Cours Système n°7 : les *sockets*

Éric Gaudefroy

École Hexagone
Cursus Cyberdéfense - M1

1. Gestion du réseau par l'OS

L'étude de l'organisation des différentes couches protocolaires fait l'objet du cours Réseau. En particulier, le fonctionnement théorique des protocoles Ethernet, ARP, IP, ICMP, TCP et UDP (et leurs interactions) est supposé connu dans le cadre du présent cours. Celui-ci se concentre sur la mise en œuvre de ces différentes couches protocolaires par le système d'exploitation et la couche utilisateur. Par un abus de langage classique, le terme de « pile TCP/IP » sera utilisé par la suite pour désigner l'ensemble de la pile réseau d'un système d'exploitation, tous protocoles confondus. Par ailleurs, dans la suite de ce document, l'utilisation de la version 4 du protocole IP (IPv4, par opposition à IPv6) sera supposée en l'absence de précisions contraires.

1.1. Interfaces réseau

Contrairement aux autres périphériques, les interfaces réseau ne sont pas gérées comme des fichiers de type périphérique (l'existence de fichiers */dev/<intf>* est exceptionnelle et ne correspond pas à un usage standard de l'interface concernée). Elles sont utilisées et configurées au travers de l'API des *sockets* et de la gestion des tables de routage du système. Dans la pratique, on peut se représenter la pile TCP/IP de la machine locale comme une entité unique, intégrée à la couche noyau, dont les points d'entrée et de sortie sont :

- par le haut, les *sockets* ouvertes par les applications, objets abstraits du système d'exploitation similaires à des descripteurs de fichiers, dans lesquels les programmes de la couche utilisateur peuvent lire et écrire des données sans se préoccuper des entêtes protocolaires ou du pilotage des interfaces réseau (de la même manière qu'un programme écrivant dans un descripteur de fichier local n'a pas à se préoccuper de l'organisation physique des données sur le disque, ni du pilotage de ce dernier) ;
- par le bas, les interfaces réseau de la machine (qui gèrent les couches liaison et physique s'il y en a).

La correspondance entre *sockets* et interfaces réseau est déterminée par les correspondances entre adresses et interfaces au niveau de la table de routage. Les données écrites sur une *socket* seront ainsi mises en forme (ajout des différents en-têtes) par la pile réseau et transmises pour émission à l'interface réseau appropriée. Inversement, les paquets reçus sur une interface réseau seront interprétés au niveau protocolaire (extraction et interprétation des en-têtes) par la pile réseau, qui en déduira la *socket* utilisateur à laquelle remonter les données de ces paquets.
Le nom de chaque interface est relatif à son type. Les interfaces IPv4 les plus courantes sont :

- la boucle locale : **lo** (ou lo0), interface « virtuelle » utilisée pour les communications IP locales au système, qui correspond aux adresses réservées IPv4 127.0.0.1/8 et à l'adresse IPv6

(unique) : :1 ; tout paquet émis par la boucle locale est immédiatement reçu par cette même interface ;

- les interfaces Ethernet au sens large : eth<n> ou wlan<n> (Wifi) sous Linux, ou <Ethernet Connection>/<Wireless-AC> sous Windows, qui correspondent aux interfaces physiques reliées à des réseaux locaux Ethernet ou assimilés ;
- les autres interfaces physiques : slip<n> (IP sur un port série), plip<n> (IP sur un port parallèle), ppp<n> (liaison point-à-point via un modem), etc...
- des interfaces virtuelles spécifiques : vmnet<n> (hub virtuel VMWare), tun<n> (interface virtuelle tun/tap),etc...

Les paramètres de configuration de ces interfaces sont accessibles et modifiables en ligne de commande par des commandes *ifconfig*. Dans tous les cas, seul root peut modifier les propriétés d'une interface (sous Linux, ce privilège correspond plus spécifiquement à la capacité CAP_NET_ADMIN). Les commandes **ifconfig** les plus classiques sont les suivantes (voir *man ifconfig* pour plus d'informations) :

- **ifconfig -a** : afficher toutes les interfaces et leurs propriétés ;
- **ifconfig <intf> <IPaddr>** : modifier l'adresse IPv4 d'une interface ;
- **ifconfig <intf> hw ether <MACaddr>** : modifier l'adresse MAC d'une interface Ethernet ;
- **ifconfig <intf> up/down** : activer/désactiver une interface ;
- **ifconfig <intf> :<p> <IPaddr>** : ajouter une adresse supplémentaire à une interface, sous Linux uniquement.

On pourra aussi se référer à **man arp** pour la gestion des tables ARP par l'OS (correspondance entre adresses IP et adresses MAC pour les machines directement joignables par une interface Ethernet). Sous Windows, La commande **ipconfig**, ou **Get-NetIPConfiguration**, permet d'afficher les valeurs de la configuration actuelle de votre réseau TCP/IP, et d'actualiser au besoin les paramètres DHCP et DNS.

