisateur sans traiter les détails laissés à la pile TCP/IP :
  - liés à la communication (paquets à former et encapsulation).
  - liés à l'utilisation d'un matériel donné (pilote de périphérique réseau).
- Sockets :
  - Objets système permettant la communication entre deux processus (sans contrainte de lien de parenté, même pas forcément sur la même machine).
- Pour du réseau, ou au sein d'un seul système UNIX (alternative aux pipes).

Interfaces réseau (Network Interface Card, NIC)

- Classiquement présentes :
  - boucle locale, ou loopback (pas de réalité physique) (lo)
  - cartes réseau Ethernet (eth)
  - cartes Wifi (wlan)
  - interfaces virtuelles (e.g.veth, br), on peut créer un réseau virtuel au sein de sa propre machine entre VMs.

- Attributs notables d'une carte réseau configurables par l'OS :
  - adresse MAC (Media Access Control)
  - adresse IP

- Consulter/configurer les interfaces avec la commande ip, ou avec ifconfig (dans /sbin).
  - Allumer et éteindre l'interface au niveau de la pile IP peut aussi se faire avec ces utilitaires.

Abstraction de la gestion du réseau par l'OS

- Une pile réseau interagit avec l'applicatif d'une part, le matériel d'autre part.

Le routage

- Qui prend les décisions de routage ? Sur quelle base ?

- Informations contenues dans une table de routage
  - sous-réseaux joignables directement par une interface,
  - pour un sous-réseau joignable, adresse de la passerelle qui peut router des paquets vers ce sous-réseau,
  - route par défaut (cas de lien avec Internet).

- Utilitaires :
  - lire la table de routage présente : netstat -rn,
  - ajouter/supprimer des routes : avec *route*
  - choix plus avancés de critères de routage : avec *ip rule*.

Autres éléments de configuration

- Il existe aussi...
  - filtrage (pare-feu) :
    - Netfilter/ iptables (Linux), packet filter pf (OpenBSD)
  - usage d'IPsec (configuration à la main ou par interaction avec un démon).

Autres éléments de configuration

- Il existe aussi...
  - filtrage (pare-feu) :
    - Netfilter/ iptables (Linux), packet filter pf (OpenBSD)
  - usage d'IPsec (configuration à la main ou par interaction avec un démon).
  - sysctl
    - Paramétrage fin de chaque couche protocolaire global au système :
      - soit avec l'utilitaire sysctl,
      - soit de manière programmatique avec l'appel système sysctl,
      - soit dans Linux avec /proc/sys (bien !).
- Attention, ce genre de choses n'est pas très portable...

Les sockets, kezako ?

- Sockets :
  - objets système permettant la communication entre deux processus (sans contrainte de lien de parenté, même pas forcément sur la même machine).
  - Parfois entre un processus et le noyau.
- Les sockets fonctionnent en général par paires.
  - Sauf cas moins classique d'utilisation pour communication entre processus et noyau.
- Descripteurs de fichiers manipulables en espace utilisateur
- API des sockets dédiée (les appels systèmes usuels risquent d'avoir des comportements différents).

Les sockets, quelle logique de fonctionnement ?

- Logique producteur-consommateur avec buffer d'émission et de réception disjoints (donc pas de risque de lire ce qu'on a écrit soi-même).

Principales familles/domaines de sockets

- AF_UNIX :
  - communication locale (au sens même système de fichiers) entre deux processus.
  - Sockets nommées.
    - Leur création engendre un inode dans le VFS, le fichier apparaissant toujours vide et avec un type spécial.
    - Pour communiquer avec une socket donnée, utiliser le nom du fichier.
    - il existe aussi des sockets nommées abstraites (nom commençant par un caractère nul (le @ vu dans netstat) qui n'auront pas d'existence réelle en tant qu'inode dans le VFS .
  - Sockets anonymes, créées par paires et connectées l'une à l'autre : pas besoin de nom.

