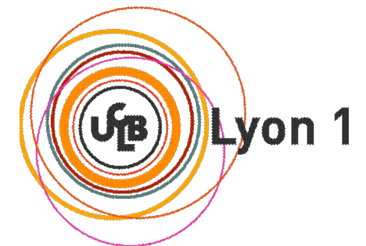
Partie 1 : Architecture et communications Client/Serveur

Olivier GLÜCK

Université LYON 1/Département Informatique

Olivier.Gluck@univ-lyon1.fr

http://perso.univ-lyon1.fr/olivier.gluck



Copyright

- Copyright © 2023 Olivier Glück; all rights reserved
- Ce support de cours est soumis aux droits d'auteur et n'est donc pas dans le domaine public. Sa reproduction est cependant autorisée à condition de respecter les conditions suivantes :
  - Si ce document est reproduit pour les besoins personnels du reproducteur, toute forme de reproduction (totale ou partielle) est autorisée à la condition de citer l'auteur.
  - Si ce document est reproduit dans le but d'être distribué à des tierces personnes, il devra être reproduit dans son intégralité sans aucune modification. Cette notice de copyright devra donc être présente. De plus, il ne devra pas être vendu.
  - Cependant, dans le seul cas d'un enseignement gratuit, une participation aux frais de reproduction pourra être demandée, mais elle ne pourra être supérieure au prix du papier et de l'encre composant le document.
  - Toute reproduction sortant du cadre précisé ci-dessus est interdite sans accord préalable écrit de l'auteur.

Remerciements

- Certains transparents sont basés sur des supports de cours de :
  - Olivier Aubert (LYON 1)
  - Bénédicte Le Grand (UPMC)
- Des figures sont issues des livres cités en bibliographie

Plan de la première partie

- Organisation pratique et contenu du module
- Bibliographie
- Quelques rappels : Internet et le modèle TCP/IP
- Architecture Client/Serveur
- Communications inter-processus
- Les sockets
- Les appels de procédures distantes

Organisation pratique et contenu du module

Lyon 1

Le module AdminSR

AdminSR : 19,5h de cours + 24h TP (Admin. Unix et Windows)

- Travaux pratiques :
  - Salles Réseaux : TPR1, TPR2, TPR3 (Linux/Windows 2000)
    - pas d'accès extérieur
    - possibilité de câblage
    - root sur les machines
- Evaluation : un contrôle final, des notes de CC

Le module AdminSR : objectifs

- Former des administrateurs systèmes et réseaux
  - → connaître le modèle Client/Serveur (90% des applications de l' Internet)
  - → avoir des notions de conception d'applications Client/Serveur
  - → connaître les protocoles applicatifs de l'Internet et savoir mettre en place les services associés sous Linux **et** sous Windows

Le module AdminSR : contenu (1)

- Modèle Client/Serveur et applications
  - Architecture et communication de type Client/Serveur
    - Modèle Client/Serveur, middleware
    - Conception d'une application Client/Serveur
    - Les modes de communication entre processus
    - Les sockets TCP/IP
    - Les serveurs multi-protocoles et multi-services
    - Les appels de procédures distantes, l'exemple des RPC

Le module AdminSR : contenu (2)

- Applications Client/Serveur sur TCP/IP
  - Connexions à distance (telnet, rlogin, ssh, X11, …)
  - Transfert de fichiers et autres (FTP, TFTP, NFS, SMB)
  - Gestion d’utilisateurs distants (NIS)
  - Le courrier électronique (POP, IMAP, SMTP, WebMail)
  - Les serveurs de noms (DNS)
  - Un annuaire fédérateur (LDAP)
  - Le web, protocole HTTP, serveur apache, caches Web
  - L’administration de réseaux et le protocole SNMP
- Les architectures pour le calcul et les communications distribuées (s’il reste du temps)

Le module AdminSR : contenu (3)

- Administration système et réseaux des technologies Windows NT (NT4, 2000, 2003 et XP) :
  - Architecture en Domaines
  - Gestion des utilisateurs (Active Directory)
  - Profils errants, stratégie de groupe
  - Système de fichiers et sécurité
  - Services réseaux
  - Scripts, base de registre
  - Gestion des disques (partitions et raid)
  - Sauvegardes et surveillance d'un parc, cluster

Bibliographie

- « *Réseaux* », 4ième édition, Andrew Tanenbaum, Pearson Education, ISBN 2-7440-7001-7
- « La communication sous Unix », 2ième édition, Jean-Marie Rifflet, Ediscience international, ISBN 2-84074-106-7
- « *Analyse structurée des réseaux* », 2ième édition, J. Kurose et K. Ross, Pearson Education, ISBN 2-7440-7000-9
- « *TCP/IP Illustrated Volume 1, The Protocols* », W. R. Stevens, Addison Wesley, ISBN 0-201-63346-9
- « *TCP/IP, Architecture, protocoles, applications* », 4ième édition, D. Comer, Dunod, ISBN 2-10-008181-0
- Internet…
  - http://www.w3.org/
  - http://www.rfc-editor.org/ (documents normatifs dans TCP/IP)

Quelques rappels : Internet et le modèle TCP/IP

Lyon 1

Le visage de l'Internet (1)

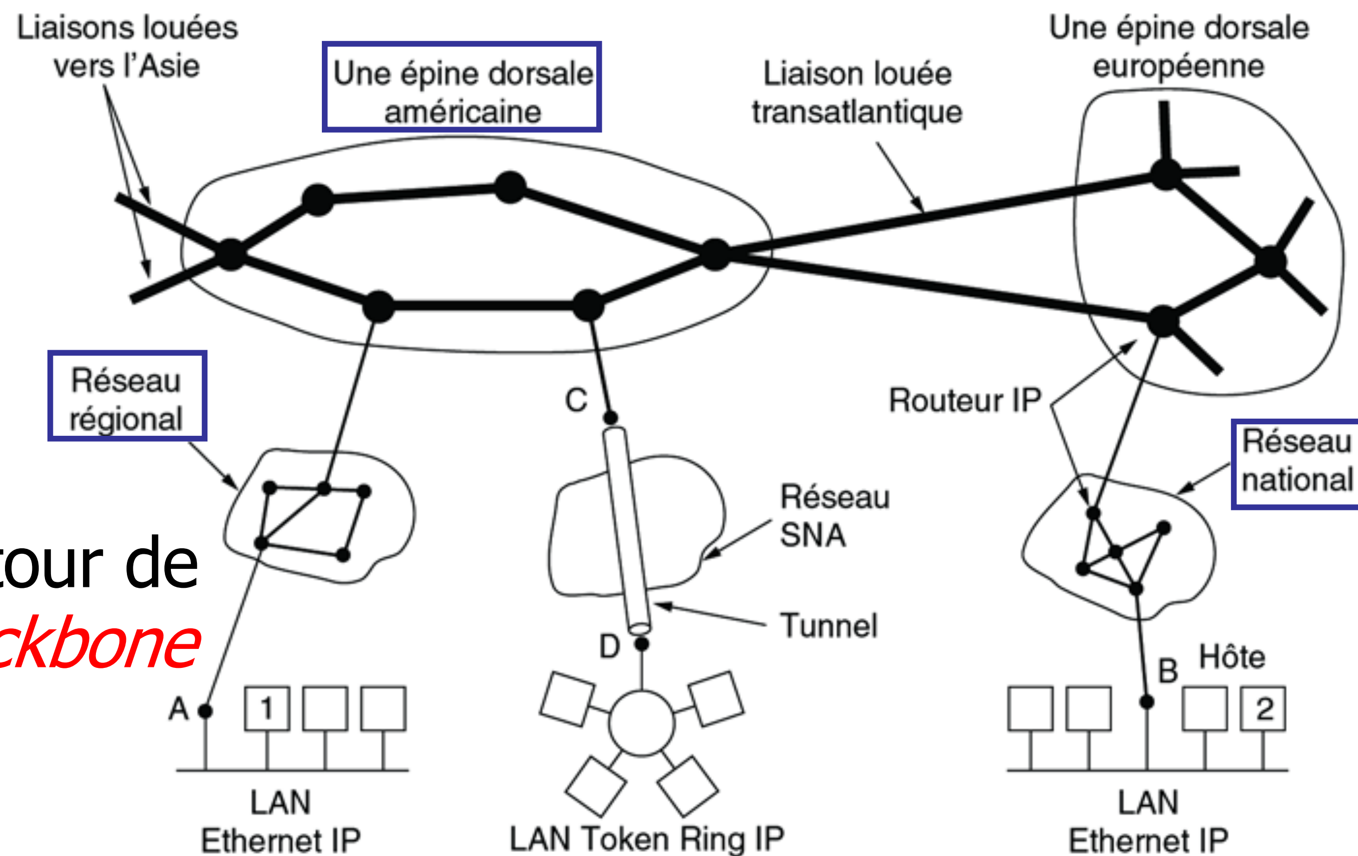
- Un réseau de réseaux
- Un ensemble de logiciels et de protocoles
- Basé sur l'architecture TCP/IP
- Fonctionne en mode Client/Serveur
- Offre un ensemble de **services** (e-mail, transfert de fichiers, connexion à distance, WWW, …)
- Une somme « d'inventions » qui s'accumulent
  - mécanismes réseau de base (TCP/IP)
  - gestion des noms et des adresses
  - des outils et des protocoles spécialisés
  - le langage HTML

- Une construction à partir du « bas »
  - réseau local (laboratoire, département)
  - réseau local (campus, entreprise)
  - réseau régional
  - réseau national
  - réseau mondial
- 3 niveaux d'interconnexion
  - postes de travail (ordinateur, terminal...)
  - liaisons physiques (câble, fibre, RTC...)
  - routeurs (équipement spécialisé, ordinateur...)

Le visage de l'Internet (3)

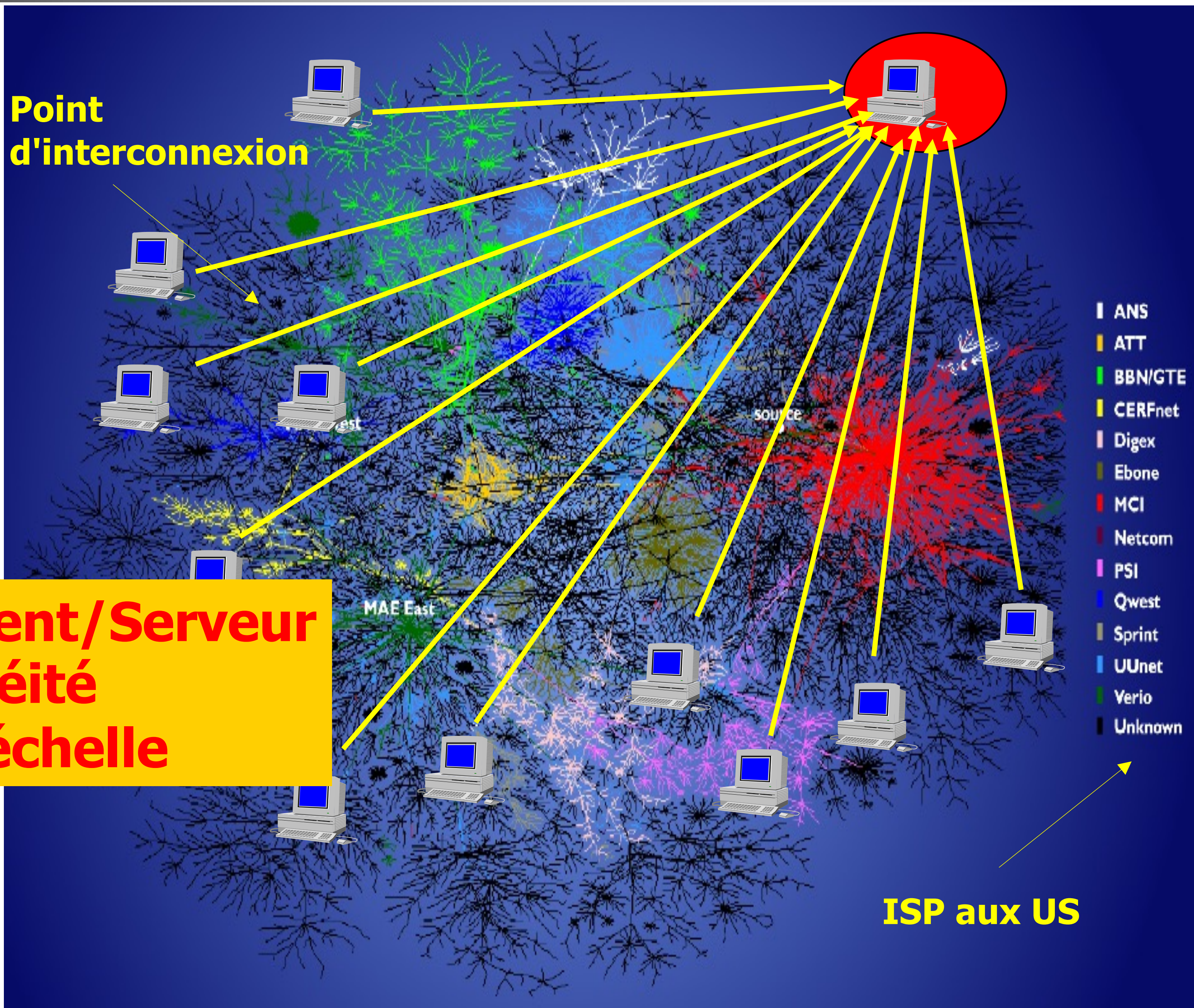
- Un ensemble de sous-réseaux indépendants (*Autonomous System*) et hétérogènes qui sont interconnectés (organisation hiérarchique)

S'articule autour de plusieurs *backbone*



© Pearson Education France

Le visage de l'Internet (4)



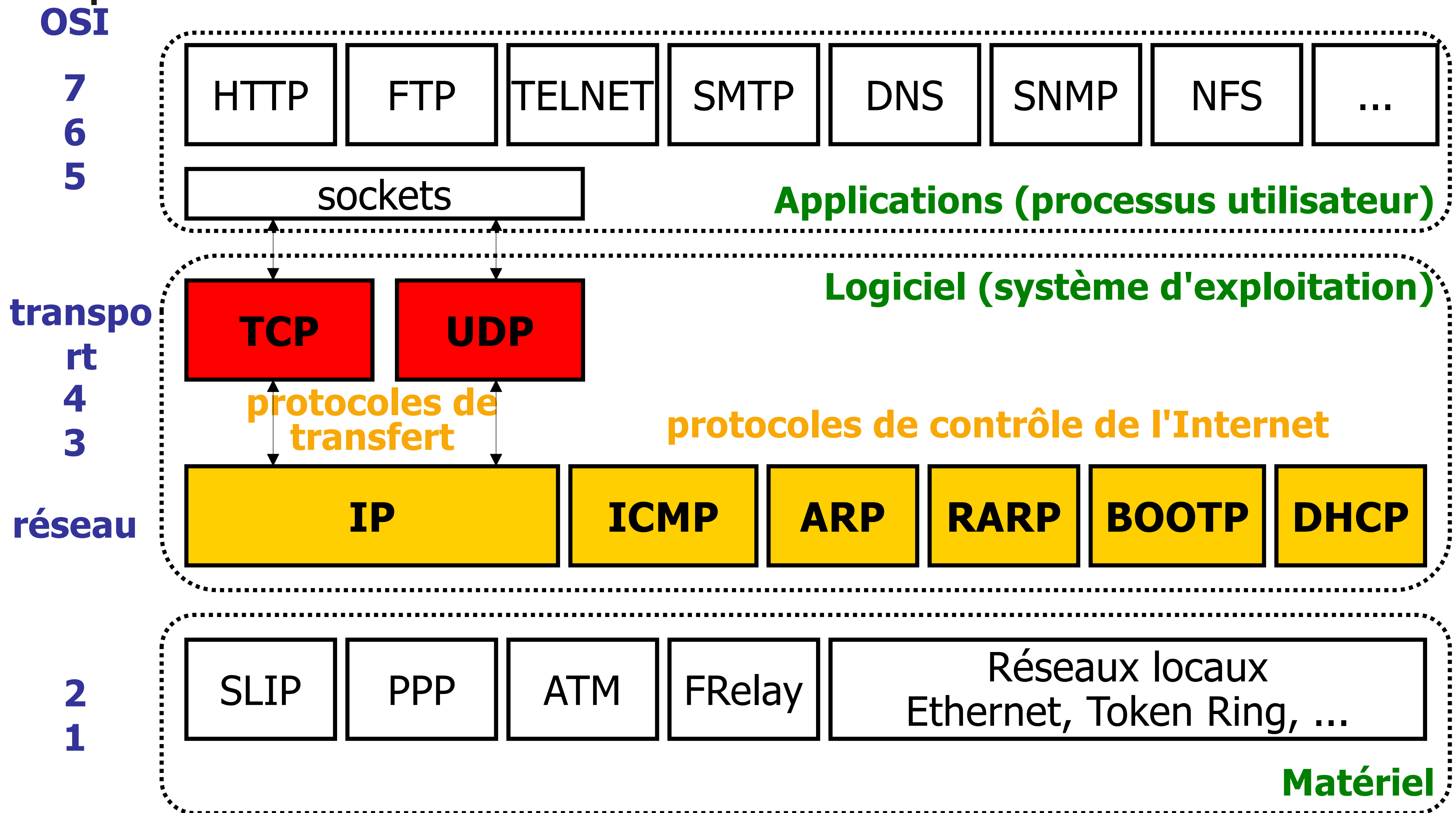
Modèle Client/Serveur
Hétérogénéité
Facteur d'échelle

L'architecture de TCP/IP (1)

- Une version simplifiée du modèle OSI
  - Application	FTP, WWW, telnet, SMTP, …
  - Transport	TCP, UDP (entre 2 processus aux extrémités)
    - TCP : transfert fiable de données en mode connecté
    - UDP : transfert non garanti de données en mode non connecté
  - Réseau	IP (routage)
  - Physique	transmission entre 2 sites

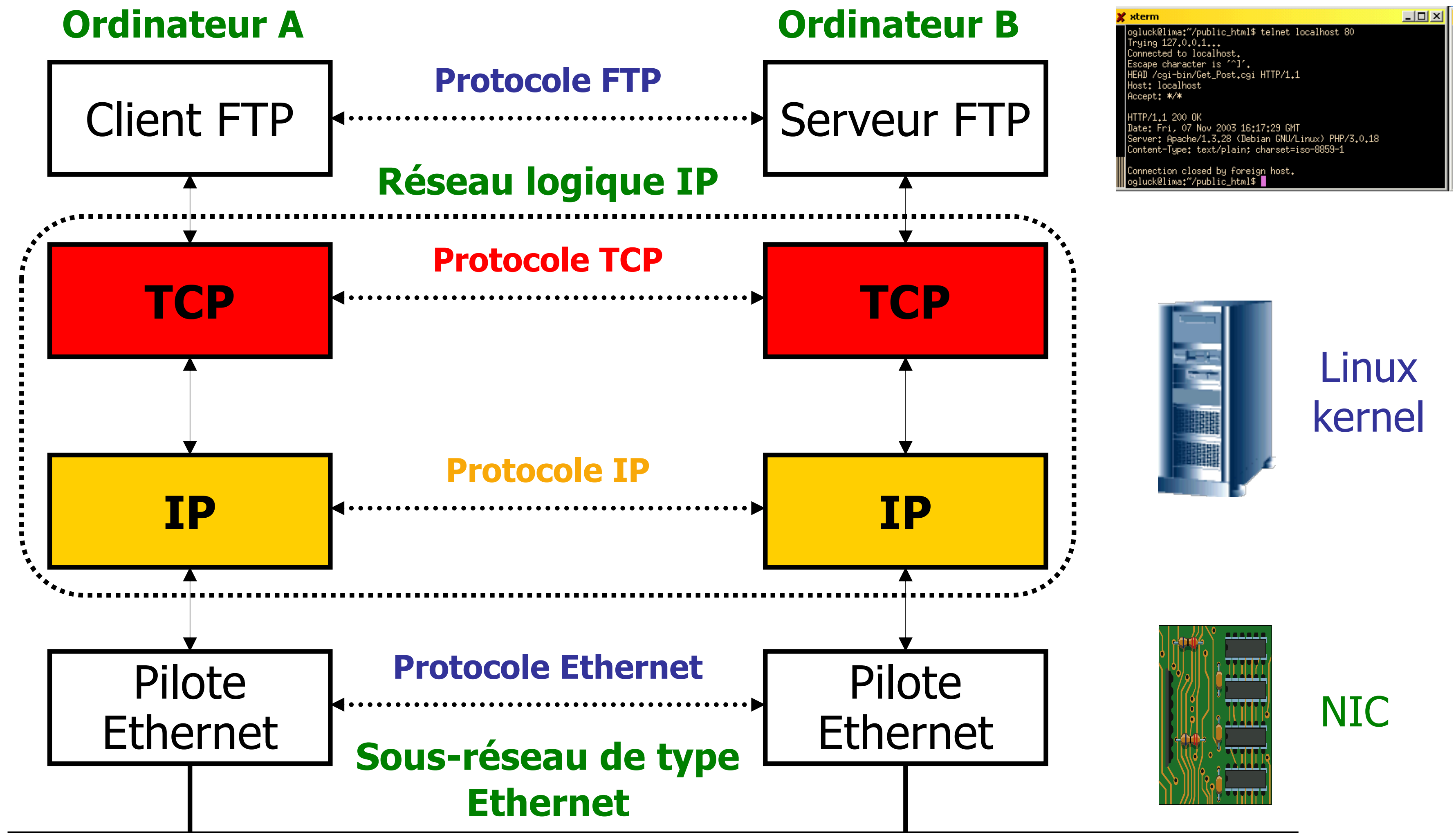
TCP	*Transport Control Protocol*
UDP	*User Datagram Protocol*
IP	*Internet Protocol*

L'architecture de TCP/IP (2)



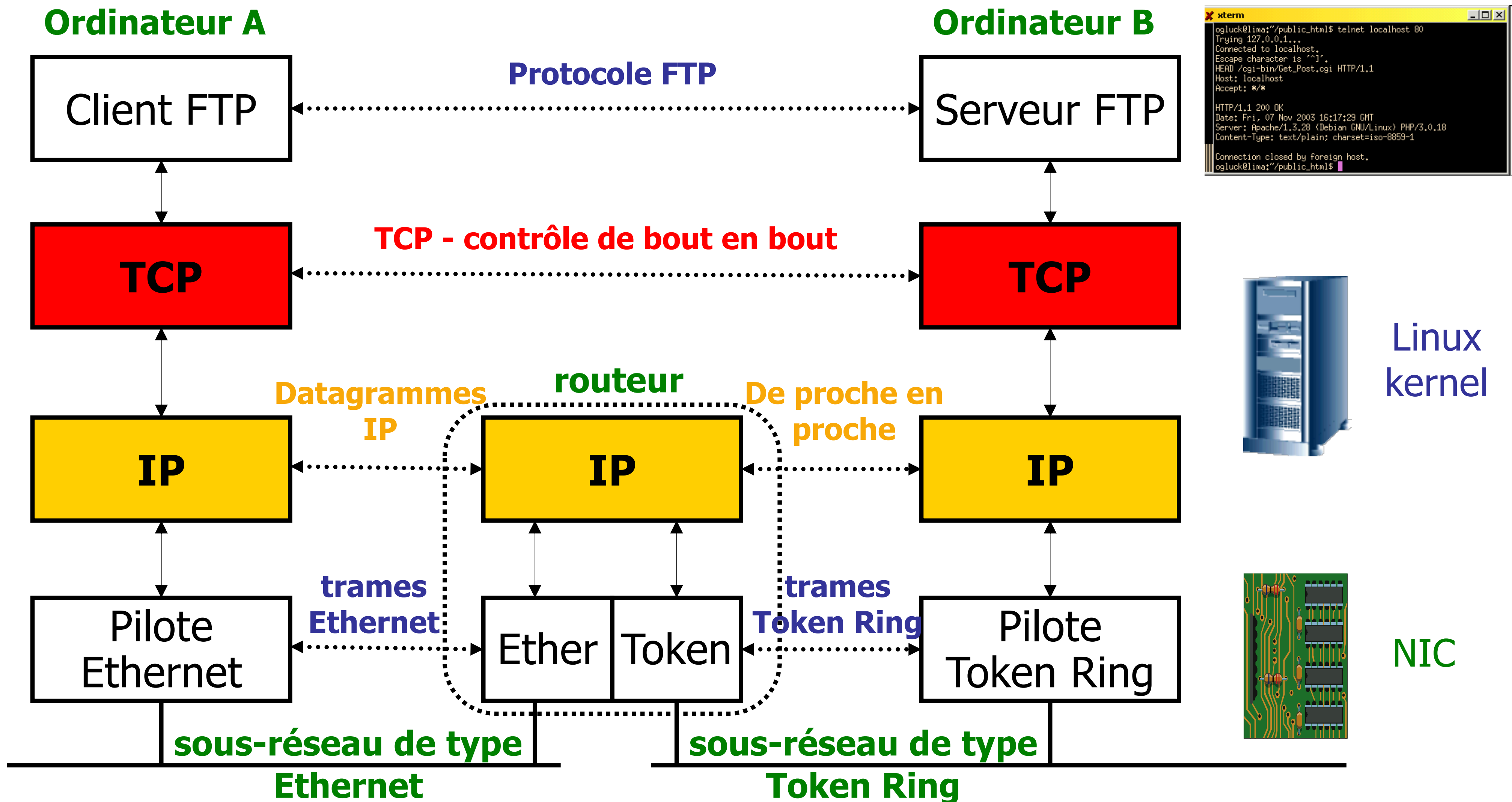
L’architecture de TCP/IP (3)

- Deux machines sur un même sous réseau IP



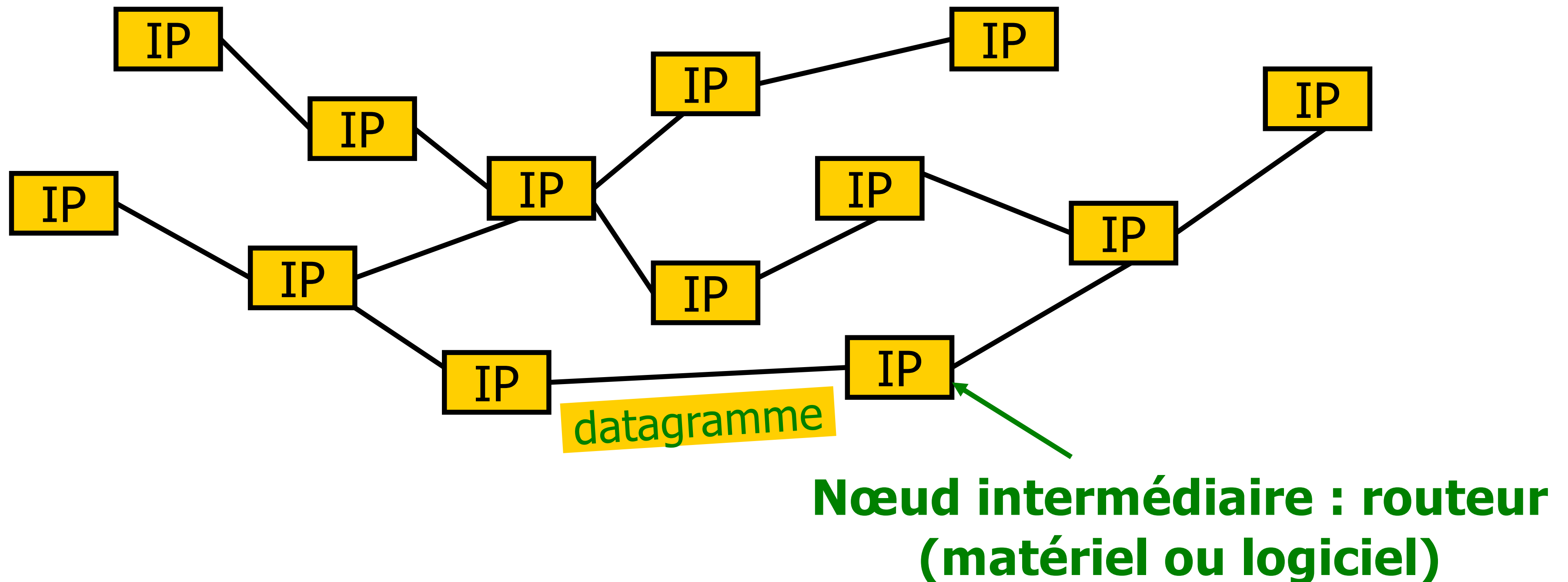
L’architecture de TCP/IP (4)