1.2. Table de routage

Les applications exploitent les interfaces pour réaliser des communications, via le réseau, avec des machines distantes. Les couches d'abstraction protocolaire rendent transparent le choix de l'interface (sauf pour les rares applications qui souhaitent un accès direct aux couches inférieures, voir plus bas) : c'est le système qui choisit une interface en fonction de la destination affichée. Ce choix repose sur la table de routage de l'OS.
La table de routage contient les informations suivantes :

- pour chaque interface, l'ensemble des adresses IP directement joignables (sous-réseau) par cette interface ;
- pour chaque sous-réseau distant connu, l'adresse IP de la passerelle directement joignable qui est capable de router un datagramme vers ce sous-réseau (ainsi que le nom de l'interface à utiliser pour atteindre cette passerelle) ;
- si la machine est reliée à un WAN (Internet), on ajoute généralement une route par défaut (passerelle à utiliser pour router le trafic vers les adresses ne bénéficiant pas d'un traitement particulier).

La manipulation de la table de routage s'effectue en ligne de commande par la commande route :

- **route -n** ou **route show** (Linux), **route PRINT** ou **netstat -rn** (compatibles Windows et plus génériques) : afficher les routes actives ;
- **route add -net <IPnet> (gw) <IPpasserelle>** : ajouter une route vers un sous-réseau particulier ;
- **route del -net <IPnet>** : supprimer une route (ajouter les paramètres de la route s'il y a ambiguïté) ;
- voir man route pour plus de détails.

On peut utiliser -net 0.0.0.0/0 ou default pour manipuler la route par défaut. La route relative au sous-réseau directement accessible par une interface donnée est généralement définie directement par **ifconfig**.
Nota : Sous Linux, de nombreux paramètres réseau (statistiques, tables diverses : arp, routage, etc.) sont accessibles en lecture via les fichiers de */proc/net*.

1.3. Autres éléments de configuration

Au delà de la définition des interfaces et tables de routage, différents paramètres complémentaires définissant le comportement de la pile réseau peuvent également être configurés depuis la couche utilisateur.

1.3.1. Filtrage

La plupart des systèmes disposent de fonctions natives de filtrage au niveau de la couche IP. Les possibilités offertes ainsi que le mode de configuration diffèrent d'un OS à l'autre :

- Linux : Netfilter/iptables
- xBSD : IPfilter (ipf)
- Windows : netsh firewall
- etc...

Certains de ces filtres seront étudiés plus en détail dans des cours ultérieurs. Globalement, la mise en place de ces filtres visera à restreindre les flux autorisés à traverser la couche IP (rejet silencieux ou non des paquets non autorisés).

1.3.2. IPsec

La plupart des systèmes disposent également de fonctions natives IPsec, qui permettent de réaliser des opérations cryptographiques sur certains paquets émis, correspondant typiquement à un chiffrement des données pour en assurer la confidentialité et/ou à une authentification des données et en-têtes protocolaires. La gestion des traitements IPsec s'effectue au niveau de la couche IP du noyau. Leur paramétrage, qui inclut la définition de la politique de sécurité applicable aux différents types de flux, peut être réalisé :

- manuellement, via des outils en ligne de commande, et/ou
- par un démon en couche utilisateur implémentant le protocole IKE, qui réalise la négociation dynamique des tunnels et les échanges de clés.

Ces points seront étudiés dans des cours ultérieurs.

1.3.3. Sysctl

Chaque couche protocolaire se voit enfin associer des paramètres fins permettant de contrôler son comportement à l'échelle du système (par opposition à d'autres types de paramètres fins, configurables au cas par cas, c'est-à-dire *socket* par *socket*, cf. **setsockopt** plus bas). Le détail de ces paramètres diffère d'un système à l'autre, voire d'une version à l'autre d'un même système.
Comme la plupart des paramètres globaux du noyau de l'OS qui sont configurables dynamiquement, ils sont accessibles par l'API des sysctl. L'OS dispose en effet d'une arborescence de paramètres dynamiques organisée en catégories, sous-catégories, etc. On désigne communément ce type d'arborescence sous le nom de MIB (Management Information Base). Selon la nature des éléments de la MIB et les privilèges de l'appelant, il est possible d'y réaliser des accès en lecture et/ou en écriture. En ligne de commande, la syntaxe est la suivante :

- **sysctl -a** : afficher la MIB sysctl ;
- **sysctl <rep>.<srep>...<elem>** : lire un élément ;
- **sysctl -w <rep>.<srep>...<elem>=<valeur>** : modifier un élément.