Principales familles/domaines de sockets

- AF_INET / AF_INET6 :
  - sockets réseau IPv4 / IPv6
  - Pour communiquer avec une socket, il faut connaître une adresse IP et un port.
- AF_NETLINK, PF_KEY : exemples sockets / processus userland et noyau
- AF_PACKET : sockets réseau bas niveau en couche liaison pour sniffer par exemple (nécessite des droits étendus, notamment CAP_NET_RAW pour les créer et les utiliser) .

Modes de fonctionnement des sockets

- Mode connecté (flot, stream)
  - deux correspondants identifiés fixés une fois pour toute, connectés l'un à l'autre,
  - deux flots unidirectionnels d'octets entre les correspondants : garantit la réception ordonnée des messages émis dans une socket (sans que l'émetteur se soucie de leur taille) ou la remontée d'une erreur
  - buffers d'émission et de réception = FIFO d'octets

- Pour les sockets réseau, cela correspond au protocole TCP.
- Mode connecté généralement utilisé pour les sockets UNIX.

Modes de fonctionnement des sockets

- Mode datagramme
  - correspondant spécifié au cas par cas,
  - émission de messages de taille bornée, l'un après l'autre, possiblement adressés à des destinataires différents,
  - remontée d'une erreur si échec d'émission, pas d'accusé de réception par leur destinataire
  - buffers d'émission et de réception = FIFO de messages

- Pour les sockets réseau, cela correspond au protocole UDP.

Modes de fonctionnement des sockets

- Mode pseudo-connecté
  - possibilité de spécifier un correspondant distant unique (pas forcément avant toute utilisation de la socket) : destinataire ensuite implicite et réception uniquement des messages de ce correspondant,
- comme le mode datagramme pour le reste,
- la socket qui reçoit les messages n'a pas à être pseudo-connectée (e.g. serveur de logs)

Types de sockets

- SOCK_STREAM : sockets en mode connecté ;
- SOCK_DGRAM : sockets en mode datagramme ;
- SOCK_RAW : sockets bas niveau permettant dans une certaine mesure de contrôler les en-têtes de la couche réseau (pour une socket de la famille AF_INET par exemple). Capacité CAP_NET_RAW nécessaire.

- La transmission des données entre buffers d'émission et de réception est gérée par le noyau.

- Pour des sockets réseau, cela comprend la création et la manipulation des paquets nécessaires, ce n'est pas à faire par l'utilisateur de la socket.

- Les propriétés de la socket (famille, type, etc.) dans laquelle on écrit vont influer sur le contenu des paquets

Rôles dans la communication

- Deux rôles ont été identifiés dans la communication : client et serveur. Quelle différence ?

Rôles dans la communication

- Deux rôles ont été identifiés dans la communication : client et serveur. Quelle différence ?

- Le client parle au serveur dont il connaît l'adresse (nom, IP :port).
- Le serveur doit déjà être en écoute quand le client émet une demande.
- Le serveur n'a pas besoin de connaître avant la première demande du client l'adresse du client.
- Le rôle client ou serveur joué par une socket n'est pas une propriété intrinsèque de la socket. Ce sont des différents appels à l'API *socket* qui détermine le rôle.

Se servir de sockets… les étapes !

- Création d'une socket
- Détermination de l'IP utilisée par la socket
- Détermination du port
- Activation de la socket, attente de requête du côté serveur
- Activation de la socket pour envoi de requête du côté client
- Lecture dans une socket, écriture dans une socket (... et inversement pour une réponse).
- Fermeture d'une socket

Création

- C :
  - int sfd = socket(int domaine, int type, int protocole)
- Python :
  - socket = socket.socket(family=AF_INET, type=SOCK_STREAM, proto=0, fileno=None)

- Création d'un objet de type socket et obtention du descripteur de fichier correspondant (pour la lecture et pour l'écriture)
- Domaine : famille de socket (AF_UNIX, AF_INET, ...)
- Type : mode de la socket
- Protocole peut en général être mis à 0, dans la mesure où il n'existe dans la plupart des cas qu'un protocole supporté pour un couple domaine/type donné