- Prise en compte de l'hétérogénéité



L'architecture de TCP/IP (5)

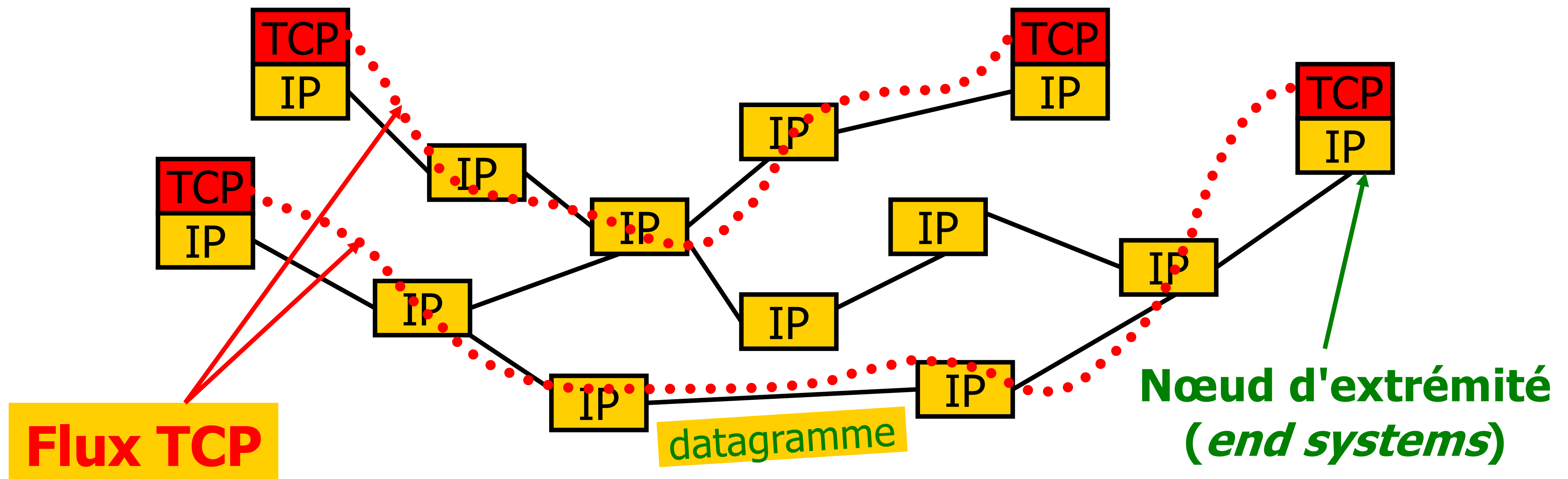
Couche réseau : communications entre machines



- IP - protocole d'interconnexion, best-effort
  - acheminement de **datagrammes** (mode non connecté)
  - peu de fonctionnalités, pas de garanties
  - simple mais robuste (défaillance d'un nœud intermédiaire)

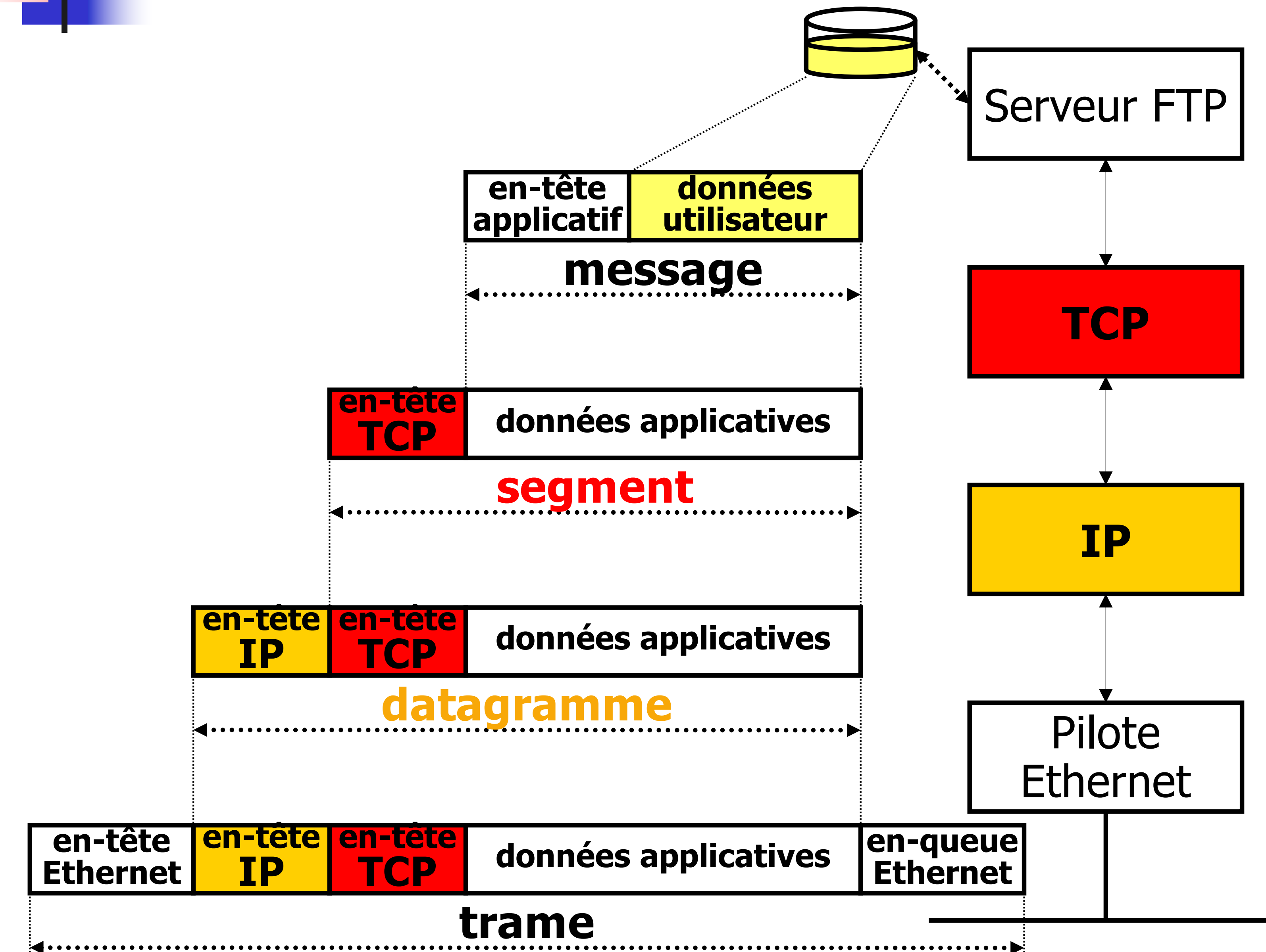
L'architecture de TCP/IP (6)

Couche transport : communications entre applis

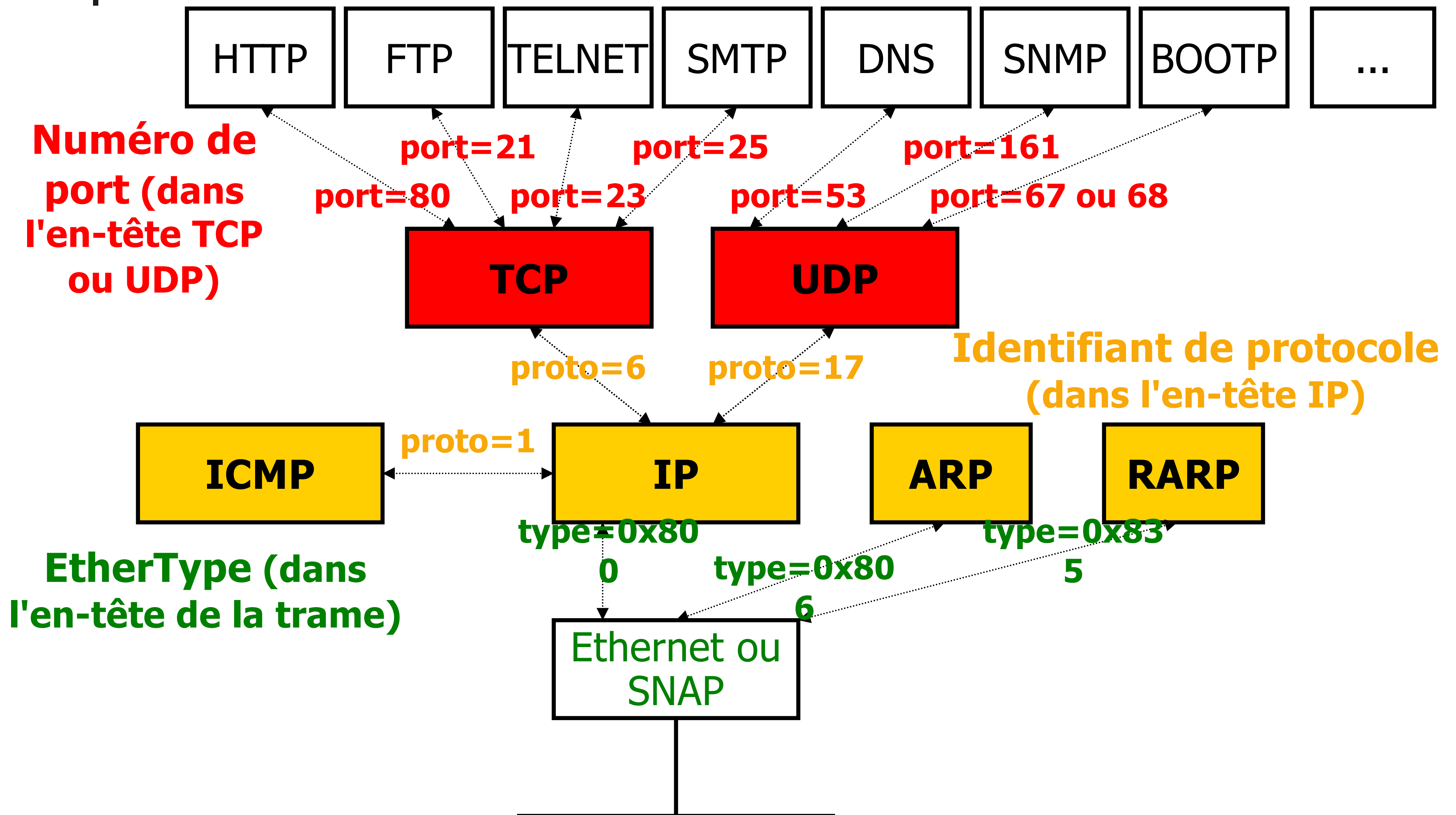


- TCP - protocole de transport **de bout en bout**
  - uniquement présent aux extrémités
  - transport **fiable** de **segments** (mode connecté)
  - protocole complexe (retransmission, gestion des erreurs, séquencement, …)

L' architecture de TCP/IP (7)



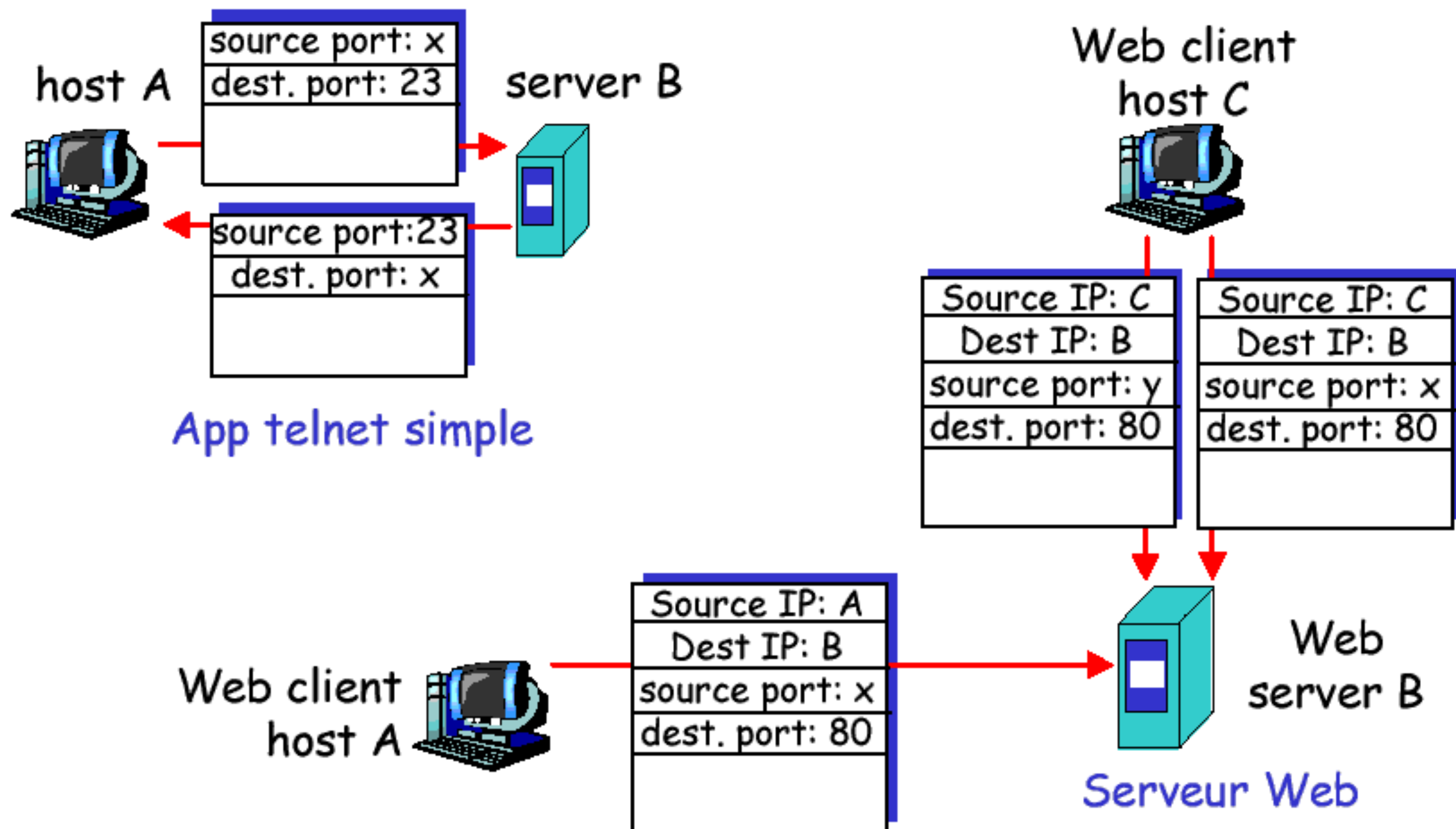
Identification des protocoles (1)



Identification des protocoles (2)

- Une adresse de transport = une adresse IP + un numéro de port (16 bits) -> adresse de socket
- Une connexion s'établit entre une socket source et une socket destinataire -> une connexion = un quintuplé (proto, @src, port src, @dest, port dest)
- Deux connexions peuvent aboutir à la même socket
- Les ports permettent un multiplexage ou démultiplexage de connexions au niveau transport
- Les ports inférieurs à 1024 sont appelés **ports réservés**

Identification des protocoles (3)



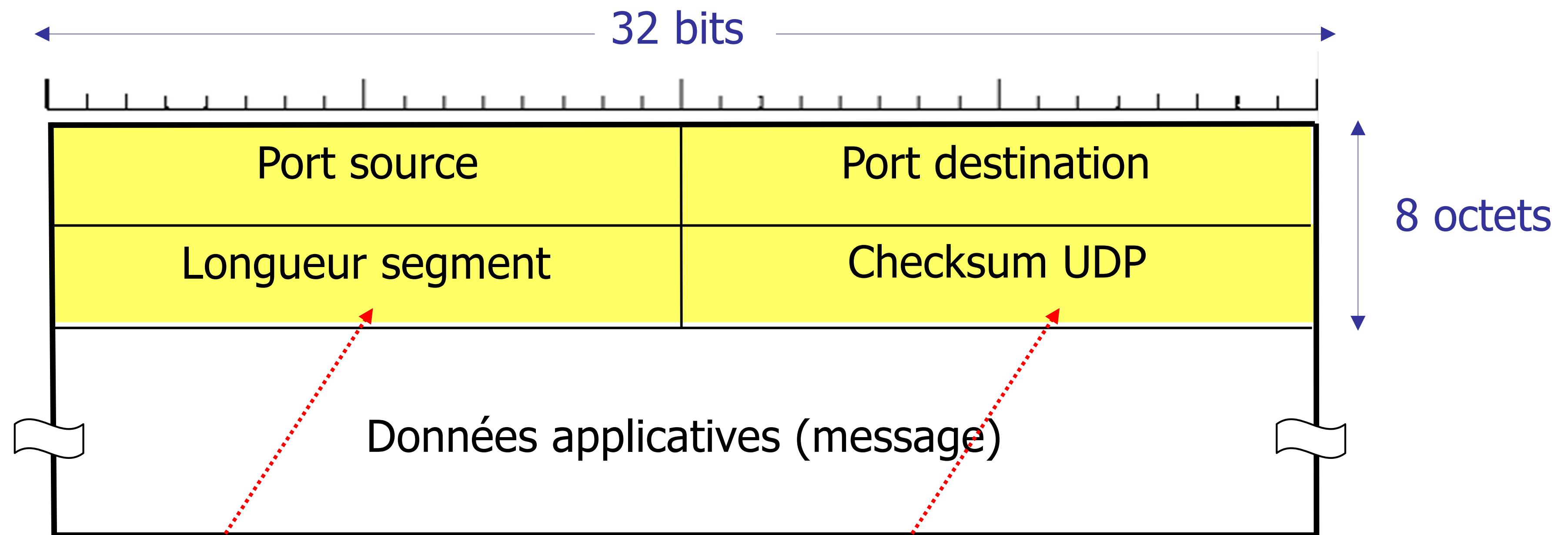
Le protocole UDP

- UDP (RFC 768) - User Datagram Protocol
  - protocole de transport le plus simple
  - service de type best-effort (comme IP)
    - les datagrammes UDP peuvent être perdus
    - les datagrammes UDP peuvent arriver dans le désordre
  - mode non connecté : chaque segment UDP est traité indépendamment des autres
- Pourquoi un service non fiable sans connexion ?
  - simple donc rapide (pas de délai de connexion, pas d'état entre émetteur/récepteur)
  - petit en-tête donc économie de bande passante
  - sans contrôle de congestion donc UDP peut émettre aussi rapidement qu'il le souhaite

Les utilisations d'UDP

- Performance sans garantie de délivrance
- Souvent utilisé pour les applications multimédias
  - tolérantes aux pertes
  - sensibles au débit
- Autres utilisations d'UDP
  - applications qui envoient peu de données et qui ne nécessitent pas un service fiable
  - exemples : DNS, SNMP, BOOTP/DHCP
- Transfert fiable sur UDP
  - ajouter des mécanismes de compensation de pertes (reprise sur erreur) au niveau applicatif
  - mécanismes adaptés à l'application

Le datagramme UDP



Taille totale du segment (en-tête+données)

Total de contrôle du segment (en-tête+données)

optionnel : peut être à 0

UDP = IP + multiplexage (adresse de transport) !!

Le protocole TCP

- Transport Control Protocol (RFC 793, 1122, 1323, 2018, 2581)

Attention: les RFCs ne spécifient pas tout - beaucoup de choses dépendent de l'implémentation

- Transport fiable en mode connecté
  - point à point, bidirectionnel : entre deux adresses de transport (@IP src, port src) --> (@IP dest, port dest)
  - transporte un flot d'octets (ou flux)
    - l'application lit/écrit des octets dans un tampon
  - assure la délivrance des données en séquence
  - contrôle la validité des données reçues
  - organise les reprises sur erreur ou sur temporisation
  - réalise le contrôle de flux et le contrôle de congestion (à l'aide d'une fenêtre d'émission)

Exemples de protocole applicatif (1)

- HTTP - HyperText Transport Protocol
  - protocole du web
  - échange de requête/réponse entre un client et un serveur web
- FTP - File Transfer Protocol
  - protocole de manipulation de fichiers distants
  - transfert, suppression, création, ...
- TELNET - TELetypewriter Network Protocol
  - système de terminal virtuel
  - permet l'ouverture d'une session distante

Exemples de protocole applicatif (2)

- 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

Lyon 1

Les applications réseau (1)

- Applications = la raison d'être des réseaux infos
- Profusion d'applications depuis 30 ans grâce à l'expansion d'Internet
  - années 1980/1990 : les applications "textuelles"
    - messagerie électronique, accès à des terminaux distants, transfert de fichiers, groupe de discussion (forum, *newsgroup*), dialogue interactif en ligne (chat), la navigation Web
  - plus récemment :
    - les applications multimédias : vidéo à la demande (*streaming*), visioconférences, radio et téléphonie sur Internet
    - la messagerie instantanée (ICQ, MSN Messenger)
    - les applications *Peer-to-Peer* (MP3, ...)

Les applications réseau (2)

- L'application est généralement répartie (ou distribuée) sur plusieurs systèmes
- Exemples :
  - L'application Web est constituée de deux logiciels communiquants : le navigateur client qui effectue une requête pour disposer d'un document présent sur le serveur Web
  - L'application *telnet* : un terminal virtuel sur le client, un serveur *telnet* distant qui exécute les commandes
  - La visioconférence : autant de clients que de participants
- --> Nécessité de disposer d'un protocole de communication applicatif !

Terminologie des applications réseau

- Processus :
  - une entité communicante
  - un programme qui s'exécute sur un hôte d'extrémité
- Communications inter-processus locales :
  - communications entre des processus qui s'exécutent sur un même hôte
  - communications régies par le système d'exploitation (tubes UNIX, mémoire partagée, …)
- Communications inter-processus distantes :
  - les processus s'échangent des **messages** à travers le réseau selon un **protocole** de la couche applications
  - nécessite une infrastructure de transport sous-jacente

Protocoles de la couche Applications

- Le protocole applicatif définit :
  - le format des messages échangés entre les processus émetteur et récepteur
  - les types de messages : requête, réponse, …
  - l'ordre d'envoi des messages
- Exemples de protocoles applicatifs :
  - HTTP pour le Web, POP/IMAP/SMTP pour le courrier électronique, SNMP pour l'administration de réseau, …
- Ne pas confondre le protocole et l'application !
  - Application Web : un format de documents (HTML), un navigateur Web, un serveur Web à qui on demande un document, un protocole (HTTP)

Le modèle Client / Serveur

- Idée : l'application est répartie sur différents sites pour optimiser le traitement, le stockage...
- Le client
  - effectue une demande de service auprès du serveur (**requête**)
  - initie le contact (parle en premier), ouvre la session
- Le serveur
  - est la partie de l'application qui offre un service
  - est à l'écoute des requêtes clientes
  - répond au service demandé par le client (**réponse**)

Le modèle Client / Serveur

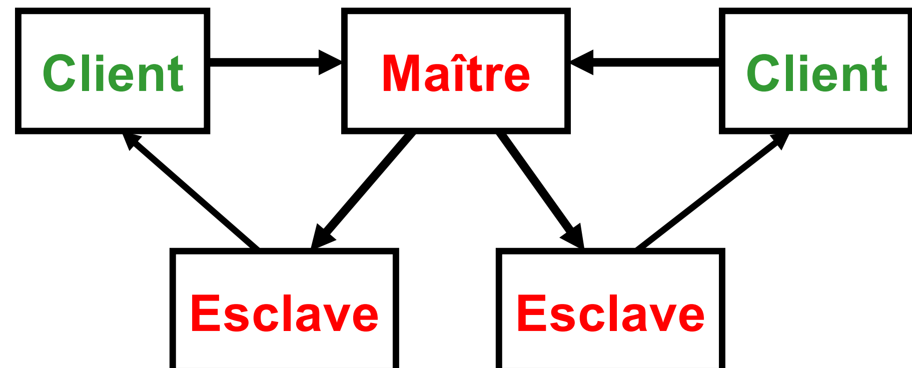
- Le client et le serveur ne sont pas identiques, ils forment un système coopératif
  - les parties client et serveur de l'application peuvent s'exécuter sur des systèmes différents
  - une même machine peut implanter les côtés client ET serveur de l'application
  - un serveur peut répondre à plusieurs clients simultanément

Des clients et des serveurs...