Sous Linux, les principaux répertoires et éléments sont :

- **net.*** : paramètres réseau ;
- **net.ipv4.*** : paramètres IPv4 et couches supérieures (BSD : net.inet) ;
- **net.ipv4.conf.*** : paramètres de gestion des interfaces réseau,
- **net.ipv4.tcp*** : paramètres TCP (BSD : net.inet.tcp.*) ;
- **net.ipv4.ip_forward** : activation du routage entre les interfaces, i.e. choix entre le rôle d'hôte terminal et celui de routeur (BSD : net.inet.ip.forwarding) ;
- **net.core.*** : paramètres réseau génériques (sockets) ;
- **kernel.*** : paramètres génériques (nom de l'OS, nom de la machine, etc.) ;
- etc...

Les MIBs différant d'un OS à l'autre, l'utilisation de sysctl n'est pas portable. Un paramètre existant sous un OS peut ne pas exister, ou exister sous un nom différent (par exemple **forwarding** sous BSD, contre **ip_forward** sous Linux), sur un autre OS. S'agissant de paramètres fins dont l'interprétation exacte est par construction propre à chaque OS, il n'existe pas de manière générale d'alternative portable.

2. Les sockets : généralités

L'API des sockets, permettant à des applications en couche utilisateur d'utiliser de manière abstraite la pile TCP/IP du système, a été définie initialement par les systèmes BSD. Elle s'est rapidement imposée sur la quasi-totalité des systèmes. La variante proposée par System V, TLI (Transport-Level Interface), n'aura jamais eu le même succès.

2.1. Typologie des sockets

Les sockets sont des objets système permettant à des processus de communiquer, qu'ils soient situés ou non sur la même machine. Ces objets se manipulent, en couche utilisateur, sous forme de descripteurs de fichiers. Les sockets constituent une abstraction commune pour un ensemble de mécanismes et de protocoles de communication, aussi bien locaux que distants (à travers le réseau). Ainsi, l'API des sockets définit un certain nombre de fonctions spécifiques, appelables sur tout type

de socket, mais avec des effets différents selon le mode de transmission (cf. plus bas) et le protocole associés à la socket considérée. De même, les fonctions supportées de manière générique par les descripteurs de fichiers (typiquement au moins **read** et **write**) auront des effets différents, voire ne seront pas du tout supportées, selon le type de socket.
Les données (ou messages) transitant par les sockets sont gérées selon la logique producteur/consommateur, la réception et l'émission possédant des buffers séparés. Un message émis sur une socket sera reçu in fine (s'il n'est pas perdu) par une autre socket, choisie selon les paramètres d'émission et/ou la configuration de la socket d'émission.
Les sockets fonctionnent donc en général au moins par paires, l'appariement en l'absence d'héritage commun se réalisant grâce au nommage (VFS pour les sockets fichiers, adresse/protocole/port pour les sockets réseau). La seule exception à cette règle concerne les familles de sockets spécifiques (hors programme) dédiées à la communication entre la couche utilisateur et le noyau, pour lesquelles une unique socket est réellement mise en œuvre à la fois en couche utilisateur.

2.1.1. Familles de protocoles (domaines)

Concrètement, une paire de sockets peut relier :

- deux processus d'une même machine (voire deux threads d'un même processus, mais il y a des moyens de communication mieux adaptés à ce type de situation) ;
- deux processus tournant sur des machines distinctes (communication via le réseau), pouvant utiliser des processeurs et des OS différents (utilisation de protocoles IP standards, donc d'une convention commune côté réseau) ;
- (hors programme) un processus et le noyau, afin de réaliser des opérations de configuration spécifiques (IPsec, routage, etc.).

A ces différents usages correspondent différentes « familles » de sockets, également dénomées « domaines ». La famille d'une socket lui est affectée lors de sa création, et ne peut pas être modifiée ensuite. Les principales familles de sockets sont (liste non exhaustive) :

- PF_UNIX (ou PF_LOCAL, ou PF_FILE) : les sockets locales ;
- PF_INET : les sockets réseau IPv4 (peuvent aussi travailler en local grâce à l'interface lo) ;
- PF_INET6 : les sockets réseau IPv6 (hors programme) ;
- PF_PACKET : (hors programme) les sockets réseau bas niveau en couche liaison (typiquement Ethernet) utiles pour les sniffeurs et les forgeurs de paquets ; spécifiques à Linux et hors programme), réservées à root (capacité CAP_NET_RAW) ;
- PF_NETLINK, PF_KEY : (hors programme) exemples de sockets utilisées pour la communication entre un processus en couche utilisateur et le noyau.

Du point de vue local du système d'exploitation, une socket réseau relie en fait un processus à la pile TCP/IP du noyau. Le processus est producteur sur le buffer d'émission, et la pile consomme au fur et à mesure qu'elle émet des paquets sur le réseau (ou, selon le protocole, lorsqu'elle a reçu un acquittement). Inversement, la pile produit sur le buffer de réception, sur lequel le processus se positionne en tant que consommateur.
La commande **netstat -an** permet de lister les sockets actives du système. Sous Linux, l'option -p donne accès à l'identité du processus attaché à chaque socket. On peut aussi restreindre la liste aux seules sockets d'une famille donnée, par exemple IP en ajoutant –inet sous Linux ou -f inet sous BSD.