Configuration

- C :
  - int ret = bind(int sfd, const struct sockaddr *local_addr, socklen_t addrlen)
- Python :
  - ret = socket.bind(address, port)

- choix de l'adresse d'écoute pour un serveur, et des coordonnées source,
- sous UNIX (mais pas sous Windows), ports privilégiés. Sous Linux, le choix d'un port privilégié nécessite la capacité CAP_NET_BIND_SERVICE.
- cas de sockets UNIX nommées : la création du ficher dans le VFS par bind nécessite les droits d'écriture dans le répertoire visé.
- cas de sockets réseau en mode datagramme : bind active la socket.

Lecture & ecriture

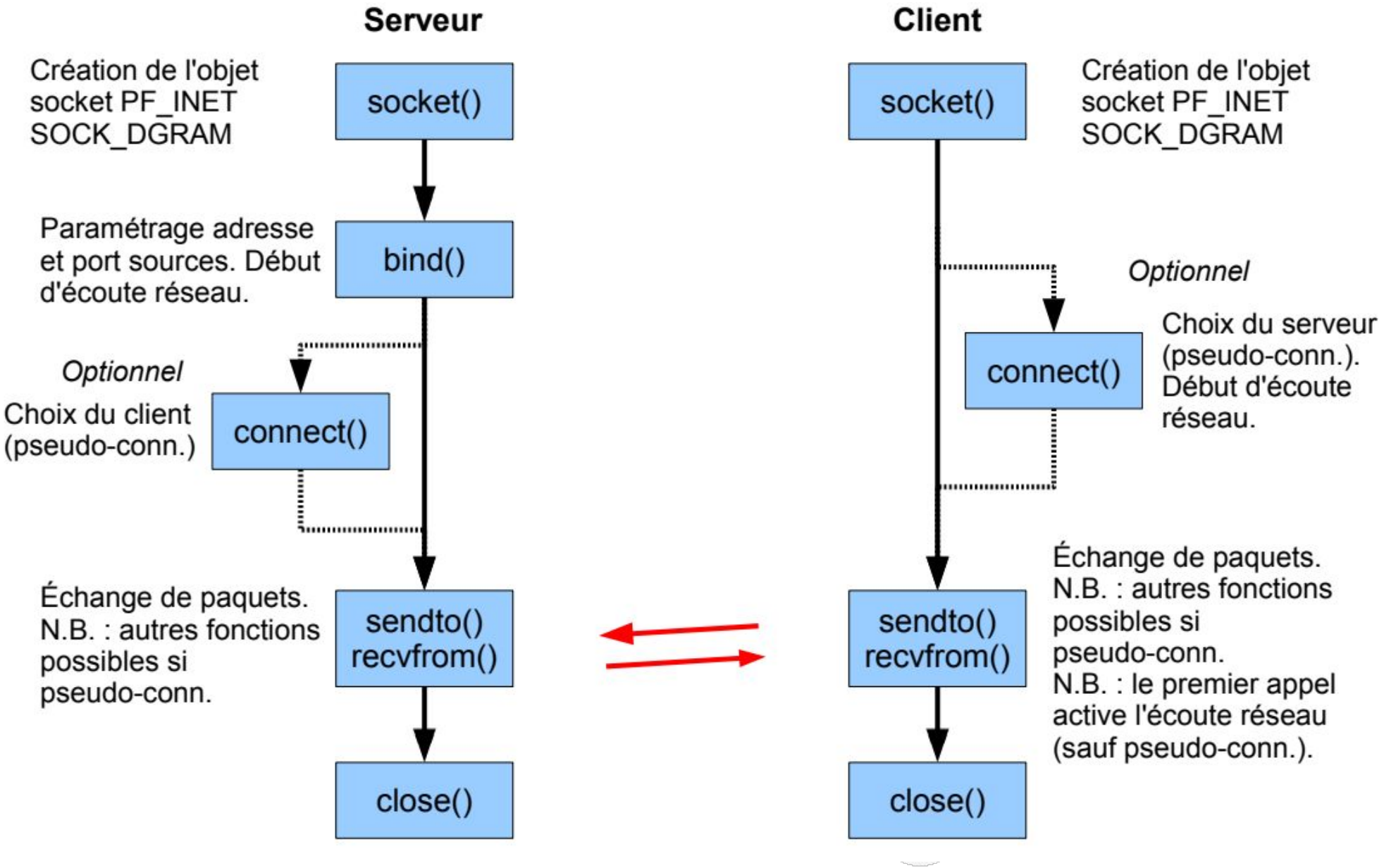
- Lecture / écriture pour sockets fonctionnant en mode non connecté :
  - Envoi de données à un destinataire identifié à travers une socket non connectée
  - C :
    - int s_len = sendto(int sfd, const void *msg, size_t len, int flags const struct sockaddr *dst, socklen_t addrlen)
  - Python :
    - bytes_sent = socket.sendto(bytes, flags, address)
- Réception de données à travers une socket non connectée et identification de leur émetteur :
  - C :
    - int r_len = recvfrom(int sfd, void *msg, size_t len, int flags, struct sockaddr *src, socklen_t *addrlen)
  - Python :
    - bytes_received = socket.recvfrom(bufsize[, flags])
- Attention : les fonctions de réception consomment exactement un message (donc adapter la taille du buffer).