Un client, un serveur :

Requête/Réponse

Plusieurs clients, un serveur :

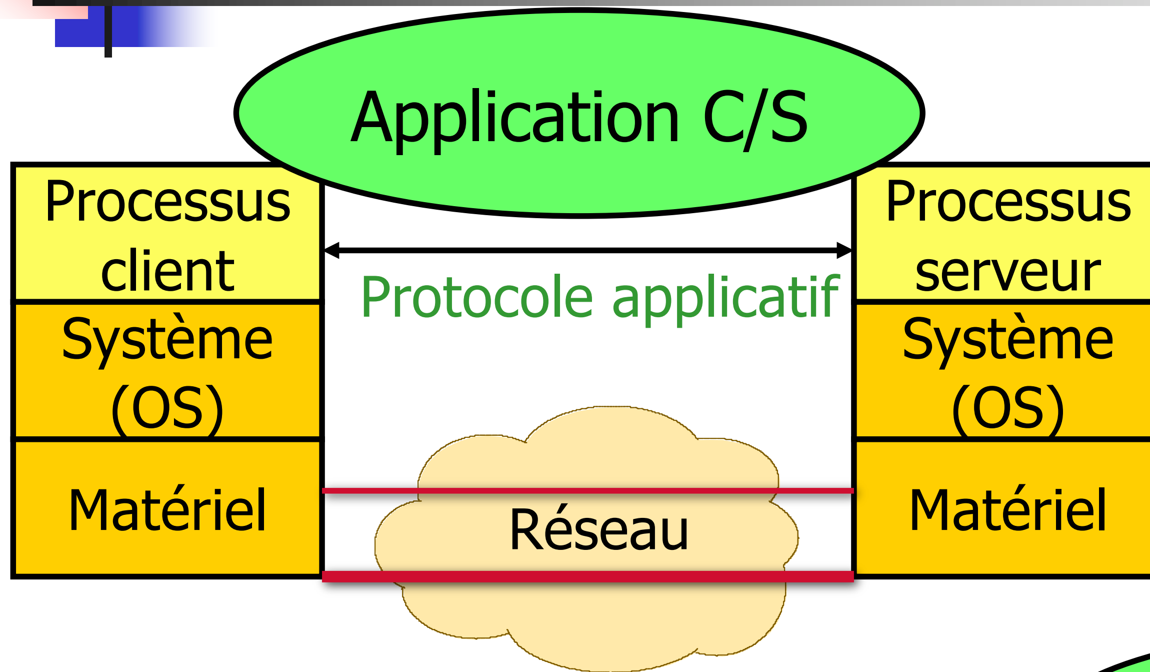


Le serveur traite plusieurs requêtes simultanées

Un client, plusieurs serveurs :

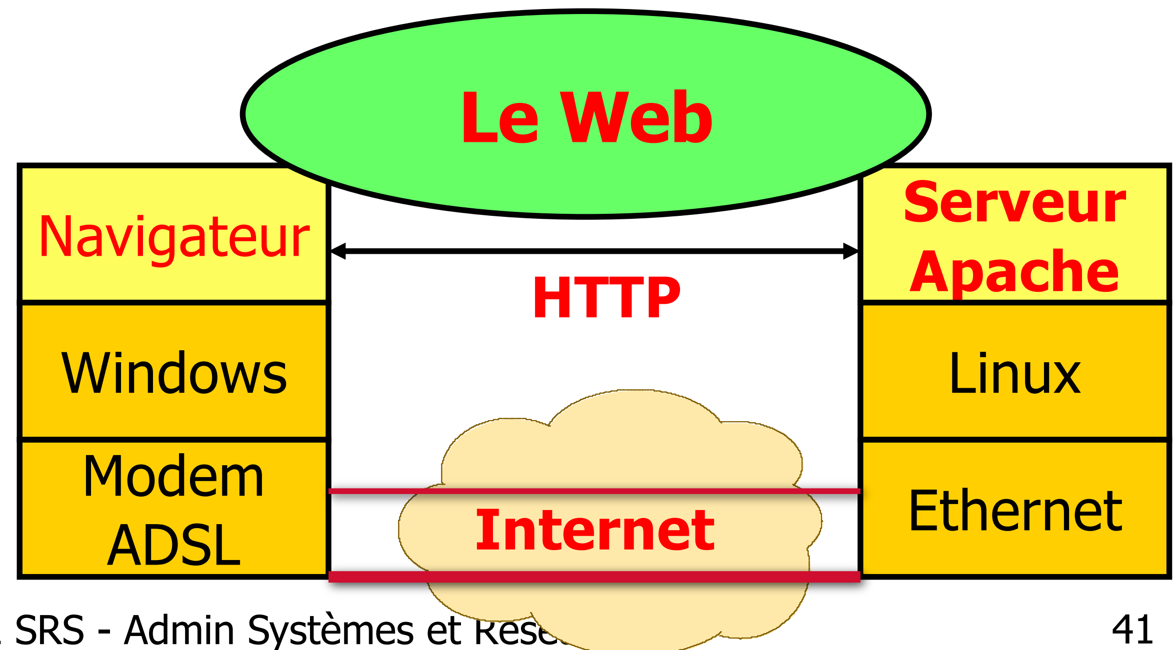
Le serveur contacté peut faire appel à un service sur un autre serveur (ex. SGBD)

Le modèle Client / Serveur

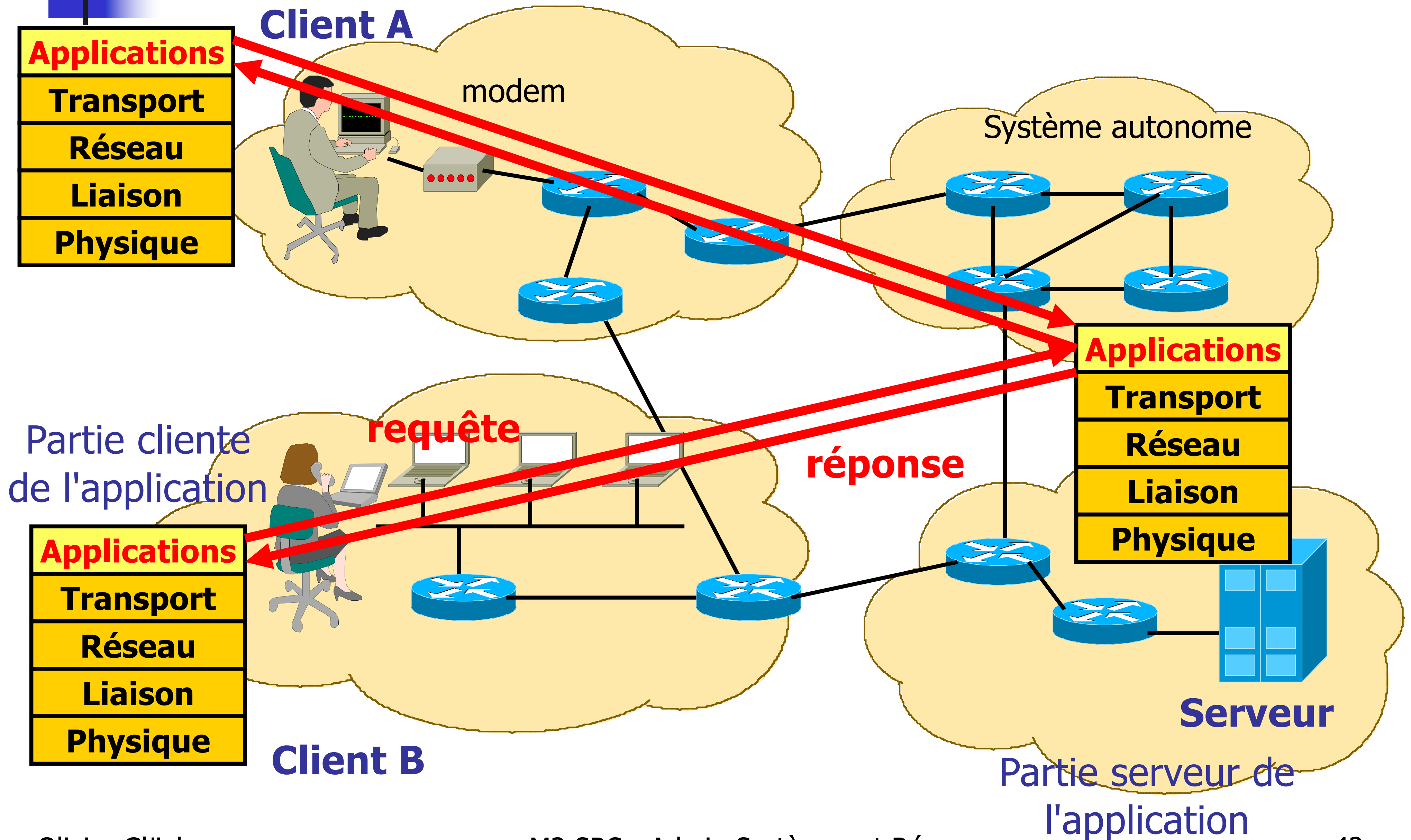


L'application est répartie sur le client et le serveur qui dialoguent selon un protocole applicatif spécifique

L'exemple du Web



Le modèle Client / Serveur



Exemple d'application client/serveur

- Le client lit une ligne à partir de l'entrée standard (clavier) et l'envoie au serveur
- Le serveur lit la ligne reçue et la convertit en majuscules
- Le serveur renvoie la ligne au client
- Le client lit la ligne reçue et l'affiche sur la sortie standard (écran)

Exemple d'application client/serveur

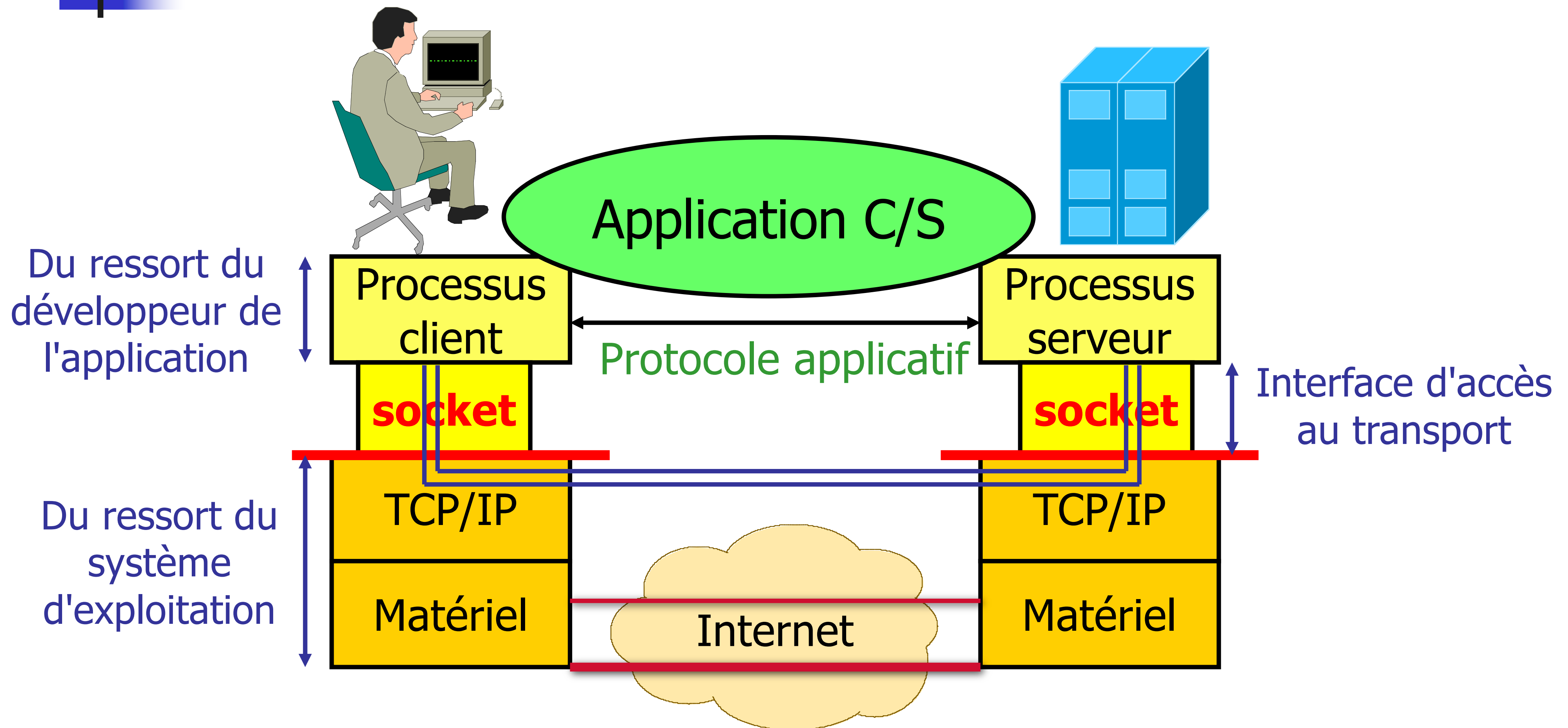
- DAYTIME (RFC 867) permet au client d'obtenir la date et l'heure du serveur
- Le protocole spécifie
  - l'échange des messages :
    - dès qu'un serveur reçoit un message d'un client, il renvoie une chaîne de caractères contenant la date et l'heure
    - le contenu du message client n'est même pas regardé
  - le format de la chaîne renvoyée : 1 ligne ASCII
    - Par exemple "*Weekday, Month Day, Year Time-Zone*"

      "*Tuesday, February 22, 1982 17:37:43-PST*"

Interface de programmation réseau

- Il faut une interface entre l'application réseau et la couche transport
  - le transport n'est qu'un tuyau (TCP ou UDP dans Internet)
  - l'API (*Application Programming Interface*) n'est que le moyen d'y accéder (interface de programmation)
- Les principales APIs de l'Internet
  - les sockets
    - apparus dans UNIX BSD 4.2
    - devenus le standard de fait
  - les RPC : Remote Procedure Call - appel de procédures distantes

Interface de programmation réseau



Une socket : interface locale à l'hôte, créée par l'application, contrôlée par l'OS

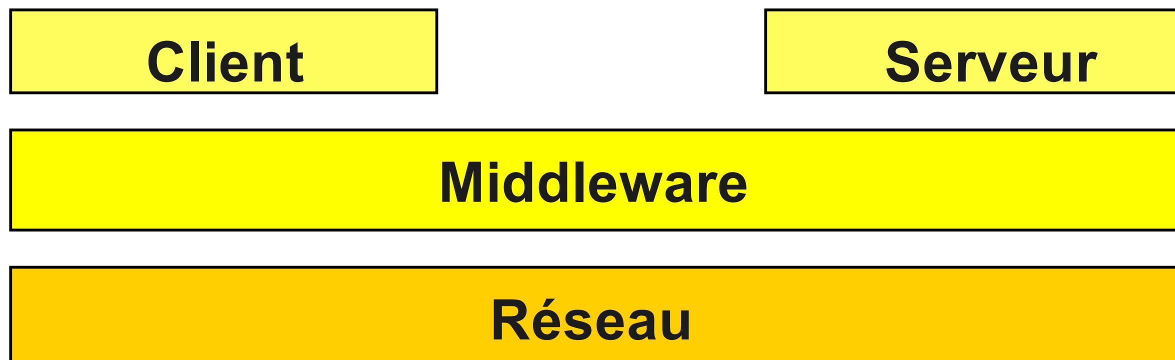
Porte de communication entre le processus client et le processus serveur

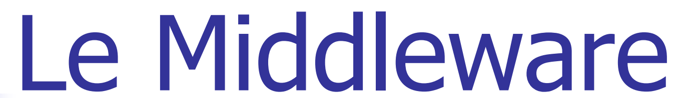
Application C/S - récapitulatif

- Une application Client/Serveur, c'est
  - **une partie cliente** qui exécute des requêtes vers un serveur
  - **une partie serveur** qui traite les requêtes clientes et y répond
  - **un protocole applicatif** qui définit les échanges entre un client et un serveur
  - **un accès via une API** (interface de programmation) à la couche de transport des messages
- Bien souvent les parties cliente et serveur ne sont pas écrites par les mêmes programmeurs (Navigateur Netscape/Serveur apache) --> rôle important des RFCs qui spécifient le protocole !

Le Middleware

- Grossièrement : la gestion du protocole applicatif+l'API d'accès à la couche transport+des services complémentaires
- C'est un ensemble de services logiciels construits au dessus d'un protocole de transport afin de permettre l'échange de requête/réponse entre le client et le serveur de manière transparente





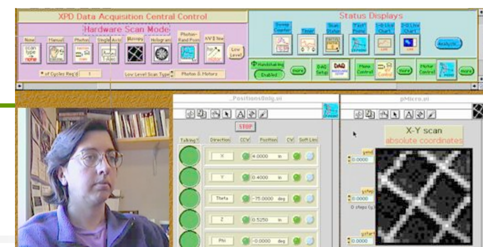
Le Middleware

- Complément de services du réseau permettant la réalisation du dialogue client/serveur :
  - prend en compte les requêtes de l'application cliente
  - les transmet de manière transparente à travers le réseau jusqu'au serveur
  - prend en compte les données résultat du serveur et les transmet vers l'application cliente
- L'objectif essentiel du *middleware* est d'offrir aux applications une interface unifiée permettant l'accès à l'ensemble des services disponibles sur le réseau : l'API

Fonctions d’un Middleware

- Procédures d’établissement/fermeture de connexion
- Exécution des requêtes, récupération des résultats
- Initiation des processus sur différents sites
- Services de répertoire
- Accès aux données à distance
- Gestion d'accès concurrents
- Sécurité et intégrité (authentification, cryptage, …)
- Monitoring (compteurs, …)
- Terminaison de processus
- Mise en cache des résultats, des requêtes

Architecture of a Grid



Portals

Portals that are Web Services based, shell scripts, specialized (e.g. high end vis workstations, PDAs)

Encapsulation as Web Services, as Script Based Services, as Java Based Services

Advanced Services

Grid Services:
- Resource brokering
- Workflow management / Fault management
- Authorization / Accounting
- Data replication and metadata management / Grid MPI / CORBA, DCOM, …

Application Services:
- Visualization / Data analysis / Data integration / Collaboration tools
- Applications

Encapsulation as Web Services, as Script Based Services, as Java Based Services

Basic Grid Services

- Resource Discovery
- Resource Scheduling
- Uniform Computing Access
- Uniform Data Access
- Monitoring and Events
- Authentication
- Identity Credentials

Grid Communication Functions (transport services, security services)

Communications

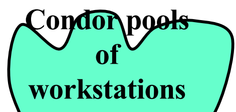
space-based networks | optical networks | Internet | …

Distributed Resources

Resource access and functionality | Resource access and functionality | Resource access and functionality | Resource access and functionality | Resource access and functionality

scientific instruments

clusters



Condor pools of workstations

tertiary storage

national supercomputer facilities

job initiation, event generators, GridFTP servers

Operational Support

Conception d'une application C/S

- Comment découper une application informatique en clients et serveurs ?
- 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)

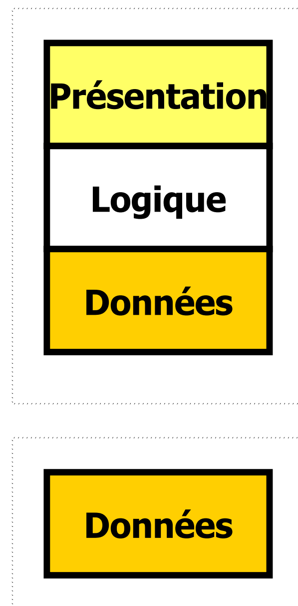
Conception d'une application C/S

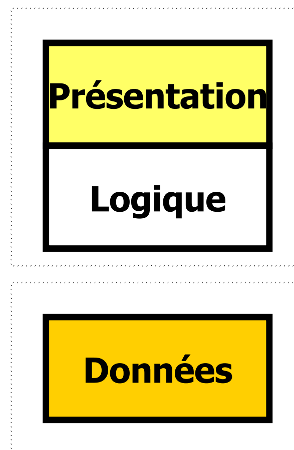
- Exemples de découpage Client/Serveur :
  - le module de gestion des données peut être hébergé par un serveur distant (SGBD, serveur web)
  - le module de gestion de l'affichage peut également être géré par un serveur distant (un terminal X par exemple)

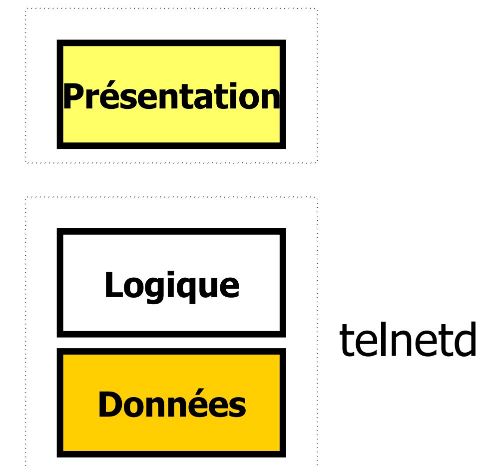
Conception d'une application C/S

- Autres exemples

BD distribuée | **Serveur de fichiers** | **Émulation de terminaux**

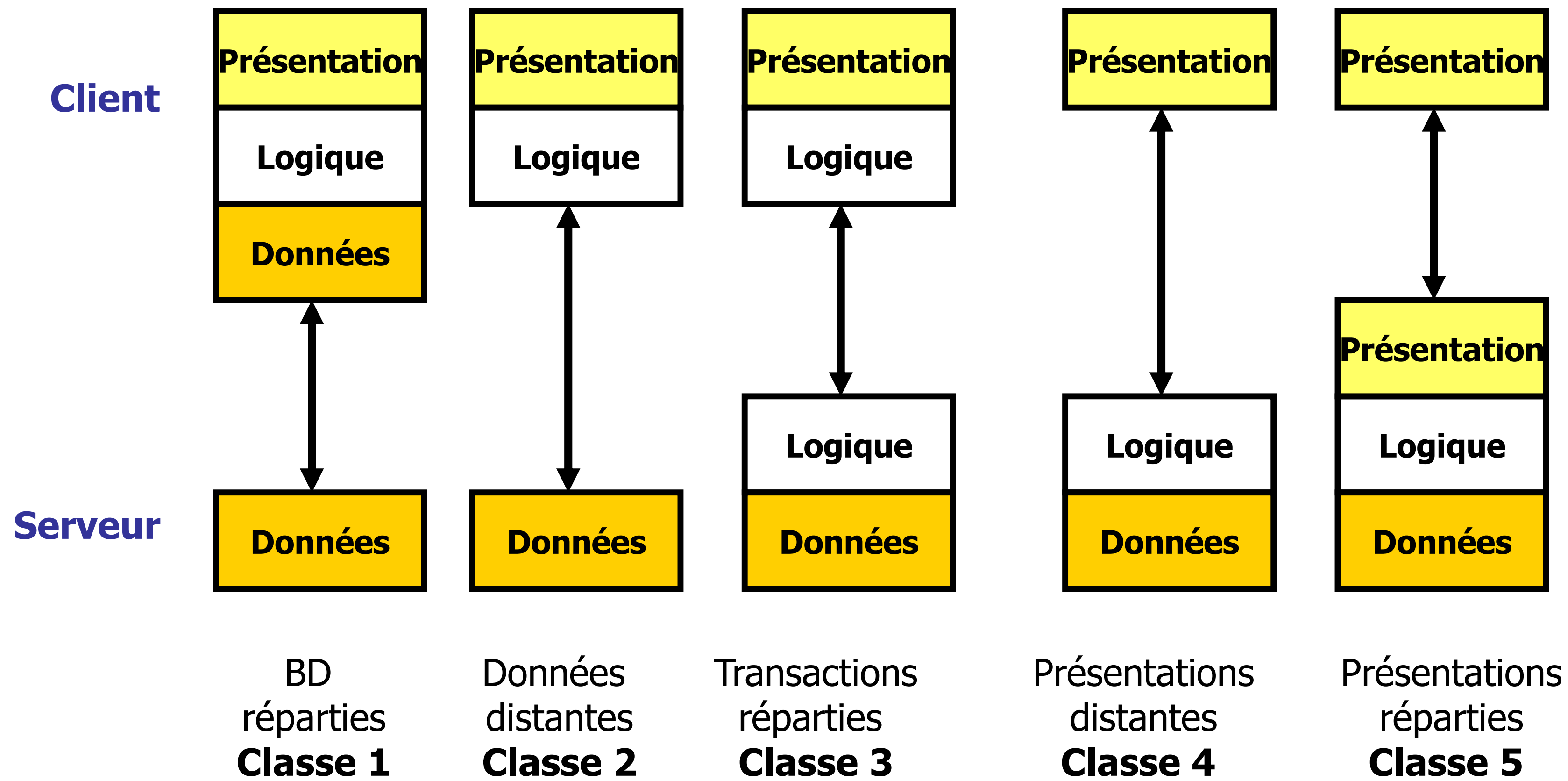






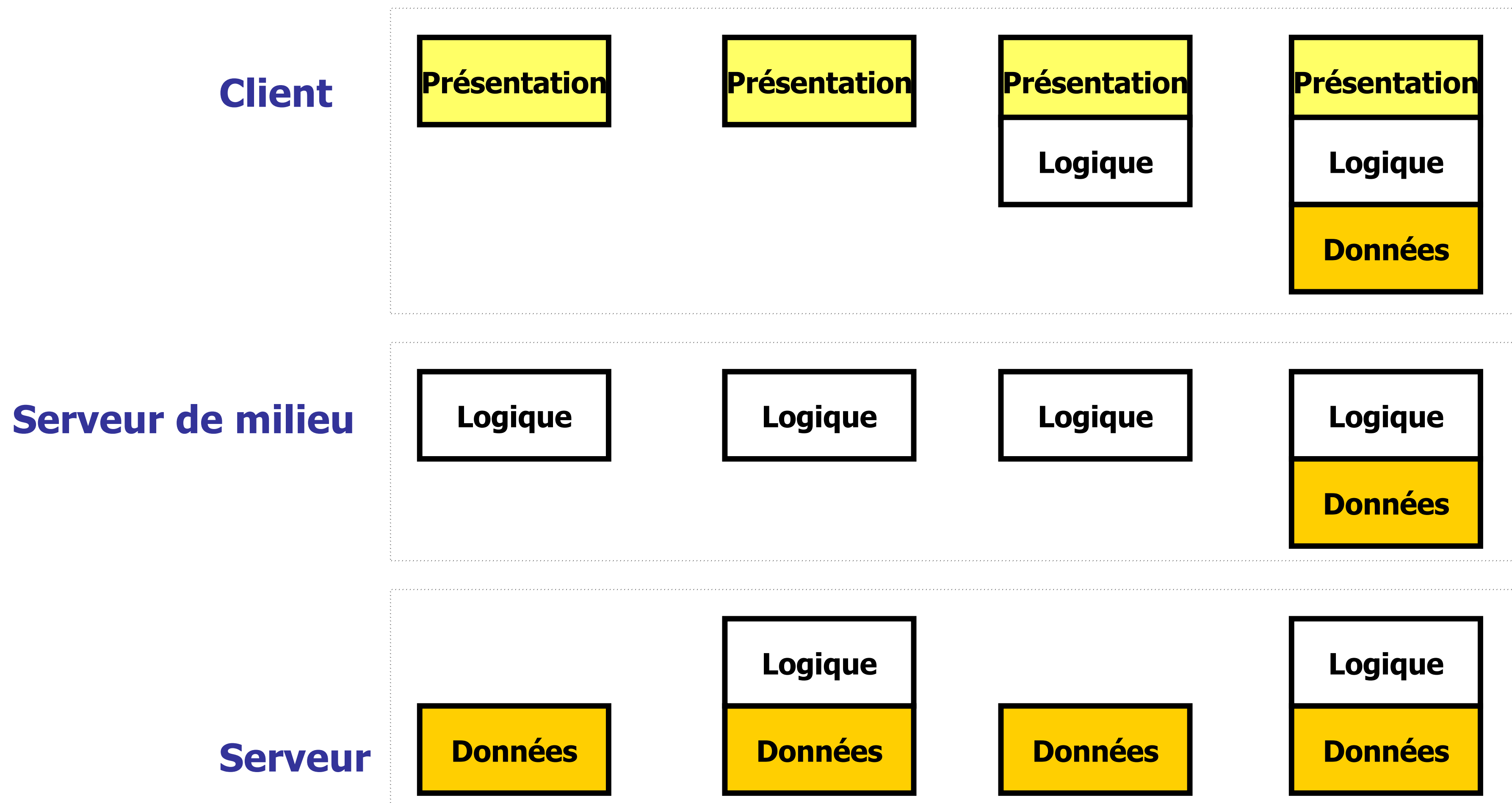
Conception d'une application C/S

- Modèle de Gartner pour les systèmes à 2 niveaux (2-tiers) :



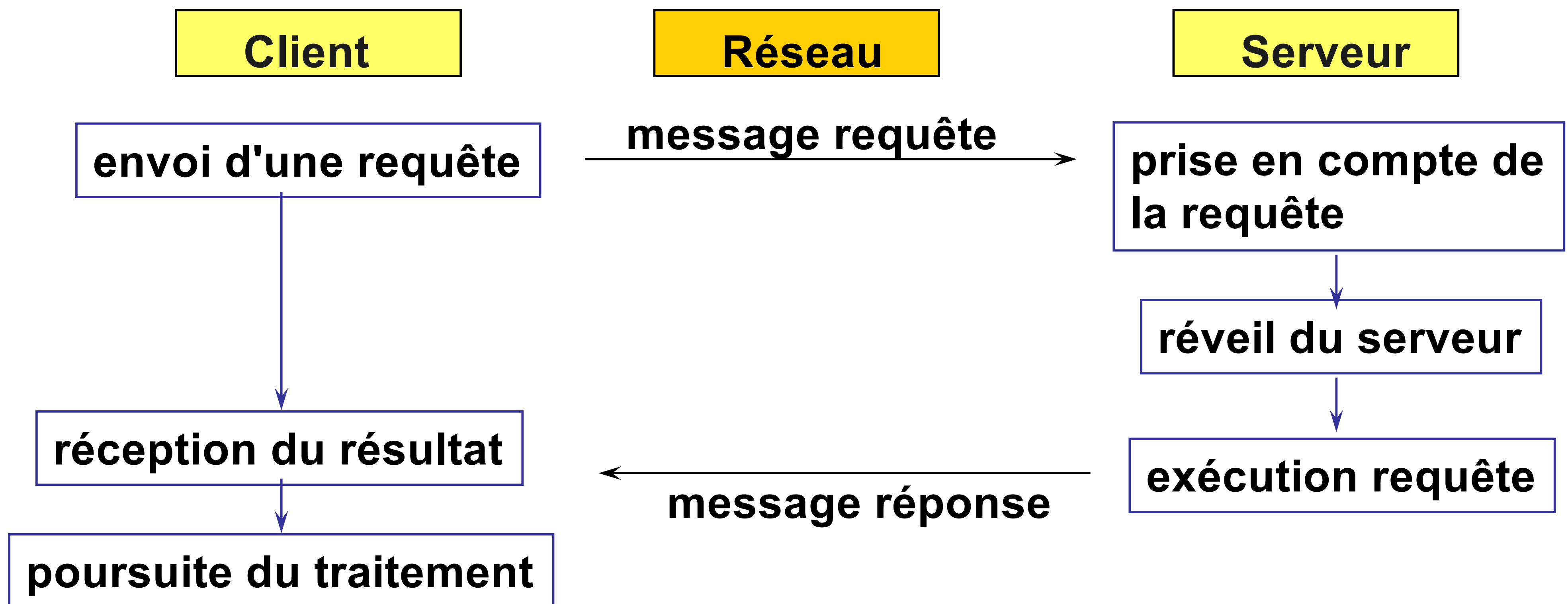
Conception d'une application C/S

- Modèle de Gartner pour les systèmes à 3 niveaux (3-tiers) :