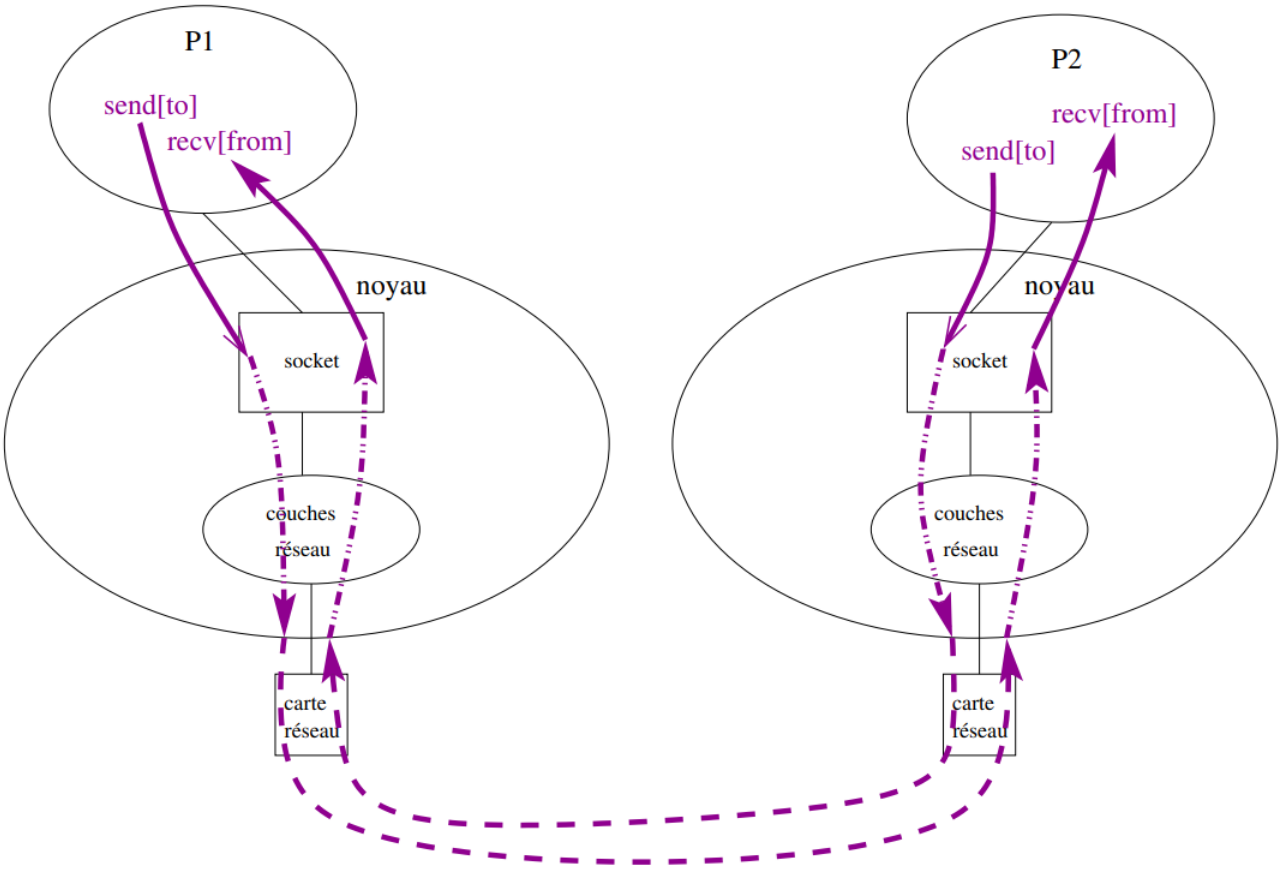

FIGURE 1 – Couche d'abstraction des sockets dans un cadre d'emploi réseau.

