Administration Systèmes et Réseaux

Partie 1 — Architecture & communications Client/Serveur

Université Claude Bernard Lyon 1

Sources et remerciements

- D'après le cours « Architecture et communications Client/Serveur » d'Olivier Glück (Lyon 1)
- Compléments : Olivier Aubert (Lyon 1), Bénédicte Le Grand (UPMC)
- Certaines figures sont issues des ouvrages de la bibliographie

Bibliographie

- A. Tanenbaum, D. Wetherall, « Réseaux », Pearson
- J. Kurose, K. Ross, « Analyse structurée des réseaux », Pearson (8e éd.)
- W. R. Stevens, « TCP/IP Illustrated, Vol. 1 », Addison-Wesley
- J.-M. Rifflet, « La communication sous Unix » (sockets / IPC)
- RFC (rfc-editor.org), documentation HTTP / TLS / QUIC (IETF, MDN)

Plan de la première partie

- Organisation pratique et contenu du module
- Rappels : Internet et le modèle TCP/IP
- Architecture Client/Serveur et middleware
- Communications inter-processus et sockets
- Appels de procédures distantes (des RPC à gRPC)
- Exercices

Plan de la première partie

- **Organisation pratique et contenu du module**
- Rappels : Internet et le modèle TCP/IP
- Architecture Client/Serveur et middleware
- Communications inter-processus et sockets
- Appels de procédures distantes (des RPC à gRPC)
- Exercices

Organisation pratique et contenu du module

Le module ASR

- Cours magistraux + travaux pratiques (administration Unix et Windows)
- TP en salles TPR
- Évaluation : un contrôle final et des notes de contrôle continu
  - Présentation
  - TP
- Pédagogie : classe inversée, présentations à préparer

Objectifs du module

- Former des administrateurs systèmes et réseaux
- Maîtriser le modèle Client/Serveur, socle de la plupart des services Internet
- Savoir concevoir une application Client/Serveur
- Déployer et sécuriser les services sous Linux et Windows

Contenu (1) — Modèle Client/Serveur

- Architecture et communication Client/Serveur, middleware
- Conception d'une application Client/Serveur
- Modes de communication entre processus
- Sockets TCP/IP (IPv4 et IPv6)
- Serveurs multi-protocoles et multi-services
- Appels distants : RPC, gRPC, API REST

Contenu (2) — Services de l'Internet

- Connexion à distance : SSH (telnet et rlogin abandonnés)
- Transfert de fichiers : SFTP/FTPS, NFS, SMB
- Messagerie : SMTP, IMAP, webmail
- Annuaires : DNS, LDAP
- Web : HTTP / HTTPS

Contenu (3) — Technologies Windows

- Architecture en domaines, Active Directory
- Gestion des utilisateurs, stratégies de groupe
- Système de fichiers et sécurité
- Services réseau, scripts, sauvegardes, cluster

Plan de la première partie

- Organisation pratique et contenu du module
- **Rappels : Internet et le modèle TCP/IP**
- Architecture Client/Serveur et middleware
- Communications inter-processus et sockets
- Appels de procédures distantes (des RPC à gRPC)
- Exercices

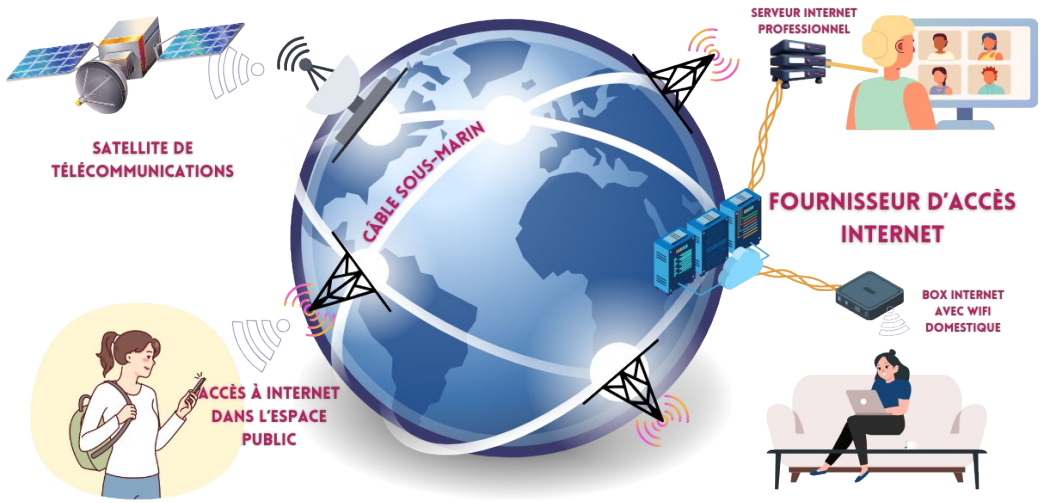
Rappels : Internet et le modèle TCP/IP

Le visage de l'Internet (1)

- Un réseau de réseaux : logiciels et protocoles
- Basé sur TCP/IP, fonctionne en mode Client/Serveur
- Offre des services : web, mail, fichiers, streaming, visioconférence…
- Une accumulation d'« inventions » : TCP/IP, noms et adresses, HTTP…

Le visage de l'Internet (2)

- Construction « par le bas » : réseau local → campus → régional → mondial
- **Trois niveaux d'interconnexion :**
  - postes de travail (ordinateurs, terminaux…)
  - liaisons physiques (cuivre, fibre, Wi-Fi, 4G/5G…)
  - routeurs (équipements ou ordinateurs spécialisés)



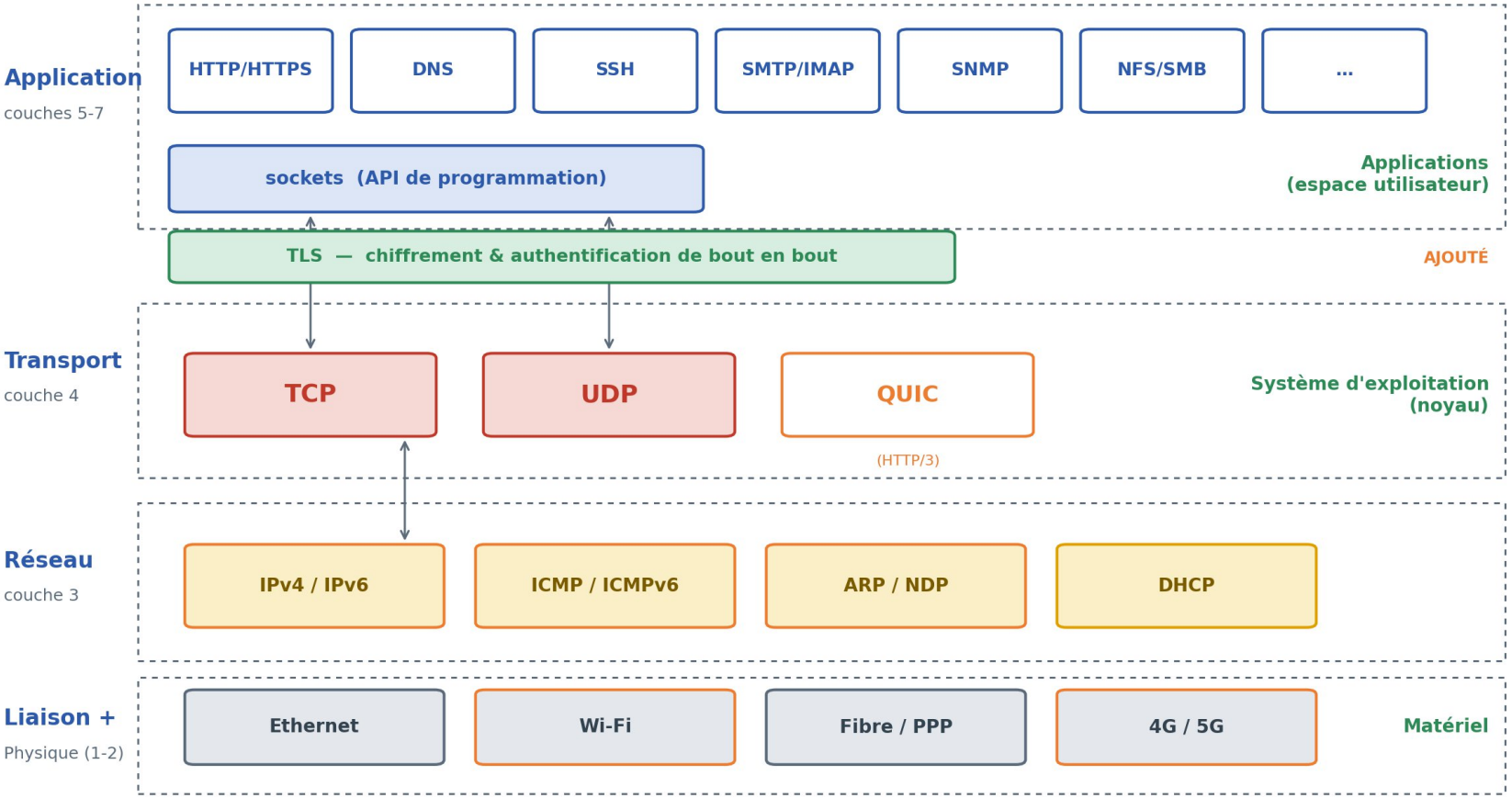
Le visage de l'Internet (3)

- Un ensemble de systèmes autonomes (AS) hétérogènes interconnectés
- Organisation hiérarchique autour de plusieurs dorsales (backbones)
- Points d'échange entre opérateurs (IXP)
- Maîtres-mots : hétérogénéité, passage à l'échelle, Client/Serveur

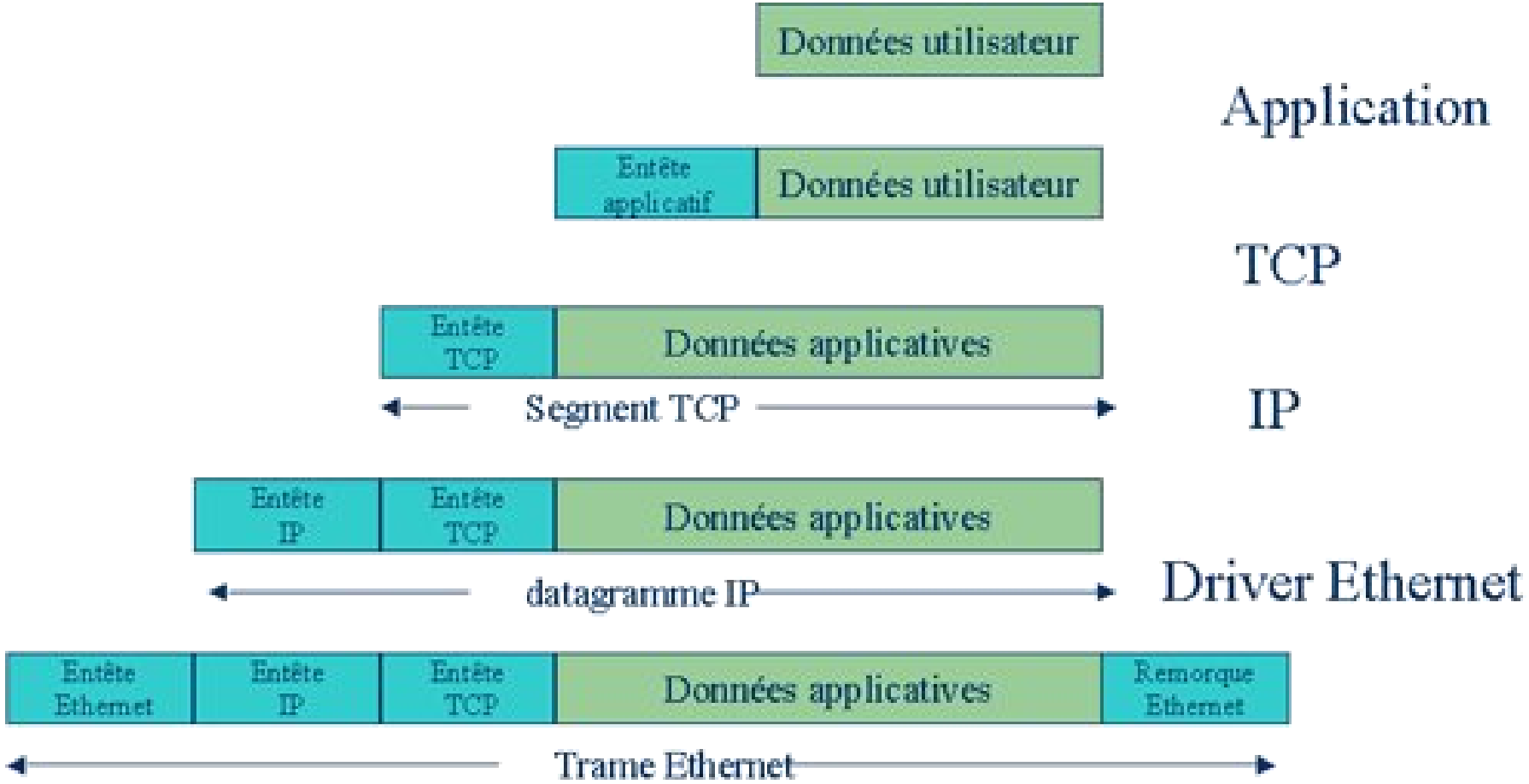
L'architecture TCP/IP (1)

- Une version simplifiée du modèle OSI
- Application : HTTP/S, DNS, SMTP… ; Transport : TCP, UDP, QUIC
- Réseau : IP (IPv4 et IPv6) ; Liaison/physique : Ethernet, Wi-Fi…
- TCP : fiable et connecté ; UDP : non garanti et non connecté

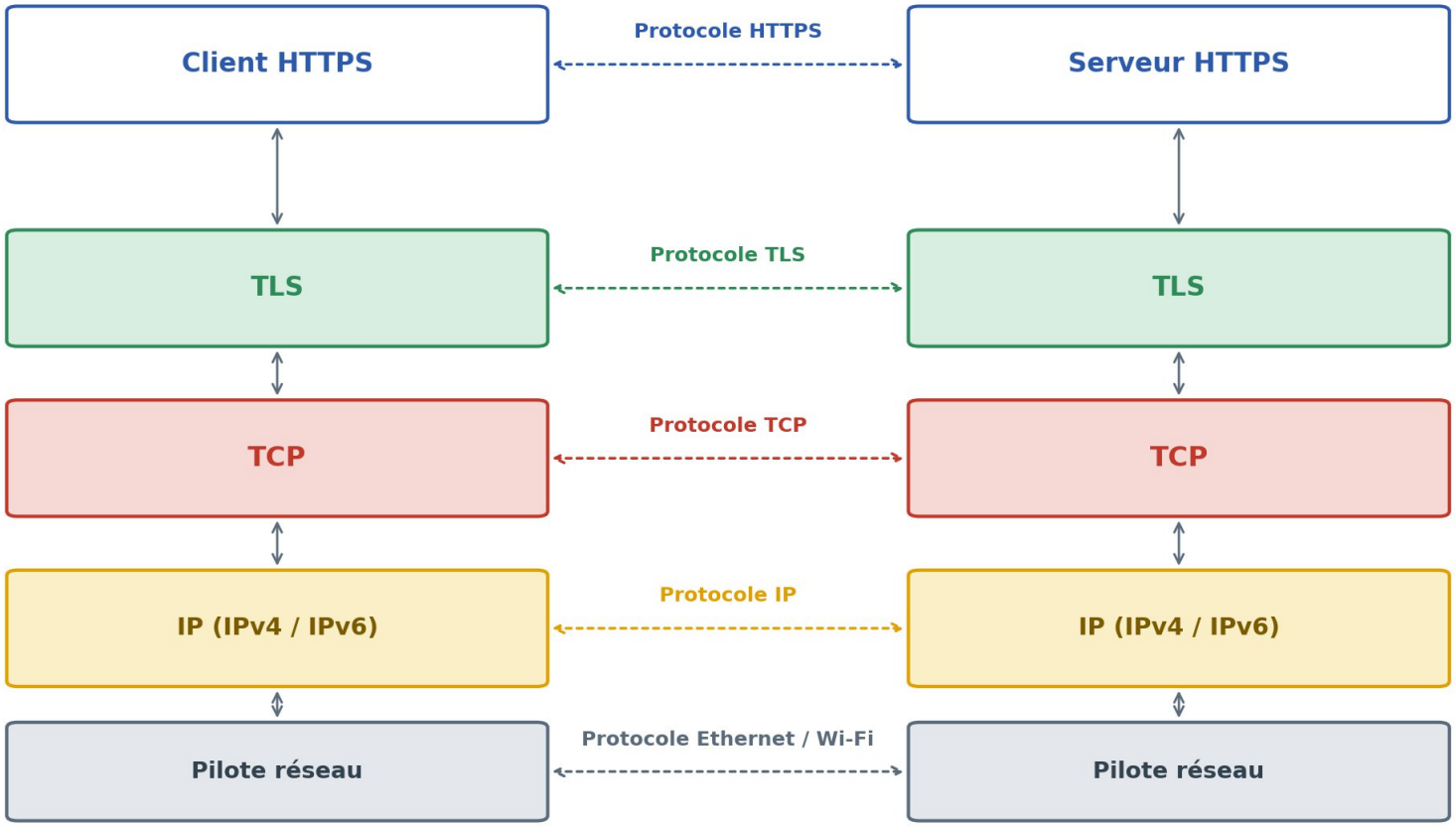
L'architecture TCP/IP — le modèle en couches