Connexion

- C :
  - int ret = connect(int sfd, const struct sockaddr *remote_addr, socklen_t addrlen)
- Python :
  - ret = socket.connect(address)

- En mode datagramme (client ou serveur) :
  - Permet de passer en mode "pseudo- connecté" en désignant un interlocuteur particulier.

  - Fonctions de lecture/écriture du mode connecté utilisables, même si ce sont toujours des datagrammes (et non des flots d'octets) qui sont reçus et envoyés.


Serveur
Client
Création de l'objet socket PF_INET SOCK_DGRAM
socket()
socket()
Création de l'objet socket PF_INET SOCK_DGRAM
Paramétrage adresse et port sources. Début d'écoute réseau.
bind()
Optionnel
Choix du client (pseudo-conn.)
connect()
Optionnel
Choix du serveur (pseudo-conn.). Début d'écoute réseau.
connect()
Échange de paquets. N.B. : autres fonctions possibles si pseudo-conn.
sendto()
recvfrom()
sendto()
recvfrom()
Échange de paquets. N.B. : autres fonctions possibles si pseudo-conn. N.B. : le premier appel active l'écoute réseau (sauf pseudo-conn.).
close()
close()

Accueillir le client

- Pour le mode connecté uniquement :
  - C :
    - int ret = listen(int sfd, int taille)
  - Python :
    - ret = socket.listen([backlog])

- indique à la pile qu'il faudra accepter les demandes de connexion active la socket pour l'écoute.
- Suite à activation, envoi de paquets SYN/ACK par la pile TCP/IP en réponse aux demandes de connexions entrantes (paquets SYN) si le backlog l'autorise.