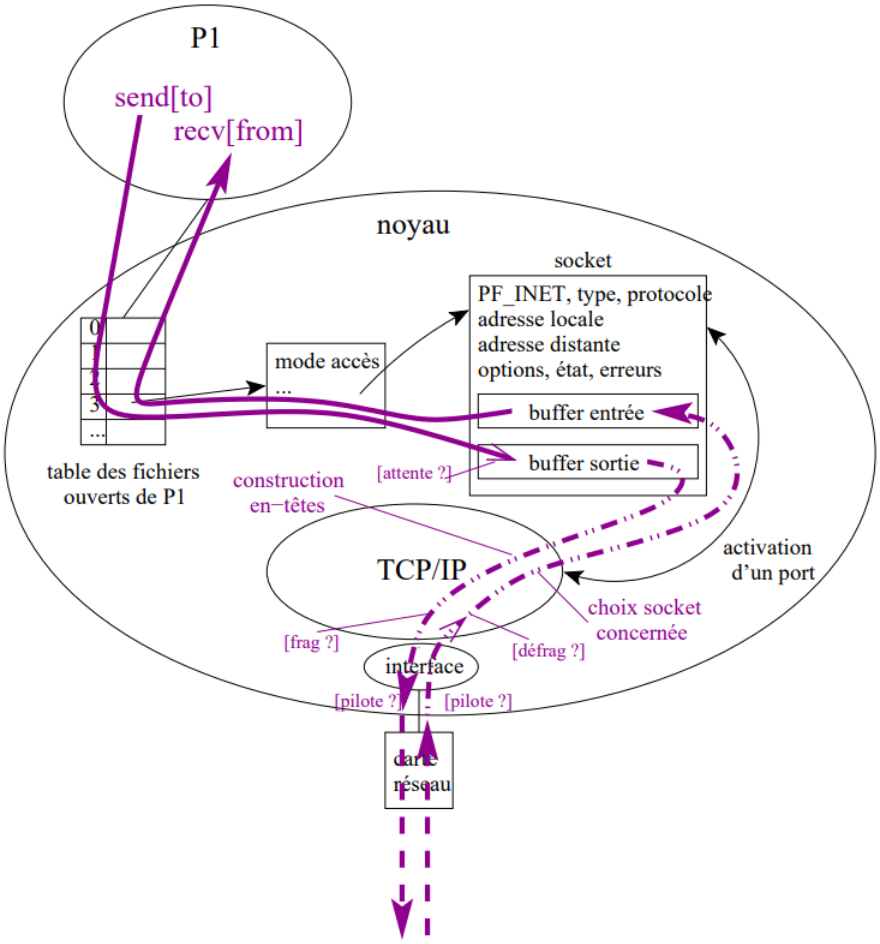

FIGURE 2 – Couche d'abstraction des sockets : point de vue local pour une socket réseau TCP.

2.1.2. Modes de fonctionnement

On distingue deux modes principaux de fonctionnement pour les sockets :

- le mode **flot** ou **connecté** (ou encore stream en anglais) : relie deux correspondants bien identifiés et sert à établir deux flots unidirectionnels d'octets entre les deux sockets associées à ces correspondants ; fonctionnellement, on peut se représenter cette paire de sockets comme deux tubes anonymes (un par sens de transmission) :

  s1 = tube1[0], tube2[1] (lecture, écriture) ;

  s2 = tube2[0], tube1[1] (lecture, écriture). mais ici, les extrémités de chaque « tube » peuvent être situées de part et d'autre du réseau ; les buffers d'entrées/sorties associés à une telle socket sont les FIFO des octets reçus depuis, ou à émettre vers, la socket distante ; dans le cas de sockets réseau, ce mode correspond en général à l'utilisation du protocole TCP ; il est également le mode le plus souvent utilisé dans le cadre d'une communication purement locale (sockets UNIX) ;
- le mode **datagramme** : permet l'envoi à la demande de messages de taille bornée en spécifiant au cas par cas l'identité du correspondant (socket distante) ; on peut ici se représenter un ensemble de sockets s'échangeant des messages isolés en changeant régulièrement d'interlocuteur ; les buffers d'entrées/sorties associés à une telle socket sont des FIFO de messages

reçus ou à émettre auxquels sont attachées les coordonnées de la socket distante concernée ; ce mode est par exemple adapté à l'utilisation du protocole UDP dans le cas de sockets réseau.

Il existe une variante du mode **datagramme**, dite mode « pseudo-connecté », dans laquelle un interlocuteur particulier a été désigné comme extrémité distante associée à la socket. La socket ne reçoit alors que les datagrammes provenant de cet interlocuteur et le considère comme destinataire implicite de ceux qu'elle émet.

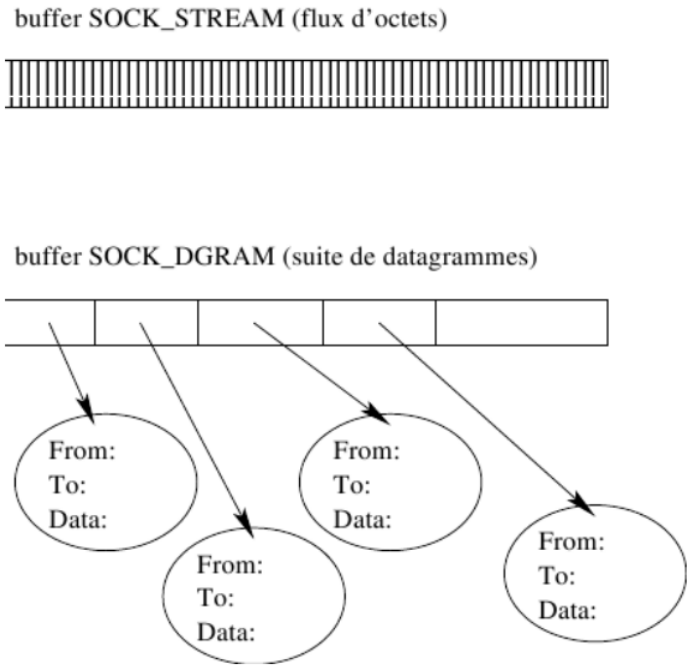

FIGURE 3 – Principe des buffers d'entrée / sortie en mode flot ou datagramme.