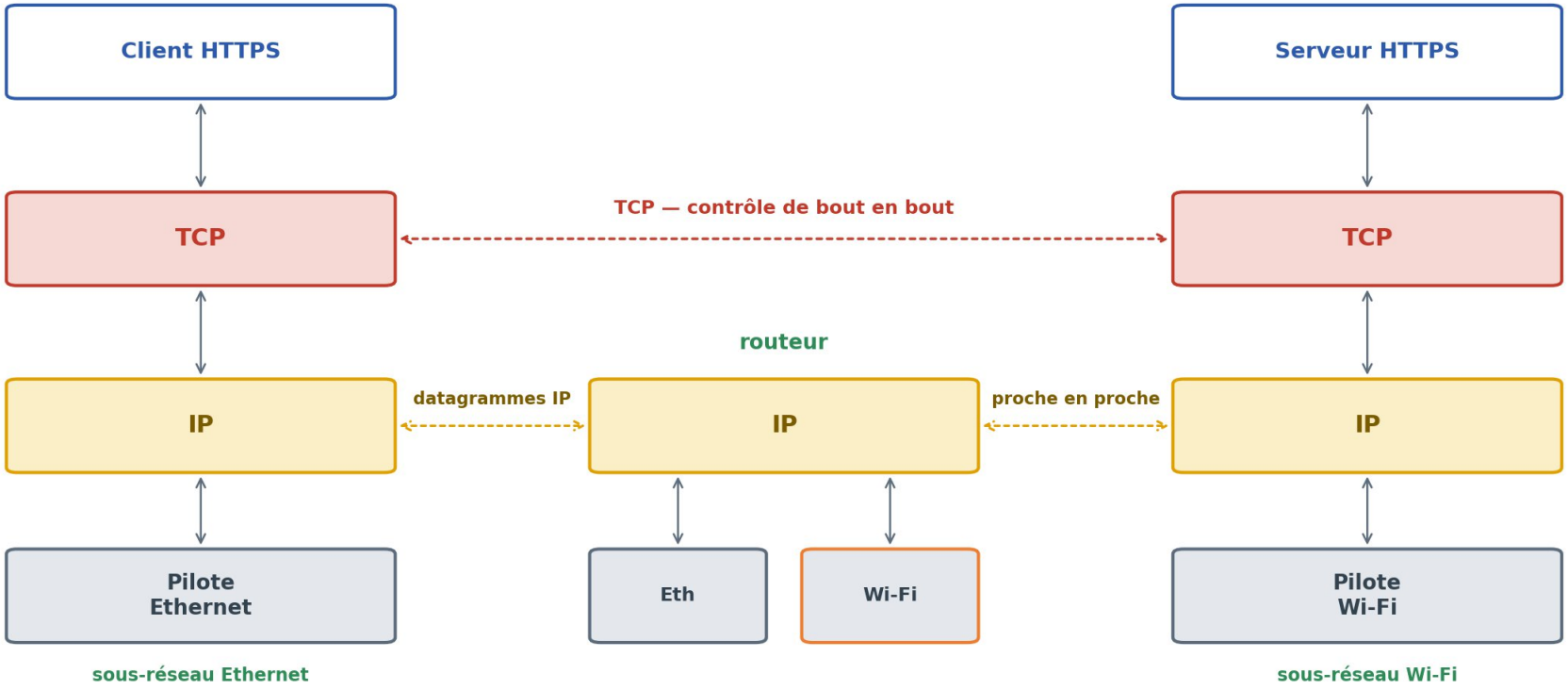
L'architecture TCP/IP — encapsulation des données



L'architecture TCP/IP — deux machines, même sous-réseau



L'architecture TCP/IP — hétérogénéité des sous-réseaux



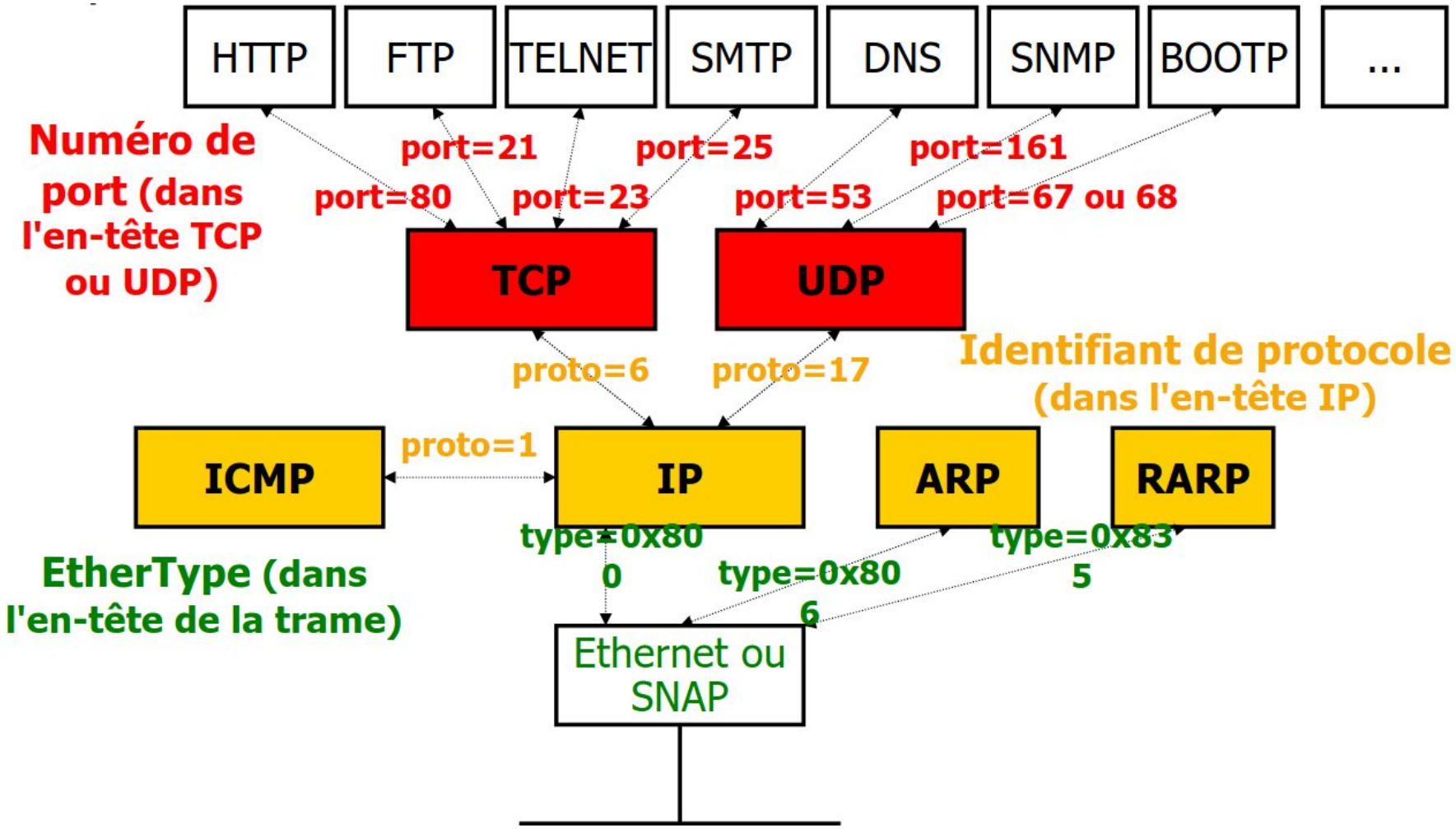
La couche réseau : IP

- Achemine des datagrammes de proche en proche (mode non connecté)
- Service best-effort : pas de garantie, mais simple et robuste
- Le routage est assuré par les routeurs
- IPv6 (128 bits) déployé à côté d'IPv4 — fonctionnement en double pile

La couche transport : TCP / UDP

- Communication de bout en bout, présente uniquement aux extrémités
- TCP : fiable et connecté (retransmission, séquencement, contrôle de flux)
- UDP : non fiable, non connecté, faible latence
- QUIC : transport moderne au-dessus d'UDP, base de HTTP/3

Identification des protocoles



Identification des protocoles

- EtherType (trame) → protocole réseau ; champ Protocol (IP) → transport
- Numéro de port (TCP/UDP, 16 bits) → application
- Adresse de transport = adresse IP + port = adresse de socket
- Une connexion = quintuplé (proto, @src, port src, @dst, port dst)

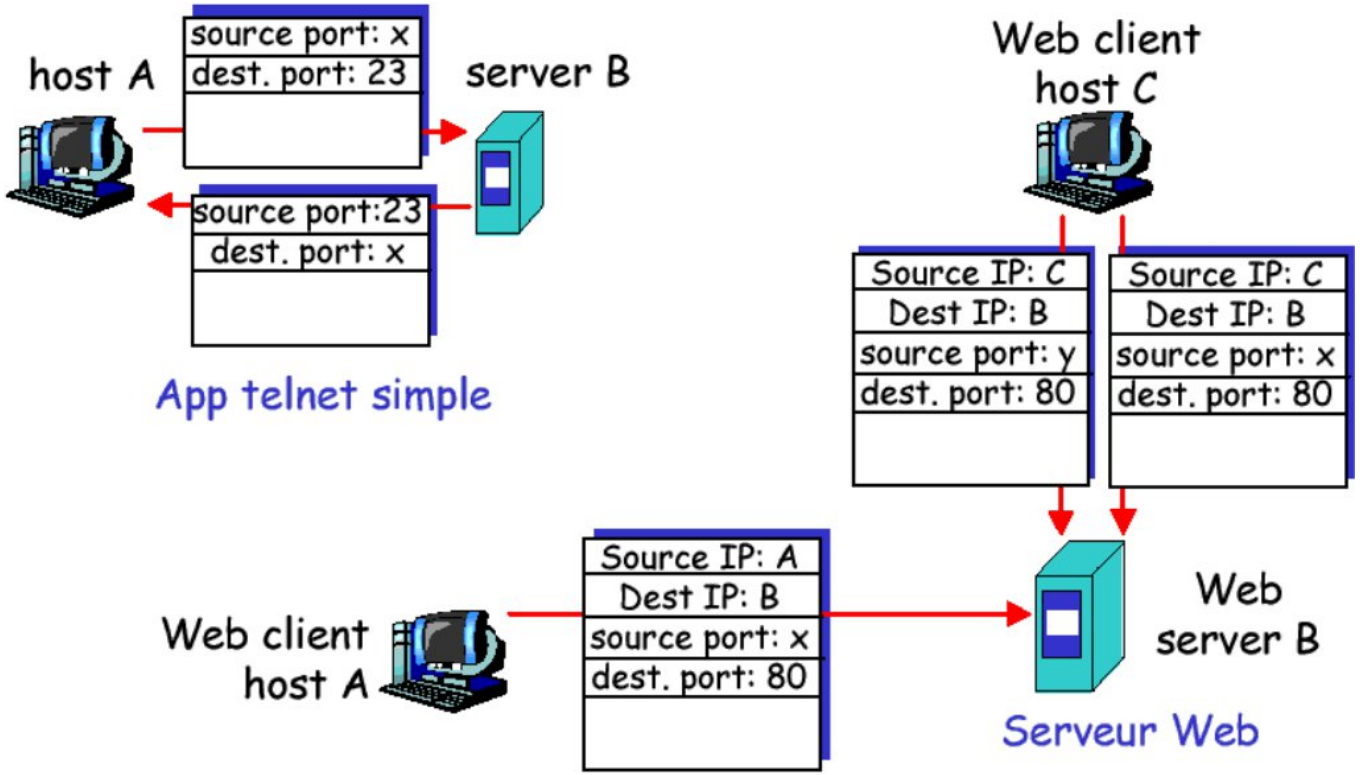
Ports et multiplexage

- Les ports inférieurs à 1024 sont réservés (well-known)
- Une socket : @IP, port

PORT NUMBER	TRANSPORT PROTOCOL	SERVICE NAME	RFC
20, 21	TCP	File Transfer Protocol (FTP)	RFC 959
22	TCP and UDP	Secure Shell (SSH)	RFC 4250-4256
23	TCP	Telnet	RFC 854
25	TCP	Simple Mail Transfer Protocol (SMTP)	RFC 5321
53	TCP and UDP	Domain Name Server (DNS)	RFC 1034-1035
67, 68	UDP	Dynamic Host Configuration Protocol (DHCP)	RFC 2131
69	UDP	Trivial File Transfer Protocol (TFTP)	RFC 1350
80	TCP	HyperText Transfer Protocol (HTTP)	RFC 2616
110	TCP	Post Office Protocol (POP3)	RFC 1939
119	TCP	Network News Transport Protocol (NNTP)	RFC 8977
123	UDP	Network Time Protocol (NTP)	RFC 5905
135-139	TCP and UDP	NetBIOS	RFC 1001-1002
143	TCP and UDP	Internet Message Access Protocol (IMAP4)	RFC 3501
161, 162	TCP and UDP	Simple Network Management Protocol (SNMP)	RFC 1901-1908, 3411-3418
179	TCP	Border Gateway Protocol (BGP)	RFC 4271
389	TCP and UDP	Lightweight Directory Access Protocol	RFC 4510
443	TCP and UDP	HTTP with Secure Sockets Layer (SSL)	RFC 2818
500	UDP	Internet Security Association and Key Management Protocol (ISAKMP) / Internet Key Exchange (IKE)	RFC 2408 - 2409
636	TCP and UDP	Lightweight Directory Access Protocol over TLS/SSL (LDAPS	RFC 4513
989/990	TCP	FTP over TLS/SSL	RFC 4217
			https://ipwithease.com

Ports et multiplexage

- Les ports inférieurs à 1024 sont réservés (well-known)
- Multiplexage/démultiplexage des connexions au niveau transport
- Plusieurs connexions peuvent aboutir à la même socket d'écoute
- Exemples : 22 SSH, 53 DNS, 443 HTTPS



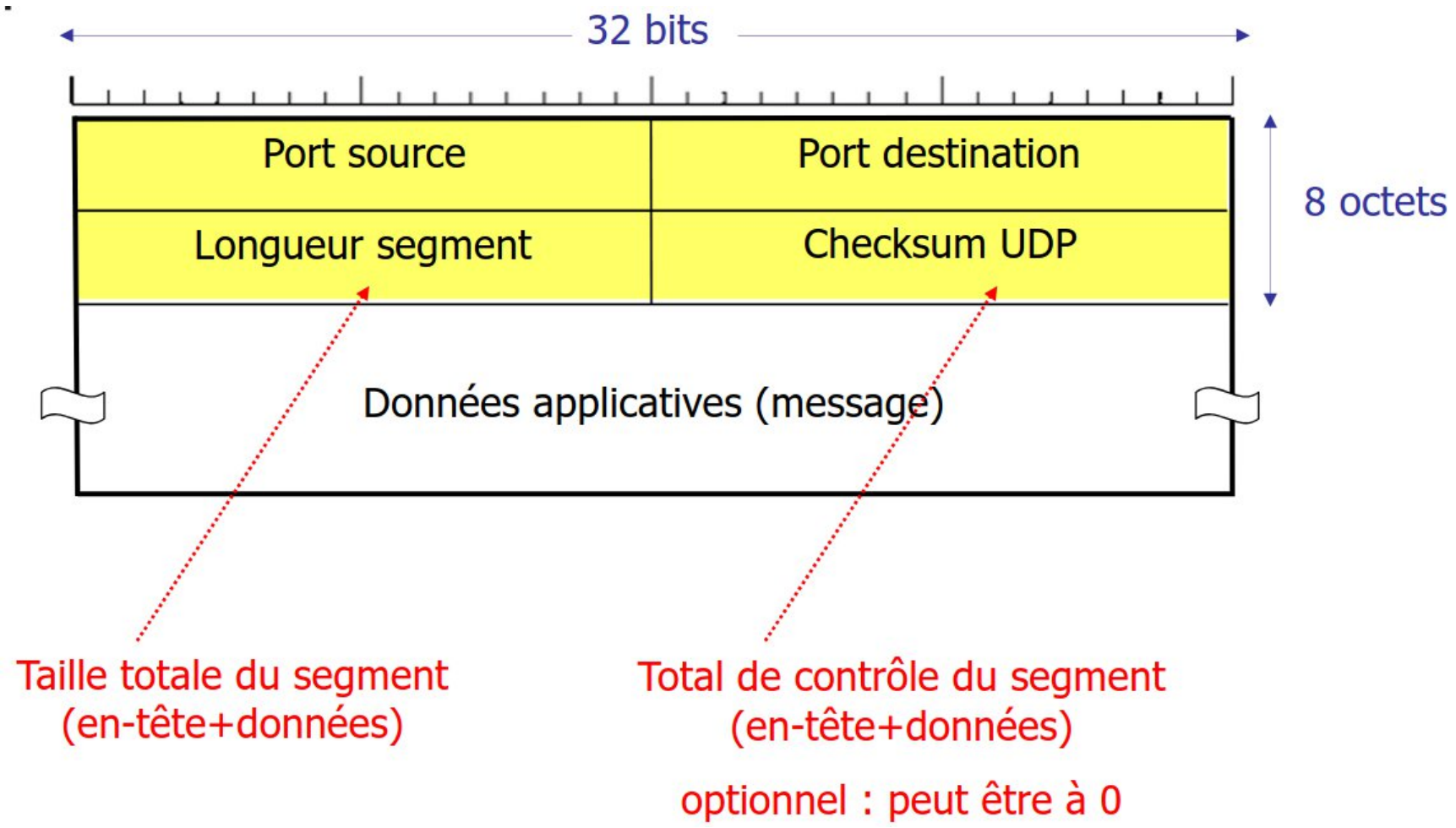
Le protocole UDP

- RFC 768 — le protocole de transport le plus simple
- Best-effort : datagrammes perdus ou déséquencés possibles
- Mode non connecté : chaque datagramme est traité indépendamment
- En-tête réduit : ports, longueur, somme de contrôle

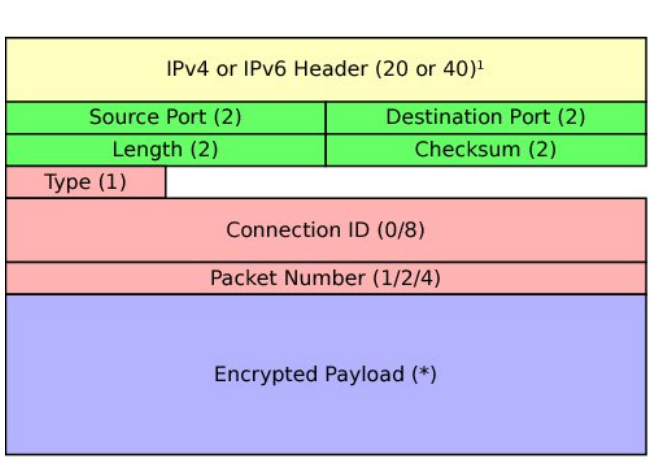
Les utilisations d'UDP

- Applications tolérant des pertes ou sensibles à la latence
- DNS, DHCP, voix et vidéo temps réel, jeux en ligne
- Fiabilité éventuellement gérée au niveau applicatif
- QUIC ajoute fiabilité et chiffrement au-dessus d'UDP

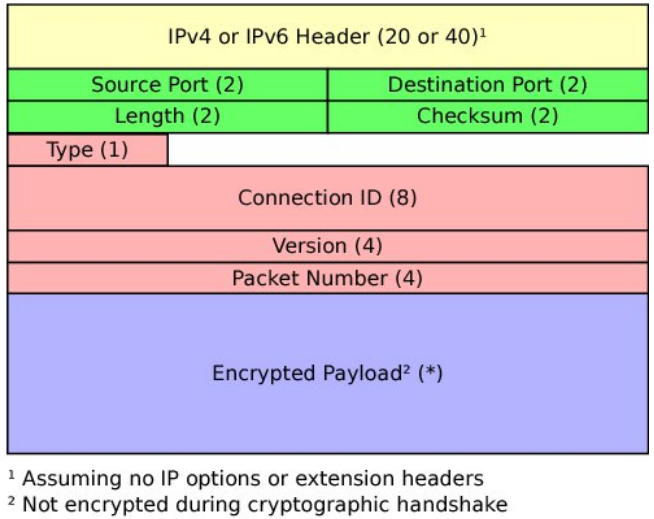
Les utilisations d'UDP



Quick



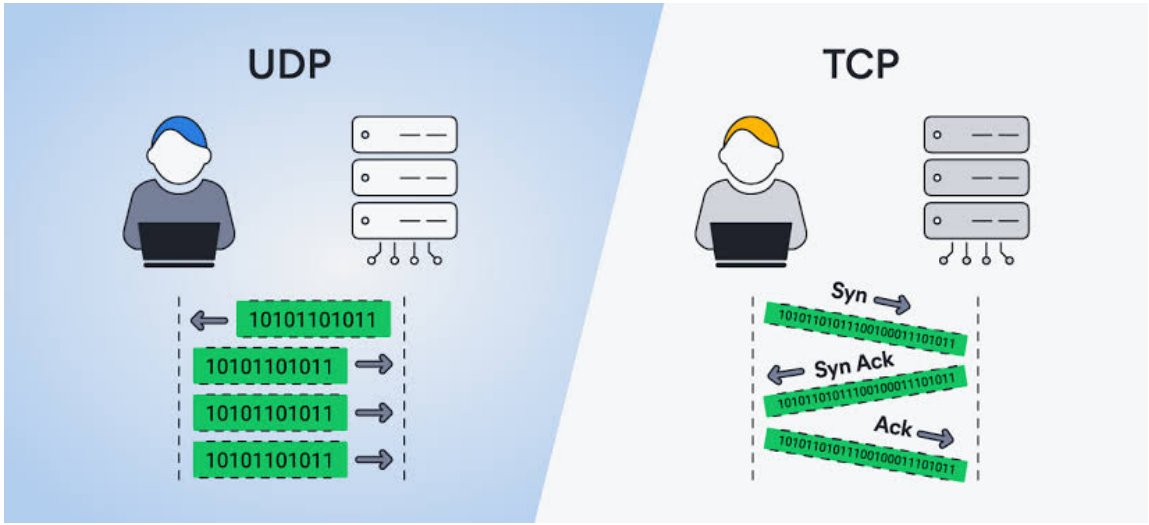
(a) With a short header.