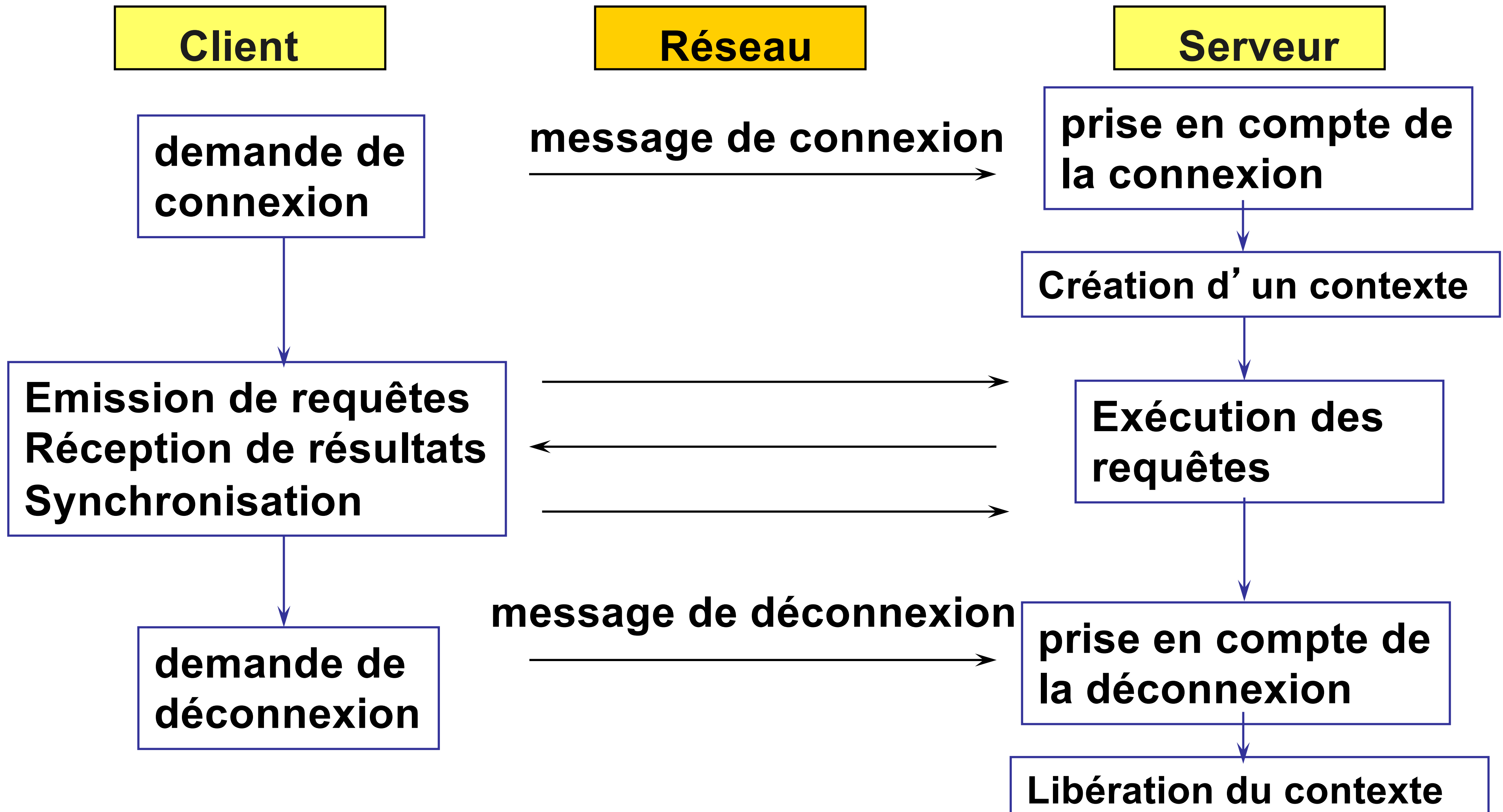
Les modes de communication

- Communication en mode non connecté



Les modes de communication

- Communication en mode connecté



Serveur itératif ou concurrent

- Serveur itératif
  - traite séquentiellement les requêtes
  - adapté aux requêtes qui peuvent s'exécuter rapidement
  - souvent utilisé en mode non connecté (recherche de la performance)
- Serveur concurrent
  - le serveur accepte les requêtes puis les "délègue" à un processus fils (traitement de plusieurs clients)
  - adapté aux requêtes qui demandent un certain traitement (le coût du traitement est suffisamment important pour que la création du processus fils ne soit pas pénalisante)
  - souvent utilisé en mode connecté

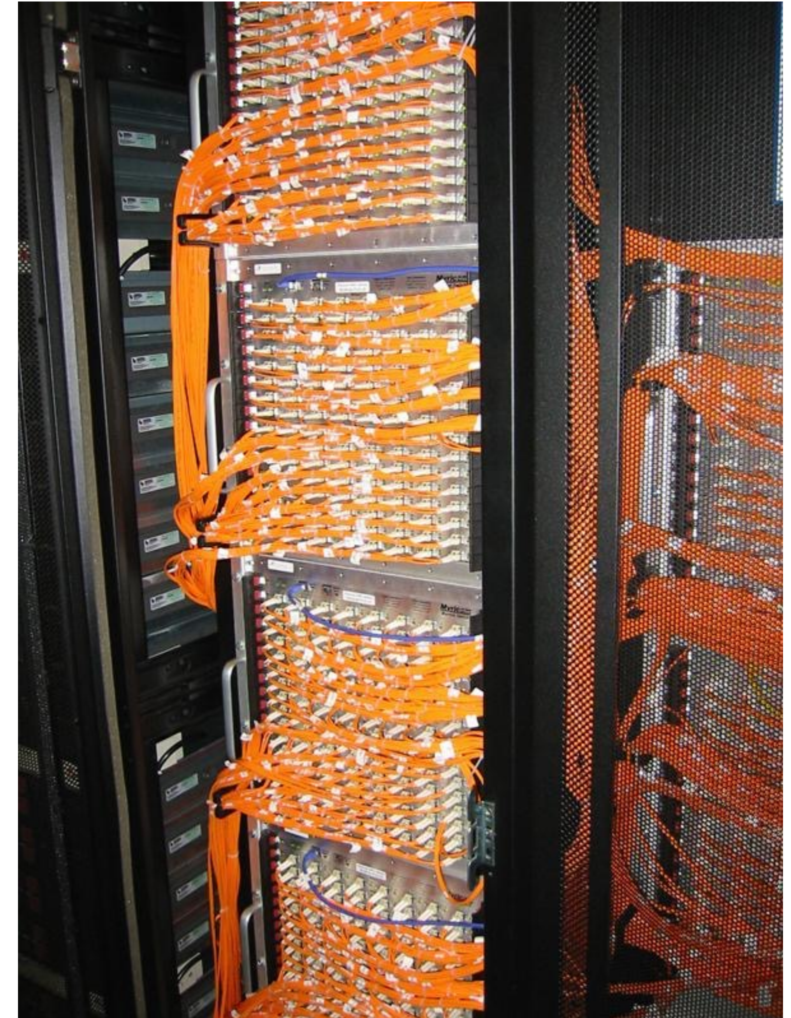
Service avec ou sans état(s)

- Service avec états
  - le serveur conserve localement un état pour chacun des clients connectés : informations sur le client, les requêtes précédentes, …
- Service sans état
  - le serveur ne conserve aucune information sur l'enchaînement des requêtes...
- 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
- Exemple : accès à un fichier distant
  - RFS avec états, NFS sans état (pointeur de fichier…)

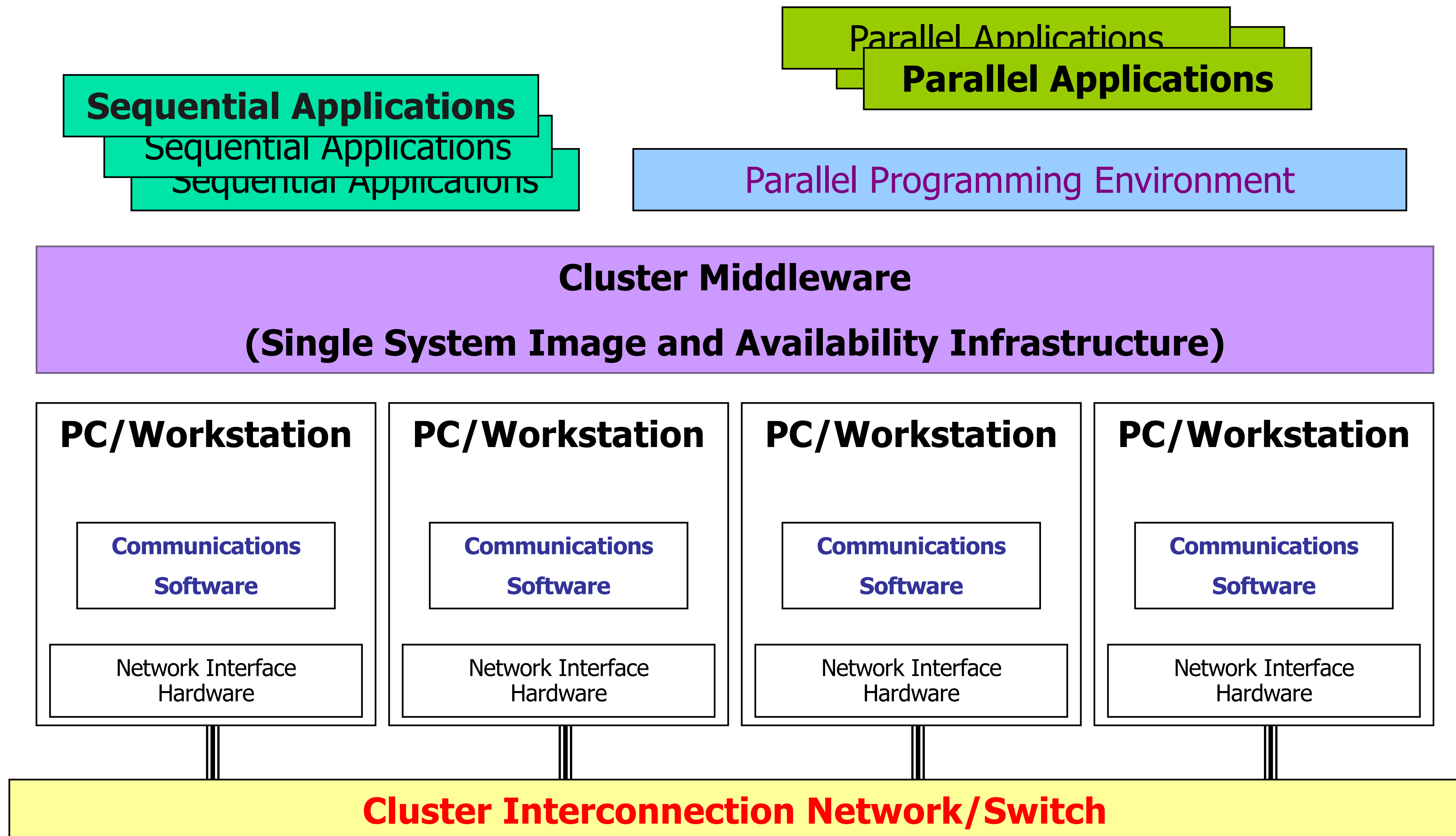
Les communications inter-processus

Lyon 1

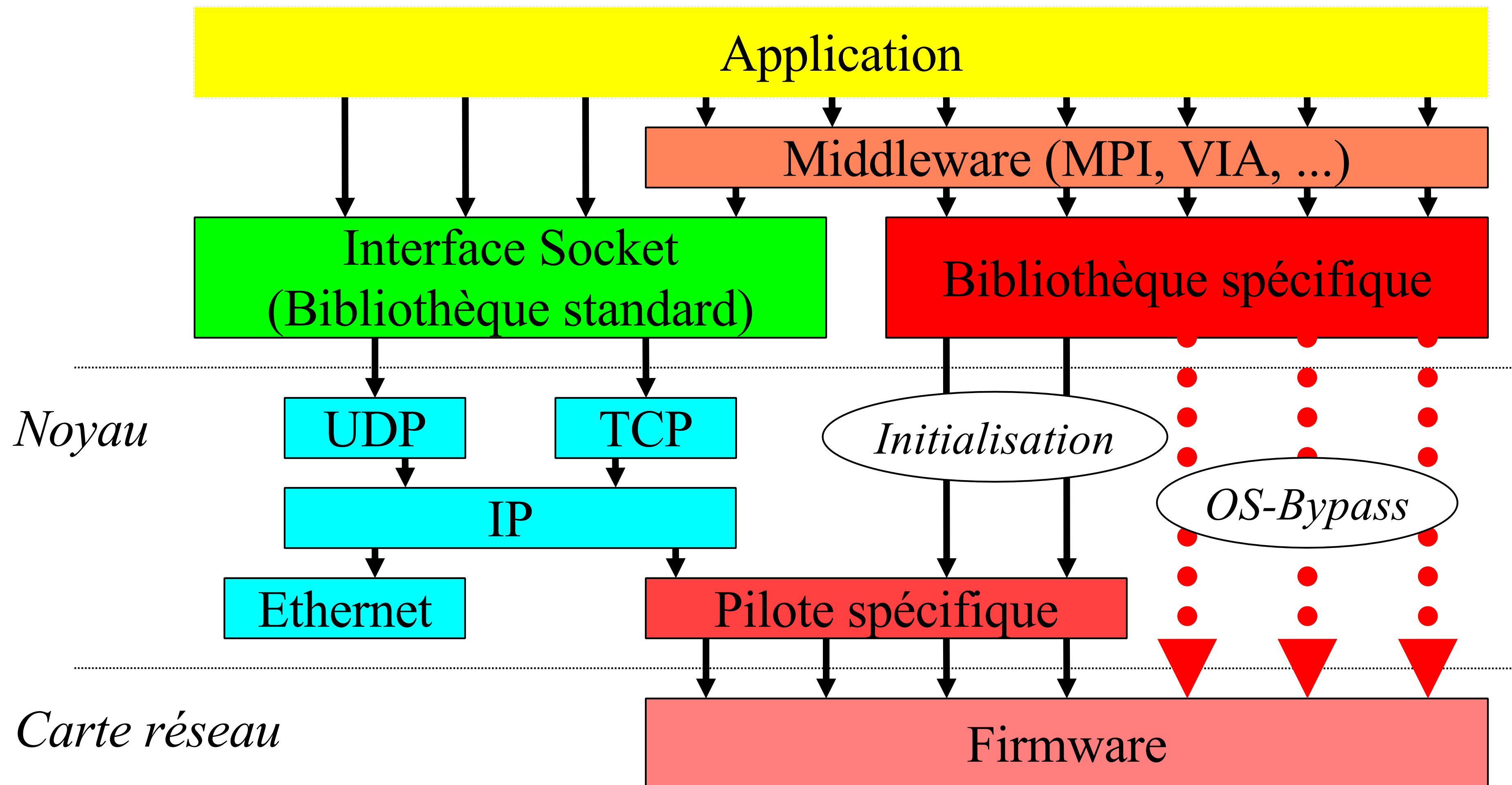
Clusters



Cluster Architecture



Modèle de fonctionnement

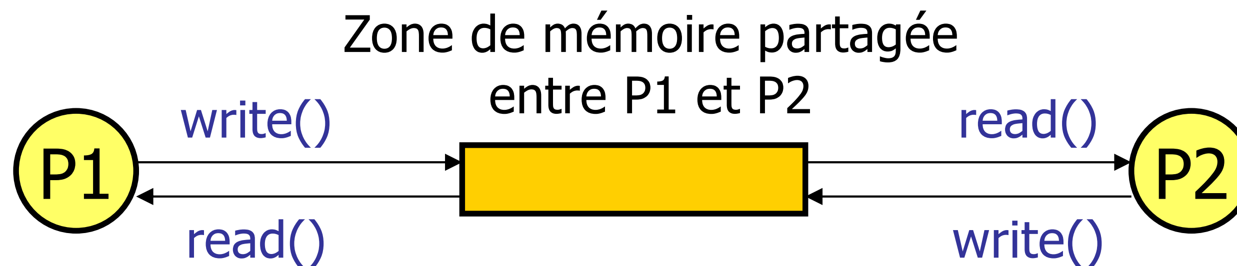


Les schémas de communication

- Dès lors qu'une application est répartie, elle se décompose en plusieurs processus qui doivent communiquer (échanges de données)
- Deux grands types de schéma de communication
  - communication par mémoire partagée (ou fichier)
  - communication par passage de messages
- On retrouve ces deux schémas de communication
  - dans des **communications locales** : entre processus s'exécutant sur le même hôte
  - dans des **communications distantes** : entre processus s'exécutant sur des hôtes distants

Communication par mémoire partagée

- Les processus se partagent une zone de mémoire commune dans laquelle ils peuvent lire et/ou écrire



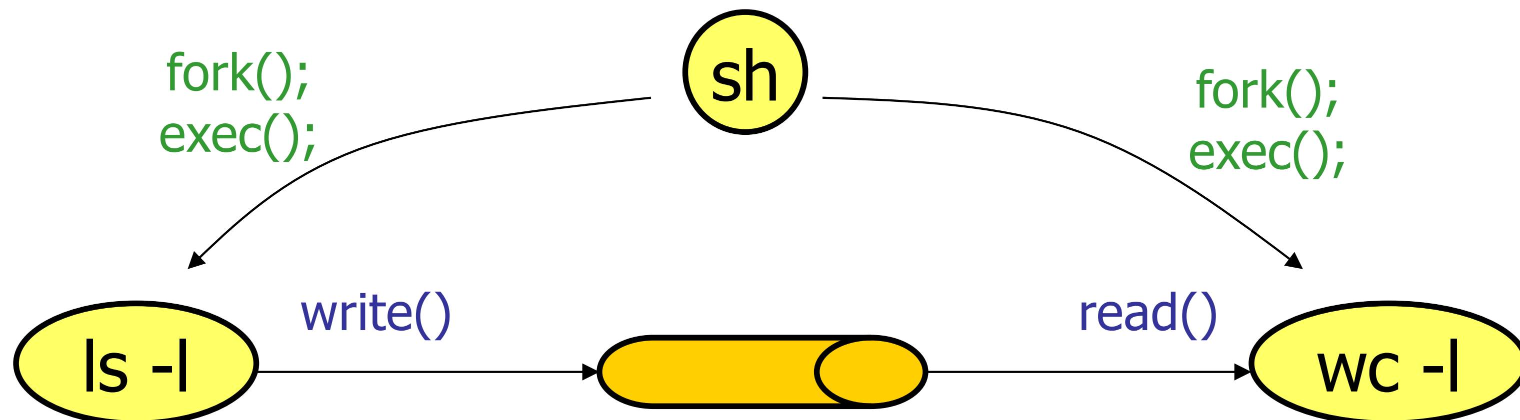
- Intérêt : communications transparentes, limitation des copies mémoire
- Problème : gestion de l'accès à une ressource partagée
  - problème si deux écritures simultanées (ordre d'ordonnancement, atomicité des opérations)
  - les processus P1 et P2 doivent se synchroniser pour accéder au tampon partagé (verrou, sémaphore, …)

Communication par mémoire partagée

- Communications locales
  - les deux processus s'exécutent sur la même machine donc peuvent se partager une partie de leur espace d'adressage
  - exemple : les *threads* s'exécutent dans le contexte d'un même processus
- Communications distantes
  - la mémoire partagée est physiquement répartie
  - le gestionnaire de mémoire virtuelle permet de regrouper les différents morceaux selon un seul espace d'adressage
  - problème de cohérence mémoire...

Les tubes de communication (*pipes*)

- Communications locales type mémoire partagée
  - le canal de communication est unidirectionnel (pas de problème de synchronisation)
  - communications entre 2 processus uniquement : l'un écrit dans le tube, l'autre lit
- Exemple : `sh$ ls -l | wc -l`

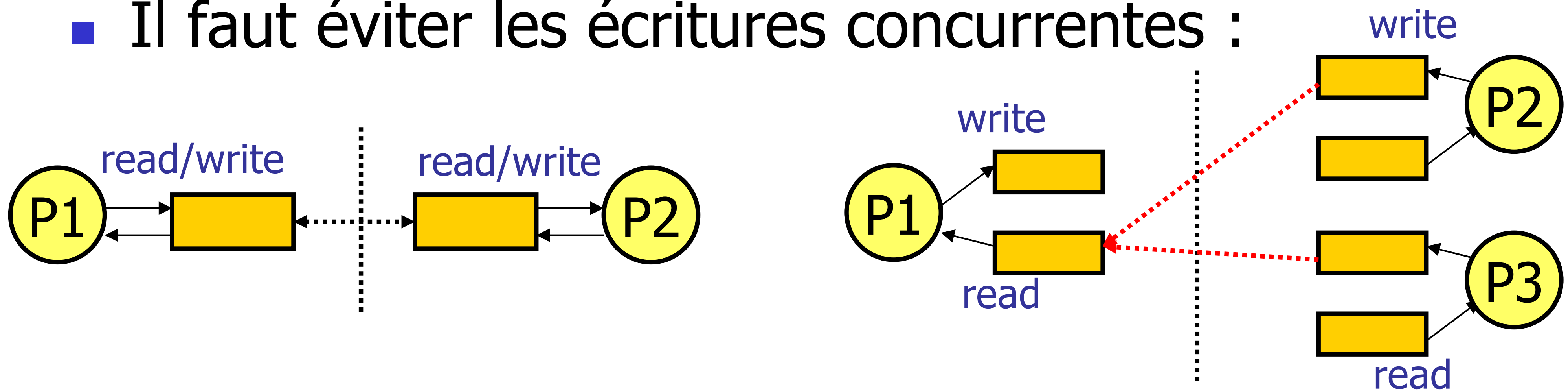


Communication par passage de msg

- Les processus n'ont pas accès à des "variables" communes
- Ils communiquent en s'échangeant des messages
  - au moins deux primitives : *send*() et *recv*()
  - des zones de mémoire locales à chaque processus permettent l'envoi et la réception des messages
  - l'émetteur/récepteur doit pouvoir désigner le récepteur/émetteur distant
- Problèmes
  - zones d'émission et réception distinctes ?
  - nombre d'émetteurs/récepteurs dans une zone ?
  - opérations bloquantes/non bloquantes ?

Communication par passage de msg

- Il faut éviter les écritures concurrentes :



- Pour se ramener à des communications point-à-point
  - --> dissocier le tampon d'émission et de réception
  - --> avoir autant de tampons de réception que d'émetteurs potentiels
  - --> il ne reste plus alors au protocole qu'à s'assurer que deux émissions successives (d'un même émetteur) n'écrasent pas des données non encore lues (contrôle de flux)

Opérations bloquantes/non bloquantes

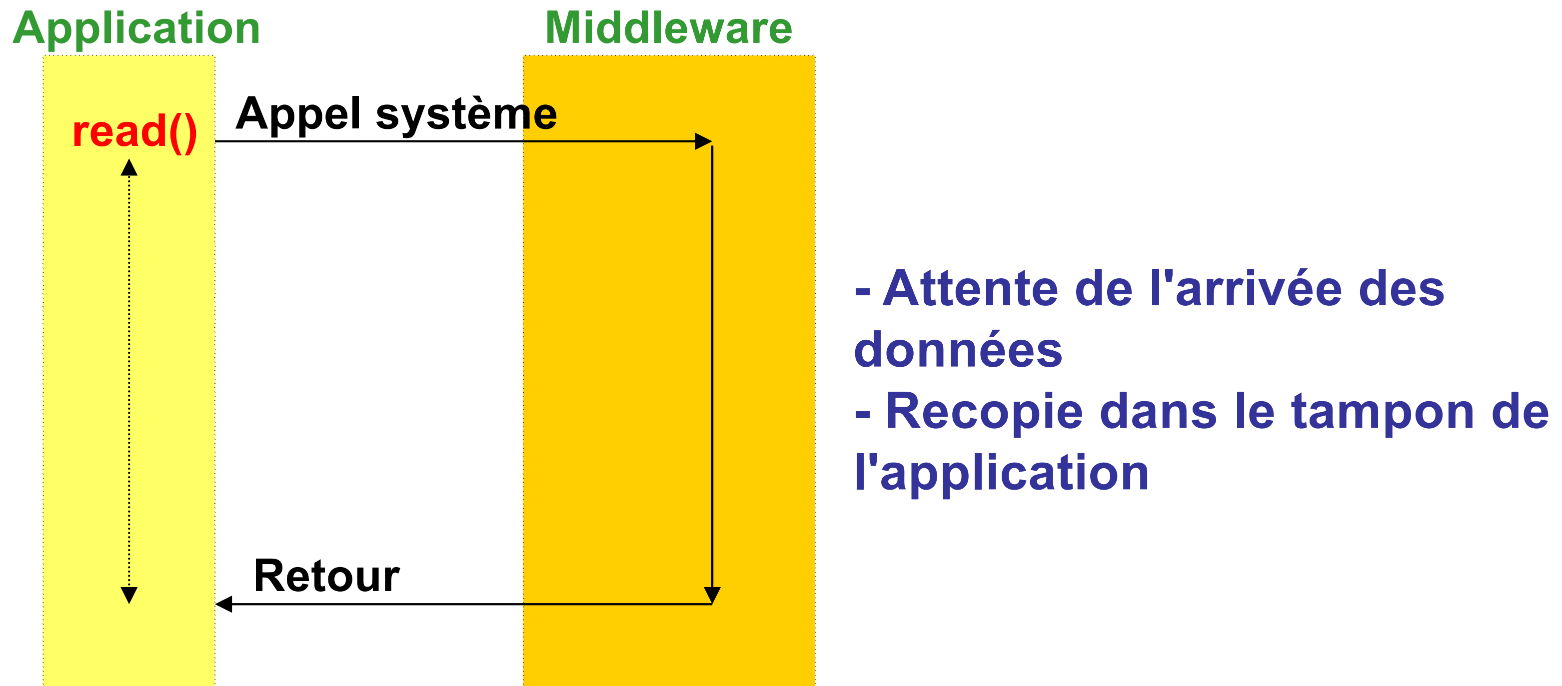
- Quand un appel à une primitive *send*() ou *recv*() doit-il se terminer ?
- Plusieurs sémantiques en réception :
  - *recv*() peut rendre la main
    - aussitôt (*recv*() non bloquant)
    - quand les données ont été reçues et recopiées depuis le tampon de réception local (le tampon de réception est de nouveau libre)

Opérations bloquantes/non bloquantes

- Plusieurs sémantiques en émission :
  - *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)

Opérations bloquantes

- Le processus se bloque jusqu'à ce que l'opération se termine :