Le mode de fonctionnement d'une socket lui est affecté lors de sa création, et ne peut pas être modifié ensuite. Ce mode correspond au paramètre « type » de l'appel socket utilisé pour créer la socket (voir plus bas). Les principaux types de socket sont (liste non exhaustive) :

- SOCK_STREAM : sockets en mode connecté (modèle = TCP), qui peuvent inclure une notion de transmission hors de bande (délicat, hors programme) ;
- SOCK_DGRAM : sockets en mode datagramme (modèle = UDP), pour les envois ponctuels de messages ;
- SOCK_RAW : sockets bas niveau permettant dans une certaine mesure de contrôler les entêtes de la couche réseau (pour une socket de la famille PF_INET par exemple), par ailleurs assez proches des sockets en mode datagramme, et réservées à root (il faut disposer de la capacité CAP_NET_RAW pour pouvoir créer une telle socket sous Linux).

On notera bien que le protocole réseau sous-jacent est totalement abstrait au niveau de l'interface des sockets. Ainsi, pour créer une socket TCP/IP, il faut et il suffit de créer une socket de la famille PF_INET et de mode SOCK_STREAM : la mise en œuvre de TCP sera automatiquement choisie par la pile réseau comme protocole permettant d'assurer les propriétés abstraites attendues du mode connecté.

2.1.3. Rôles au sein de la communication

La mise en place des communications conduit généralement à distinguer deux rôles :

- le **serveur**, associé à une adresse connue et lancé en premier, qui attend d'être sollicité par un client (le plus souvent en vue de lui fournir un service) ;
- le **client** qui, lorsqu'il est prêt, contacte de façon asynchrone un serveur à l'adresse convenue (le plus souvent pour lui poser une question et récupérer sa réponse).

Cette répartition des rôles peut être plus floue dans certains types d'applications (par exemple dans une logique « pair-à-pair »), avec par exemple un changement au cours du temps de la répartition des rôles client et serveur. Dans tous les cas, le rôle associé à une socket n'est pas, au contraire de son domaine et de son mode de fonctionnement, une propriété intrinsèque de la socket, mais plutôt le résultat des appels qui sont réalisés avec cette socket.

2.1.4. Opérations sur les sockets

La mise en œuvre de sockets nécessite un enchaînement d'opérations, qui dépend du mode et du rôle (client ou serveur) de la socket considérée. Ces appels, qui sont détaillés dans la section suivante et résumés dans les figures 6 et 7 en fin de document, consistent principalement à :

- Créer un objet socket ;
- Définir au besoin les paramètres locaux de la socket (adresse et port sources, typiquement) - obligatoire pour un serveur, optionnel pour un client ;
- Définir au besoin les paramètres distants de la socket (adresse et port destination) - obligatoire pour un client, optionnel pour un serveur en mode datagramme ;
- Activer l'écoute sur la socket (serveur uniquement) et accepter une connexion (serveur en mode connecté uniquement) ;
- Ou initier une connexion (client en mode connecté uniquement) ;
- Envoyer et / ou recevoir des données ;
- Fermer la connexion (en mode connecté) et désallouer les ressources locales.

On notera bien que les étapes "génériques" décrites ici ne correspondent pas une à une à autant d'appels de fonctions : par exemple la définition des paramètres locaux et l'activation de l'écoute pourront être réalisées en un seul appel (mode non connecté) ou en deux (mode connecté).

2.2. Principes d'adressage

2.2.1. Notion d'adresse

L'envoi de messages via des sockets nécessite en général de spécifier l'adresse de la socket distante destinataire des messages. Le type de cette adresse dépend de la famille de socket considérée :

- Pour une socket réseau, il s'agit naturellement de l'adresse IP (ou IPv6) de la machine distante, complété du port correspondant au service à joindre lorsque le protocole intègre une notion de port (TCP, UDP, mais pas ICMP par exemple) ;
- Pour une socket locale UNIX, il s'agit d'un nom de fichier du VFS, sauf dans le cas d'un nom abstrait (voir remarque dans la sous-section suivante). Ce fichier apparaît bien au sein du VFS, avec un type spécial (type 's' dans la sortie d'une commande **ls -l**).