(b) With a long header.

Le protocole TCP

- RFC 793 et suivantes — transport fiable en mode connecté
- Transporte un flot d'octets ; contrôle d'erreurs et de flux
- Établissement de connexion (3-way handshake) puis fermeture
- Contrôle de congestion à l'aide d'une fenêtre d'émission



Applications réseaux - exemples

- SMTP - Simple Mail Transfer Protocol
  - service d'envoi de courrier électronique
  - réception (POP, IMAP, IMAPS, …)
- DNS - Domain Name System
  - assure la correspondance entre un nom symbolique et une adresse Internet (adresse IP)
  - bases de données réparties sur le globe
- SNMP - Simple Network Management Protocol
  - protocole d'administration de réseau (interrogation, configuration des équipements, …)
- Les sockets - interface de programmation permettant l'échange de données (via TCP ou UDP)

Architecture Client/Serveur

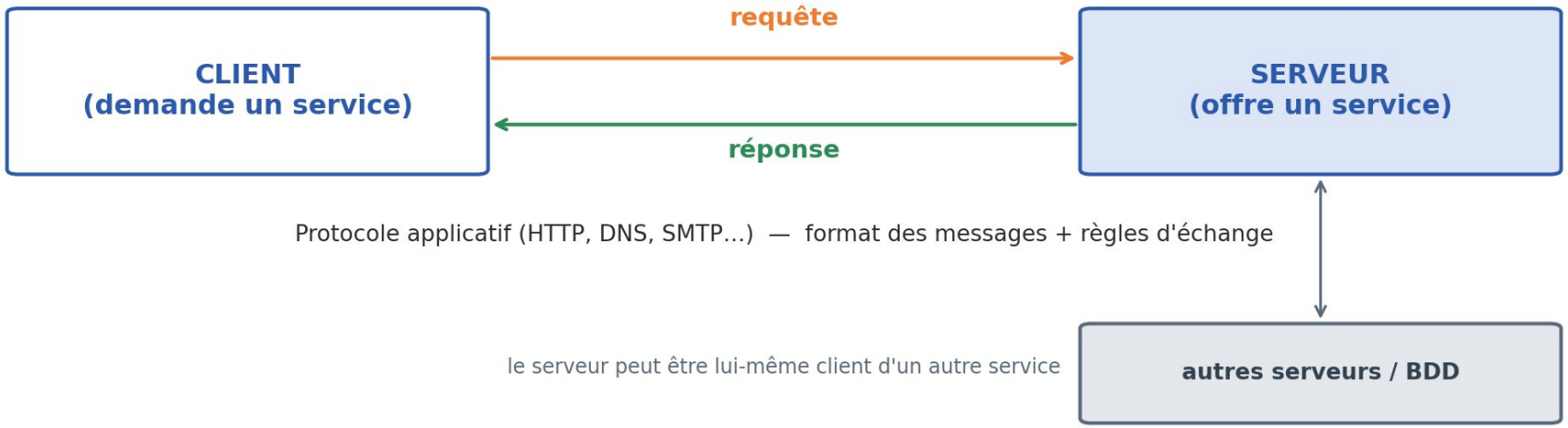
Les applications réseau

- Les applications sont la raison d'être des réseaux
- Profusion depuis 40 ans : web, mail, fichiers, streaming, visio, téléphonie…
- Elles reposent sur une infrastructure de transport (les sockets)
- Exemple du web : le navigateur (client) dialogue avec un serveur

Terminologie des applications réseau

- Processus client / processus serveur
- Ne pas confondre le protocole applicatif et l'application
- Le protocole définit : format des messages, ordre des échanges, actions
- Exemples de protocoles : HTTP, SMTP, DNS

Le modèle Client/Serveur

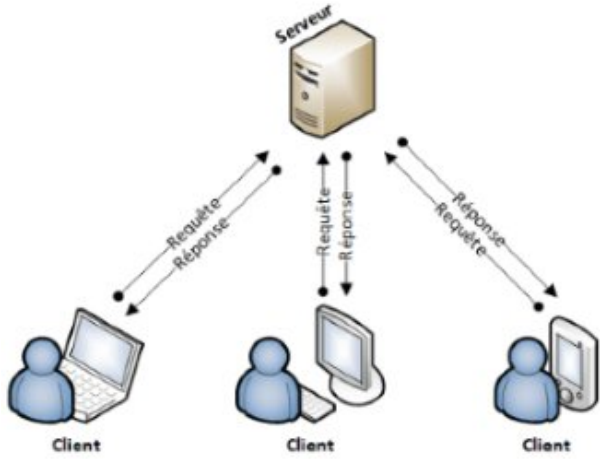


Le modèle Client/Serveur

- L'application est répartie sur plusieurs entités
- Le client demande un service, le serveur le rend
- Système coopératif, possible sur des systèmes d'exploitation différents
- Un serveur peut être lui-même client d'un autre service

Des clients et des serveurs…

- Plusieurs clients peuvent s'adresser à un même serveur (concurrence)
- Le serveur est souvent permanent, le client lancé à la demande
- Communication via une adresse de transport connue (port)
- Exemple : un serveur web sert des milliers de clients

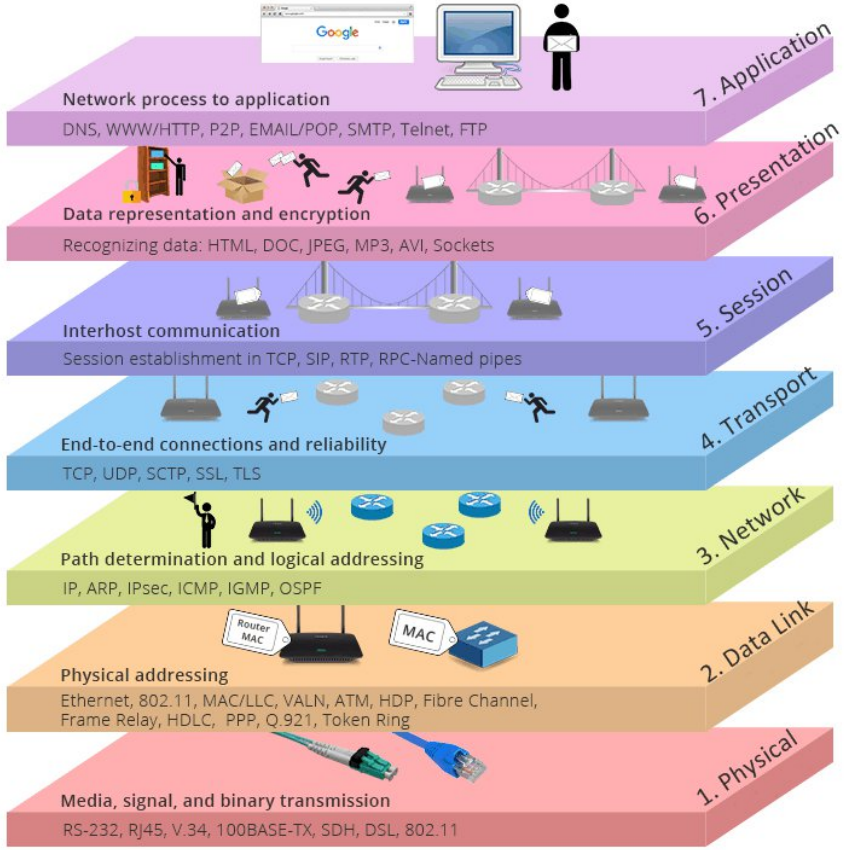


Exemple d'application Client/Serveur

- Service simple : le serveur renvoie une ligne de texte (ex. l'heure)
- Le client se connecte, lit la réponse et l'affiche
- Le protocole spécifie le format (ex. une ligne ASCII)

Interface de programmation réseau

- Il faut une interface entre l'application et la pile réseau
- Les sockets : API standard (BSD), portable
- Indépendante du transport (TCP/UDP) et de la version d'IP
- getaddrinfo() : résolution nom → adresse, IPv4 et IPv6



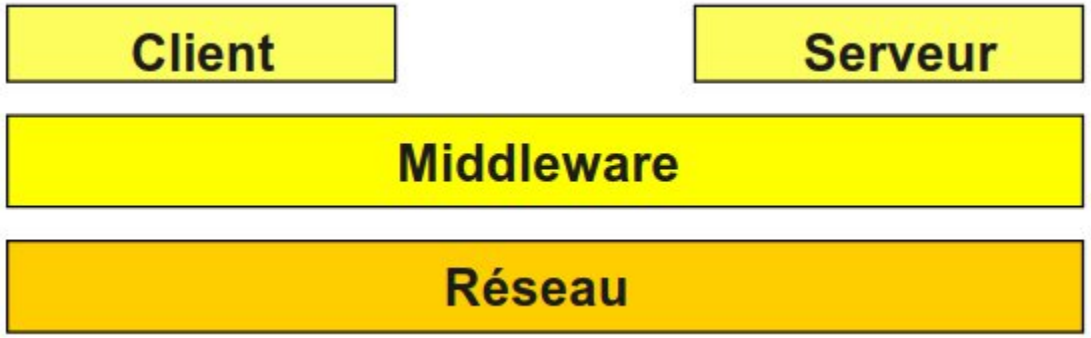
Application Client/Serveur — récapitulatif

- Choisir le transport, le mode (itératif/concurrent) et le format des messages
- Définir précisément le protocole applicatif
- Gérer les erreurs et la sécurité (TLS)
- Penser la montée en charge dès la conception

Le middleware

Le middleware

- Couche logicielle entre l'application et le système/réseau
- Masque l'hétérogénéité (matériel, OS, langages, réseau)
- Fournit des services communs aux applications réparties
- Exemples : RPC/gRPC, files de messages, API REST, ORB



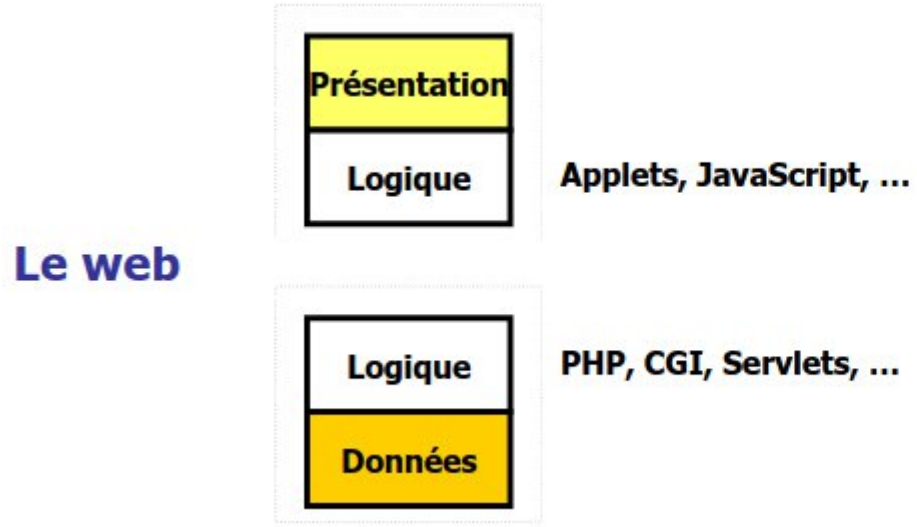
Fonctions d'un middleware

- Communication et conversion des données (sérialisation)
- Nommage et localisation des services (annuaire)
- Sécurité, transactions, tolérance aux pannes
- Aujourd'hui : passerelles d'API, service mesh, brokers de messages

Conception d'une application Client/Serveur

Découpage Client/Serveur

- Trois fonctions : présentation, logique métier, accès aux données
  - Module de gestion des données peut être hébergé sur un serveur distant
- Où placer la frontière entre client et serveur ?
- Modèle de Gartner : du client lourd au client léger
- Compromis : charge réseau, charge serveur, richesse du client

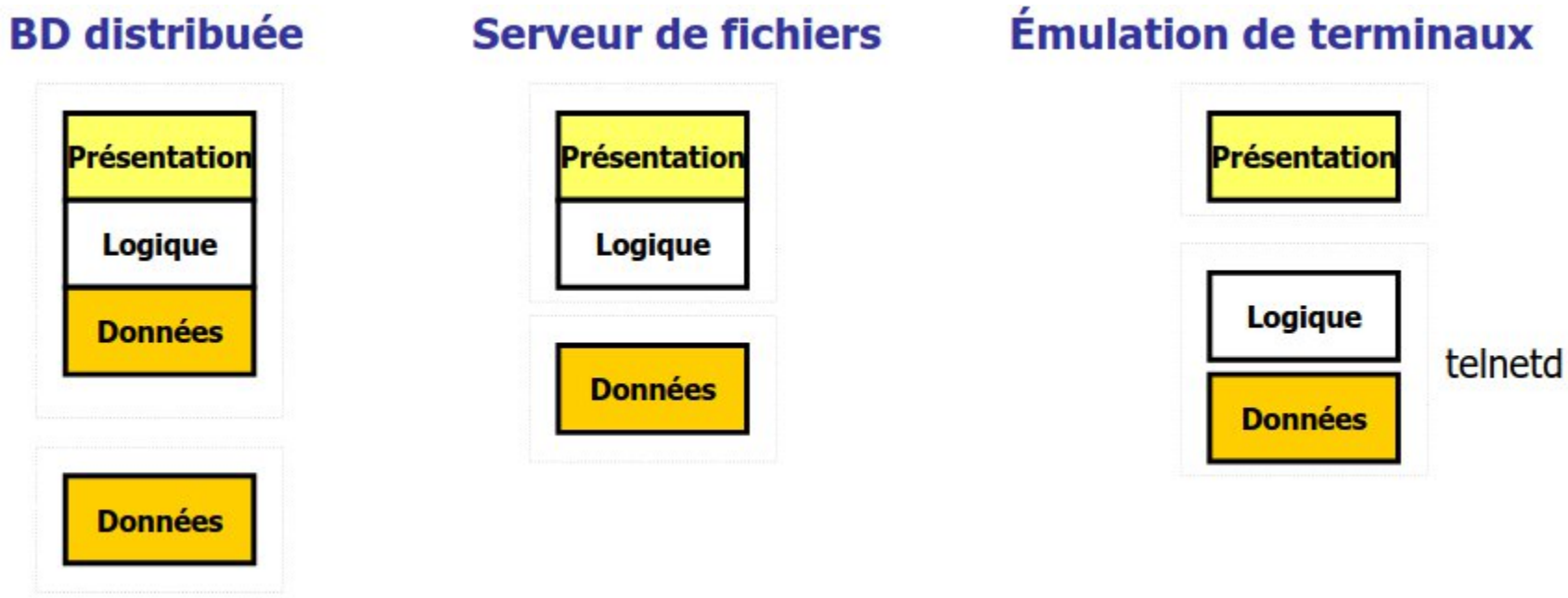


Découpage Client/Serveur

- Une application informatique est représentée
- selon un modèle en trois couches :
  - la couche **présentation** (interface Homme/Machine) :
    - gestion de l'affichage…
  - la couche traitements (ou **logique**) qui assure la fonctionnalité intrinsèque de l'application (algorithme)
  - la couche **données** qui assure la gestion des données de l'application (stockage et accès)