Accepter le client

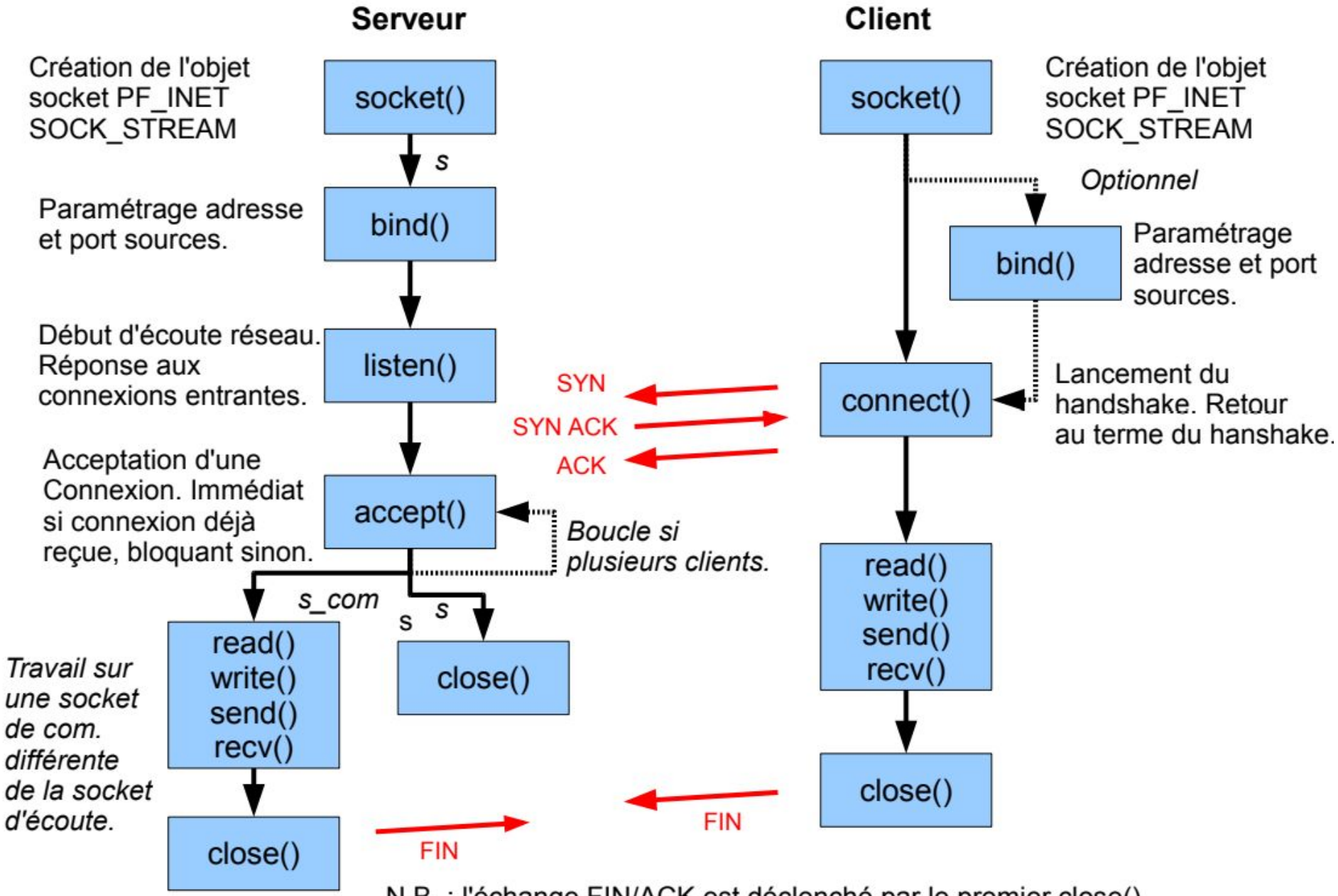
- Pour le mode connecté uniquement :
  - C :
    - int accept(int sockfd, struct sockaddr *remote_addr, socklen_t *addrlen);
  - Python :
    - (conn, address) = socket.accept()

- création d'une nouvelle socket pour parler au client, tout en conservant la socket d'écoute pour empiler de nouvelles demandes de clients.
- récupération des coordonnées dans *remote_addr ou address (avec une longueur *addrlen en C)
- accept est bloquant par défaut, pour permettre au serveur d'attendre l'arrivée du prochain client

Echanger en mode connecté

- Lecture/écriture pour une socket en mode connecté (hors socket d'écoute) ou pseudo-connecté :
  - Fonctions génériques *read* et *write* utilisables,
  - permet de rediriger les entrées / sorties (stdin, stdout, stderr) de programmes en ligne de commande dans des sockets

- Appels dédiés qui fournissent plus d'options :
  - C :
    - int s_len = send(int sfd, const void *msg, size_t len, int flags)
    - int r_len = recv(int sfd, void *msg, size_t len, int flags)
  - Python :
    - bytes_sent = socket.send(bytes[, flags])
    - bytes_received = socket.recv(bufsize[, flags])



N.B. : l'échange FIN/ACK est déclenché par le premier close().