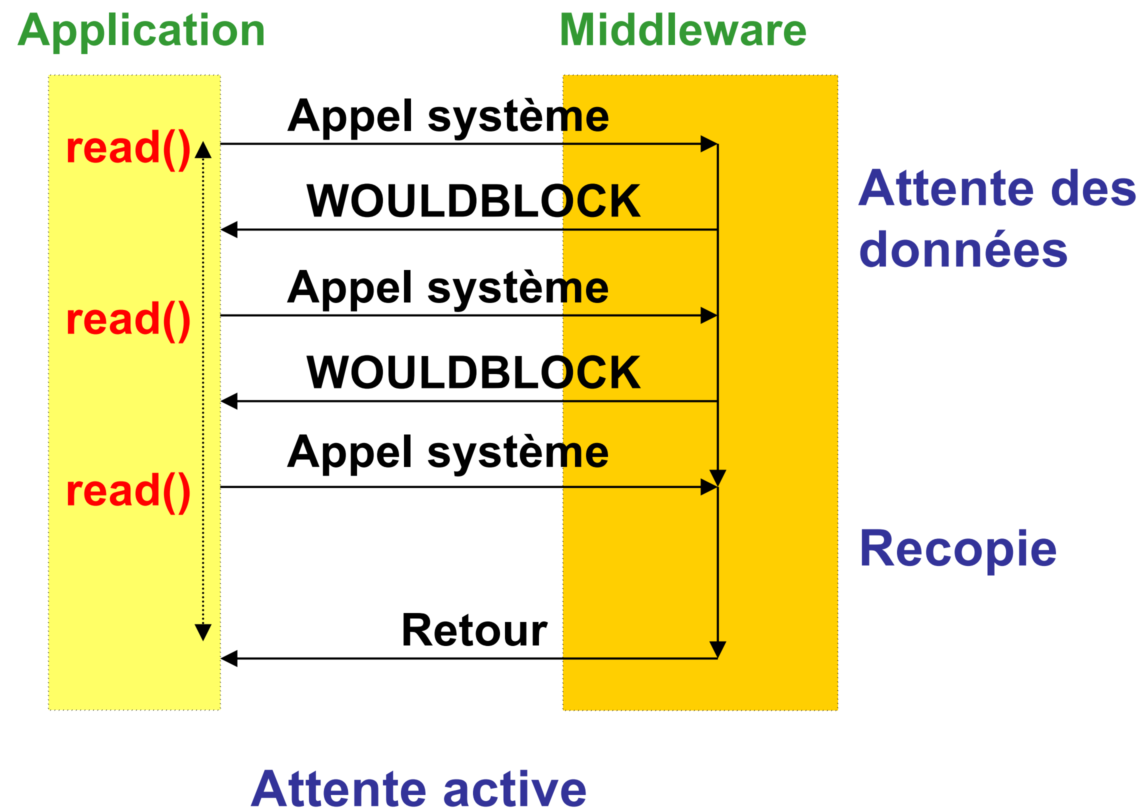
Opérations non bloquantes

- Intérêt :
  - le processus peut faire autre chose en attendant que les données soient émises ou reçues
- Le processus a tout de même besoin d'être informé de la complétion de l'opération (lecture ou écriture)
- Deux possibilités :
  - 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)

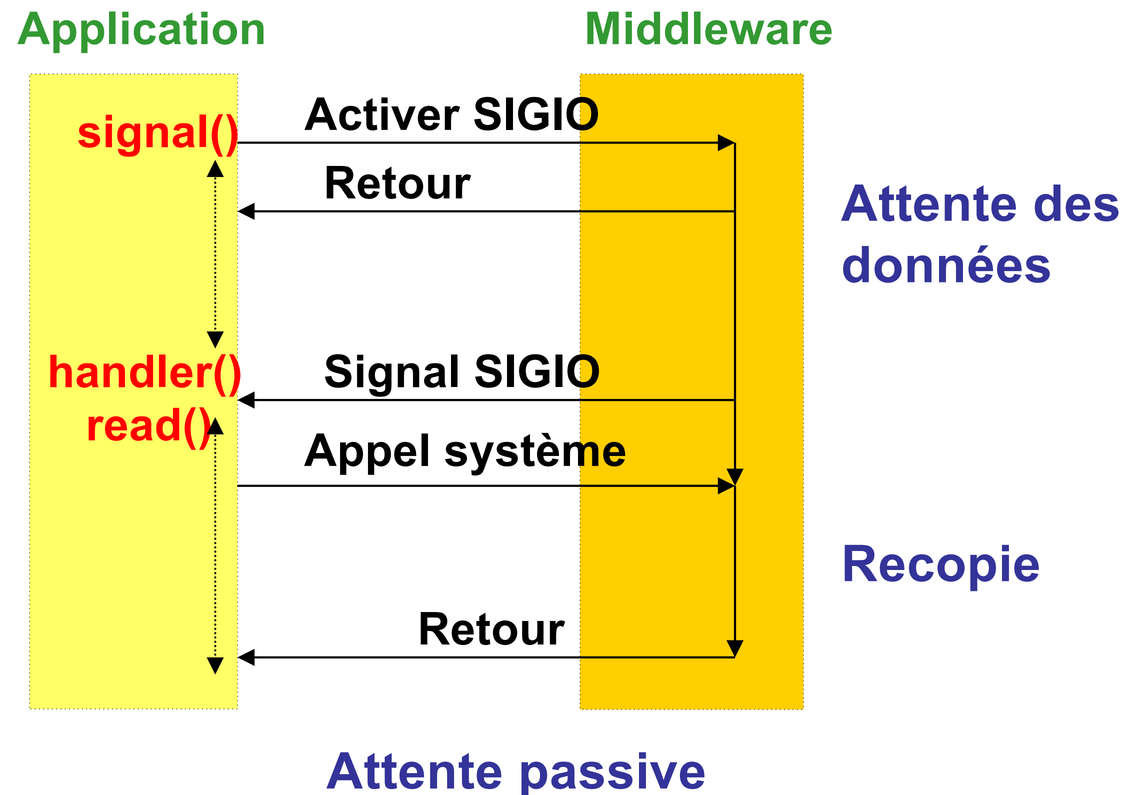
Communication par signaux

- Mécanisme de communications **locales** inter-processus (ou depuis le noyau vers un processus) permettant de notifier un événement
- 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
- Exemple : arrivée de données urgentes sur une socket

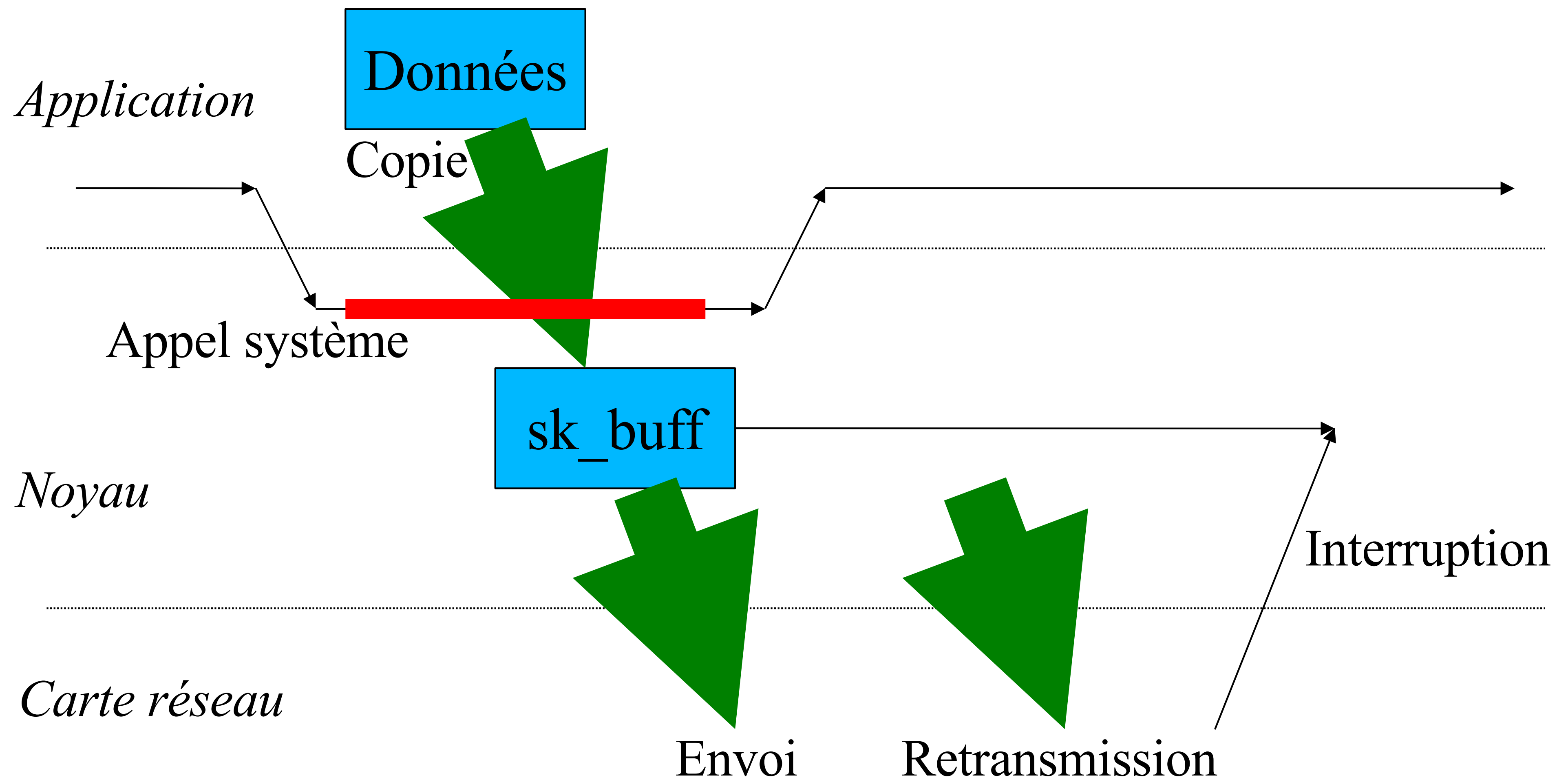
Opérations non bloquantes



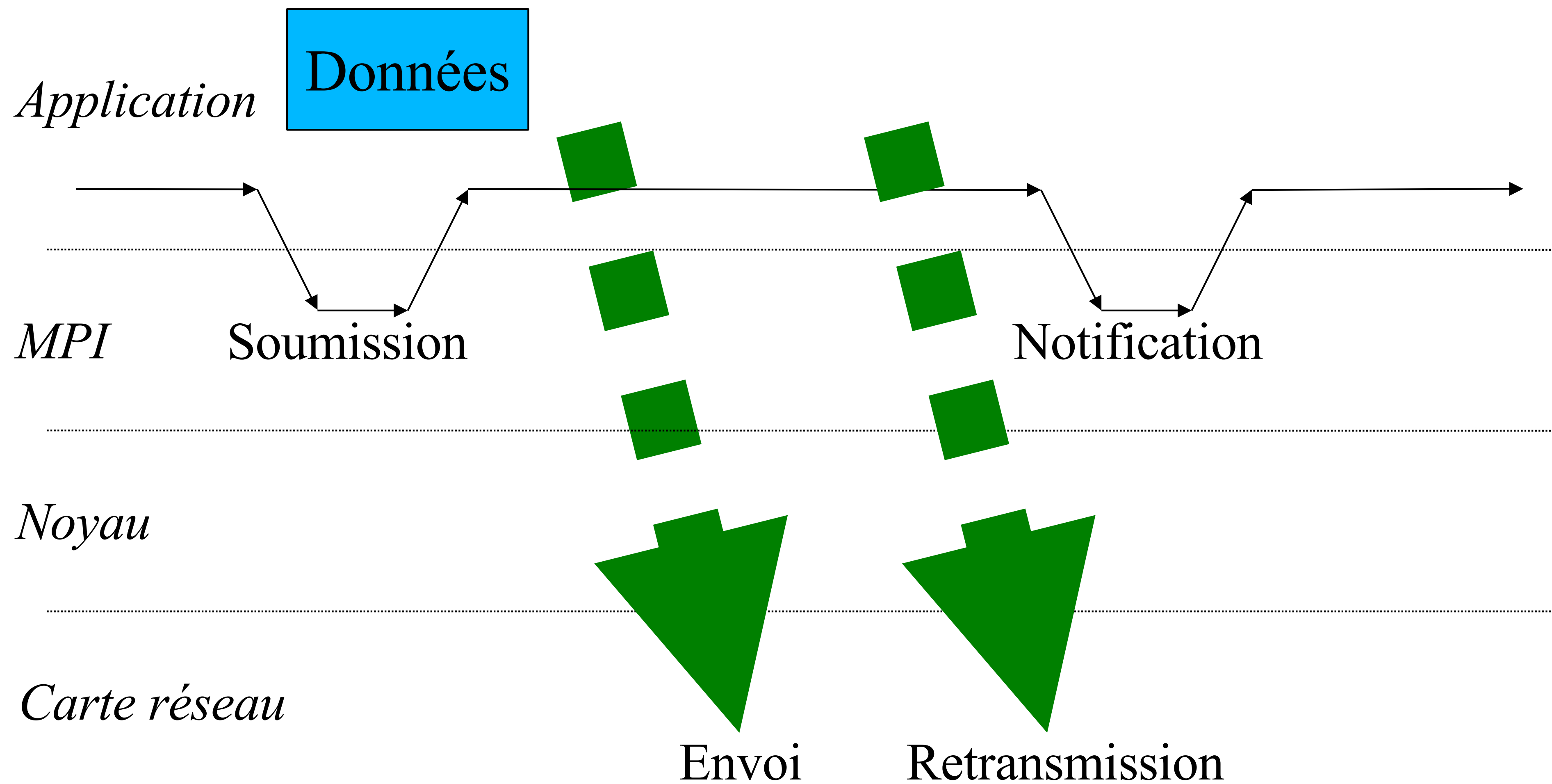
Opérations non bloquantes



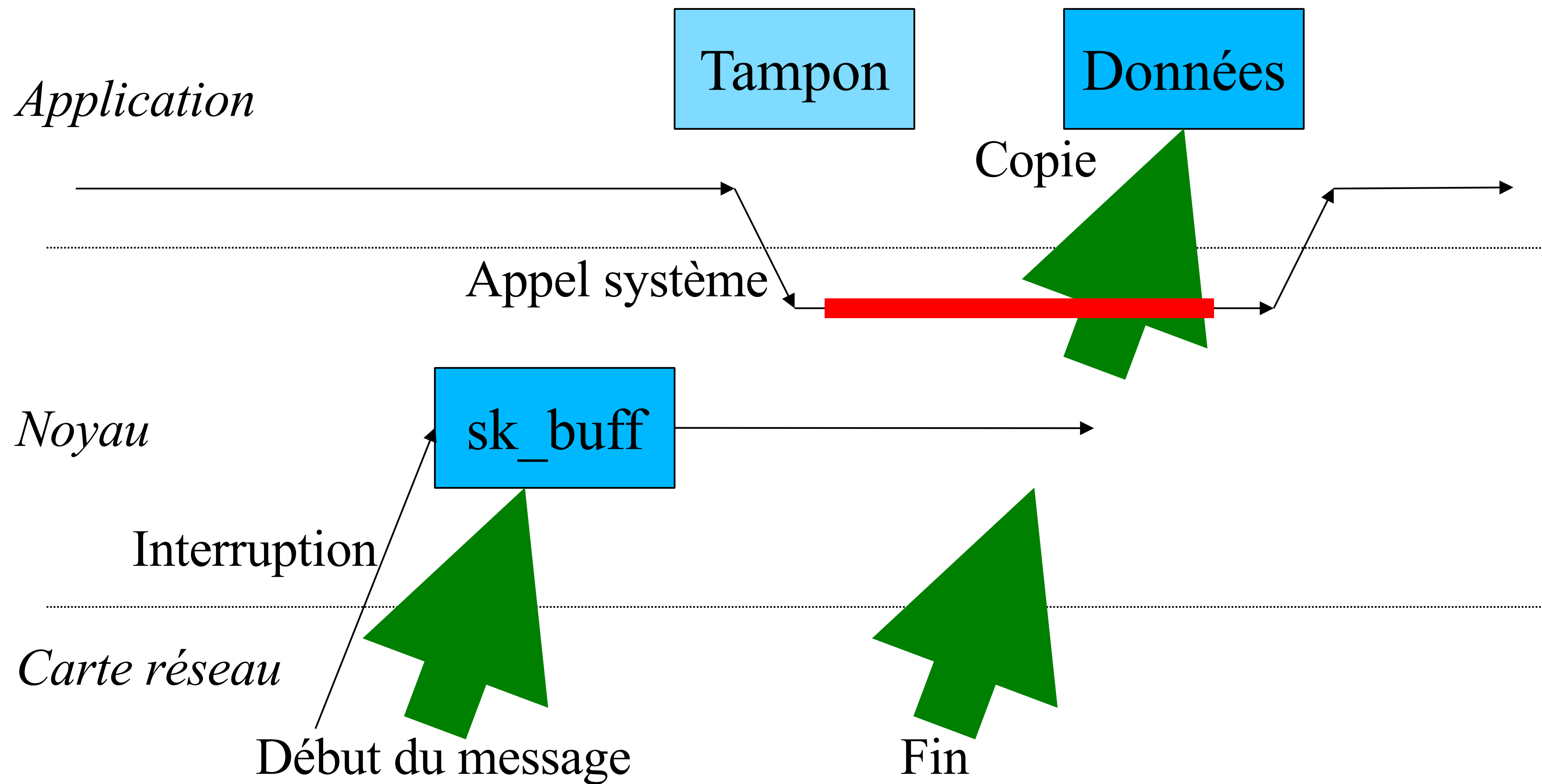
Emission bloquante



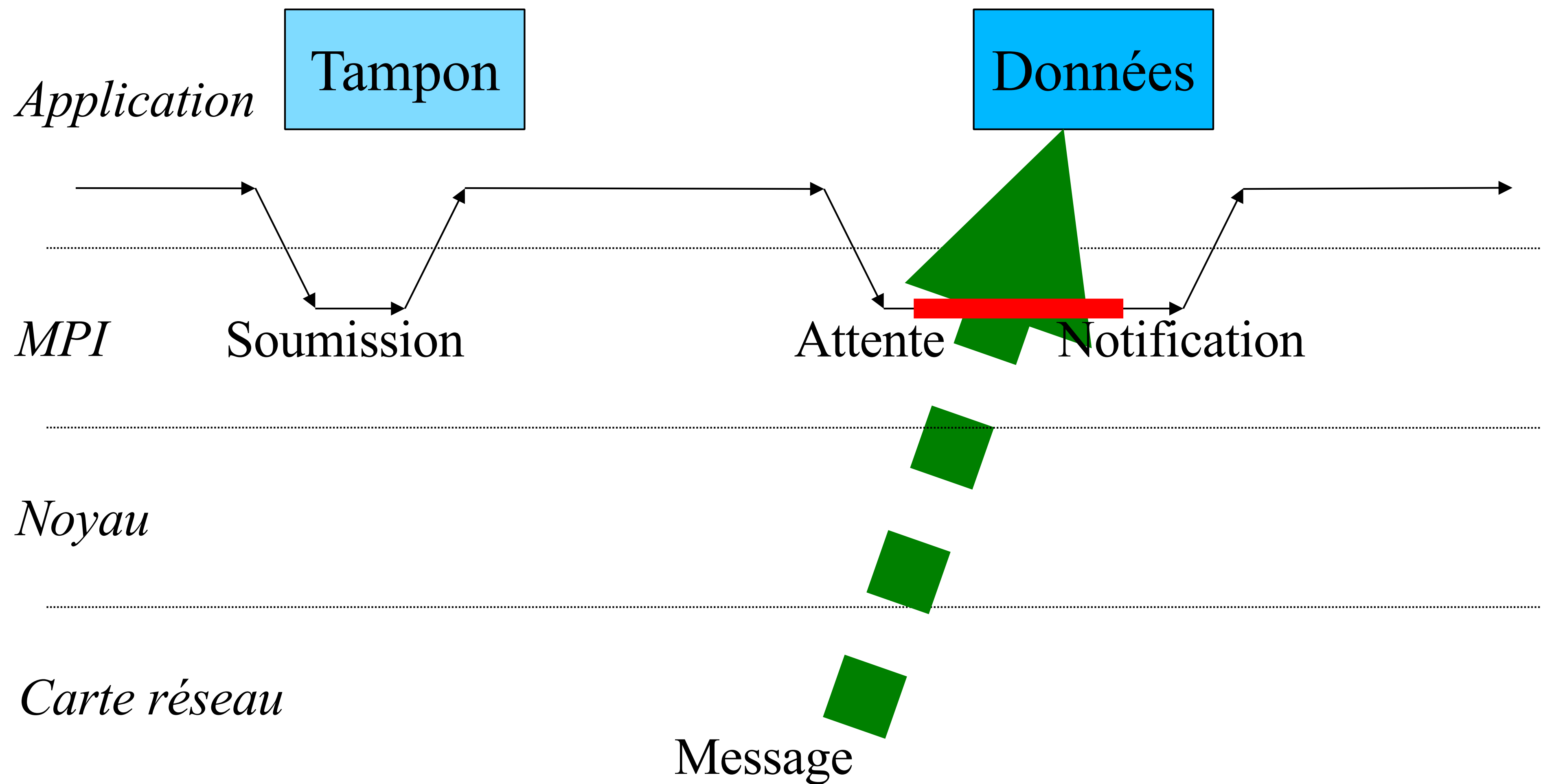
Emission non-bloquante



Réception bloquante



Réception non-bloquante

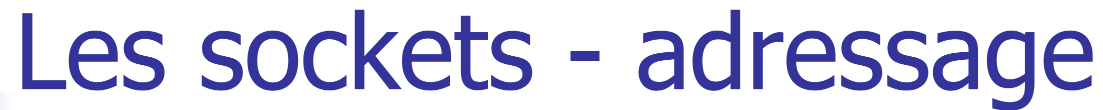


Désignation du destinataire/émetteur

- Pour faire du passage de messages, il est nécessaire de désigner l'autre extrémité de la communication
- Désignation explicite
  - du ou des processus destinataire(s)/émetteurs
- Désignation implicite
  - recevoir un message de n'importe qui
  - émettre un message à n'importe qui (diffusion)
  - une phase d'établissement de connexion désigne les deux entités communicantes

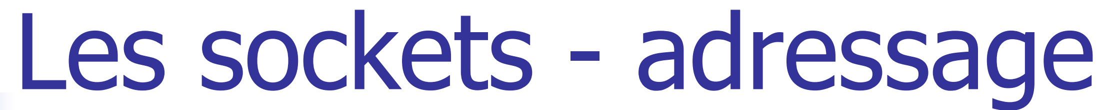
Les sockets

UCB Lyon 1



Les sockets - adressage

- Deux processus communiquent en émettant et recevant des données via les sockets
- Les sockets sont des portes d'entrées/sorties vers le réseau (la couche transport)
- Une socket est identifiée par une adresse de transport qui permet d'identifier les processus de l'application concernée
- Une adresse de transport = un numéro de port (identifie l'application) + une adresse IP (identifie le serveur ou l'hôte dans le réseau)



Les sockets - adressage

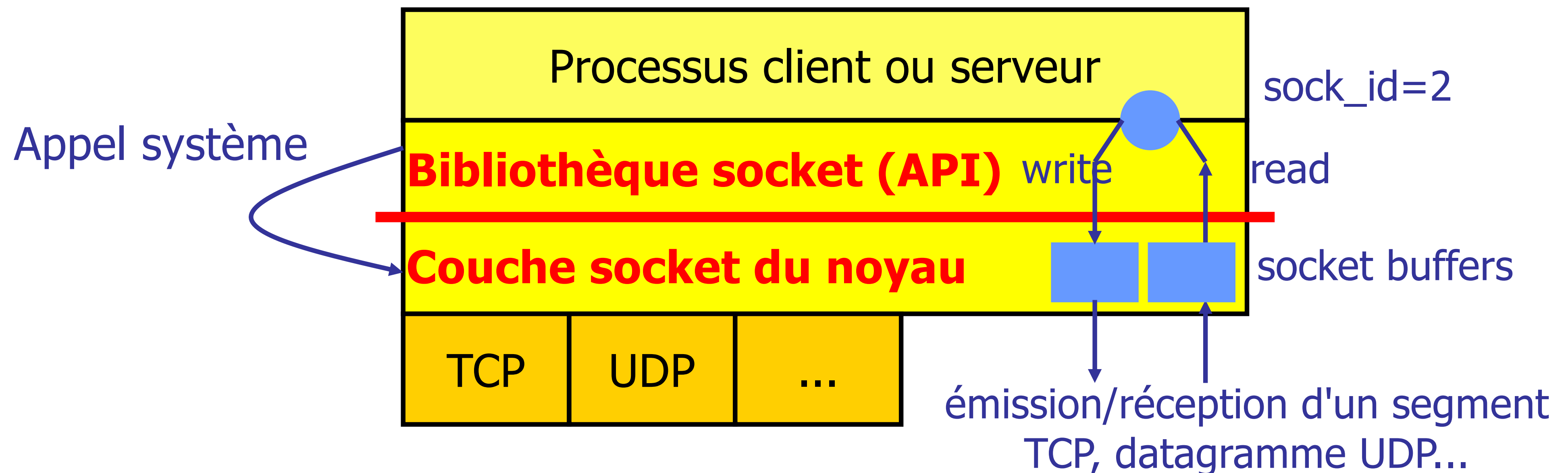
- Le serveur doit utiliser un numéro de port fixe vers lequel les requêtes clientes sont dirigées
- Les ports inférieurs à 1024 sont réservés :
  - "*well-known ports*"
  - ils permettent d'identifier les serveurs d'applications connues
  - ils sont attribués par l'IANA
- Les clients n'ont pas besoin 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, …
- Ces opérations sont des appels système
- Il existe différents types de socket associés aux différents services de transport :
  - *stream sockets (connection-oriented)* - SOCK_STREAM
    - utilise TCP qui fournit un service de transport d'octets fiable, dans l'ordre, entre le client et le serveur
  - *datagram sockets (connectionless)* - SOCK_DGRAM
    - utilise UDP (transport non fiable de datagrammes)
  - *raw sockets* - SOCK_RAW
    - utilise directement IP ou ICMP (ex. *ping*)