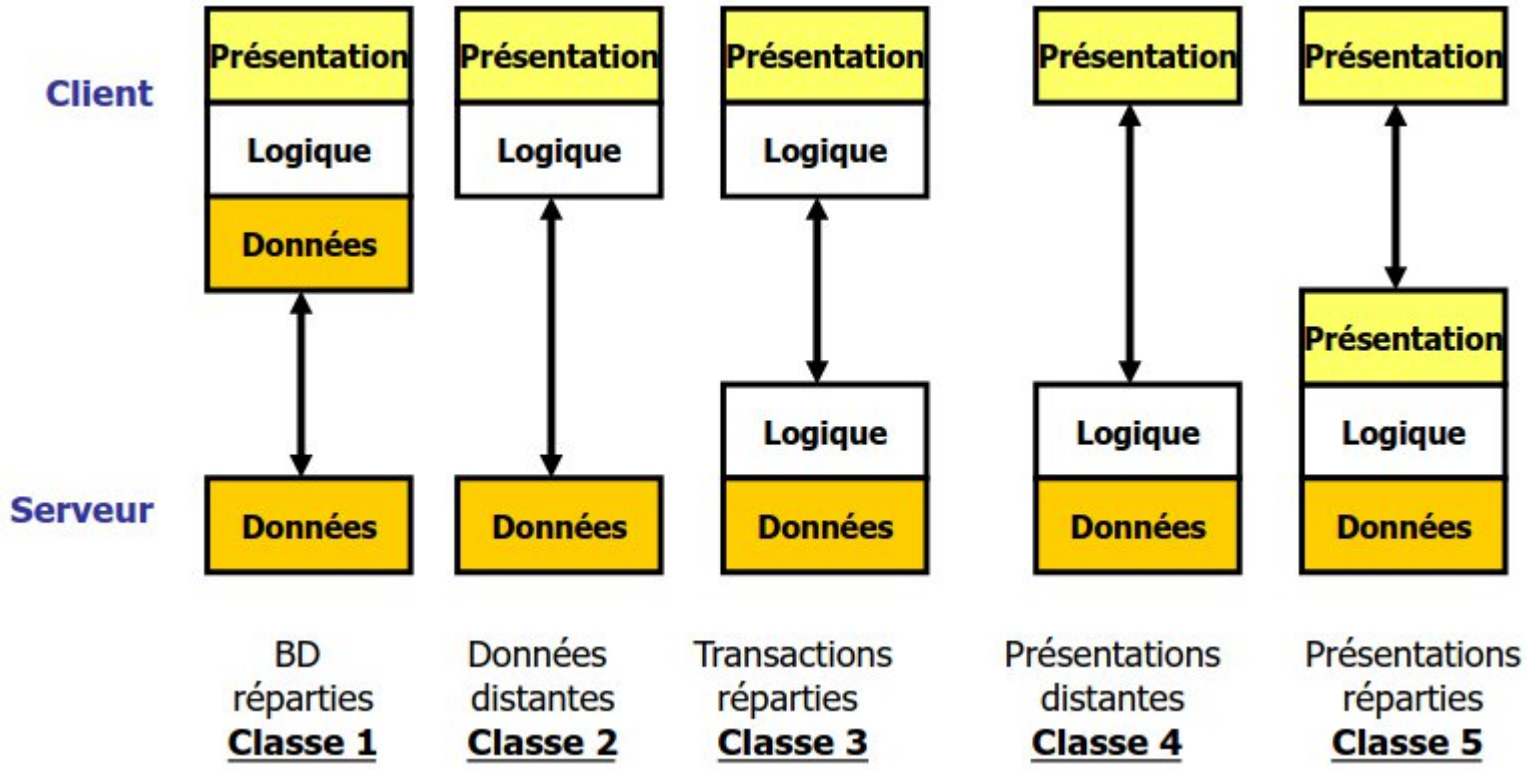
Découpage Client/Serveur

- Autres exemples de découpage



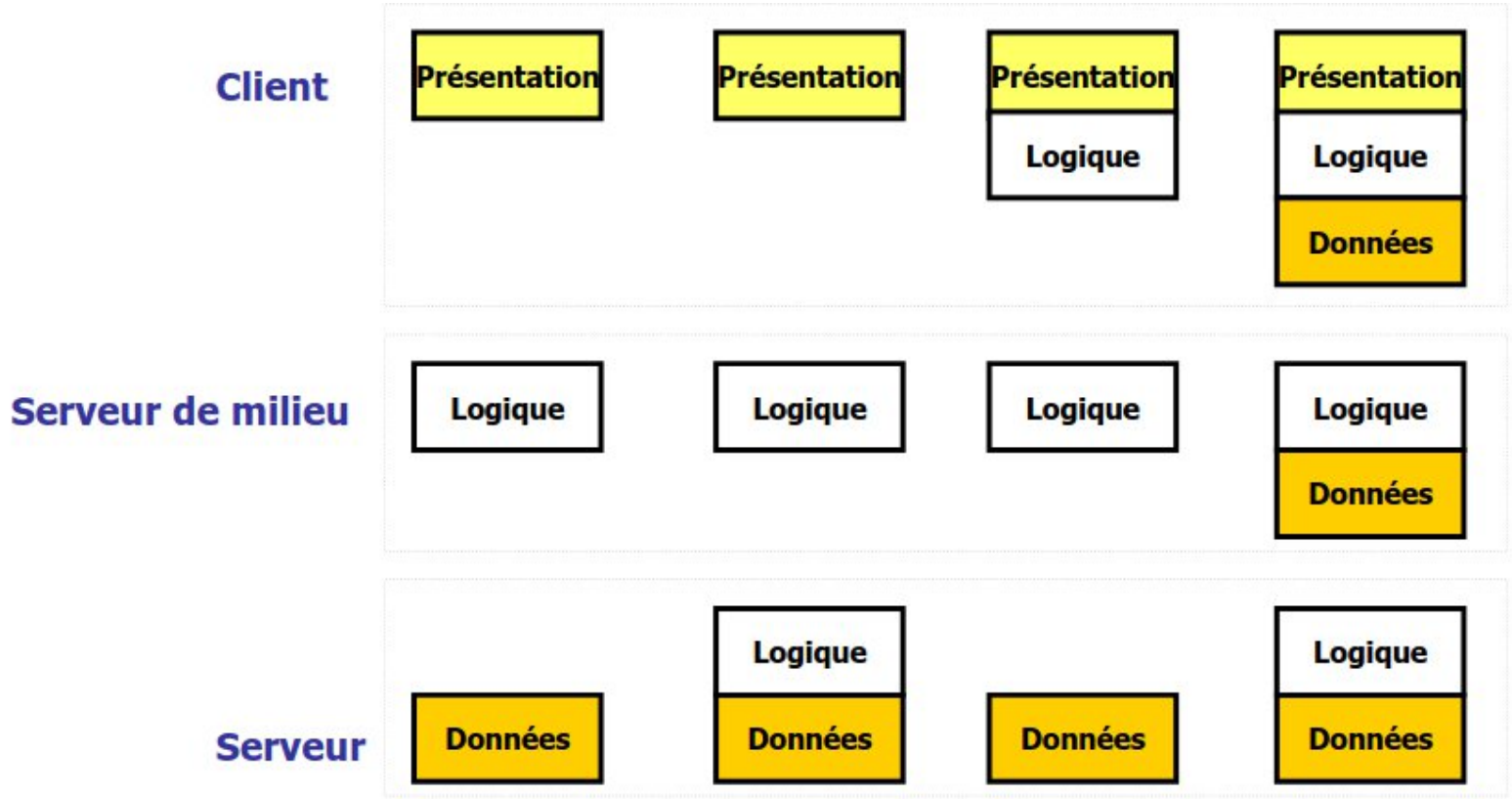
Architectures 2-tiers et 3-tiers

- **2-tiers : client riche ↔ serveur de données**
- 3-tiers : présentation / logique métier / données séparées
- Meilleure évolutivité et maintenance
- Socle des architectures web



Architectures 2-tiers et 3-tiers

- 2-tiers : client riche ↔ serveur de données
- **3-tiers : présentation / logique métier / données séparées**
- Meilleure évolutivité et maintenance
- Socle des architectures web



Du 2-tiers aux microservices

communication : REST (HTTP/JSON), gRPC, files de messages — conteneurs & orchestration

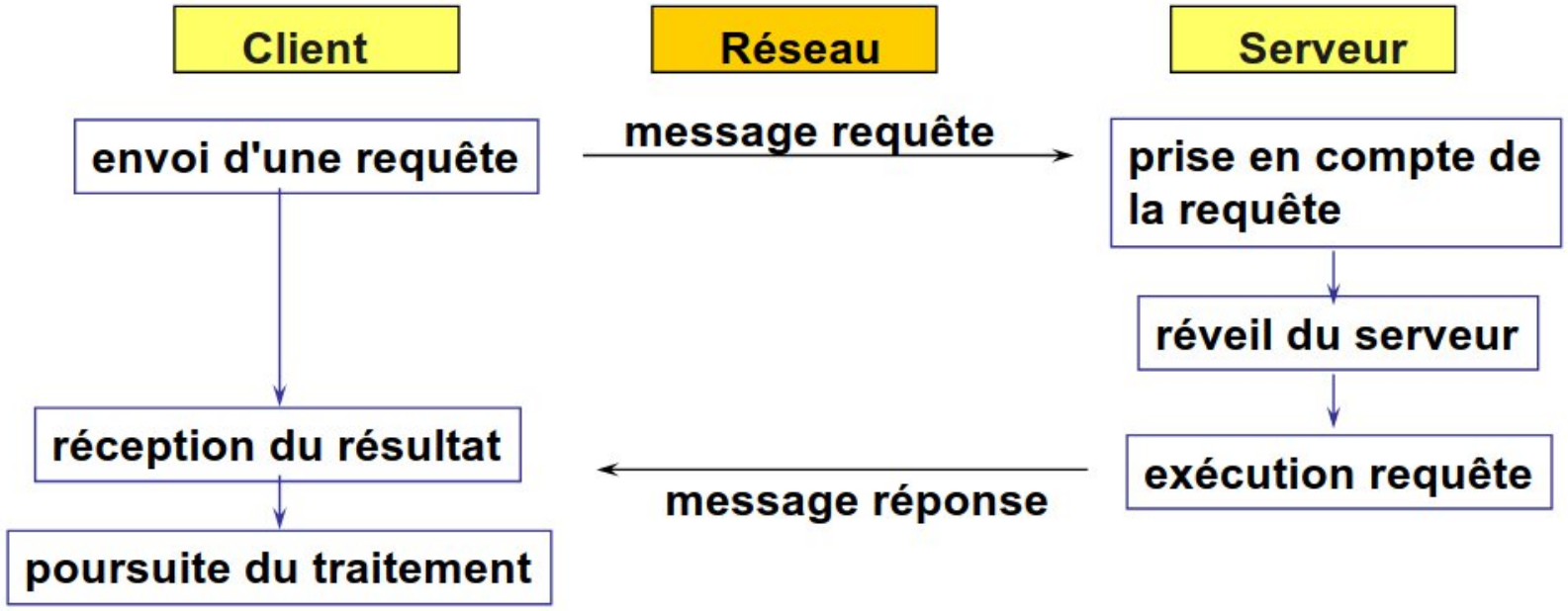
Architectures modernes

- Microservices : services indépendants, déployables séparément
- Conteneurs (Docker) et orchestration (Kubernetes)
- Cloud et serverless : élasticité, facturation à l'usage
- Communication : REST, gRPC, files de messages

Communications entre processus

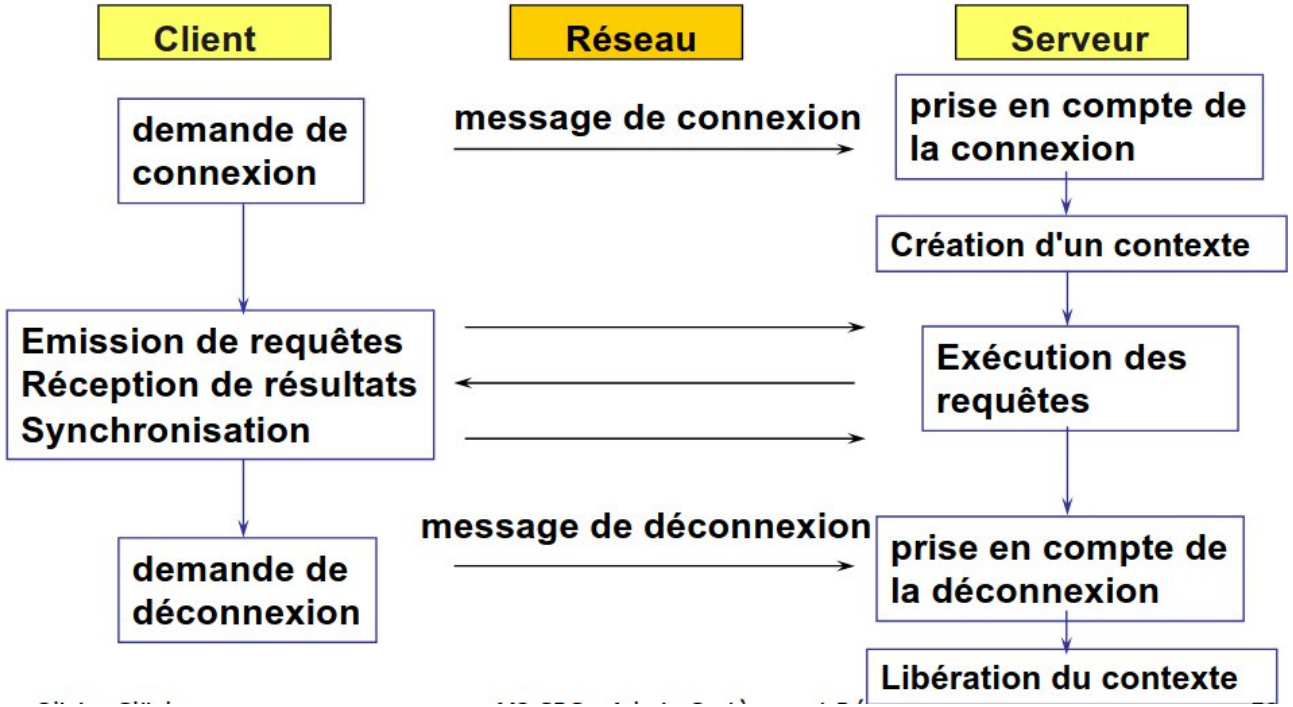
Les modes de communication

- Local (même machine) ou distant (réseau)
- Non connecté (datagrammes, UDP)



Les modes de communication

- Local (même machine) ou distant (réseau)
- Non connecté (datagrammes, UDP) ou Connecté (flux, TCP)



Les modes de communication

- Local (même machine) ou distant (réseau)
- Connecté (flux, TCP) ou non connecté (datagrammes, UDP)
- Synchrone (bloquant) ou asynchrone (non bloquant)
- Avec ou sans état côté serveur

Serveur itératif ou concurrent

- Itératif : une requête à la fois (simple, services courts)
  - Traite séquentiellement les requêtes
  - Mode non connecté (UDP)
- Concurrent : plusieurs clients en parallèle (fork, threads, select)
  - Mode connecté (TCP)
- Le mode concurrent est indispensable pour les services longs
- Compromis simplicité / performance

Service avec ou sans état

- Sans état : chaque requête est autonome (HTTP, DNS) → scalable
  - Le serveur ne conserve aucune information sur l'enchaînement des requêtes...
- Avec état : le serveur mémorise un contexte (session)
- Incidence sur les performances et la tolérance aux pannes dans le cas où un client fait plusieurs requêtes successives
  - performance --> service sans état
  - tolérance aux pannes --> service avec états

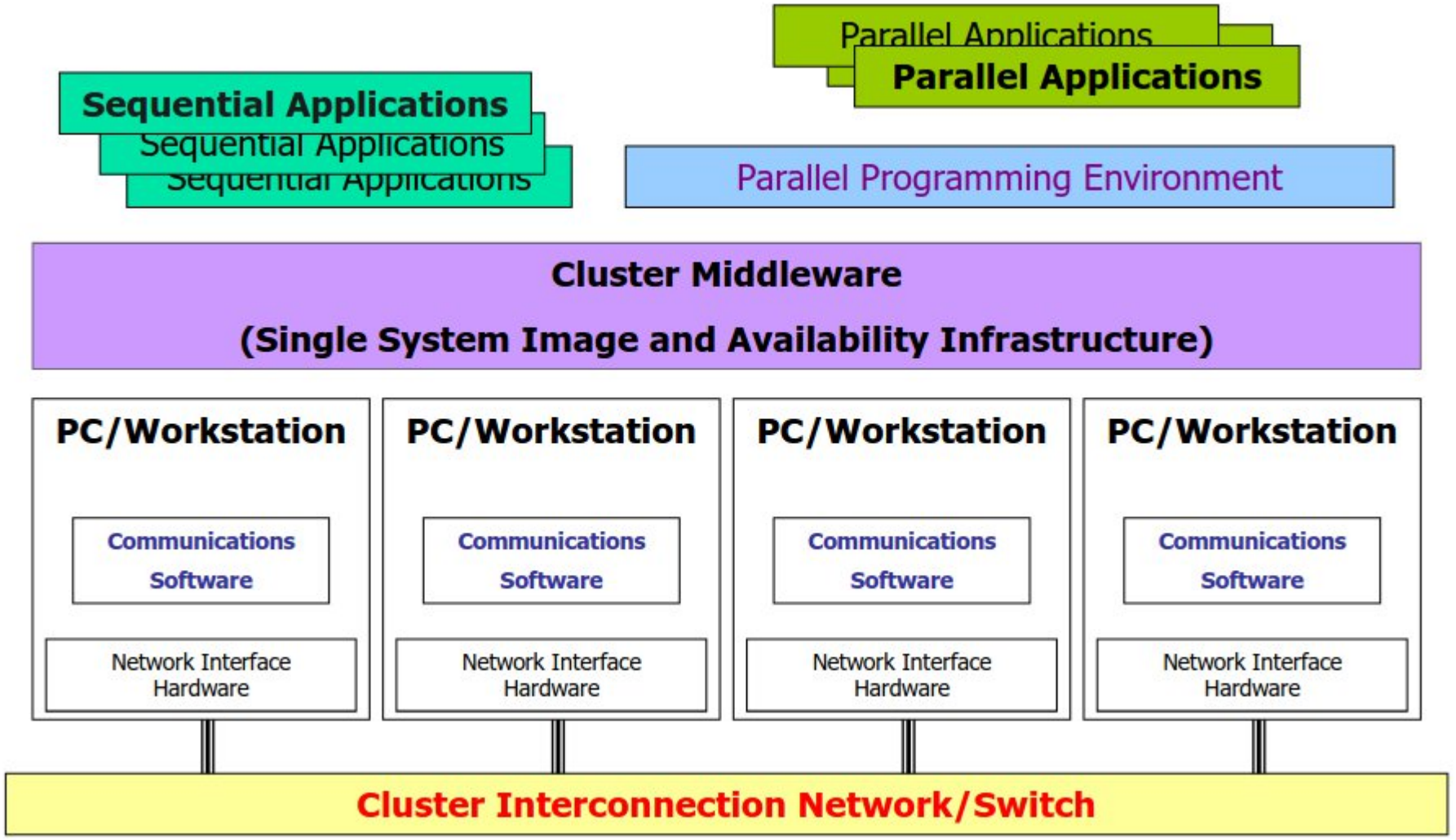
- Tendance moderne : services sans état + état externalisé

Les communications inter-processus (IPC)