Précisions

- Si l'accès à la socket s'effectue en mode bloquant (c'est le cas par défaut) :
  - lecture bloquante tant que l'OS n'a rien reçu
  - écriture est bloquante tant que l'OS ne peut pas prendre en compte le message à envoyer

- La gestion des erreurs est difficile.
  - Evénements locaux et distants susceptibles de modifier l'état de la socket ⇒ erreurs non-imputables à l'éventuel appel qui les a provoquées (écriture, etc.) mises en attente sur la socket.

Servir plusieurs clients

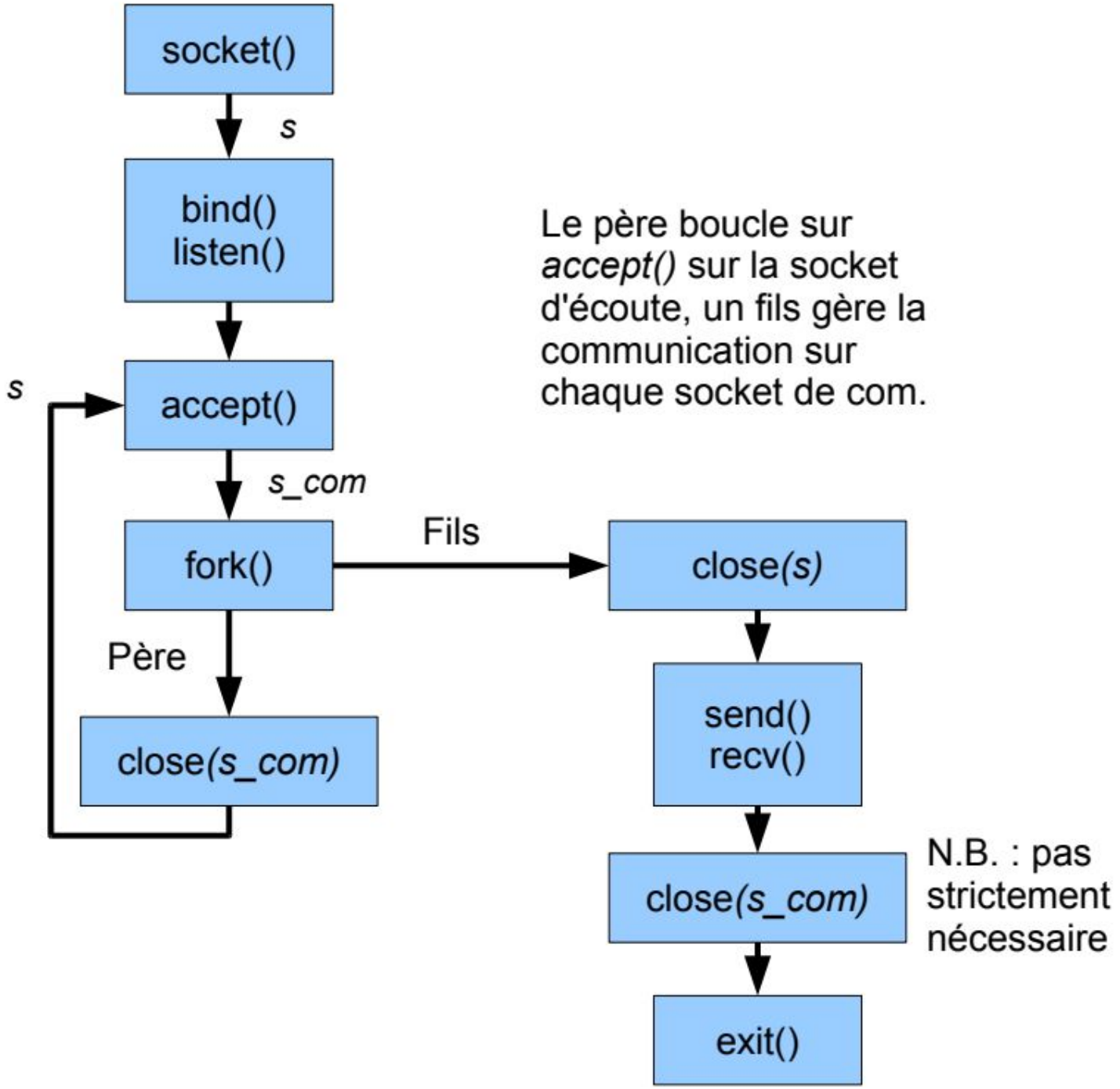
- Opérations bloquantes :
  - lecture sur des sockets
  - attente de connexions côté serveur (fonction *accept*)
  - opérations d'écriture sur une ressource potentiellement saturée.
- Comment peut-on optimiser le service des clients ?