Les sockets en pratique

Un descripteur de socket (sock_id) n'est qu'un point d'entrée vers le noyau

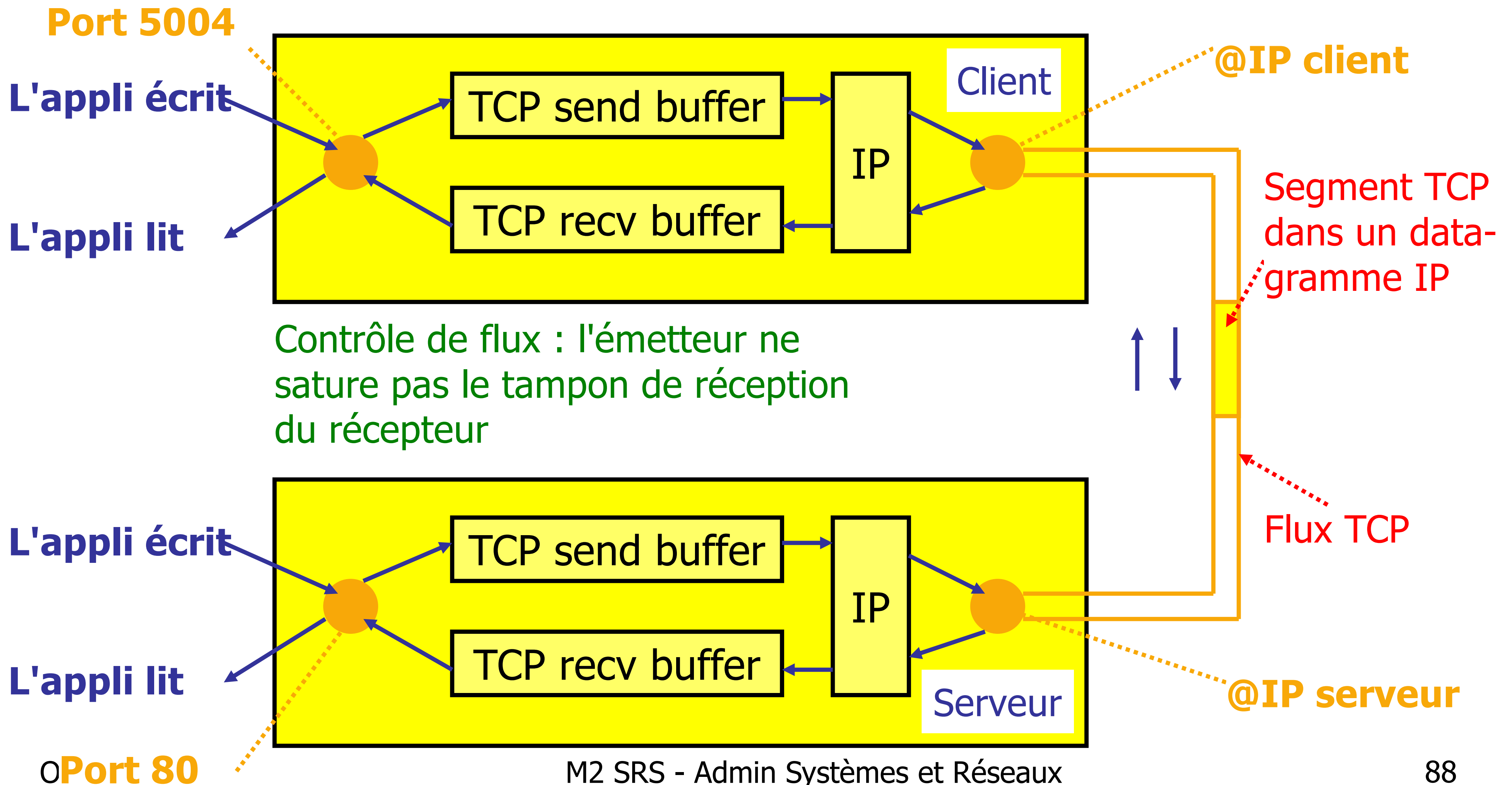


- la bibliothèque socket est liée à l'application

- la couche socket du noyau réalise l'adaptation au protocole de transport utilisé

Rappel - une connexion TCP

- Une connexion = **(proto, @IP_src, port_src, @IP_dest, port_dest)**



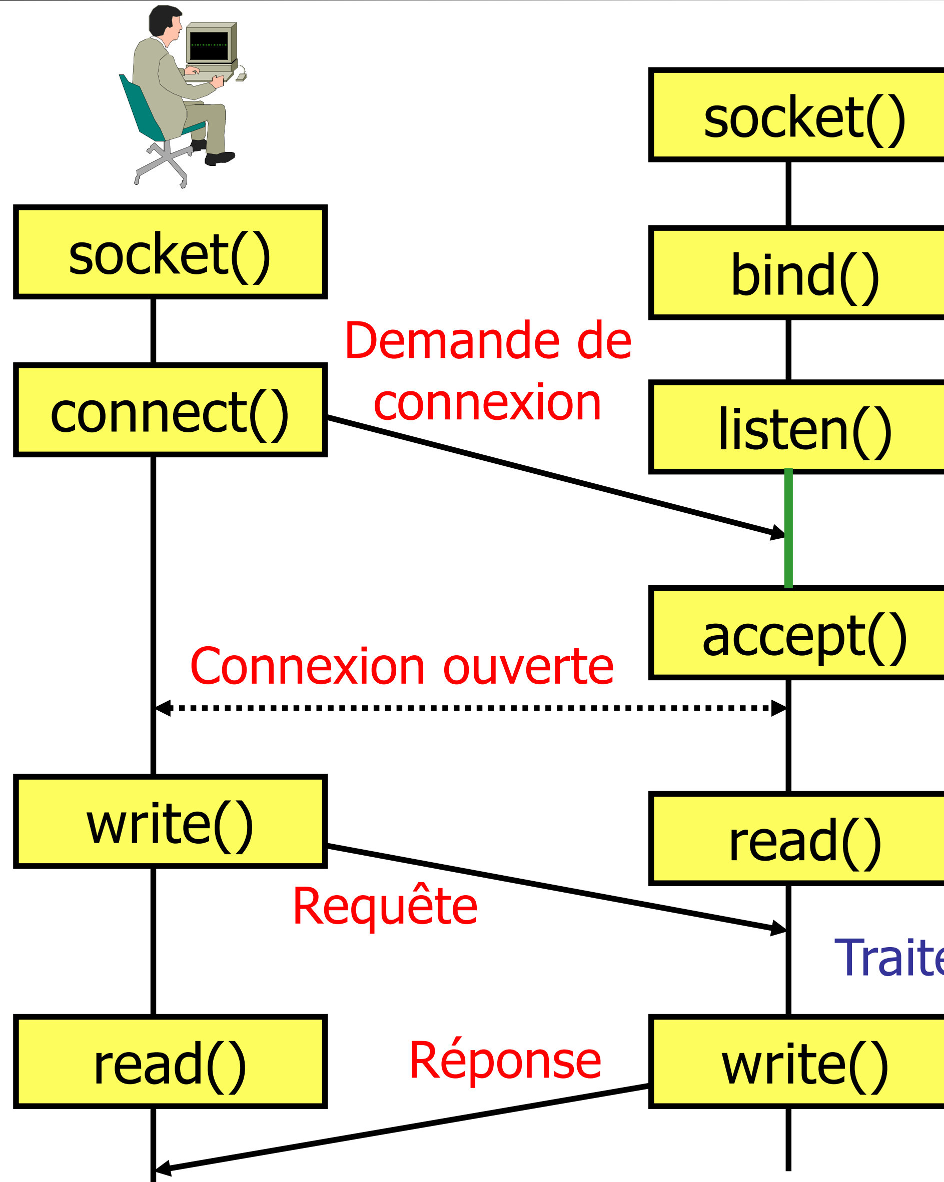
En mode connecté...

- 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
- 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

En mode connecté...

Création du descripteur local

Demande d'ouverture de connexion



Attachement d'un numéro de port à la socket

Le serveur autorise NMAX connexions (le service est ouvert !)

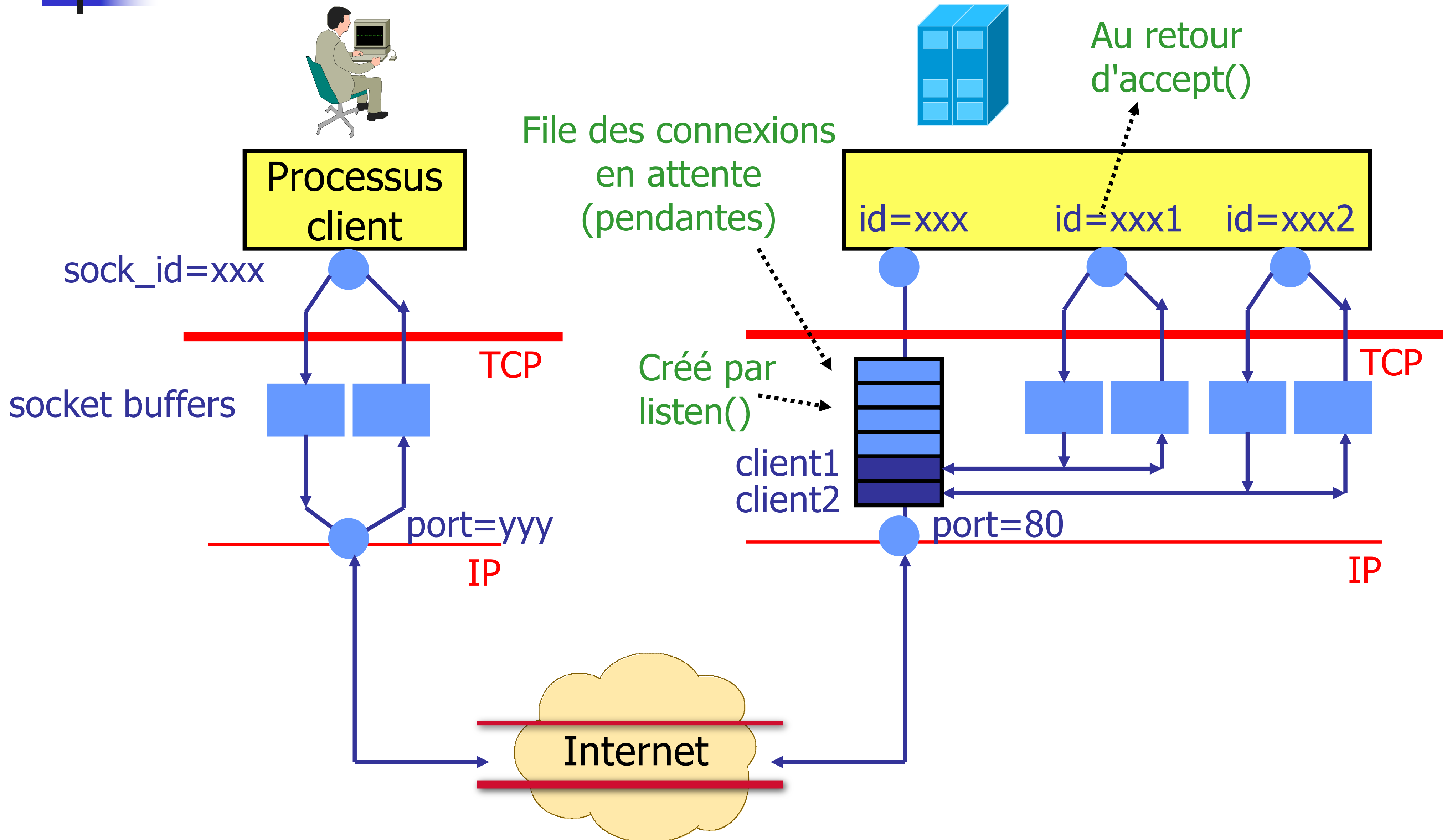
Le serveur accepte (ou attend) une connexion pendante et crée une nouvelle socket dédiée au client

Traitement de la requête

En mode connecté...

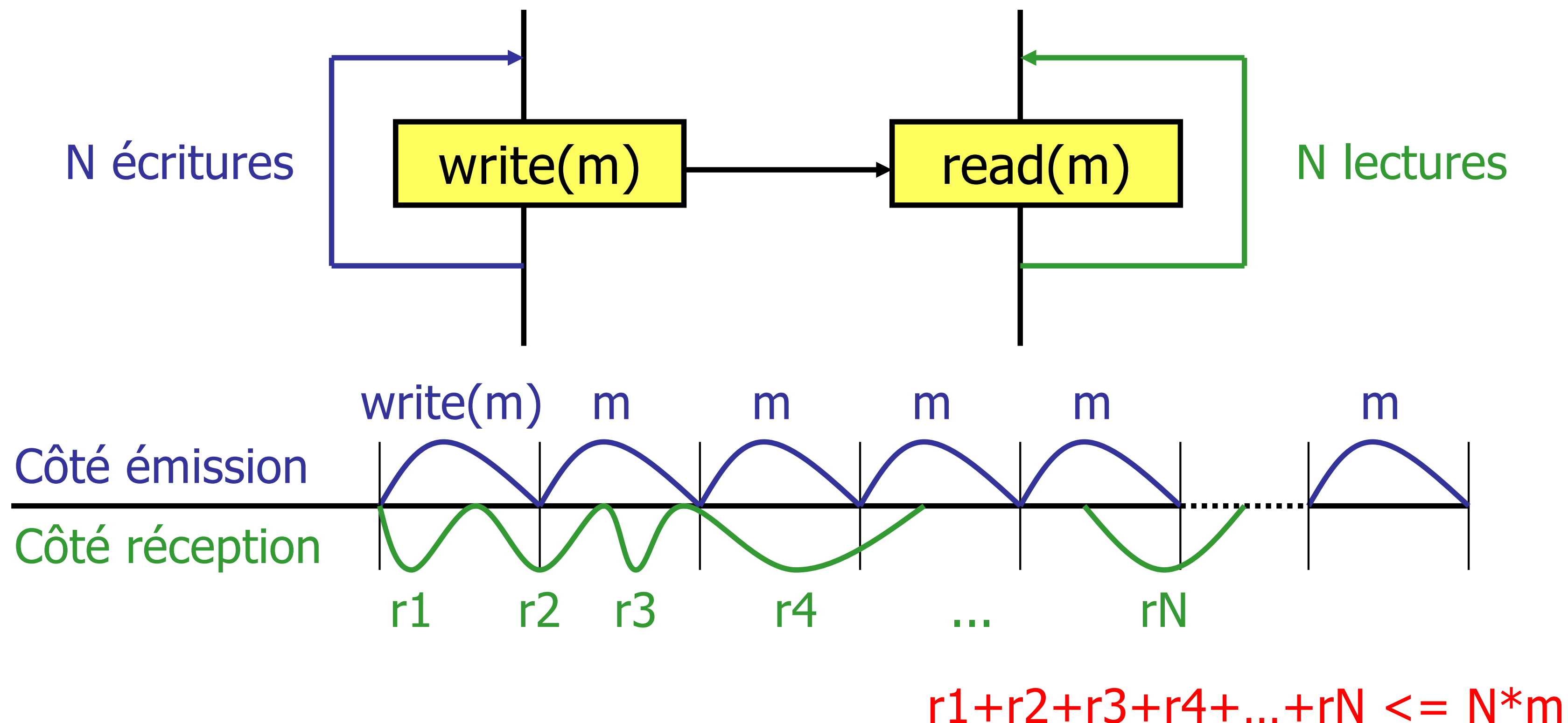
	Paramètres en entrée	Paramètres en sortie
socket()	type, domaine, protocole	sock_id
bind()	sock_id, port	
listen()	sock_id, NMAX	
connect()	sock_id, @sock_dest	
accept()	sock_id	@sock_src, client_sock_id
read()	client_sock_id, @recv_buf, lg	read_lg
write()	client_sock_id, @send_buf, lg	write_lg

En mode connecté...



En mode connecté...

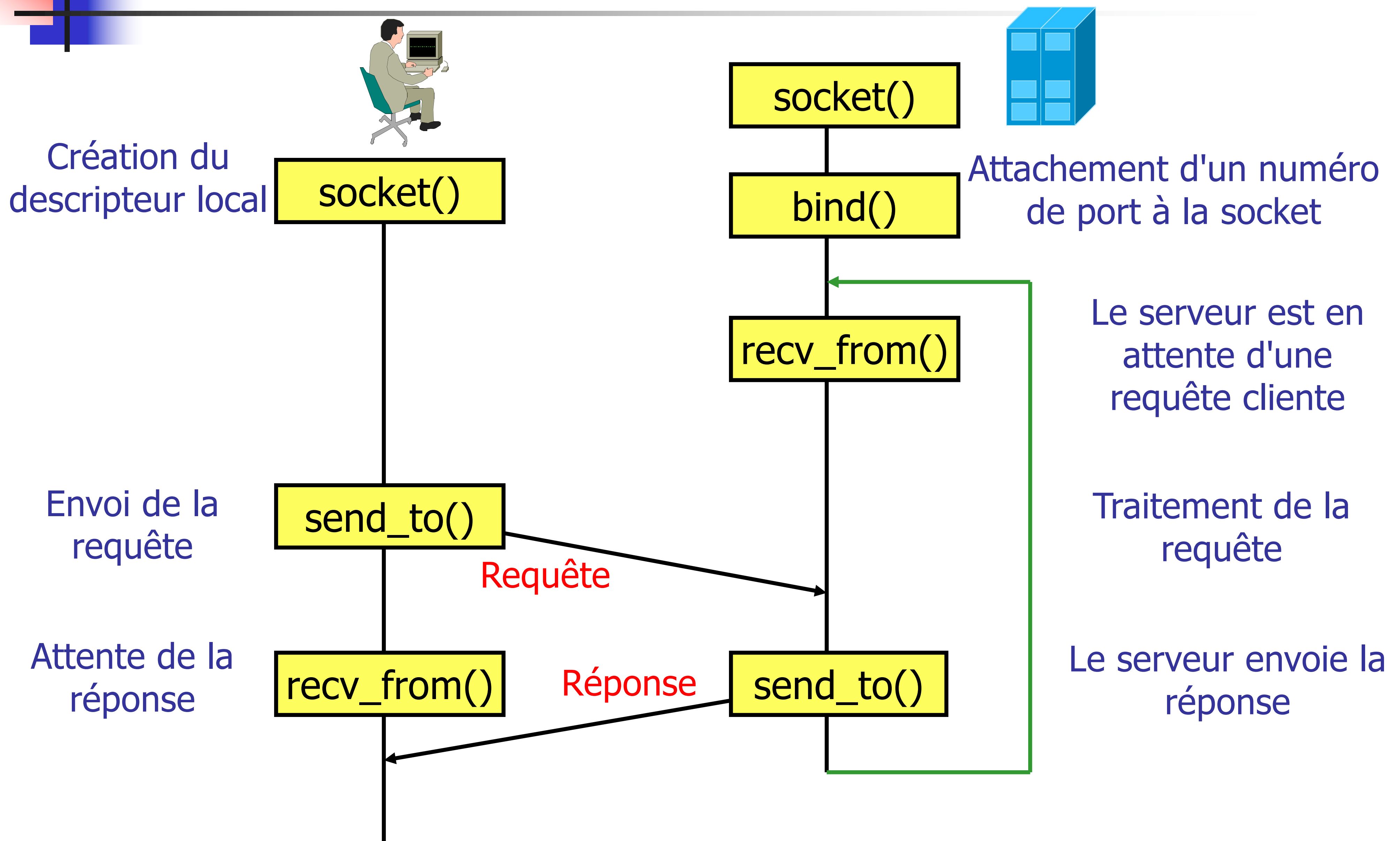
- Attention : les émissions/réceptions ne sont pas synchrones
  - *read(m)* : lecture **d'au plus** m caractères
  - *write(m)* : écriture de m caractères



En mode non connecté...

- 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

En mode non connecté...



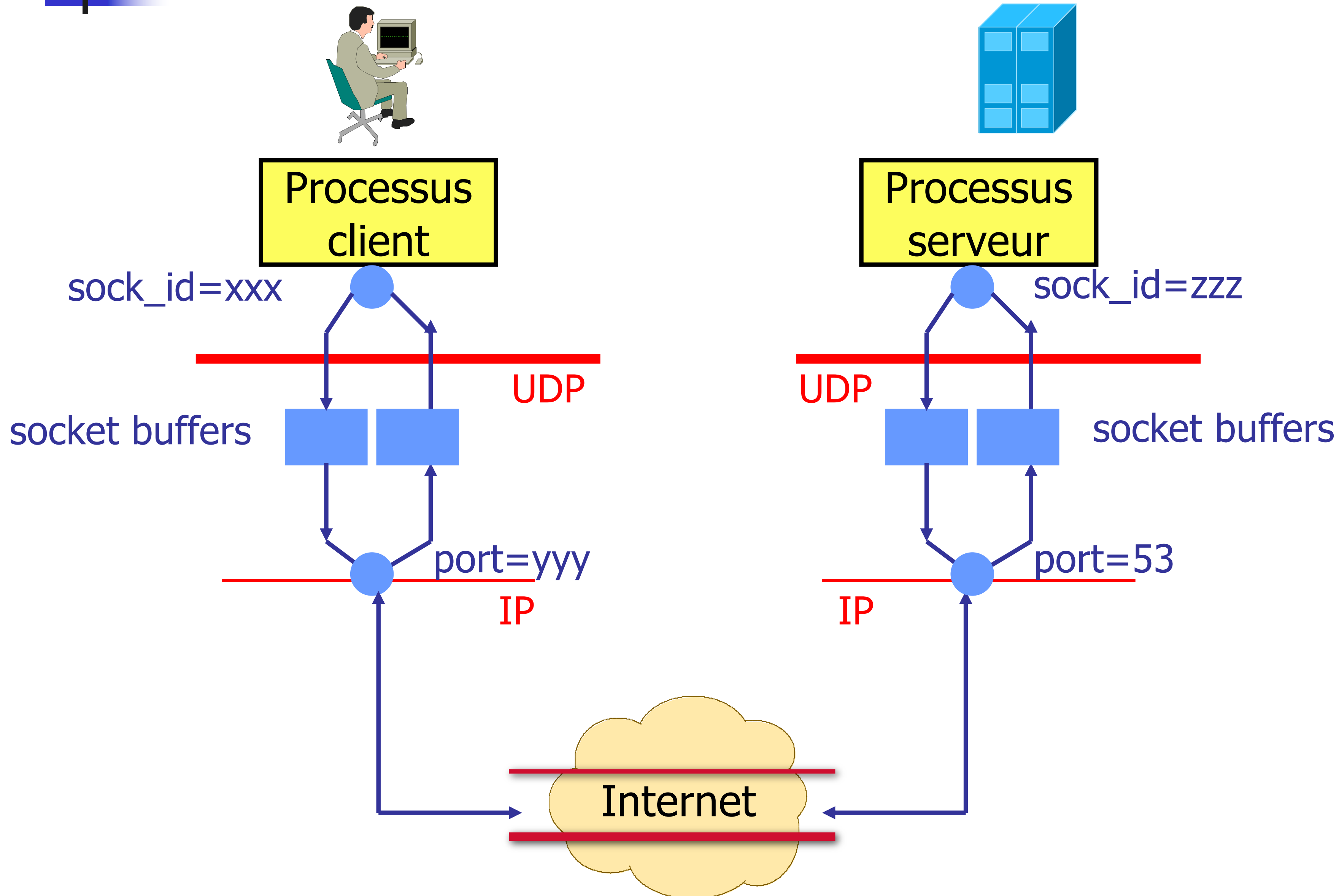
En mode non connecté...

	Paramètres en entrée	Paramètres en sortie
socket()	type, domaine, protocole	sock_id
bind()	sock_id, port	
recv_from()	sock_id, @recv_buf, lg	read_lg, **@sock_src**
send_to()	sock_id, **@sock_dest**, @send_buf, lg	write_lg

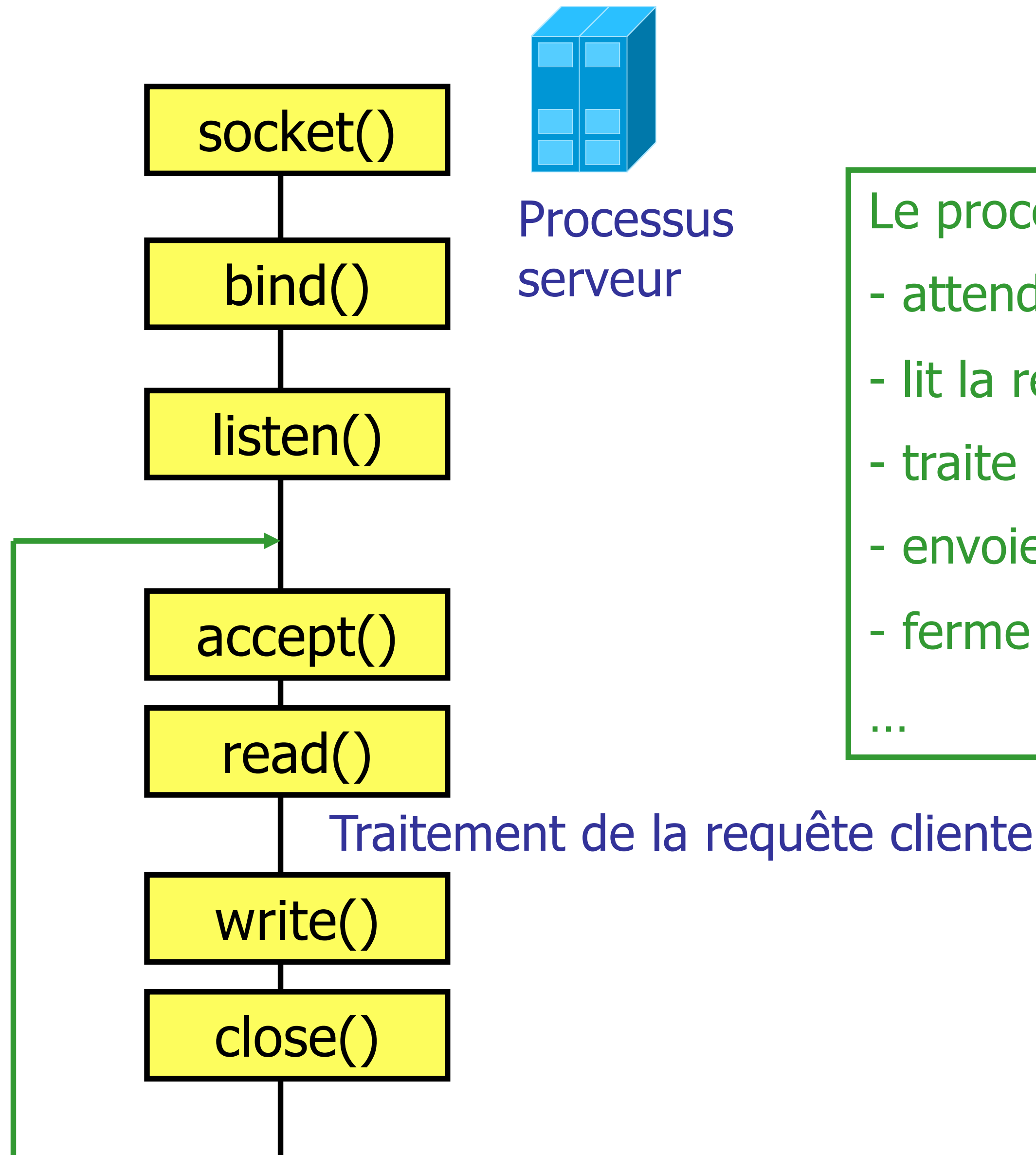
Rappel en mode connecté :

read()	client_sock_id, @recv_buf, lg	read_lg
write()	client_sock_id, @send_buf, lg	write_lg

En mode non connecté...