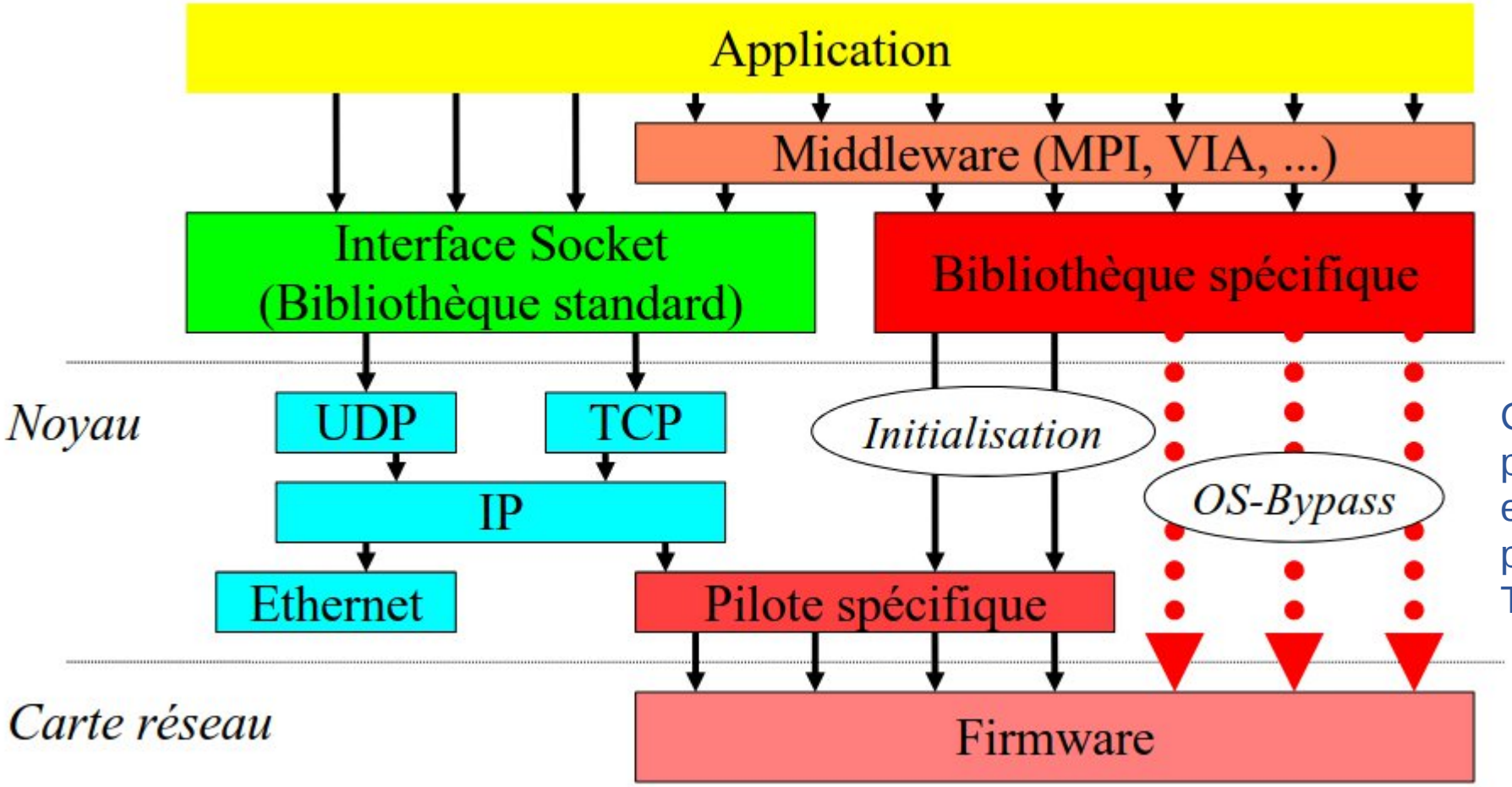
Clusters

- Ensemble de machines vues comme une ressource unique
- Répartition de charge et haute disponibilité
- Communication par passage de messages entre nœuds
- Base du calcul distribué et des fermes de serveurs

Architecture



Mode de fonctionnement



Calcul haute performance par exemple (plusieurs choix possibles) – a latence de TCP/IP trop importante

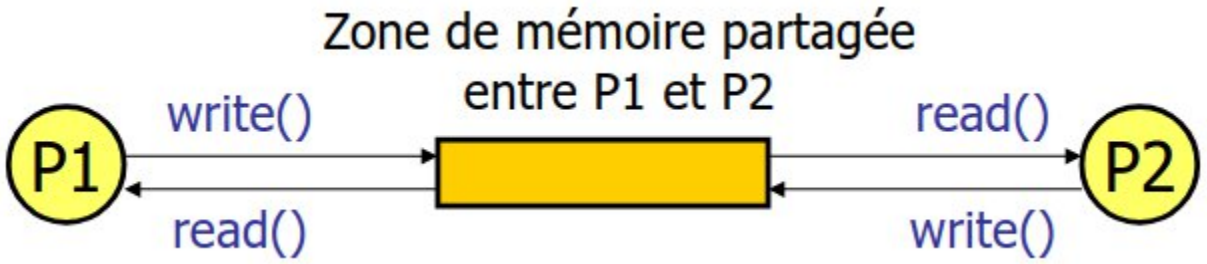
Les schémas de communication locale

- Sur une **seule machine**, on peut partager de la mémoire (rapide, mais impossible à distance) ;
- Les schémas de communication locale. Deux grandes familles :
  - Mémoire partagée : les processus partagent une zone mémoire (rapide, accès direct) ; possible uniquement sur une même machine.
  - Échange de messages : les processus ne partagent rien, le noyau transmet l'information :
    - Tubes (pipes) (même machine mais pas de partage de mémoire)
    - Passage de messages (files, sockets locales)
    - Signaux: notifications (même machine)

- Seul l'échange de messages s'étend entre machines différentes (ex. un cluster), via les sockets réseau

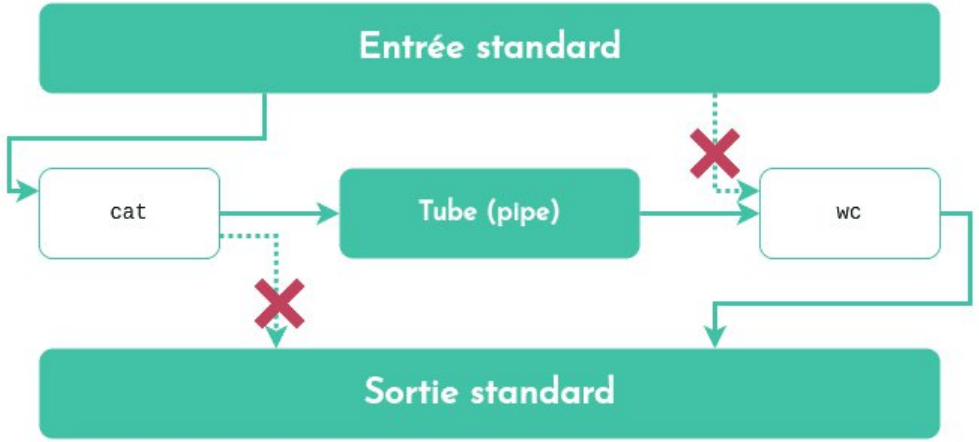
Communication par mémoire partagée

- Les processus partagent une zone mémoire commune
- Communication très rapide
- Nécessite une **synchronisation** (sémaphores, mutex)
- Éviter les écritures concurrentes
- Problème : gestion de l'accès à une ressource partagée



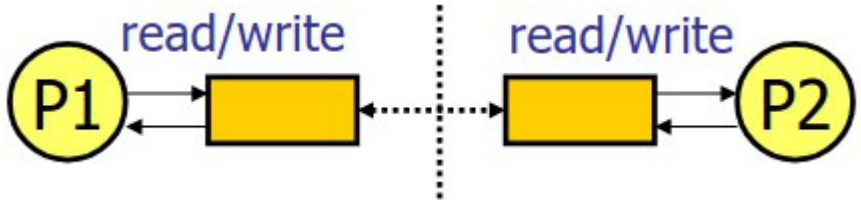
Les tubes de communication (pipes)

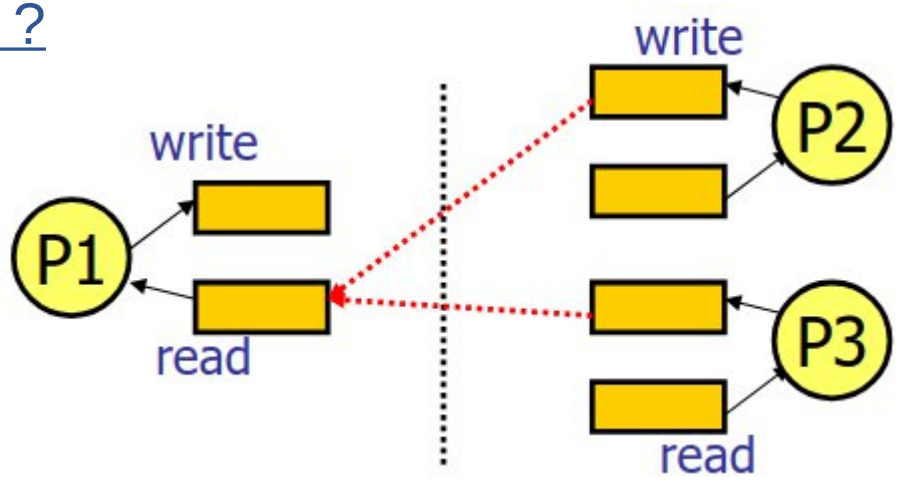
- Canal **unidirectionnel** entre processus apparentés
  - Pas de problème de synchronisation
- Tubes nommés (FIFO) entre processus quelconques
- Modèle producteur / consommateur
- Largement utilisés dans le shell (|). Exemple : cat test.txt | wc -l



Communication par passage de messages

- Les processus échangent des messages explicites
  - Au moins deux primitives : send() et recv()
- Pas de variables partagées
- Files de messages, sockets locales (Unix domain)
- Modèle naturel pour le réseau
- Problème : Opérations bloquantes/non bloquantes ?
  - Zones d'émission et réception distinctes ?
  - Eviter les écritures concurrentes
    - Dissocier le tampon d'émission et de réception
    - Avoir autant de tampons de réception que d'émetteurs potentiels





Opérations bloquantes / non bloquantes

- Quand un appel à une primitive (send ou recv) doit il se terminer ?
- Bloquante : l'appel attend la fin de l'opération
  - Long et bloquant ...
- Non bloquante : retour immédiat, statut à tester
  - **attente active** : appels réguliers à la primitive jusqu'à complétion
  - **attente passive** : le système informe le processus par un moyen quelconque de la complétion de l'opération (signaux par exemple)
- Compromis simplicité / réactivité
- select() / poll() / epoll() : attendre plusieurs sources

Opérations bloquantes / non bloquantes - réception

- Quand un appel à une primitive (send ou recv) doit il se terminer ?

- Plusieurs sémantiques en réception, recv() peut rendre la main :
  - aussitôt (recv() non bloquant)
  - quand les données ont été reçues et recopiées depuis le tampon de réception local (le tampon de réception est de nouveau libre)

Opérations bloquantes / non bloquantes - émission

- Quand un appel à une primitive (send ou recv) doit il se terminer ?

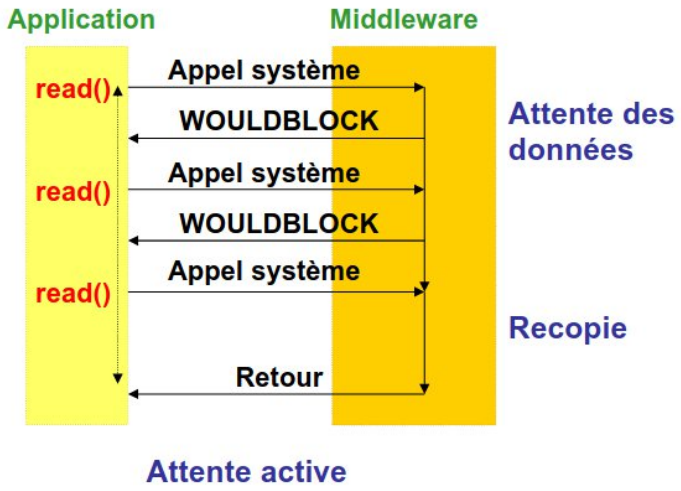
- Plusieurs sémantiques en réception, send() peut rendre la main :
  - aussitôt (send() non bloquant)
  - quand les données ont été recopiées dans le tampon d'émission local (les données peuvent être modifiées au niveau de l'application)
  - quand les données ont été recopiées dans le tampon de réception distant (le tampon d'émission local est de nouveau libre)
  - quand le destinataire a consommé les données (le tampon de réception est de nouveau libre)

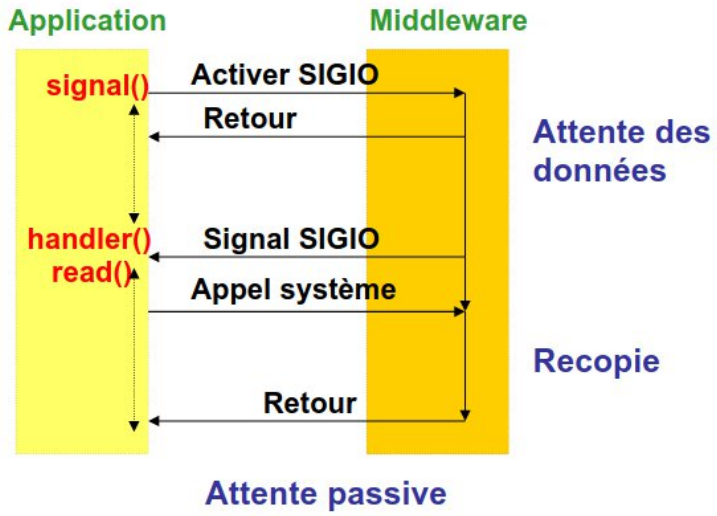
Communication par signaux

- Notification asynchrone d'un événement à un processus
- Principe : interruption logicielle quand l'événement se produit
- Le processus
  - indique les signaux qu'il souhaite capter (provoquant son interruption)
  - met en place un handler (fonction particulière) qui sera exécuté quand l'événement se produira
- Très peu de données (le numéro de signal)
- Utilisée pour le contrôle (arrêt, rechargement de configuration)
- À manipuler avec précaution (réentrance)

Communication par signaux

- Notification asynchrone d'un événement à un processus
- Principe : interruption logicielle quand l'événement se produit
- Le processus
  - indique les signaux qu'il souhaite capter (provoquant son interruption)
  - met en place un handler (fonction particulière) qui sera exécuté quand l'événement se produira





Désigner l'émetteur et le destinataire

- Comment nommer l'autre processus ?
- Localement : PID, nom de tube
- En réseau : adresse de socket (IP + port)
- Résolution de noms : DNS, getaddrinfo()

Les sockets

Les sockets

- Interface de programmation entre application et transport
  - Les sockets sont des portes d'entrées/sorties vers le réseau (la couche transport)
- Deux processus communiquent en émettant et recevant des données via les sockets
- Une socket = un « fichier virtuel » (read / write / close)
- Indépendante du protocole et de la version d'IP
- API standard (POSIX/BSD), disponible partout

Les sockets - adressage

- Le serveur écoute sur un port fixe et connu
- Le port du client est attribué automatiquement
- Adresse de socket = adresse IP (id la machine – serveur ou hôte) + port (id l'application)
- IPv4 (AF_INET) et IPv6 (AF_INET6) ; getaddrinfo() recommandé
- Les ports inférieurs à 1024 sont réservés :
  - "well-known ports" : ils permettent d'identifier les serveurs d'applications connues
  - ils sont attribués par l'IANA
- Les clients n'ont pas besoin d'utiliser des well-known ports
  - ils utilisent un port quelconque entre 1024 et 65535 à condition que le triplet <transport/@IP/port> soit unique
  - ils communiquent leur numéro de port au serveur lors de la requête (à l'établissement de la connexion TCP ou dans les datagrammes UDP)

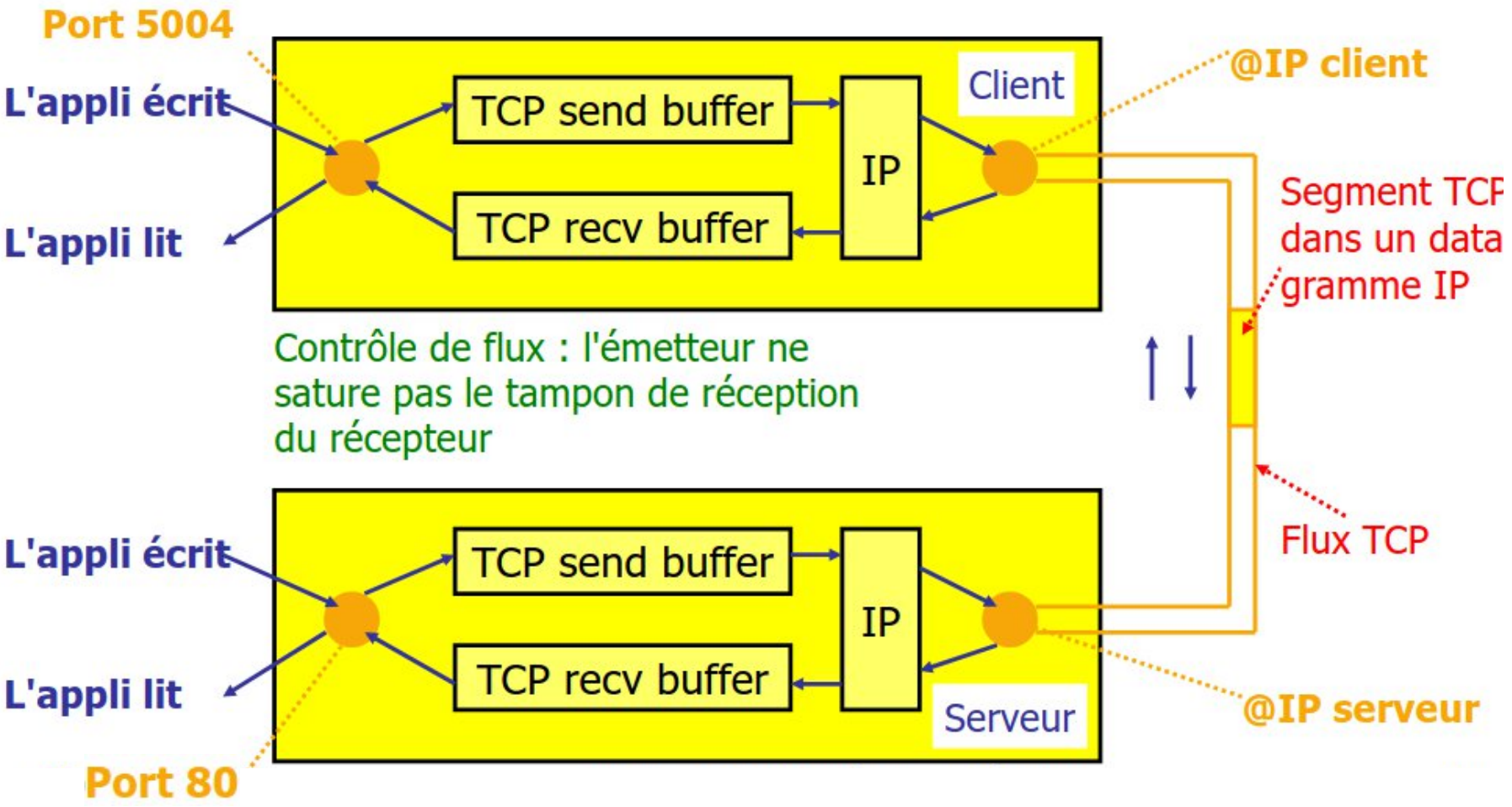
Les sockets en pratique

- Une socket est un fichier virtuel avec les opérations d'ouverture, fermeture, écriture, lecture, …
- Création : socket(domaine, type, protocole)
- type SOCK_STREAM (TCP) ou SOCK_DGRAM (UDP) ou SOCK_RAW (IP directement)
- Attacher une adresse : bind()
- Émissions/réceptions partielles possibles → boucle nécessaire