Servir plusieurs clients

- plusieurs fils d'exécution (threads ou processus), à raison d'un par ressource bloquante ;
- le passage en mode non-bloquant des descripteurs de fichier (ajout du flag O_NONBLOCK par *fcntl*) et attente active (ou signaux)
- en UNIX, *select*, qui permet de surveiller la disponibilité de plusieurs descripteurs en même temps.

Socket ou Pipe ?

- Bidirectionnel et buffers séparés, donc gestion des accès concurrents simplifiée.
- Pas de contraintes de liens entre processus communiquant (parenté, utilisateur), tout en autorisant un contrôle des clients qui s’appuie sur le VFS.
- Transmission de descripteurs de fichiers et contrôle de ’credentials’.


socket()
s
bind()
listen()
s
accept()
s_com
fork()
Fils
close(s)
Père
close(s_com)
send()
recv()
close(s_com)
exit()
Le père boucle sur accept() sur la socket d'écoute, un fils gère la communication sur chaque socket de com.
N.B. : pas strictement nécessaire

Des points à aborder…

- Correction du TD Shell
- Le MBR

Point Sécurité !

- Inetd
  - Idée :
    - Pour que les services n'aient pas à gérer les accès réseau ou plusieurs clients, leur garantir qu'ils peuvent interagir avec un client donné en lisant sur stdin et en écrivant sur stdout.
  - Mise en œuvre :
    - démon s'exécutant en root
    - parcours d'un fichier de configuration : ports + binaire à exécuter si connection / paquet reçu pour création/activation de toutes les sockets nécessaires
    - à réception demande connection : fork, redirection de stdin/stdout sur la socket et exécution du binaire
    - à réception paquet UDP : création de socket puis connect, fork, redirection de stdin/stdout sur la socket et exécution du binaire
- Quels problèmes ?

Point Sécurité !

- Un programme traitant un service ne sert qu'un client (pas d'optimisation possible).

- uid et gid root pour les fils puis les binaires exécutés peuvent à leur tour en bénéficier... Quelles sont les conséquences d'une vulnérabilité dans le code du binaire exécuté ?

Solutions : ? ? ?

Point Sécurité !

- Pistes de durcissement possibles :
  - Démarche moderne : un serveur par service.
  - Utiliser fork plutôt que select ou des threads permet d'avoir un processus par client. La compromission d'un client est potentiellement restreinte à son espace mémoire.
  - Limitation du nombre de connections entrantes (sysctl, rlimit).

Point Sécurité !

- Pistes de durcissement possibles :
  - Accorder les privilèges root est excessif. Seul bind nécessite CAP_NET_BIND_SERVICE si ouverture de port privilégié.
  - Premier pas, descente de privilèges. Plutôt dans les fils, pas très intelligent pour le père (nécessité de nouveaux appels de bind).
  - Créer une sandbox pour les fils : chroot ?
    - Plutôt seccomp-BPF (filtrage appels système + arguments) !
  - Faire en sorte que seule CAP_NET_BIND_SERVICE soit laissée au processus en ayant besoin.

Un peu de pratique ?