Serveur itératif en mode connecté

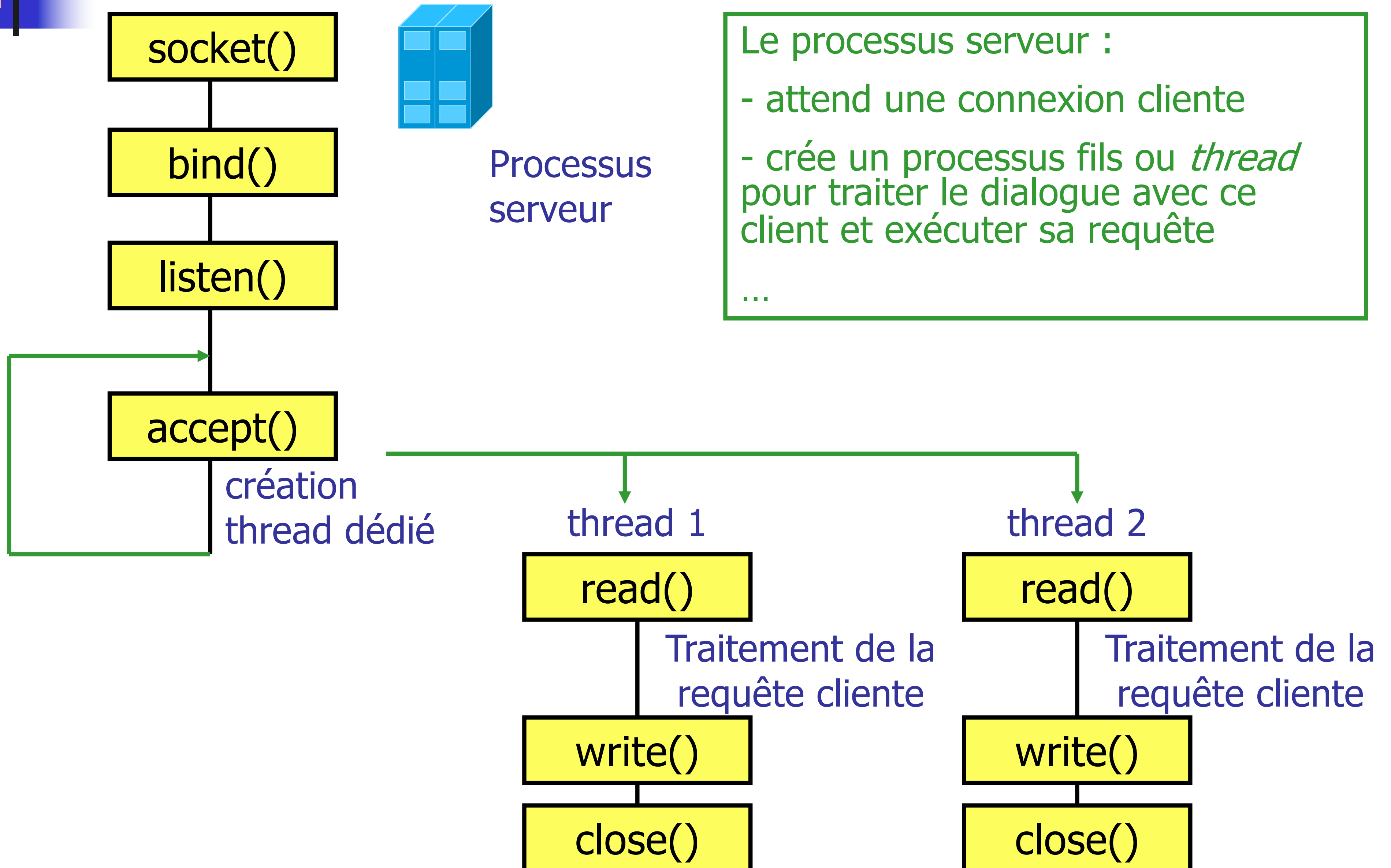


Le processus serveur :

- attend une connexion cliente
- lit la requête
- traite la requête
- envoie la réponse
- ferme la connexion cliente

…

Serveur concurrent en mode connecté



Le processus serveur :

- attend une connexion cliente
- crée un processus fils ou *thread* pour traiter le dialogue avec ce client et exécuter sa requête

…

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 paramètrer 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)

Opérations bloquantes/non bloquantes

- Comportement vis à vis de l'acceptation des connexions en mode non bloquant
  - s'il n'y a pas de connexion pendante, retourne -1 ... (l'application doit réessayer plus tard)
- Comportement vis à vis des demandes de connexions en mode non bloquant
  - la primitive `connect()` retourne immédiatement mais la demande de connexion n'est pas abandonnée au niveau TCP...

Paramétrage des sockets

- Les sockets sont paramétrables
  - fonctions `setsockopt()` et `getsockopt()`
  - options booléennes et non booléennes
- Exemples d'options booléennes
  - diffusion (dgram uniquement ; remplace l'@IP destinataire par l'@ de diffusion de l'interface)
  - `keepalive` : teste régulièrement la connexion (stream)
  - `tcpnodelay` : force l'envoi des segments au fur et à mesure des écritures dans le tampon
- Exemples d'options non booléennes
  - taille du tampon d'émission, taille du tampon de réception, type de la socket

Les serveurs multi-protocoles

- Un serveur qui offre le même service en mode connecté et non connecté
  - exemple : DAYTIME (RFC 867) port 13 sur UDP et sur TCP qui permet de lire la date et l'heure sur le serveur
  - 13/TCP : la demande de connexion du client déclenche la réponse (à une requête donc implicite) : le client n'émet aucune requête
  - 13/UDP : la version UDP de DAYTIME requiert une requête du client : cette requête consiste en un datagramme arbitraire qui n'est pas lu par le serveur mais qui déclenche l'émission de la donnée côté serveur
- Le serveur écoute sur 2 sockets distinctes pour rendre le même service

Les serveurs multi-protocoles

- Pourquoi un serveur multi-protocoles ?
  - certains systèmes ferment tout accès à UDP pour des raisons de sécurité (pare-feu)
  - non duplication des ressources associées au service (corps du serveur)
- Fonctionnement
  - 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 et en mode concurrent

Les serveurs multi-protocoles

- En 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
    - lorsque les échanges sont terminés, le serveur réitère son attente sur les deux sockets initiales

Les serveurs multi-protocoles

- En 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

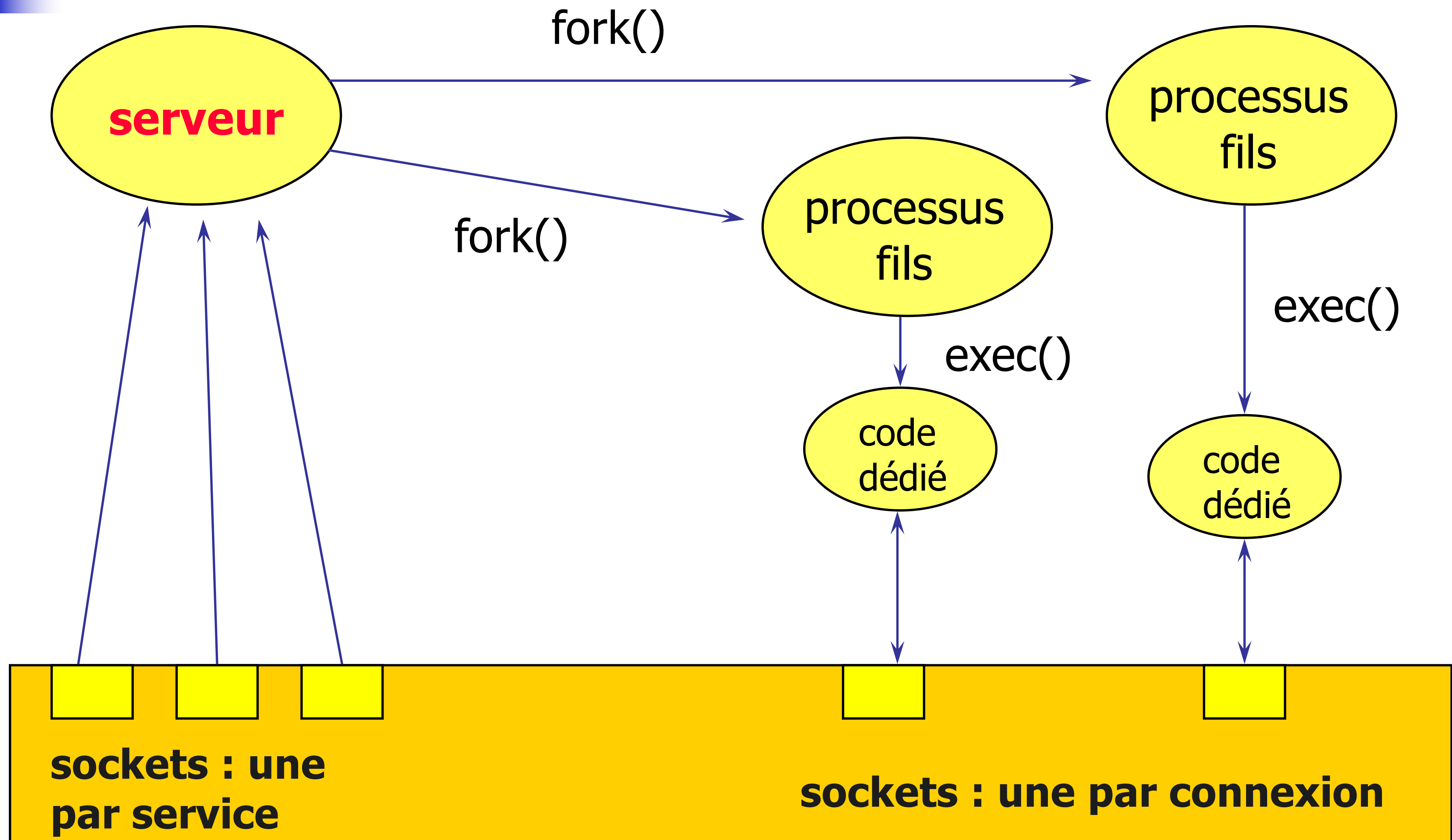
Les serveurs multi-services

- Un serveur qui répond à plusieurs services (une socket par service)
- Pourquoi un serveur multi-services ?
  - problème lié à la multiplication des serveurs : le nombre de processus nécessaires et les ressources consommées qui y sont associées
- 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 : lancement d'un programme différent selon la requête entrante
  - 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 serveurs multi-services



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
  - qui est en attente des requêtes clientes
- On parle de démon
- A priori, il faudrait un démon par service
- Problème : multiplication des services --> multiplication du nombre de démons
- Sous UNIX, un super-démon : `inetd`

Le démon *inetd*

- 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 fichier /etc/inetd.conf

```
#  Internet services syntax :
#  <service_name> <socket_type> <proto> <flags> <user> <server_pathname> <args>
#  wait : pour un service donné, un seul serveur peut exister à un instant donné
# donc le serveur traite l'ensemble des requêtes à ce service
# stream --> nowait : un serveur par connexion
ftp     stream    tcp    nowait   root   /etc/ftpd     ftpd -l
tftp    dgram     udp    wait     root   /etc/tftpd    tftpd
shell   stream    tcp    nowait   root   /etc/rshd     rshd
pop3    stream tcp   nowait  root   /usr/local/lib/popper   popper -s -d -t /var/log/poplog
# internal services :
# => service réalisé par inetd directement
time    stream    tcp    nowait    root    internal
time    dgram     udp    nowait    root    internal
```

La scrutation de plusieurs sockets

- Scrutation : mécanisme permettant l'attente d'un événement (lecture, connexion, ...) sur plusieurs points de communication
  - nécessaire dans le cas des serveurs multi-services ou multi-protocoles
- Problème lié aux caractères bloquants des primitives
  - exemple : une attente de connexion (`accept`) sur une des sockets empêche l'acceptation sur les autres…
- Première solution
  - rendre les primitives non bloquantes à l'ouverture de la socket
  - inconvénient : attente active (dans une boucle)

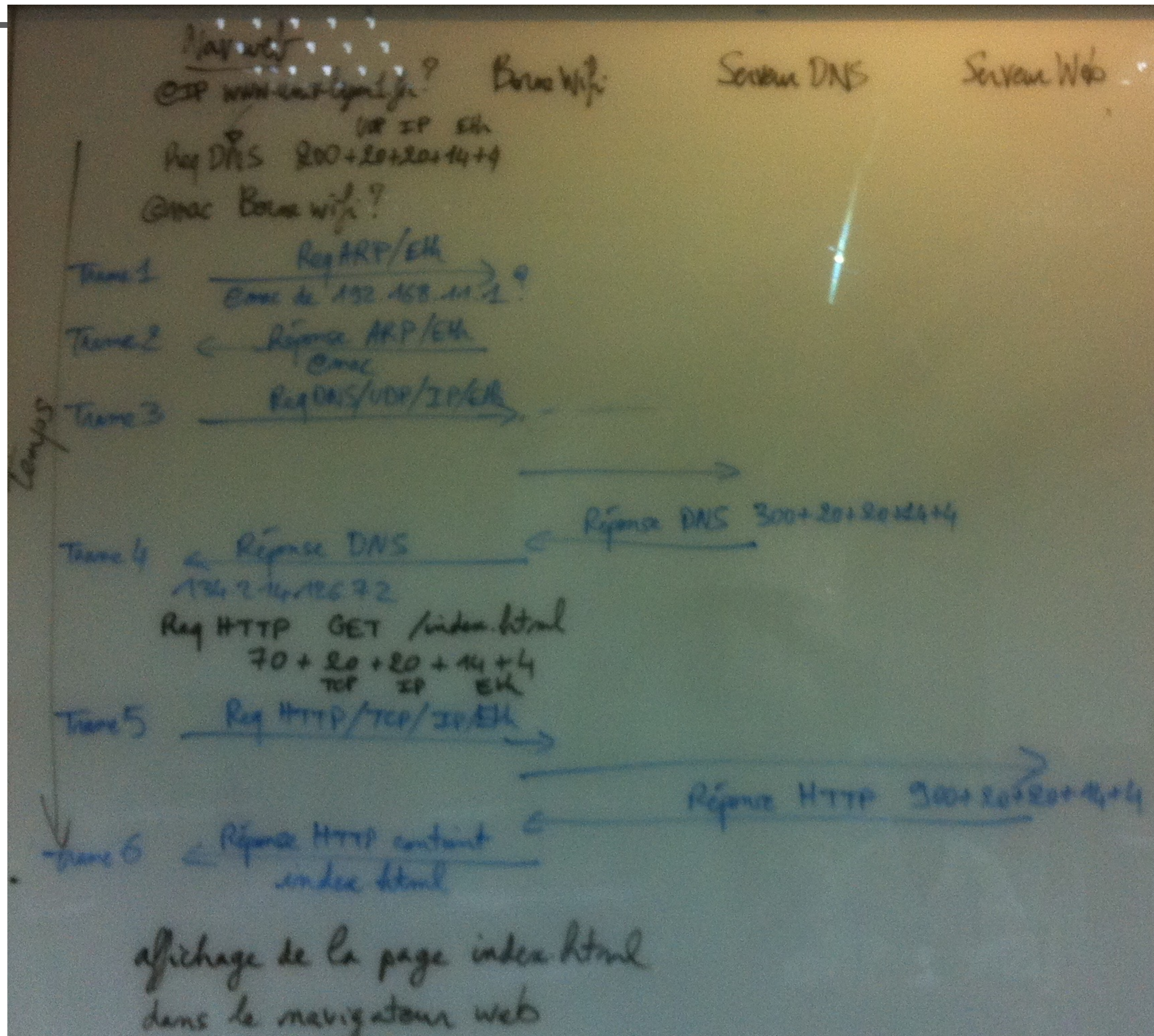
- Deuxième solution
  - créer un fils par socket pour la scrutation d'un service
  - inconvénient : lourd, gaspillage de ressources
  - mais avantage conservé d'activation à la demande
- Troisième solution : la primitive `select()`
  - permet de réaliser un multiplexage d'opérations bloquantes (scrutation) sur des ensembles de descripteurs passés en argument :
    - descripteurs sur lesquels réaliser une lecture
    - descripteurs sur lesquels réaliser une écriture
    - descripteurs sur lesquels réaliser un test de condition exceptionnelle (arrivée d'un caractère urgent)
  - un argument permet de fixer un temps maximal d'attente avant que l'une des opérations souhaitées ne soit possible

La scrutation de plusieurs sockets

- La primitive `select()` rend la main quand une de ces conditions se réalise :
  - l'un des événements attendus sur un descripteur de l'un des ensembles se réalise : les descripteurs sur lesquels l'opération est possible sont dans un paramètre de sortie
  - le temps d'attente maximum s'est écoulé
  - le processus a capté un signal (provoque la sortie de `select()`)

Exemple d’une requête HTTP

Exemple d’une requête HTTP



Les appels de procédures distantes

UCB Lyon 1

Deux approches de conception

- Un concepteur d'application distribuée peut procéder selon deux approches :
  - conception orientée communication :
    - définition du protocole d'application (format et syntaxe des messages) inter-opérant entre le client et le serveur
    - conception des composants serveur et client, en spécifiant comment ils réagissent aux messages entrants et génèrent les messages sortants
  - conception orientée application :
    - construction d'une application conventionnelle, dans un environnement mono-machine
    - subdivision de l'application en plusieurs modules qui pourront s'exécuter sur différentes machines

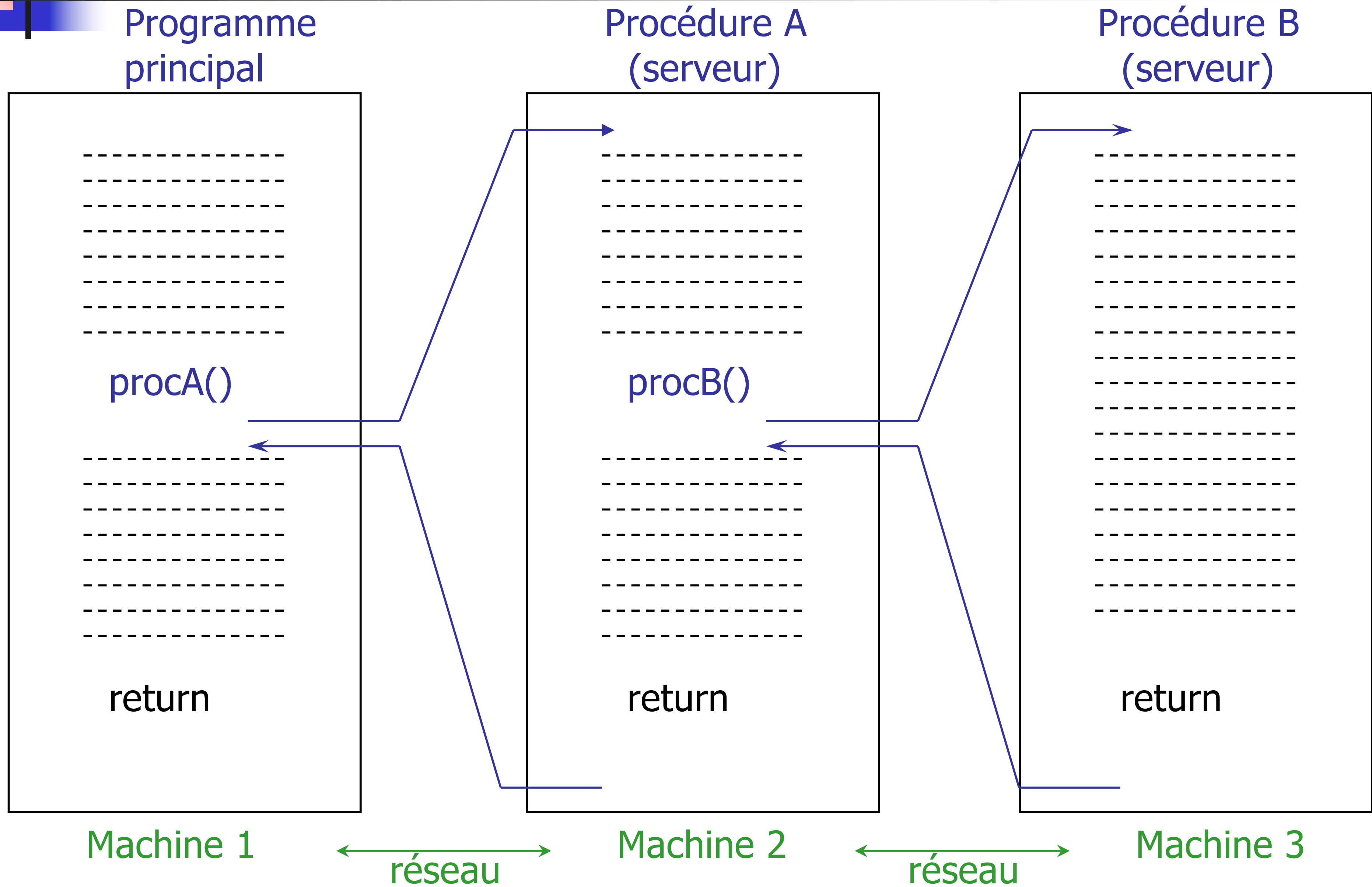
Principe général

- Souvent, quand un client envoie une requête (des paramètres), il est bloqué jusqu'à la réception d'une réponse
- Analogie avec un appel de fonction
  - la fonction ou procédure ne rend la main au programme appelant qu'une fois le traitement (calcul) terminé
- 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

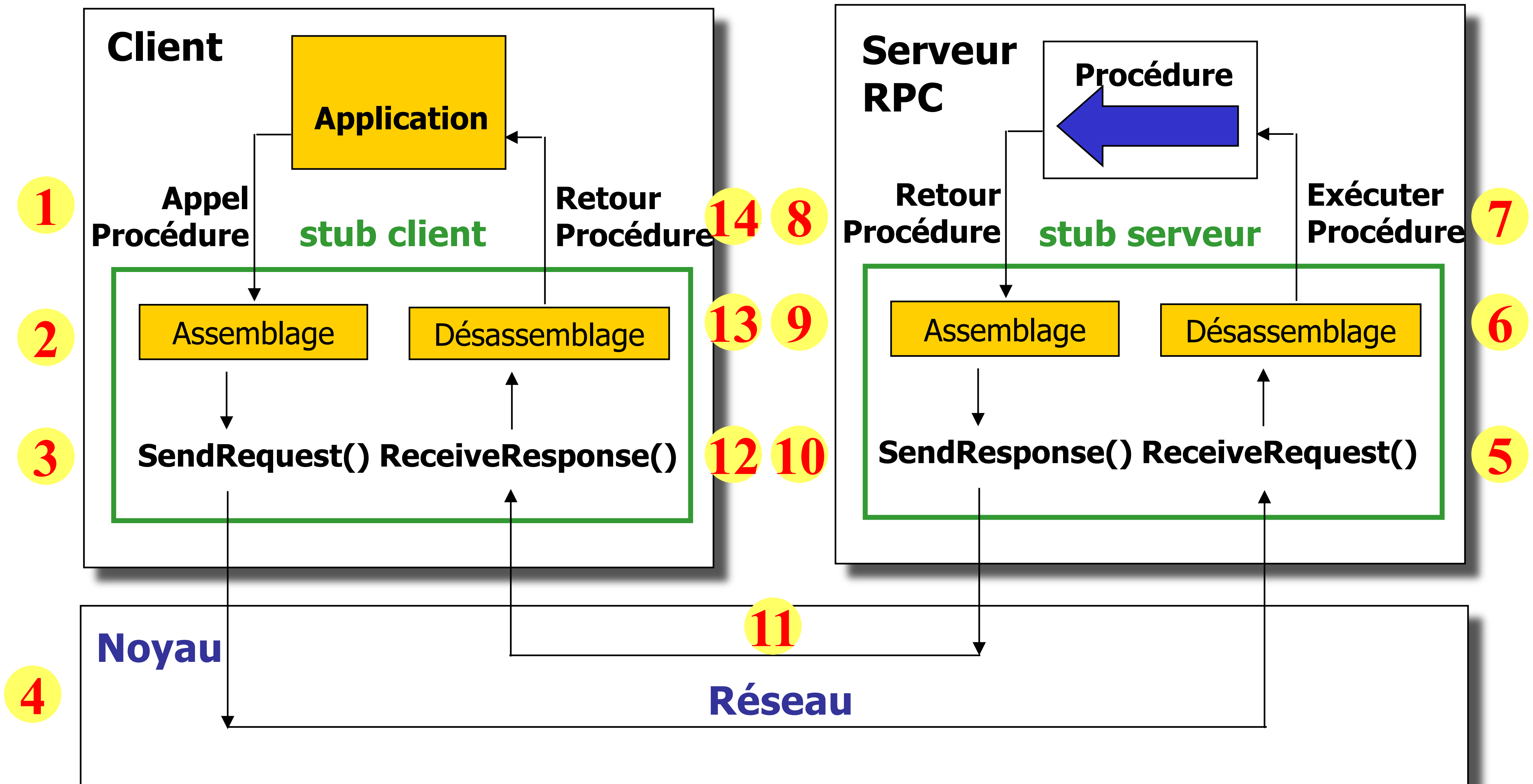
Principe général

- 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" !

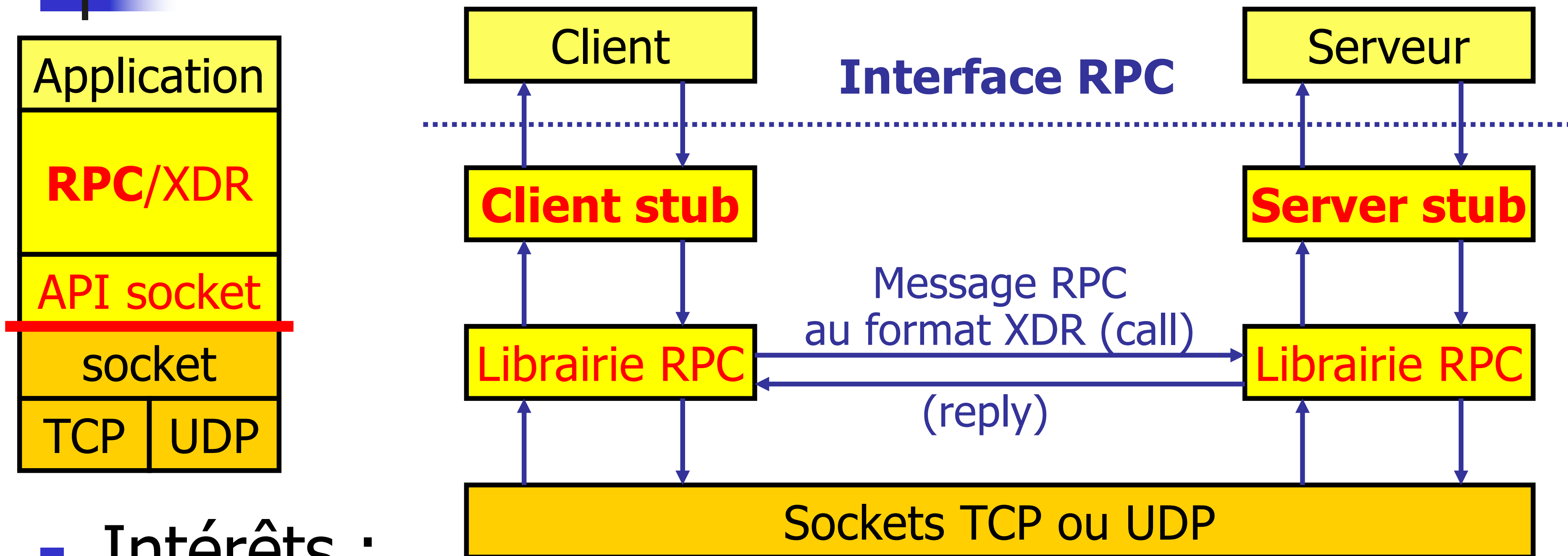
Le modèle RPC



Le modèle RPC



L'interface RPC



- Intérêts :
  - l'application n'a pas à manipuler directement les sockets (le transport des données est transparent)
  - l'implémentation des RPC est indépendante de l'OS
- Inconvénient :
  - l'utilisation des RPC est moins performante que l'utilisation directe des sockets (couches supplémentaires)

Restrictions liées aux RPC

- Pas de passage de paramètres par adresse : impossible de passer des pointeurs (ou références)
  - en effet, les espaces d'adressage du client et du serveur sont différents donc aucun sens de passer une adresse
- La procédure distante n'a pas accès aux variables globales du client, aux périphériques d'E/S (affichage d'un message d'erreur !)
- Un appel de procédure obéit à fonctionnement synchrone : une instruction suivant un appel de procédure ne peut pas s'exécuter tant que la procédure appelée n'est pas terminée

Le protocole RPC

- Il doit définir le format du `call` (message du client vers le serveur), le format des arguments de la procédure, le format du `reply` (résultats)
- Il doit permettre d'identifier la procédure à exécuter par le serveur quand un `call` arrive
- Il doit permettre d'authentifier la demande (problèmes de sécurité)
  - Quelles machines distantes sont autorisées à exécuter la procédure ?
  - Quels utilisateurs sont autorisés à exécuter la procédure ?