Rappel - une connexion TCP

- Connexion = (proto, @IP src, port src, @IP dst, port dst)
- Tampons d'émission et de réception gérés par le noyau
- Contrôle de flux : ne pas saturer le récepteur
- read()/write() manipulent un flux d'octets

Rappel - une connexion TCP



Mode connecté - côté serveur

- socket() → bind() → listen() → accept() (bloquant)
- accept() retourne une nouvelle socket dédiée au client
- read()/write() pour dialoguer, puis close()
- listen() fixe la taille de la file d'attente (backlog)

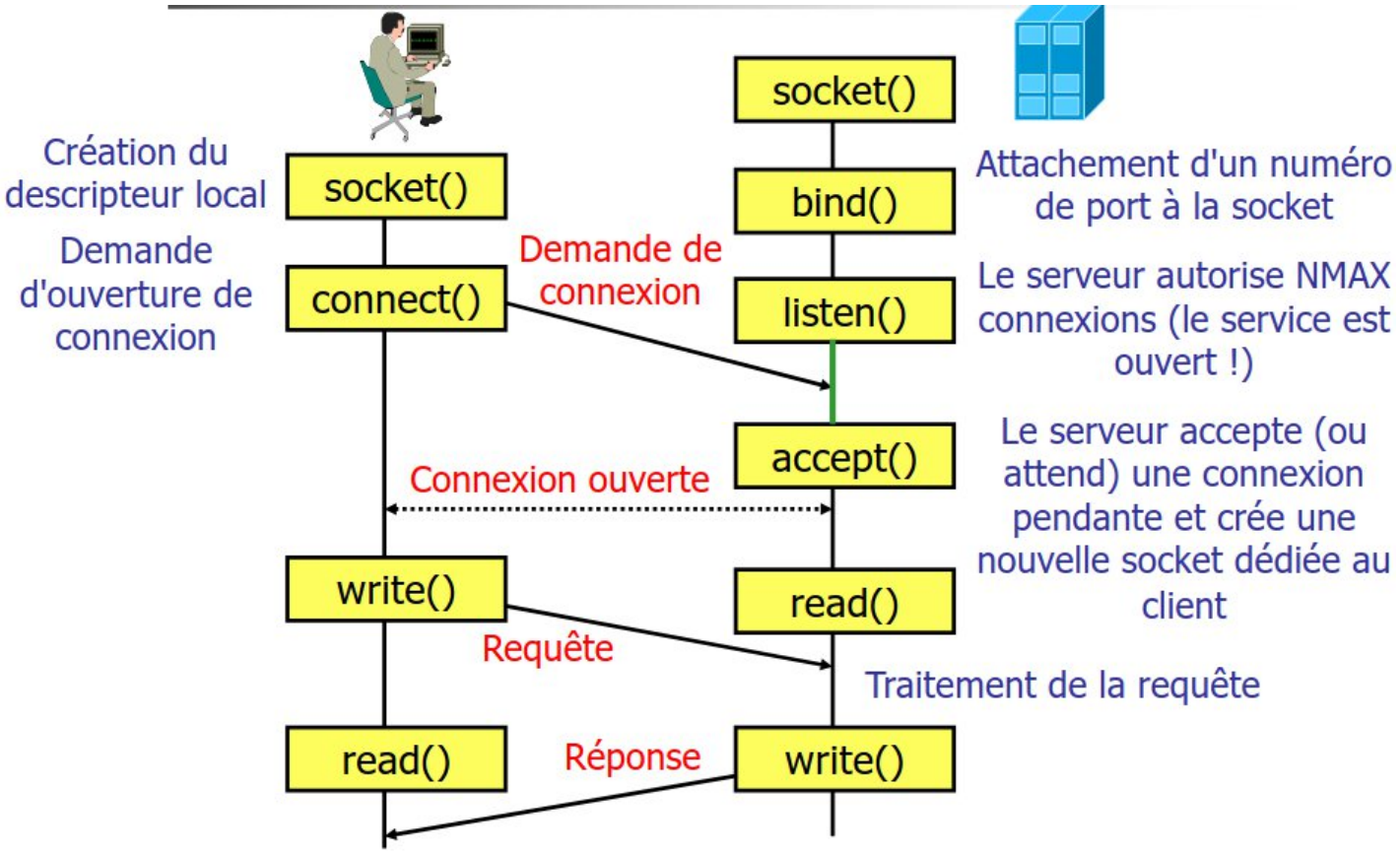
- Pour que le client puisse contacter le serveur
  - le processus serveur doit déjà tourner
  - le serveur doit avoir créé au préalable une socket pour recevoir les demandes de connexion des clients

Mode connecté - côté client

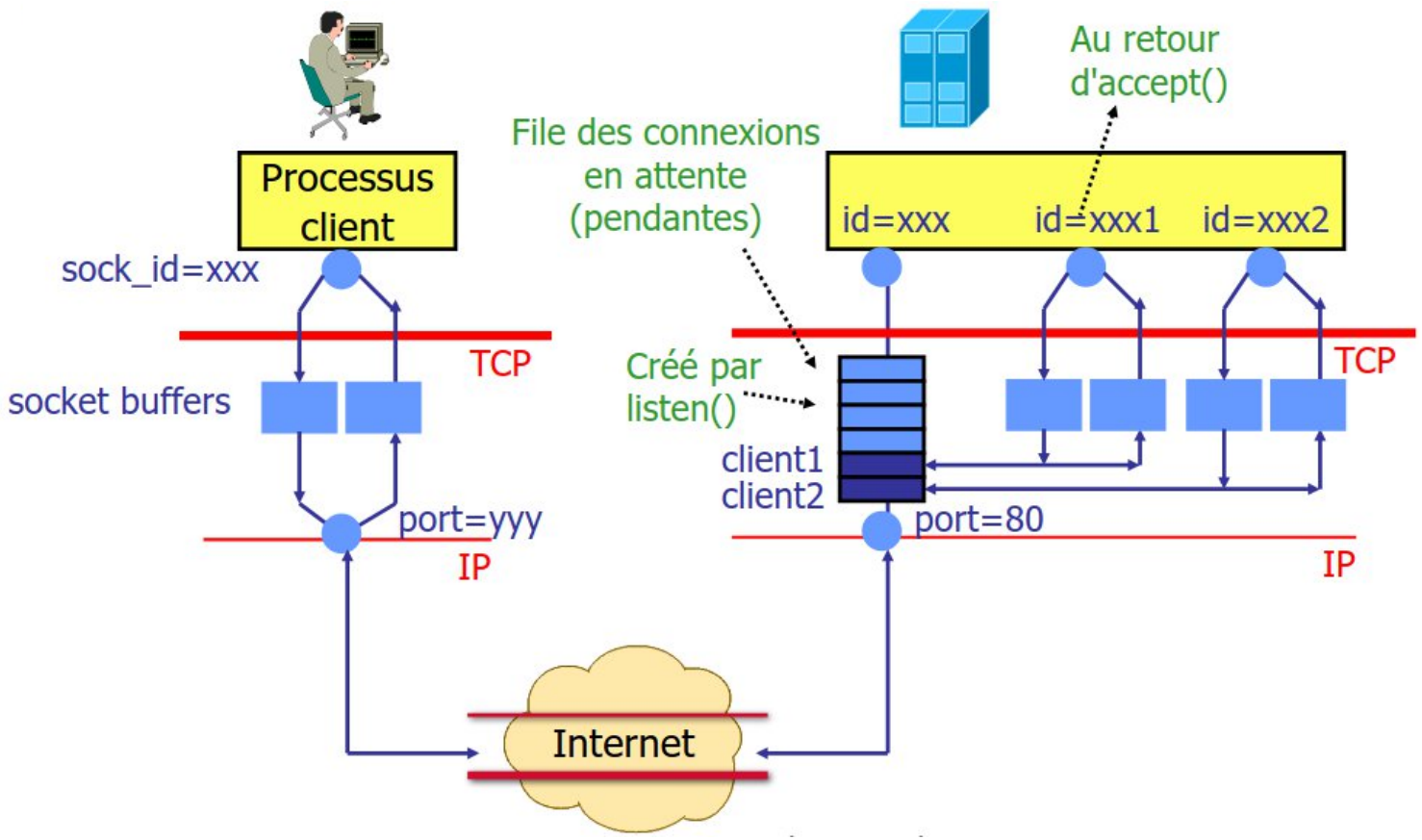
- socket() → connect() déclenche le 3-way handshake
- write()/read() pour échanger, puis close()
- Pas de bind() explicite : port attribué automatiquement
- Le flux d'octets ne préserve pas les frontières de messages

- Le client contacte le serveur
  - en créant une socket locale au client
  - en spécifiant une adresse IP et un numéro de port pour joindre le processus serveur
- Le client demande alors l'établissement d'une connexion avec le serveur
- Si le serveur accepte la demande de connexion
  - il crée une nouvelle socket permettant le dialogue avec ce client
  - permet au serveur de dialoguer avec plusieurs clients

Mode connecté (TCP) - séquence d'appels



Mode connecté (TCP) - séquence d'appels



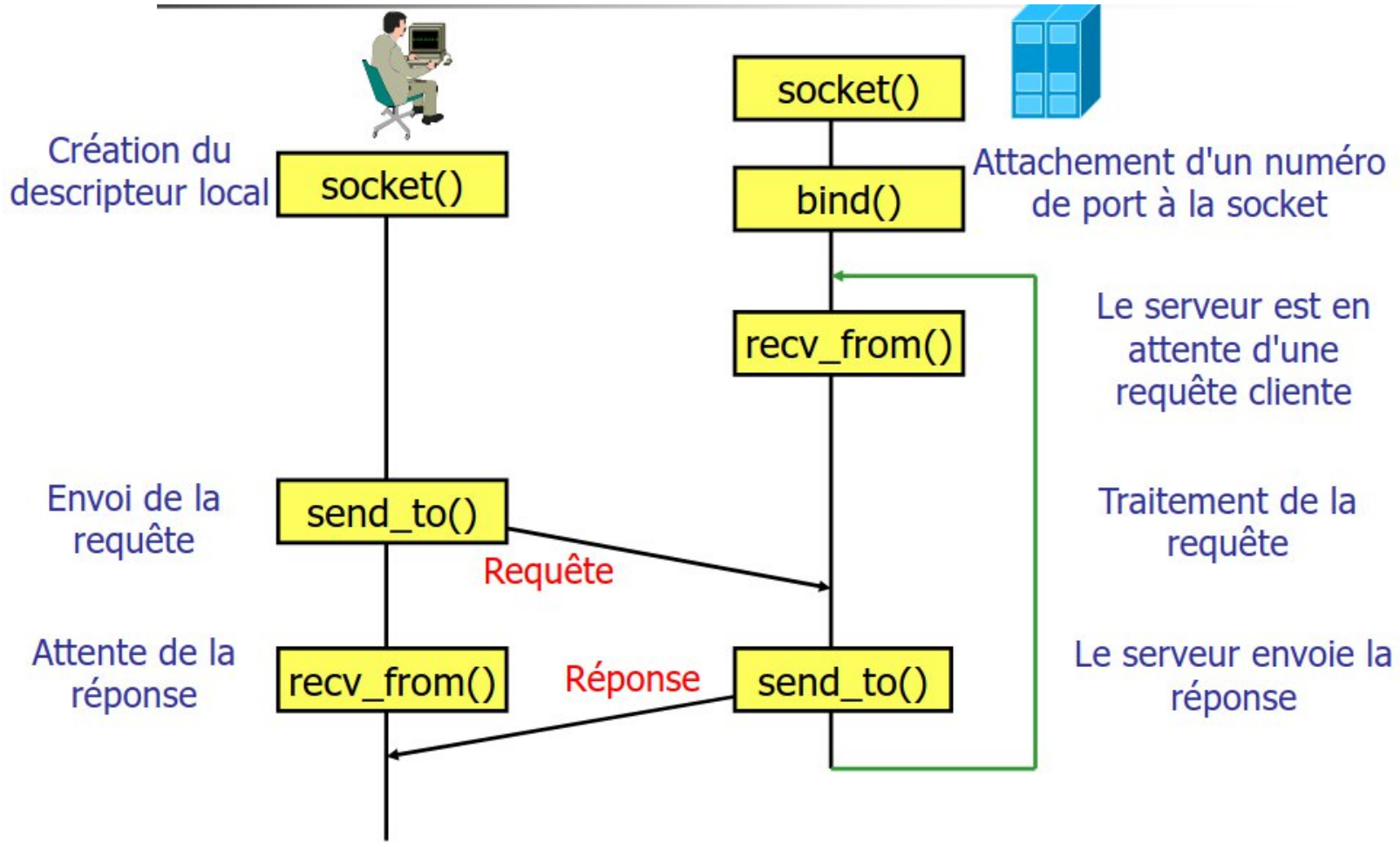
Mode non connecté - UDP

- Pour que le client puisse contacter le serveur
  - il doit connaître l'adresse de la socket du serveur
  - le serveur doit avoir créé la socket de réception
- Le client envoie sa requête en précisant, lors de chaque envoi, l'adresse de la socket destinataire
- Le datagramme envoyé par le client contient l'adresse de la socket émettrice (port, @IP)
- Le serveur traite la requête et répond au client en utilisant l'adresse de la socket émettrice de la requête

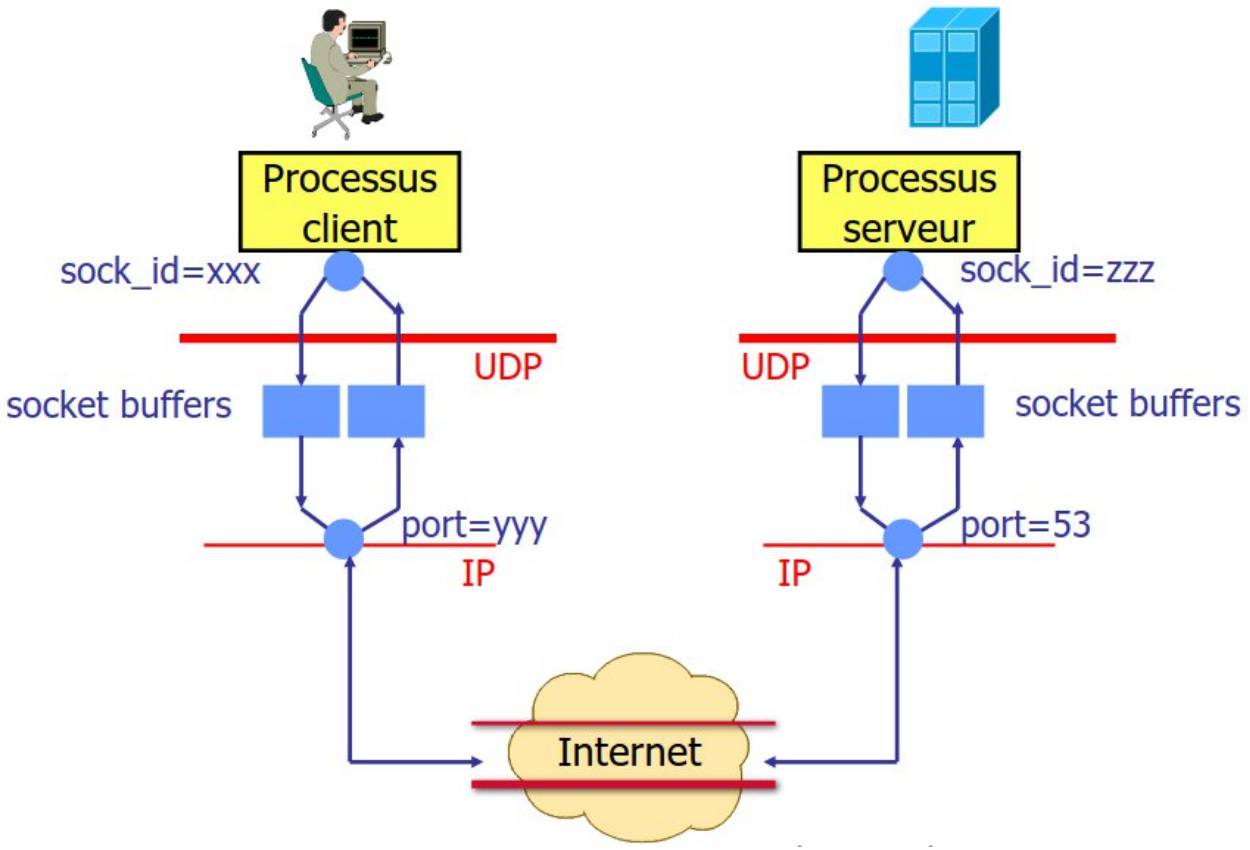
Mode non connecté - UDP

- socket() puis bind() côté serveur
- sendto() / recvfrom() : adresse fournie à chaque envoi
- Pas de connexion, chaque datagramme est autonome
- Les frontières de messages sont préservées