Contrairement à une connexion réseau, une connexion par socket UNIX ne fait pas nécessairement intervenir d'adresse. Il est en effet possible (grâce à la fonction **socketpair** décrite plus bas), de créer directement une paire de sockets connectées entre elles, sans leur associer d'adresse. De telles sockets sont normalement appelées sockets anonymes, par opposition aux sockets nommées associées à une adresse (qu'elle soit abstraite ou liée à un fichier du VFS).
Selon le protocole, la communication peut également faire intervenir une notion d'adresse locale (adresse source). C'est le cas par exemple avec des sockets réseau. La notion d'adresse source n'a en revanche pas de sens dans le cas de sockets UNIX.
Même dans le cas où elle a un sens, il n'est pas toujours nécessaire de spécifier explicitement une adresse locale. Ainsi, si l'adresse locale d'une socket IP n'est pas explicitement définie, l'OS lui attribue un "port temporaire" dès que nécessaire (connexion vers une adresse distante ou premier envoi d'un datagramme), le port choisi dépendant de sa configuration et des disponibilités, associé à une adresse IP INADDR_ANY correspondant à l'ensemble des adresses du poste.

2.2.2. Représentation des adresses

Il existe un type générique pour la manipulation d'adresses de sockets : *struct sockaddr* type générique. Concrètement, il s'agit d'un type « abstrait » qui n'est utilisé que pour le prototypage des fonctions sur les sockets.

3. Mise en oeuvre des sockets

Apparues en 1971 dans les laboratoires de l'Université de Berkeley, le concept de socket a été développé pour améliorer Arpanet. Les recherches amènent à la création en 1983 d'une API dédiée aux communications dans le système Berkeley Software Distribution (BSD).
Avec l'explosion des communications réseaux, l'adaptabilité des sockets face aux enjeux de passage à l'échelle leur permet de supporter la plupart des nouvelles applications. Les communications « Client / Serveur » reposent sur les sockets.
Aujourd'hui, malgré une nette amélioration des couches supérieures du modèle OSI (5-6-7) et des protocôles, l'API réseau bas-niveau demeure la même : la socket « API for Internet sockets », ou plus communément BSD sockets. L'architecture la plus commune est le « Client / Serveur ». Un pôle de communication est un serveur qui attend et l'autre un client qui sollicite le serveur.
Sous Python, il est nécessaire d'importer le module BSD sockets :

```
import socket
```

3.1. Création de sockets

La réservation d'une ressource de type socket s'effectue en C par la fonction :

```
int sfd = socket(int domaine, int type, int protocole)
```

Création d'un objet de type socket et obtention du descripteur de fichier correspondant (pour la lecture et pour l'écriture).
L'argument « domaine » correspond à la famille de socket telle que décrite dans la section précédente (PF_UNIX, PF_INET, ...), tandis que le « type » définit le mode connecté ou non de la socket (SOCK_STREAM ou SOCK_DGRAM). Le 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, par exemple TCP pour PF_INET et SOCK_STREAM (exception notable : les « sockets ICMP » qui sont des sockets IP en mode raw mais nécessitent de spécifier ICMP comme protocole).

Exemple de fonctionnement de la commande **ping** : le « client » ouvre une socket socket(PF_INET, SOCK_RAW, IPPROTO_ICMP) pour envoyer un ICMP ECHO REQUEST et attendre un ICMP ECHO REPLY. Il doit donc être root. Dans la pratique, le programme **ping** est installé avec un bit 's' pour permettre à un utilisateur non privilégié de s'en servir. Il n'y a pas de « serveur » **ping**, c'est la couche IP (= le noyau) distante qui répond à la requête si sa configuration (sysctl, filtrage) l'y autorise.

Des exemples d'implémentation de sockets en C sont proposées en annexe au présent document.