Mode non connecté (UDP) - séquence d'appels



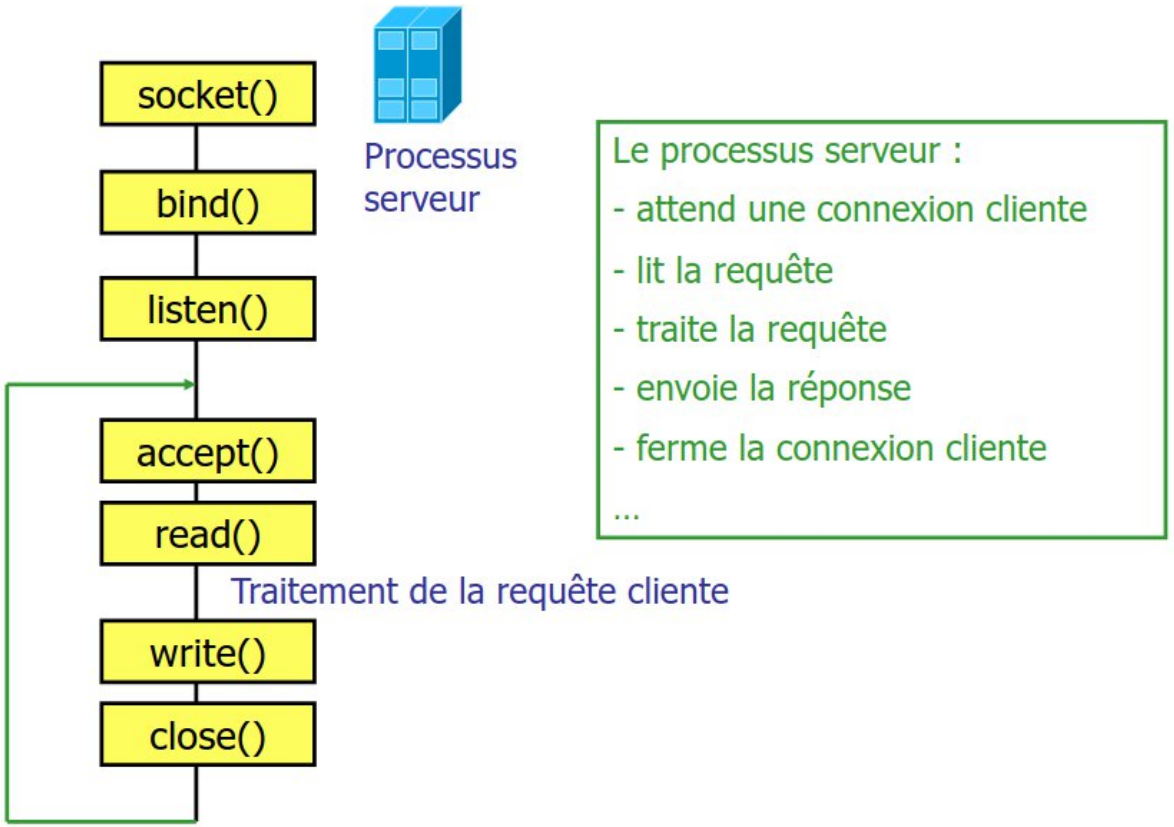
Mode non connecté (UDP) - séquence d'appels



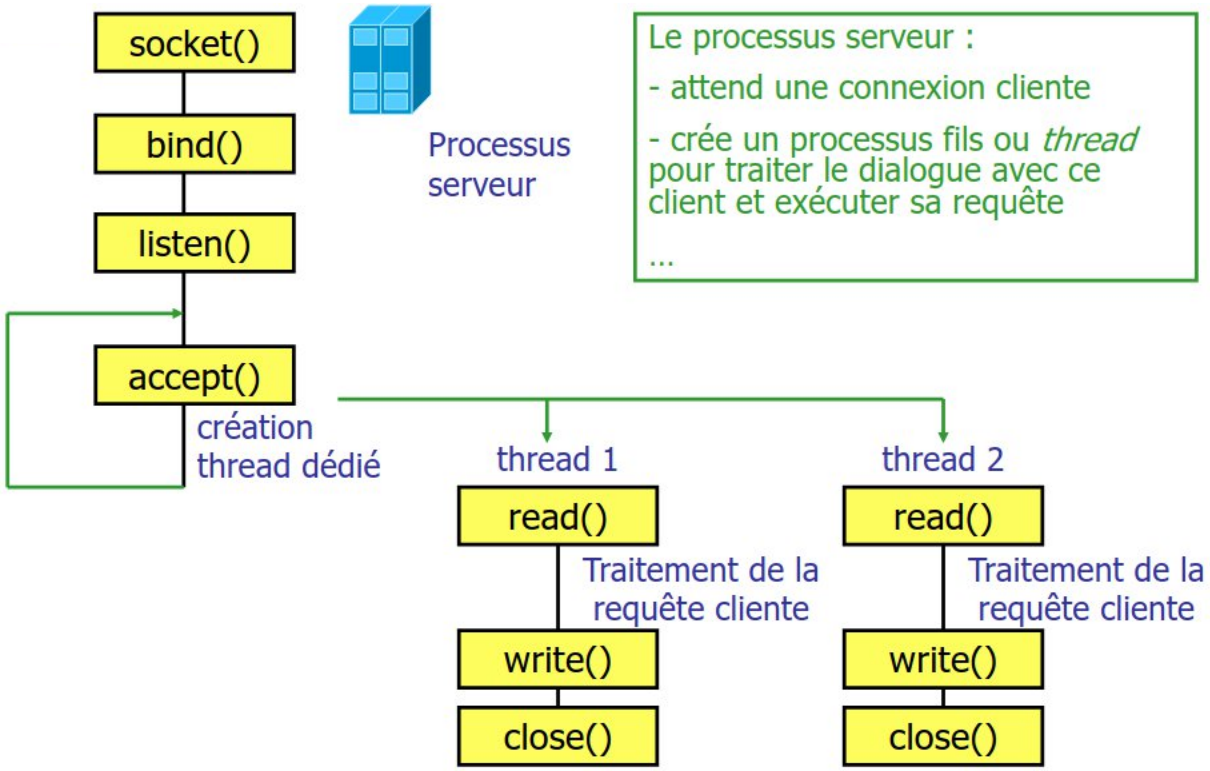
Serveur itératif vs concurrent (sockets)

- Itératif : accept() puis traitement, un client à la fois
- Concurrent : un fils par client via fork()
- Variante : threads (plus légers que fork)
- Variante : un seul processus avec select() / epoll()

Serveur itératif



Serveur itératif



Opérations bloquantes / non bloquantes

- Par défaut, les primitives connect(), accept(), send_to(), recv_from(), read(), write() sont bloquantes
  - recv() sur un tampon vide attendra l'arrivée des données pour rendre la main
  - send() sur un tampon plein attendra que les données quitte le tampon pour rendre la main
  - accept() ne rend la main qu'une fois une connexion établie (bloque si pas de connexions pendantes)
  - connect() ne rend la main qu'une fois la connexion cliente établie (sauf si pas entre listen() et accept())

Opérations bloquantes / non bloquantes

- Il est possible de paramétrer la socket lors de sa création pour rendre les opérations non bloquantes
- Comportement d'une émission non bloquante
  - tout ce qui peut être écrit dans le tampon l'est, les caractères restants sont abandonnés (la primitive retourne le nombre de caractères écrits)
  - si aucun caractère ne peut être écrit (tampon plein), retourne -1 avec errno=EWOULDBLOCK (l'application doit réessayer plus tard)
- Comportement d'une lecture non bloquante
  - s'il n'y a rien à lire dans la socket, retourne -1 ... (l'application doit réessayer plus tard

La scrutation (select / poll / epoll)

- Surveiller plusieurs sockets dans un seul processus
- Réveil dès qu'une socket est prête (lecture/écriture)
- Évite le coût d'un processus/thread par client
- epoll / kqueue : passage à l'échelle (milliers de connexions)

Paramétrage des sockets

- setsockopt() et getsockopt() : options de la socket
  - régler ou lire les options d'une socket, pour ajuster son comportement au-delà des valeurs par défaut
- SO_REUSEADDR, délais d'attente (timeouts), keep-alive
  - Sert à relancer un serveur aussitôt après son arrêt, sans l'erreur « Address already in use »
  - Keep- alive : Sert à détecter une connexion morte : sur une liaison TCP inactive, le noyau envoie des sondes ; si le pair a disparu (crash, câble débranché), la connexion est fermée et les ressources libérées
- Mode non bloquant (O_NONBLOCK)
  - Sert à ce que accept/recv/send/connect rendent la main tout de suite au lieu de bloquer
- Tailles des tampons d'émission/réception

Serveurs multi-protocoles et multi-services

Les serveurs multi-protocoles

- Offrir le même service en TCP et en UDP
- Une socket par protocole, surveillées par select()
- Choix du transport selon le besoin (fiabilité vs latence)
  - certains systèmes ferment tout accès à UDP pour des raisons de sécurité (pare-feu)
  - non duplication des ressources associées au service (corps du serveur)
- Exemple : DNS (UDP par défaut, TCP pour les grandes réponses)

Les serveurs multi-protocoles

- Un seul processus utilisant des opérations non bloquantes de manière à gérer les communications à la fois en mode connecté et en mode non-connecté
- Deux implémentations possibles :
  - En mode itératif
  - En mode concurrent

Les serveurs multi-protocoles – mode itératif

- Le serveur ouvre la socket UDP et la socket TCP puis boucle sur des appels non bloquants à accept() et recv_from() sur chacune des sockets
  - si une requête TCP arrive
    - le serveur utilise accept() provoquant la création d'une nouvelle socket servant la communication avec le client
    - lorsque la communication avec le client est terminée, le serveur ferme la socket "cliente" et réitère son attente sur les deux sockets initiales
  - si une requête UDP arrive
    - le serveur reçoit et émet des messages avec le client (pas de nouvelle socket)
    - lorsque les échanges sont terminés, le serveur réitère son attente sur les deux sockets initiales

Les serveurs multi-protocoles – mode concurrent

- Un automate gère l'arrivée des requêtes (primitives non bloquantes)
- Création d'un nouveau processus fils pour toute nouvelle connexion TCP
- Traitement de manière itérative des requêtes UDP
  - elles sont traitées en priorité
  - pendant ce temps, les demandes de connexion sont mises en attente

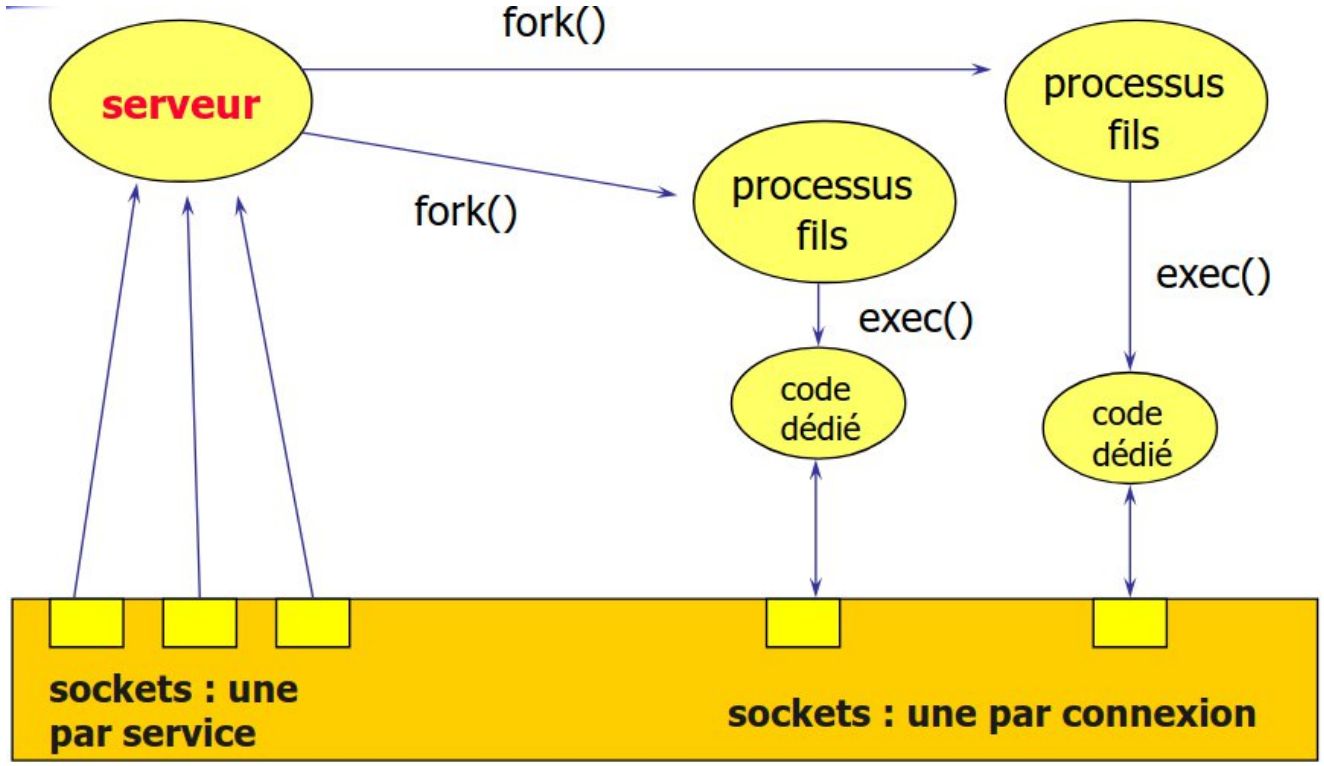
- Pour UDP
  - Pas de changement de traitement concurrent ou itératif (juste de possible compétition avec TCP)
- Pour TCP
  - Itératif : gérer les requêtes une à une
  - Concurrent : Le processus père crée des processus fils (et donc parallèlise)

Les serveurs multi-services

- Un même processus offre plusieurs services
- Une socket d'écoute par service
- select() pour aiguiller les requêtes
- Mutualise les ressources
- Avantages
  - Le code réalisant les services n'est présent que lorsqu'il est nécessaire
  - La maintenance se fait sur la base du service et non du serveur : l'administrateur peut facilement activer ou désactiver un service

Les serveurs multi-services - Fonctionnement

- Le serveur ouvre une socket par service offert, attend une connexion entrante sur l'ensemble des sockets ouvertes
- Lorsqu'une demande de connexion arrive, le serveur crée un processus fils qui prend en compte la connexion
- Le processus fils exécute (via exec() sur système UNIX) un programme dédié réalisant le service demandé

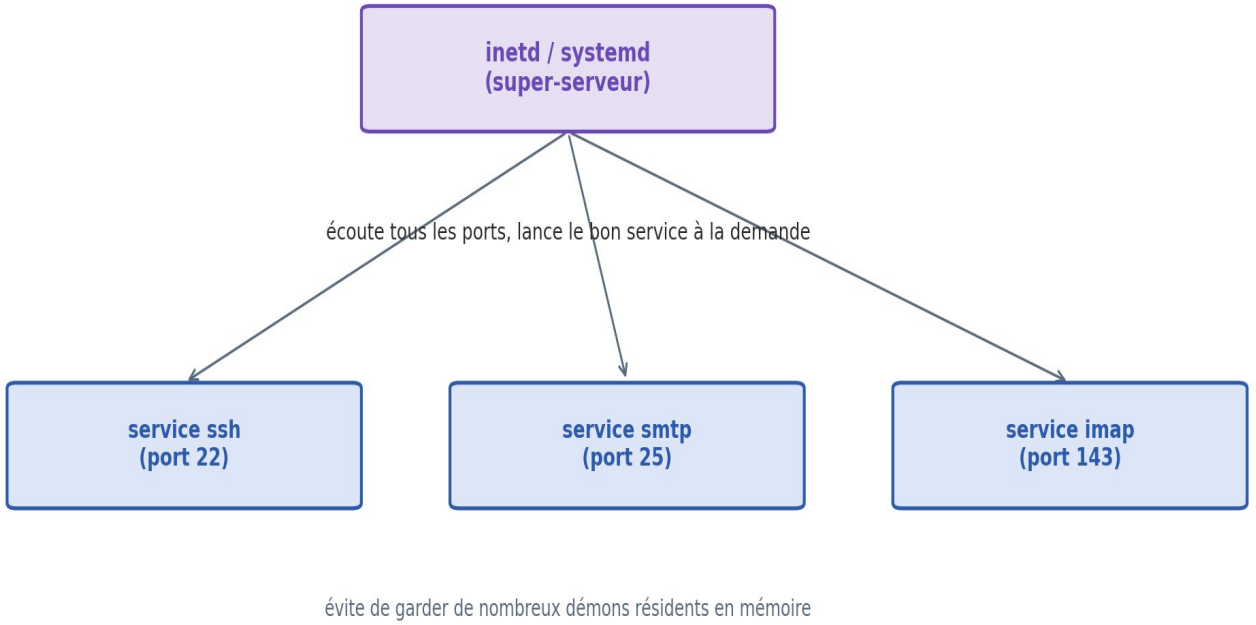


Les processus démons

- L'invocation d'un service Internet standard (FTP, TELNET, RLOGIN, SSH, …) nécessite la présence côté serveur d'un processus serveur qui tourne en permanence et qui est en attente des requêtes clientes
- Service résident, détaché du terminal
- Démarré au boot et surveillé par le système
- Aujourd'hui géré par systemd (unités, sockets, journal)
- Journalisation centralisée (journald / syslog)

Le super-serveur (inetd / systemd)

- Un super serveur
  - un processus multi-services multi-protocoles
  - un serveur unique qui reçoit les requêtes
  - activation des services à la demande
  - permet d'éviter d'avoir un processus par service, en attente de requêtes
  - une interface de configuration (fichier inetd.conf) permettant à l'administrateur système d'ajouter ou retirer de nouveaux services sans lancer ou arrêter un nouveau processus
- Le processus inetd attend les requêtes à l'aide de la primitive select() et crée un nouveau processus pour chaque service demandé (excepté certains services UDP qu'il traite lui-même)



Le super-serveur géré par systemd

```
  UNIT                                LOAD   ACTIVE SUB       DESCRIPTION          >
  acpid.socket                        loaded active running   ACPID Listen Socket
  avahi-daemon.socket                 loaded active running   Avahi mDNS/DNS-SD Sta>
  cups.socket                         loaded active running   CUPS Scheduler
  dbus.socket                         loaded active running   D-Bus System Message >
  docker.socket                       loaded active running   Docker Socket for the>
  nordvpnd.socket                     loaded active running   NordVPN Daemon Socket
  snapd.socket                        loaded active running   Socket activation for>
  syslog.socket                       loaded active running   Syslog Socket
  systemd-fsckd.socket                loaded active listening fsck to fsckd communi>
  systemd-initctl.socket              loaded active listening initctl Compatibility>
  systemd-journald-audit.socket       loaded active running   Journal Audit Socket
  systemd-journald-dev-log.socket     loaded active running   Journal Socket (/dev/>
  systemd-journald.socket             loaded active running   Journal Socket
  systemd-rfkill.socket               loaded active listening Load/Save RF Kill Swi>
  systemd-udevd-control.socket        loaded active running   udev Control Socket
  systemd-udevd-kernel.socket         loaded active running   udev Kernel Socket
  uuidd.socket                        loaded active listening UUID daemon activatio>

LOAD   = Reflects whether the unit definition was properly loaded.
ACTIVE = The high-level unit activation state, i.e. generalization of SUB.
SUB    = The low-level unit activation state, values depend on unit type.
17 loaded units listed. Pass --all to see loaded but inactive units, too.
```

inetd et son successeur

- inetd : un seul démon écoute tous les ports
- Il lance le service approprié à la demande
- Évite de garder de nombreux démons résidents en mémoire
- Remplacé aujourd'hui par l'activation par socket de systemd

Les appels de procédures distantes (RPC)

Deux approches de conception

- Orientée messages (sockets) : on échange des messages
- Orientée appels de procédures (RPC) : on appelle une fonction distante
- Les RPC masquent le réseau au programmeur
- Base des middlewares modernes

Principe général des RPC

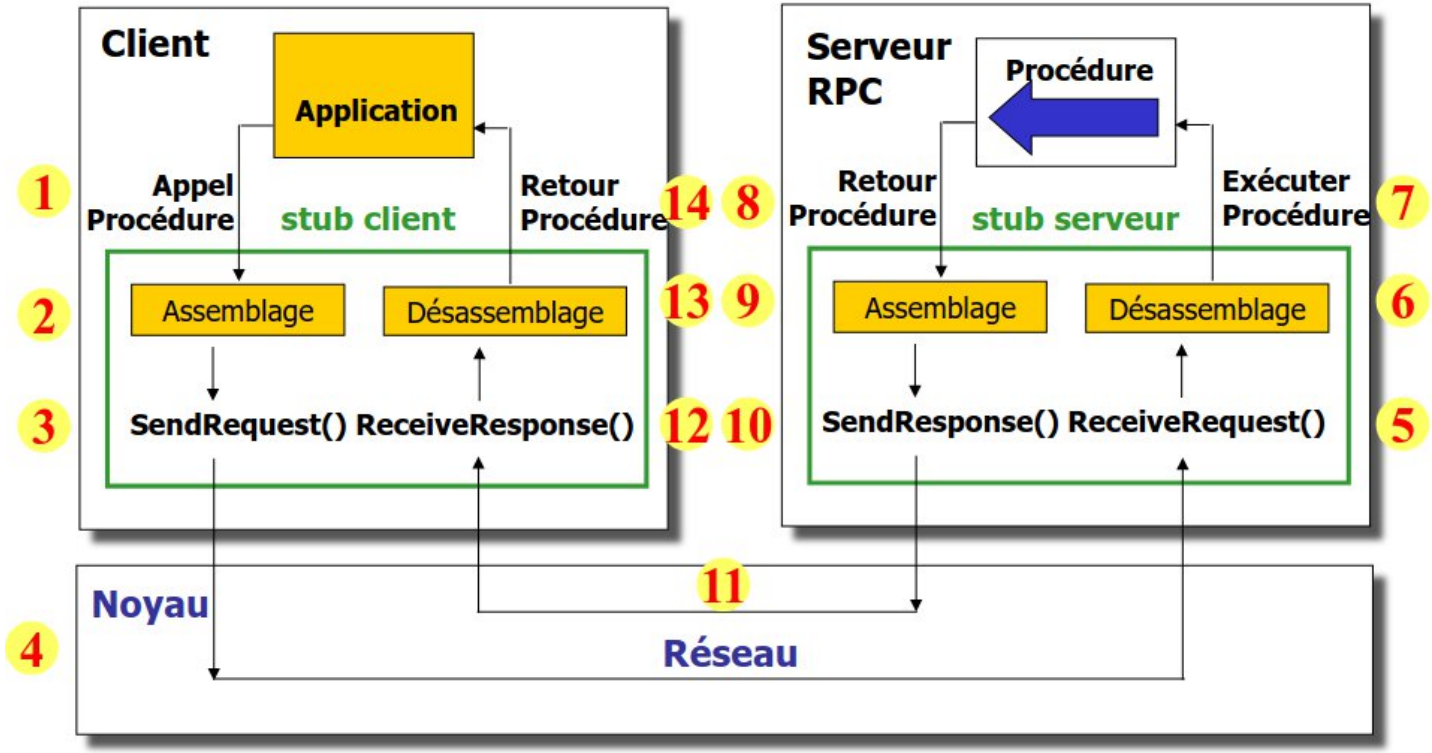
- Un programme appelle une procédure exécutée sur une autre machine
- L'appel distant ressemble à un appel local
- Réseau et sérialisation cachés par des « stubs »
- Paramètres passés par valeur (pas d'adresses)

- RPC - Remote Procedure Call
  - permettre à un processus de faire exécuter une fonction par un autre processus se trouvant sur une machine distante
  - se traduit par l'envoi d'un message contenant l'identification de la fonction et les paramètres
  - une fois le traitement terminé, un message retourne le résultat de la fonction à l'appelant

Principe général des RPC

- L'objectif des RPC est de faire en sorte qu'un appel distant ressemble le plus possible à un appel local
- Le processus client (l'appelant) est lié à une petite procédure de bibliothèque, appelée stub client, qui représente la procédure du serveur dans l'espace d'adressage du client
- Le processus serveur (l'exécutant) est lié à un stub serveur qui représente l'exécution du client
- Dissimule le fait que l'appel de la procédure n'est pas local : le programmeur de l'application utilise un appel de procédure "normal" !

Le modèle RPC (stubs)



Le modèle RPC

- Stub client : empaquette les paramètres (marshalling)
- Transport via TCP ou UDP
- Stub serveur : dépaquette, exécute, renvoie le résultat
- Une IDL décrit l'interface et génère les stubs

Restrictions liées aux RPC

- Pas de passage de paramètres par adresse / pointeur
  - En effet, les espaces d'adressage du client et du serveur sont différents donc aucun sens de passer une adresse
- Gestion des pannes réseau et des délais
- Sémantique d'exécution (au plus une / au moins une fois)
- Hétérogénéité des représentations de données

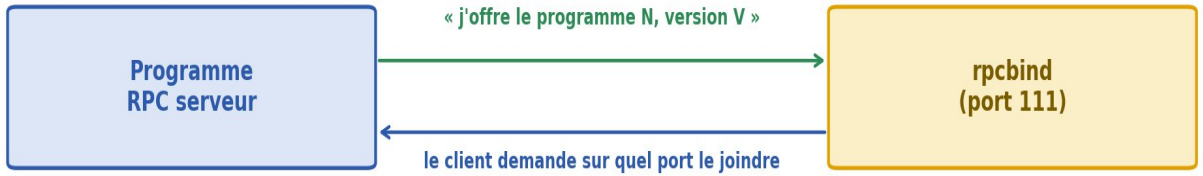
L'implémentation Sun RPC (ONC RPC)

- Technologie historique de Sun Microsystems
- Identification par (programme, version, procédure)
- UDP ou TCP pour les communications
- Représentation des données : XDR
- Toujours présente sous NFS, mais datée

- Les Sun RPC :
  - Le format des messages que l'appelant (stub client) émet pour déclencher la procédure distante sur un serveur
  - Le format des arguments de la procédure
  - Le format des résultats de la procédure

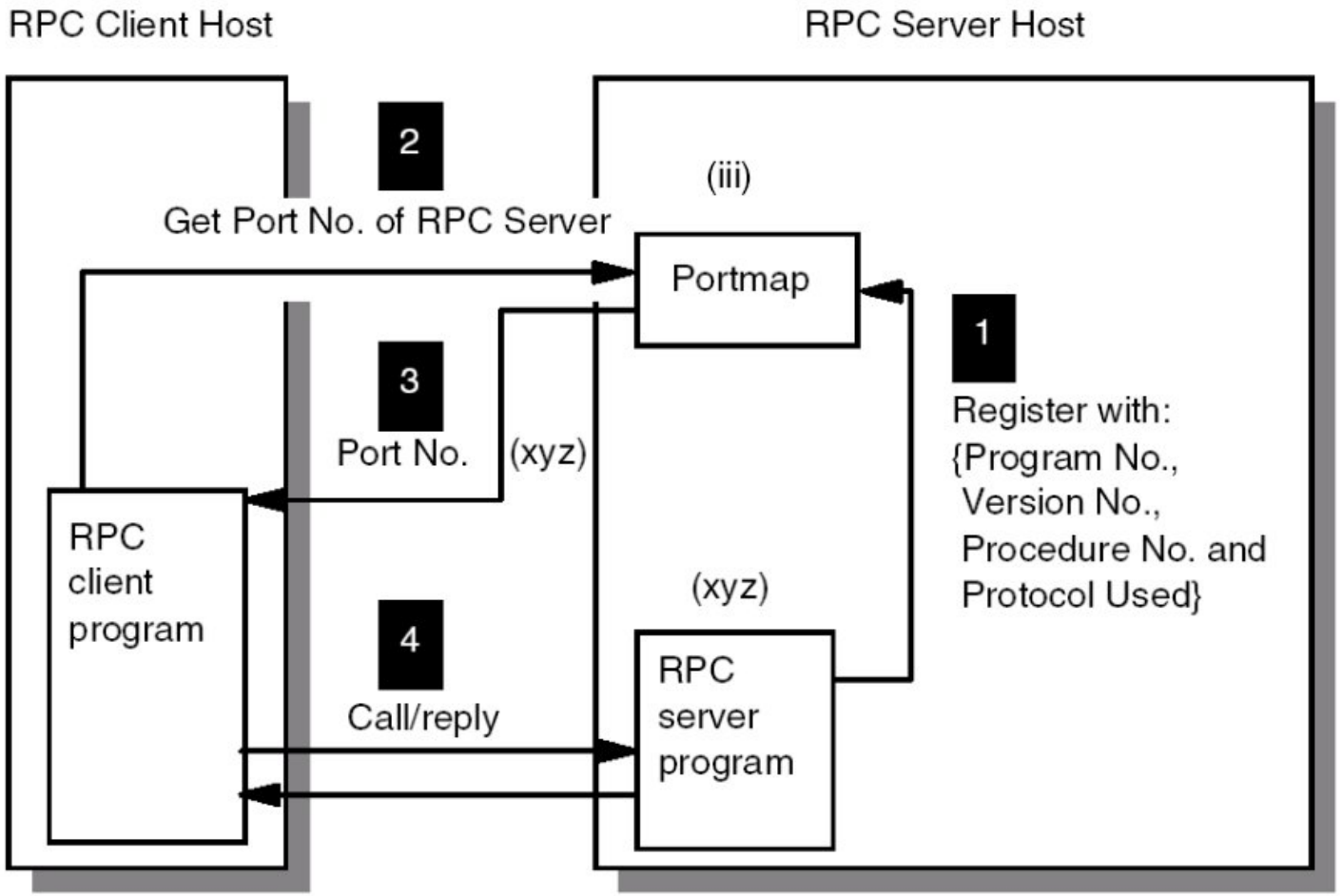
Le port mapper (rpcbind)

- Lorsqu'un programme RPC (serveur) démarre, il alloue dynamiquement un numéro de port local, contacte le port mapper de la machine sur laquelle il s'exécute, puis informe ce dernier de l'association
- lorsqu'un client désire contacter un programme RPC sur une machine distante, il interroge d'abord le port mapper de cette machine pour connaître le port de communication associé au service RPC
- le port mapper est lui même un programme RPC (100000) mais il est le seul à utiliser un port alloué statiquement : le port 111/UDP et le port 111/TCP



port dynamique du service résolu via le port fixe 111

Le port mapper (rpcbind)



Identification des procédures distantes

- Un programme distant correspond à un serveur avec ses procédures et ses données propres
- Chaque programme distant est identifié par un entier unique codé sur 32 bits utilisé par l'appelant
  - Chaque procédure a un numéro (0 = test / ping)
- Une procédure distante est identifiée par le triplet (program, version, procedure)
  - program identifie le programme distant
  - version identifie la version du programme
  - procedure identifie la procédure

- Le service s'enregistre auprès de rpcbind (port 111)
- Le client résout le port dynamique via rpcbind

Identification des procédures distantes

- Un programme distant correspond à un serveur avec ses procédures et ses données propres
- Chaque programme distant est identifié par un entier unique codé sur 32 bits utilisé par l'appelant
  - Chaque procédure a un numéro (0 = test / ping)

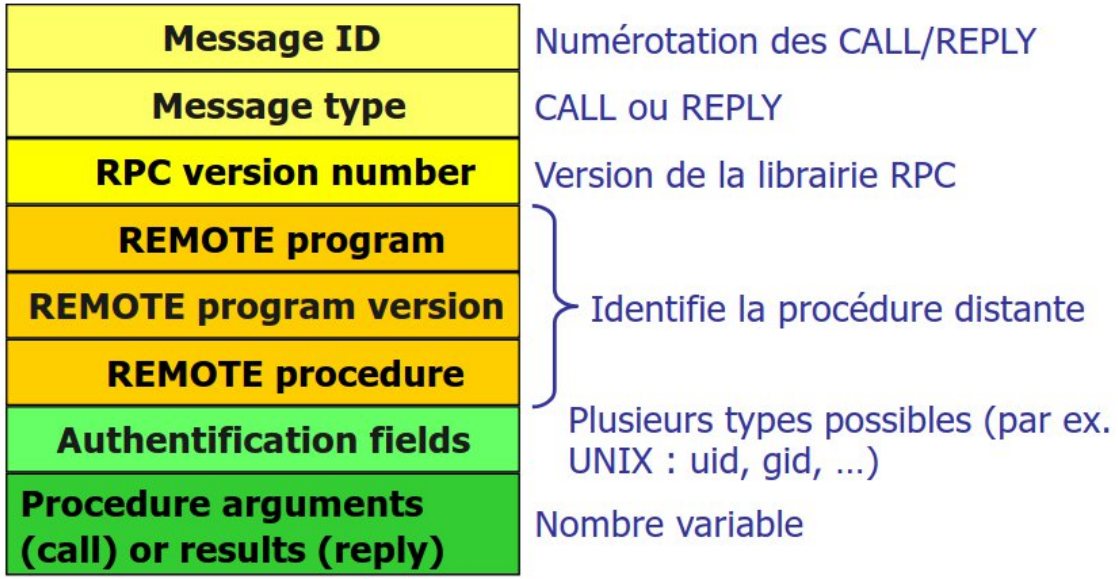
Nom	Identifiant	Description
portmap	100000	port mapper
rstat	100001	rstat, rup, perfmeter
ruserd	100002	remote users
nfs	100003	Network File System
ypserv	100004	Yellow pages (NIS)
mountd	100005	mount, showmount
dbxd	100006	debugger
ypbind	100007	NIS binder
etherstatd	100010	Ethernet sniffer
pcnfs	150001	NFS for PC

La sémantique « au moins une fois »

- En UDP, une requête peut être renvoyée après expiration d'un délai
- La procédure peut donc être exécutée plusieurs fois
- Convient aux opérations idempotentes
- Le concepteur doit en tenir compte

Format des messages RPC

- Message d'appel (CALL) et de réponse (REPLY)
- Identifiant de transaction (XID) pour apparier appel et réponse
- Réponses : succès, programme/procédure inconnu, erreur d'authentification
- Authentification optionnelle (AUTH_SYS, etc.)



La représentation XDR

- Format pivot pour échanger des données hétérogènes
- Taille et codage fixés (ex. entier sur 4 octets, big-endian)
- Sérialise entiers, chaînes, structures, tableaux
- Autres types de représentations : Protocol Buffers, JSON, MessagePack

- Pourquoi XDR ?
  - répond au problème d'échange d'informations typées ou structurées entre deux machines hétérogènes dans la représentation locale des données
  - exemple : un entier de 32 bits ayant la valeur 260 sera représenté par :
    - 00000104h sur une machine de type " big endian" c'est à dire avec les Most Significant Bytes ayant les adresses basses et les LSB ayant les adresses hautes
    - 40100000h sur une machine de type " little endian"
  - il faut adopter une représentation réseau et convertir sur les extrémités les représentations locales en représentation réseau et vice-versa

La représentation XDR

Type	Taille	Description
int	32 bits	entier signé de 32 bits
unigned int	32 bits	entier non signé de 32 bits
bool	32 bits	valeur booléenne (0 ou 1)
enum	arb.	type énuméré
hyper	64 bits	entier signé de 64 bits
unsigned hyper	64 bits	entier non signé de 64 bits
float	32 bits	virgule flot. simple précision
double	64 bits	virgule flot. double précision
opaque	arb.	donnée non convertie (sans type)
fixed array	arb.	tableau de longueur fixe de n'importe quel autre type
structure	arb.	agrégat de données
discriminated union	arb.	structure implémentant des formes alternatives
symbolic constant	arb.	constante symbolique
void	0	utilisé si pas de données
string	arb.	chaîne de car. ASCII

Les RPC sous UNIX (pratique)

- Fichier de description .x (langage IDL)
- rpcgen génère les stubs client/serveur et les routines XDR
- La commande rpcinfo interroge le rpcbind d'une machine
  - permet de s'assurer de la disponibilité d'un service RPC particulier (exécution de la procédure 0 du service)
    - rpcinfo -u host prog_num [ver_num] (UDP)
    - rpcinfo -t host prog_num [ver_num] (TCP)
- NFS reste l'usage le plus courant

```
etc/rpc
# This file contains user readable names that can be used in place of rpc
# program numbers.

portmapper      100000  portmap sunrpc
rstatd          100001  rstat rstat_svc rup perfmeter
rusersd         100002  rusers
nfs             100003  nfsprog
ypserv          100004  ypprog
mountd          100005  mount showmount
```

Des RPC classiques à gRPC

- gRPC : RPC moderne (Google), sur HTTP/2
- IDL = Protocol Buffers (.proto), multi-langages
- Streaming, TLS intégré, hautes performances
- Alternative : API REST (HTTP/JSON), plus simple à exposer

Exercices

Lecture bloquante / non bloquante

- Lire exactement 100 octets sur une socket (mode connecté)
- Décrire l’laogirhme correspondant et les avantes/inconvénients :
  - Cas d'une lecture complètement bloquante (read attend tout)
  - Cas standard des sockets (« au moins 1 octet »)
  - Cas d'une lecture non bloquante (EWOULDBLOCK) — avantages/inconvénients

Analyse d'une application Client/Serveur

- Quel service ? combien d'arguments au client ?
- Quel port serveur ? mode connecté ou non ? justifier
- Rôle des constantes BUF_SIZE et QUEUE_SIZE
- Le serveur passe-t-il à l'échelle ? proposer une solution

Un mini-inetd

- Compléter la page de manuel et le schéma algorithmique
- Rôle et signature de la fonction tcp_listen()
- Différences et similitudes entre mini-inetd et inetd
- Étendre pour gérer plusieurs couples (port, programme)