En Python, la fonction de création d'un objet « socket » est :

```
socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Les paramètres suivants sont utilisés :

- type de socket : socket.SOCK_STREAM
- protocole par défaut : AF_INET (TCP)

Le Transmission Control Protocol (TCP) a pour avantages :

- robuste : les paquets perdus dans le réseau sont détéctés et retransmis par l'envoyeur ;
- ordonné : les datas sont lues à destination dans le même ordre que leur écriture chez l'envoyeur.
- protocole par défaut : AF_INET (TCP)

Par opposition, les sockets User Datagram Protocol (UDP) créées avec socket.SOCK_DGRAM ne sont pas robustes aux erreurs de transmission et ne peuvent se baser sur l'ordre des données reçues.

Pourquoi est-ce important ?

Car les réseaux utilisent un système de délivrance des messages en *best effort* : il n'y a a pas de garantie qu'un message vous soit parvenu ou qu'il soit parti vers un destinataire. Les réseaux sont constitués d'équipements, comme des switchs et des routeurs, qui ont une mémoire et une bande passante finies. Ils utilisent des buffers (CPU, mémoire, bus, interfaces) qui génèrent des pertes et temps de traitement.

C'est donc au protocole de gérer les pertes et mélanges de paquets potentiels. TCP rempli ce rôle.

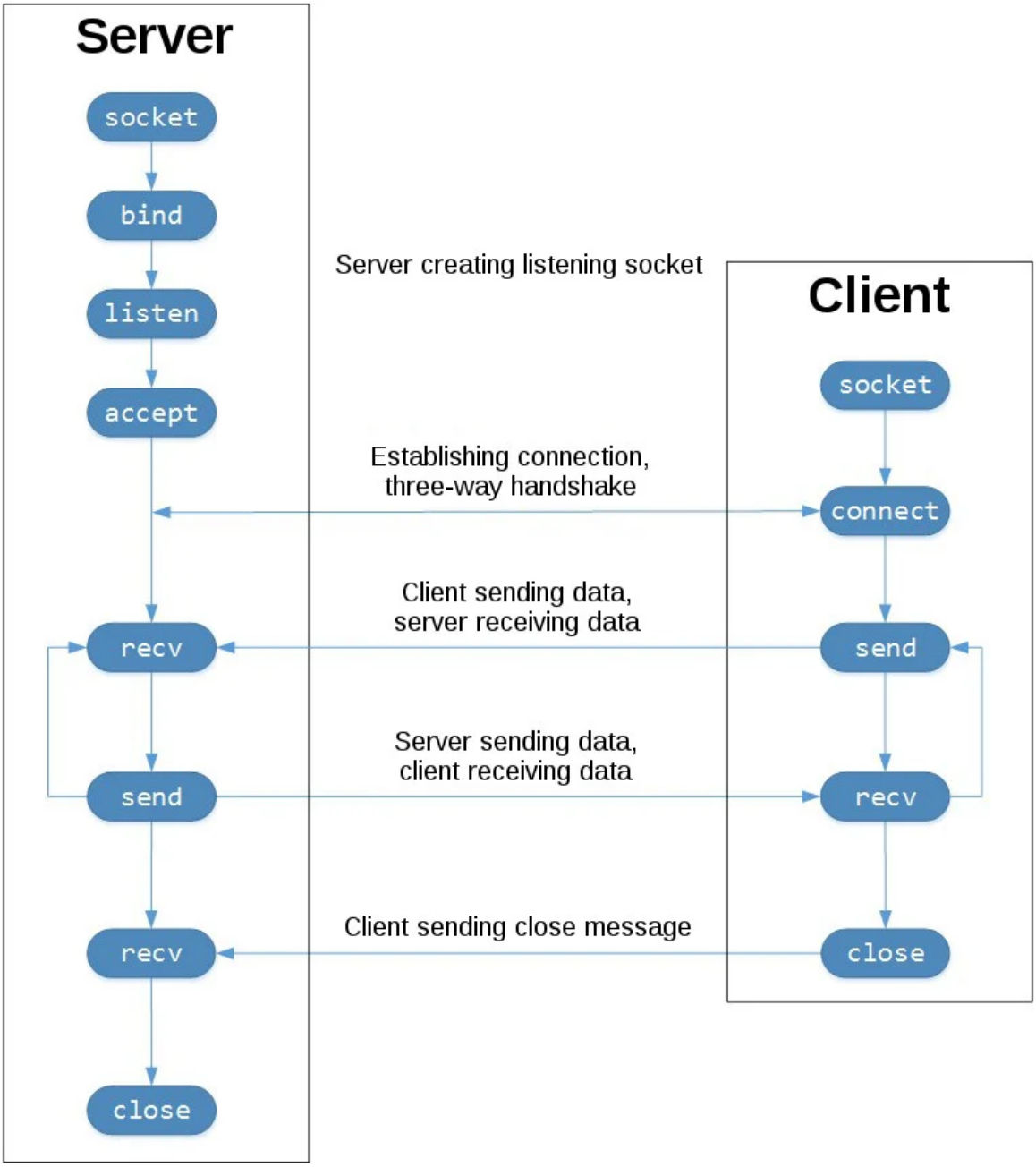

FIGURE 4 – Mécanisme de traitement d'une socket sous Python.

Comme indiqué par la figure 4, les appels à l'API nécessaires pour disposer d'une socket en mode « serveur en écoute » sont :

```
socket()
.bind()
.listen()
.accept()
```

Une socket en écoute fait exactement ce que son nom suggère : elle attend les connexions des clients. Quand un client se connecte, le serveur appelle la fonction *.accept()* pour accepter la connexion

entrante.

Côté client, la demande de connexion utilise l'appel *.connect()*. Cette fonction va établir la connexion et initier le *three-way handshake*. Ce handshake est capital car il asssure que chaque pôle de communication est joignable dans le réseau. Ensuite se déroule l'échange de données grâce aux fonctions *.send()* et *.recv()*.

Enfin, le client et le serveur ferment leurs sockets respectives.

3.2. Mise en œuvre sous Python

La mise en poeuvre des concepts vus précedemment fait l'objet du TD n°3 : sockets.

Ce document est inspiré du « Cours Système n°7 : les sockets » délivré par monsieur Vincent Strubel dans les cours dispensés à l'Agence nationale de la sécurité des systèmes d'information (Janvier 2013).