L'implémentation de SUN

- Sun Microsystems a développé une technologie RPC dite « Sun RPC » devenue aujourd'hui un standard de fait
- NFS (*Network File Sytem*) repose sur les Sun RPC
- Les Sun RPC définissent :
  - 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
- Possibilité d'utiliser UDP ou TCP pour les communications
- XDR assiste les RPC pour assurer le fonctionnement dans un environnement hétérogène (représentation standard des arguments et résultats…)

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
- Les procédures d'un programme distant sont identifiées séquentiellement par les entiers 0, 1, ..., N
- 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

Identification des procédures distantes

- La procédure de numéro 0 permet de tester la disponibilité du service
- Un identifiant de programme peut correspondre à plusieurs processus de service (`mount/showmount`)

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"

- Les RPC sur un protocole de transport non fiable (UDP)
  - si un appel de procédure distante s'exécutant sur UDP ne retourne pas, l'appelant ne peut pas savoir si la procédure a été exécutée ou si la réponse a été perdue
  - du côté de l'appelant : la réception d'un `reply` signifie uniquement que la procédure distante a été exécutée au moins une fois
  - du côté de serveur : un serveur recevant plusieurs fois la même requête ne peut pas savoir si le client s'attend à une unique exécution de la procédure ou bien s'il s'agit effectivement de N exécutions distinctes de la même proc.

La sémantique "au moins une fois"

- Le concepteur d'une application RPC utilisant UDP doit prendre en compte le fait que la non réception d'un `reply` ne signifie pas que la procédure distante n'a pas été exécutée...
- Exemple :
  - lecture dans un fichier distant : pas gênant si une demande de lecture a généré deux exécutions de la procédure
  - écriture dans un fichier distant : gênant s'il s'agit d'un ajout en fin de fichier ; la chaîne peut être ajoutée deux fois au lieu d'une seule…
- Les procédures doivent être idempotentes :
  - --> pas de procédure d'ajout en fin de fichier mais une procédure d'écriture à telle position (ajout d'un paramètre précisant où écrire dans le fichier)

Communications client/serveur

- Les sockets utilisent un *well-known* port pour contacter un serveur distant (ex: telnet=port 23)
- Les clients RPC ne connaissent que l'identifiant du programme RPC distant et le numéro de procédure (ex: 100003 pour NFS)
- Pourtant, les communications sous-jacentes se font en mode client/serveur : l'appelant doit connaître l'adresse (IP, port) utilisée par le programme RPC distant (ex: `nfsd`)

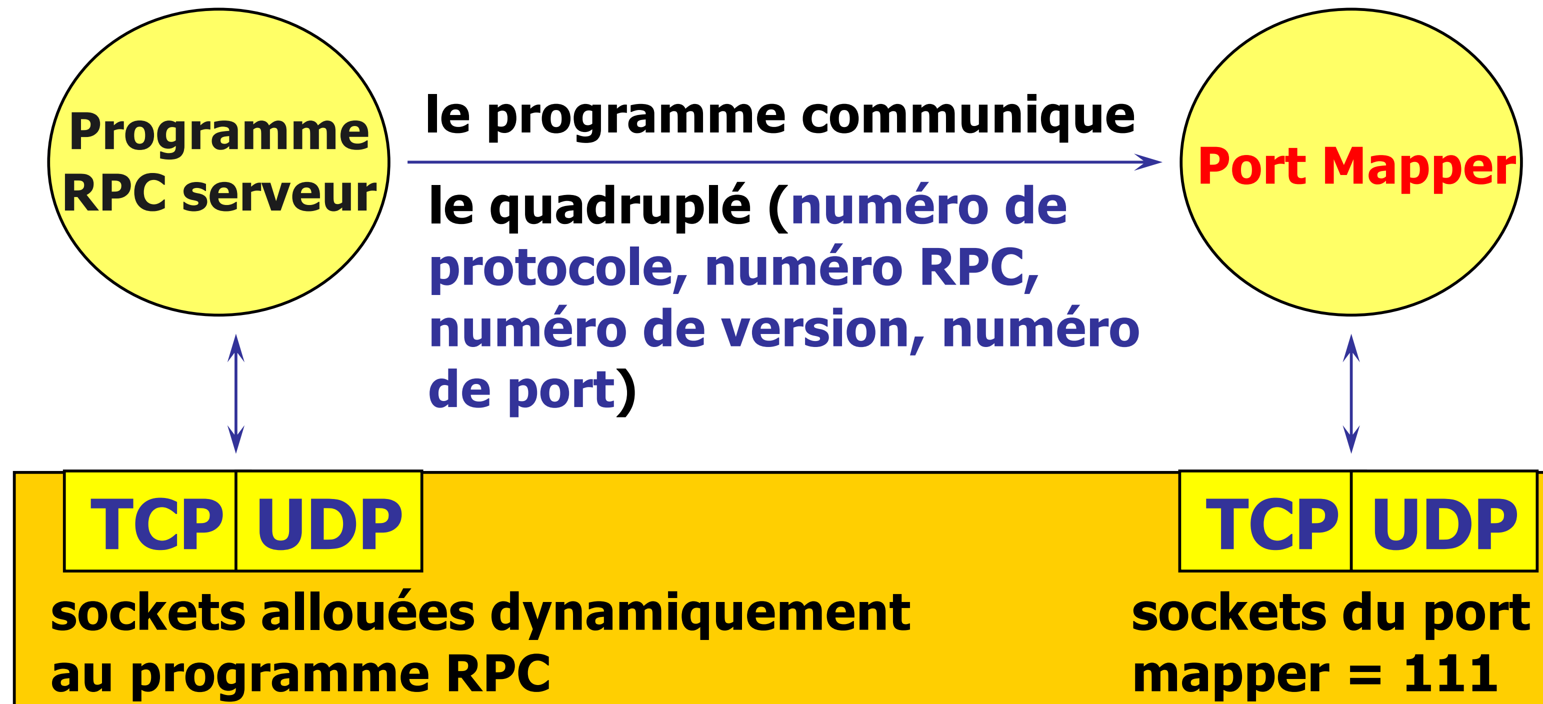
Communications client/serveur

- Le numéro de port du processus serveur est attribué dynamiquement quand il démarre
  - --> car le nombre de programmes RPC (identifiant sur 32 bits) est potentiellement supérieur au nombre de *well-known* ports (numéro de port sur 16 bits, ports réservés entre 0 et 1023)
- Un processus spécial, le démon `portmap` (ou `rpcbind`) maintient une base de données renseignant les associations locales entre numéro de port et programme RPC

Le processus portmap (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

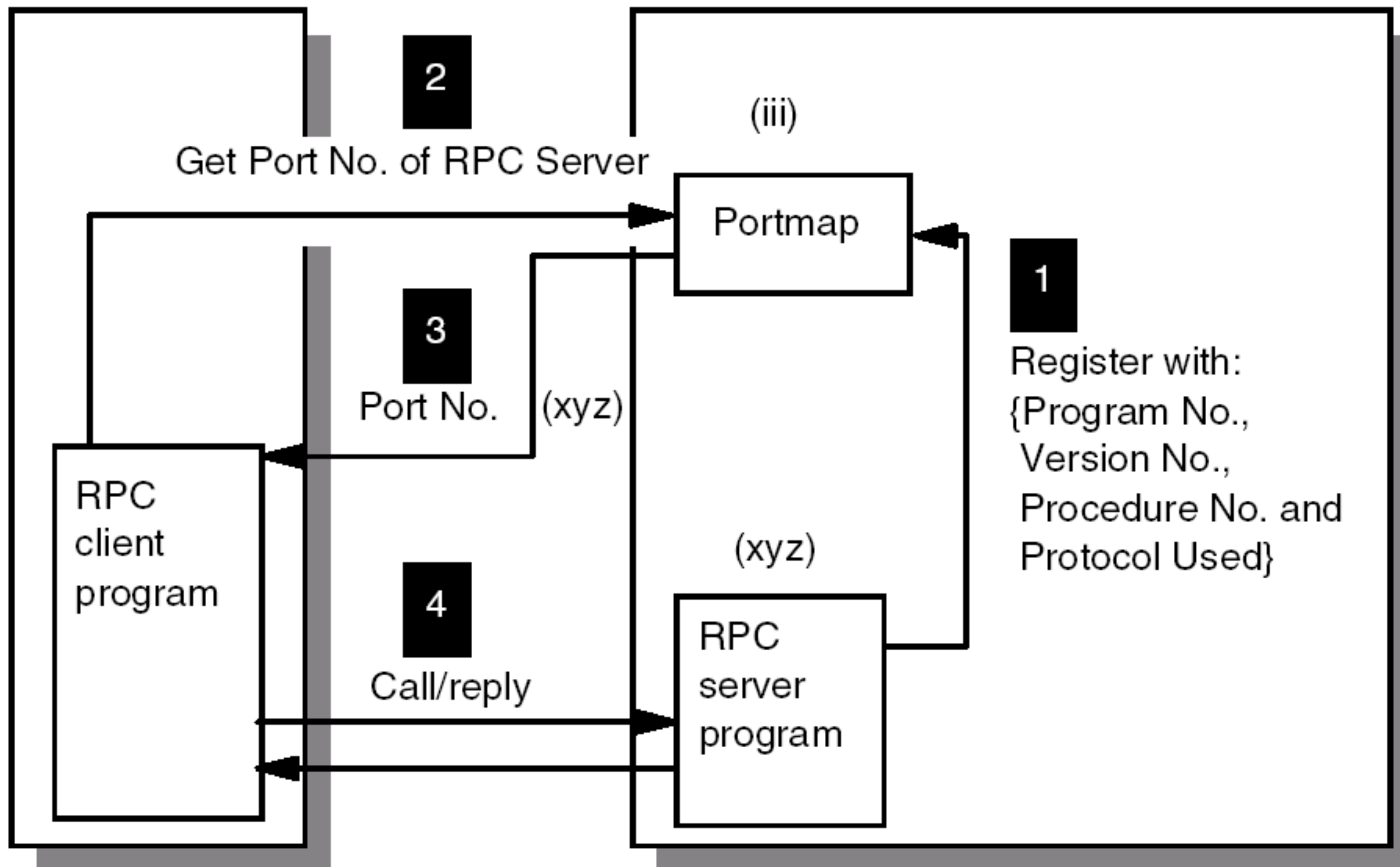
Le processus portmap (rpcbind)



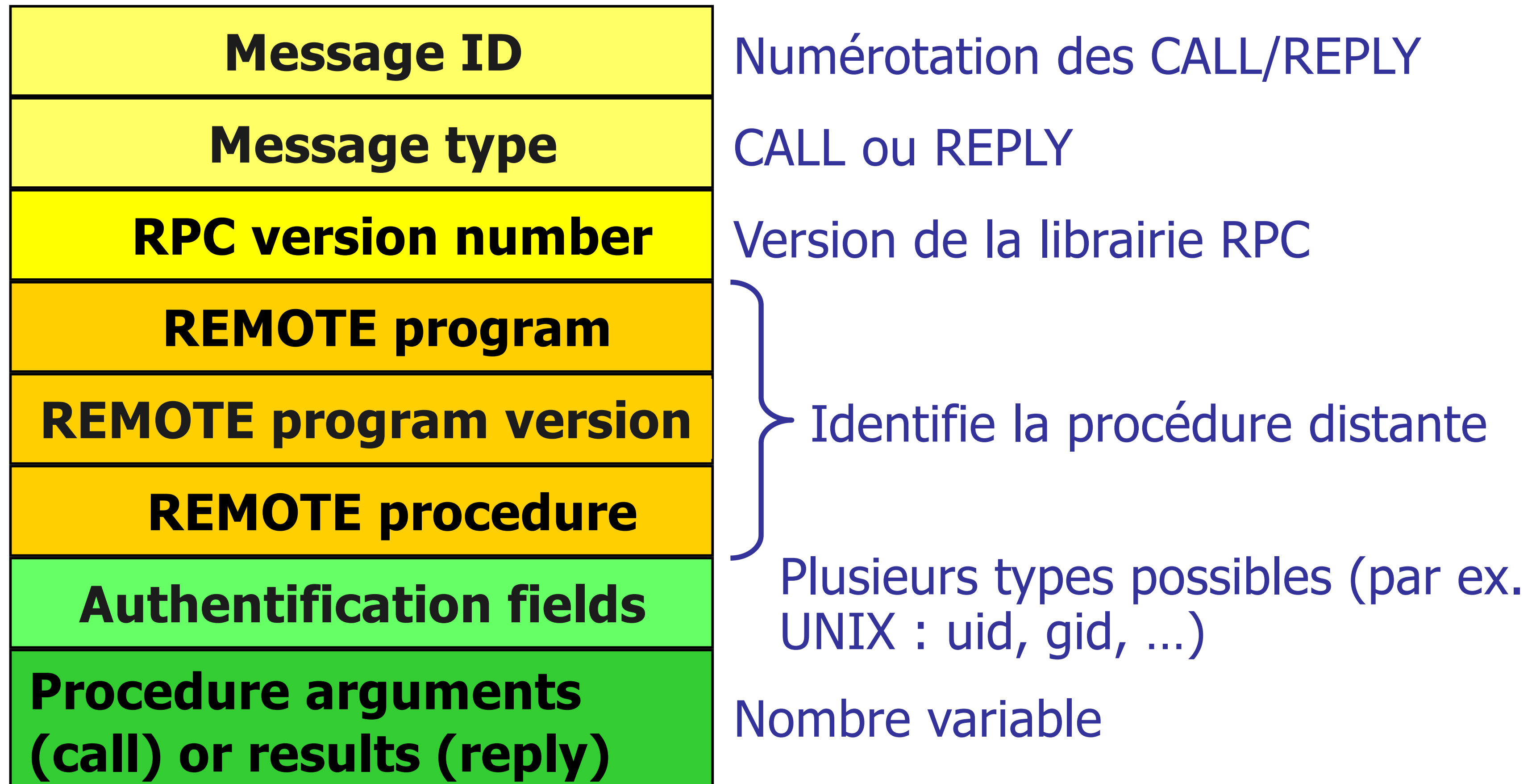
Le processus portmap (rpcbind)

- Les procédures du port mapper
  - 0 : fonction vide (teste la présence de portmap)
  - 1 : enregistrement d'un service (local)
  - 2 : annulation d'un service (local)
  - 3 : demande du numéro de port d'un service enregistré localement
  - 4 : liste tous les services enregistrés localement
  - 5 : appel d'une procédure distante via le port mapper --> permet de "pinger" une procédure distante

Utilisation du port mapper (rpcbind)



Le format des messages RPC



Le format est de longueur variable car le nombre d'arguments de la procédure appelée est variable

Les réponses possibles

- Plusieurs types de réponses possibles :
  - SUCCESS : les résultats de la procédure sont renvoyés au client
  - RPC_MISMATCH : les versions RPC du client et du serveur ne sont pas compatibles
  - AUTH_ERROR : problème d'authentification
  - PROG_MISMATCH : la procédure demandée n'est pas disponible (problème de version du programme, …)
- Plus de détails : RFC 1057

La représentation XDR

- Les champs des messages RPC sont spécifiés dans le format XDR (*eXternal Data Representation*)
- XDR : représentation des données définie par SUN Microsystems
  - définit le type et le format des données échangées sur le réseau (paramètres de la procédure distante)
  - permet d'échanger des données entre machines ayant des représentations internes différentes

La représentation XDR

- 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

- L'hétérogénéité concerne :
  - la taille des objets typés : un entier peut être codé sur 2 octets ou 4 octets…
  - l'ordre des octets : *big endian* ou *little indian*
  - la représentation proprement dite d'un objet typé : combien de bits pour la mantisse et l'exposant représentant un nombre flottant, représentation d'un entier négatif…
- Inconvénient du protocole XDR :
  - l'encodage est effectué même si les machines source et destination utilisent déjà la même représentation
  - --> perte de performance

La représentation XDR

- XDR va par exemple spécifier qu'un entier occupe 32 bits qui seront transférés dans l'ordre "*big endian*" sur le réseau
  - si l'émetteur ou le récepteur n'est pas "*big endian*", XDR fera la conversion de l'entier
- Le problème se posait déjà pour les transmissions par socket des adresses IP et numéros de port
  - fonctions de conversion :
    - `htons()` et `htonl()` : représentation locale --> représentation réseau
    - `ntohs()` et `ntohl()` : représentation réseau --> représentation locale
  - ce problème ne se pose pas pour transférer un fichier : transfert brut d'une séquence d'octets sans interpréter son contenu

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

La représentation XDR

- Une donnée avant d'être envoyée est codée au format XDR mais aucune information dans l'encodage ne donne le type de la donnée
  - si une application utilise un entier de 32 bits, le résultat de l'encodage occupera exactement 32 bits et rien n'indiquera qu'il s'agit d'un type entier
- Cette forme d'encodage implique que clients et serveurs doivent s'entendre sur le format exact des données qu'ils échangent
- La bibliothèque de fonctions de conversion XDR permet simplement aux concepteurs d'applications RPC de ne pas se soucier de la représentation locale des données

La représentation XDR

```
XDR      *xdrs;            /* pointeur vers  un buffer XDR  */
char     buf[BUFSIZE];   /* buffer pour recevoir les données encodées */
xdr_mem_create  (xdrs, buf, BUFSIZE, XDR_ENCODE);
/* maintenant un buffer XDR est créé pour encoder les données
 * chaque appel à une fonction d'encodage va placer le résultat
 * à la fin de ce buffer ; le pointeur xdrs sera mis à jour */

int i;
...
i=260;
xdr_int(xdrs, &i);   /* encode l'entier i et le place à la fin de buffer */
/* Le programme receveur devra décoder les données :
xdr_mem_create  ( ... , XDR_DECODE) */
```

La représentation XDR

Fonction	arguments	type de données converti
xdr_bool	xdrs, ptrbool	booléen
xdr_bytes	xdrs, ptrstr, strsize, maxsize	chaîne de caractères
xdr_char	xdrs, ptrchar	caractère
xdr_double	xdrs, ptrdouble	virgule flot. double précision
xdr_enum	xdrs, ptrint	type énuméré
xdr_float	xdrs, ptrfloat	virgule flot. simple précision
xdr_int	xdrs, ptrint	entier 32 bits
xdr_long	xdrs, ptrlong	entier 64 bits
xdr_opaque	xdrs, ptrchar, count	données non converties
xdr_pointer	xdrs, ptrobj	pointeur
xdr_short	xdrs, ptrshort	entier 16 bits
xdr_string	xdrs, ptrstr, maxsize	chaîne de caractères
xdr_u_char	xdrs, ptruchar	entier 8 bits non signé
xdr_u_int	xdrs, ptrint	entier 32 bits non signé

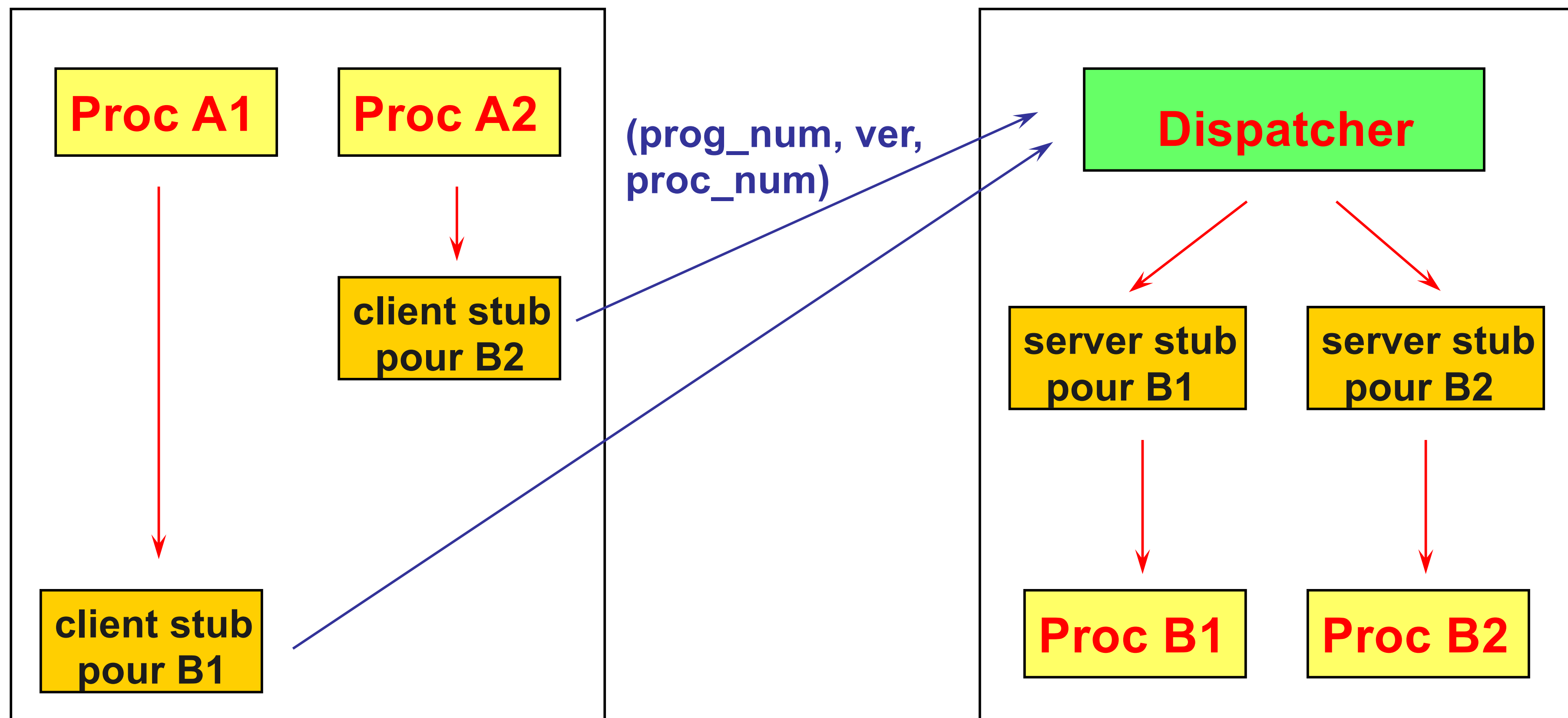
Conception d'un programme RPC

La méthodologie consiste à développer l’application distribuée comme une application conventionnelle puis à définir les procédures qui seront exécutées à distance

- côté client, entre l’appel de procédure et la procédure distante, il faut :
  - encoder les arguments
  - créer un message RPC CALL
  - émettre ce message vers le programme distant
  - attendre les résultats et décoder ces résultats selon la représentation interne de la machine locale
- côté serveur, il faut :
  - accepter une demande d'exécution RPC
  - décoder les arguments selon la représentation de la machine locale
  - dispatcher le message vers la procédure adéquate
  - construire la réponse puis encoder celle-ci
  - émettre le message correspondant vers le client

Conception d'un programme RPC

- Les procédures «stubs» remplacent les procédures conventionnelles (appel de procédure côté client, procédure appelée côté serveur)



Génération de programme RPC

- `rpcgen` est un outil de génération de programme RPC produisant automatiquement le code pour :
  - les procédures « stub » client et serveur
  - un squelette de serveur
  - les procédures XDR d'encodage/décodage des paramètres et des résultats
  - l'envoi et la réception des messages RPC
- Le concepteur du programme RPC doit fournir un fichier spécifiant
  - la description des structures de données des paramètres et des résultats
  - les fonctions réalisant le service désiré

Génération de programme RPC

- `rpcgen` produit quatre fichiers source dont les noms sont dérivés du nom du fichier de spécification en entrée
- Si le fichier en entrée est `serv.x`, les fichiers générés sont :
  - `serv.h` : déclarations des constantes et types utilisés dans le code généré pour le client et le serveur
  - `serv_xdr.c` : procédures XDR utilisées par le client et le serveur pour encoder/décoder les arguments
  - `serv_clnt.c` : procédure « stub » côté client
  - `serv_svc.c` : procédure « stub » côté serveur

- Un exemple de spécification

```
-- définitions des types --
struct couple_int {int min; int max;};
struct couple_int_float {int s; float m;};
-- … --
program VECTEUR_PROG {
  version VECTEUR_VERSION_1 {
     couple_int MIN_MAX(vecteur) = 1;
     couple_int_float SOM_MOY(vecteur) = 2;
     int PRODUIT_SCALAIRE(couple_vecteur) = 3;
     vecteur SOMME_VECTEUR(couple_vecteur) = 4;
  } = 1; -- num_version --
} = 0x22222222; -- num_prog --
```

Génération de programme RPC

- Ensuite, côté client, il suffit d'appeler `callrpc()`

```
callrpc(host,prog,progver,procnum,inproc,in,outproc,out);
```

  - `inproc` est une procédure qui encode les arguments dans le message RPC
  - `in` spécifie l'adresse des arguments de la procédure distante
  - `outproc` est une procédure qui décode les résultats dans le message RPC
  - `out` spécifie l'adresse en mémoire où les résultats seront décodés
- Inconvénients
  - uniquement au dessus d'UDP
  - pas de paramétrage possible des temporisations (temps maximal d'attente d'un résultat = 25 s, ré-émission de la requête toutes les 5 s)

Les RPC sous UNIX

- Le fichier `/etc/rpc`
  - l'équivalent de `/etc/services` pour les sockets (annuaire des services)
  - contient les informations relatives aux programmes RPC : nom du service, numéro de programme, listes d'alias

```
root@192.168.69.2# cat /etc/rpc
portmapper      100000  portmap sunrpc
rusersd         100002  rusers
nfs             100003  nfsprog
ypserv          100004  ypprog nis
mountd          100005  mount showmount
...
```

Les RPC sous UNIX

- La commande `rpcinfo`
  - permet d'interroger le port mapper pour connaître les services RPC disponibles la machine où il s'exécute (procédure n° 4 du port mapper)

`rpcinfo -p [host]`

(par défault, host = localhost)

  - 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)

(par défault, ver_num = 1)

Les RPC sous UNIX

```
root@192.168.69.1# rpcinfo -p 192.168.90.2
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100004    2   udp    954  ypserv
    100004    1   udp    954  ypserv
    100004    2   tcp    957  ypserv
    100004    1   tcp    957  ypserv
    100007    2   udp    959  ypbind
    100007    1   udp    959  ypbind
    100007    2   tcp    962  ypbind
    100007    1   tcp    962  ypbind
root@192.168.69.1# rpcinfo -u 192.168.90.2 ypserv
  program 100004 version 1 ready and waiting
  program 100004 version 2 ready and waiting
```

Les RPC sous UNIX

- Sur le client (192.168.69.1)

```
root@192.168.69.1# rpcinfo -u 192.168.69.2 nfs
rpcinfo: RPC: le programme n'est pas enregistré
Le programme 100003 n'est pas disponible.
root@192.168.69.1# rpcinfo -p 192.168.69.2
Aucun programme enregistré sur l'hôte cible
```

- Sur le serveur (192.168.69.2)

```
root@192.168.69.2# rpcinfo -p 192.168.69.2
No remote programs registered.
root@192.168.69.2# rpcinfo -p | grep nfs
   100003    2   udp   2049  nfs
   100003    3   udp   2049  nfs
```

- Il faut autoriser les connexions RPC extérieures

Les RPC sous UNIX

- Les fichiers `/etc/hosts.allow` et `/etc/hosts.deny` permettent d'autoriser ou non l'accès aux services réseaux (en particulier les connexions RPC distantes sur la machine locale)
- Syntaxe générale (hors RPC) :

`service: utilisateurs` (autorisés dans `hosts.allow`, rejetés dans `hosts.deny`)

- Exemple (voir aussi `man hosts_access`) :

```
root@192.168.69.2# cat /etc/hosts.deny
  ALL: ALL #on ne peut plus rien faire à distance
root@192.168.69.2# cat /etc/hosts.allow
  portmap nfsd sshd: 192.168.69. 192.168.90.3
```

Les RPC sous UNIX

- Comme pour les démons utilisant les sockets, il est possible de lancer dynamiquement le processus d'un serveur RPC uniquement quand un client sollicite le service (via le démon `inetd`)
- Il suffit d'ajouter une entrée par service RPC dans le fichier `/etc/inetd.conf`

```
# services RPC
rpc 100002 1-2 dgram udp wait root /sbin/ypserv ypserv -d
```

- Quand le processus `inetd` se lance, il réalise l'enregistrement des services RPC qu'il prend en compte auprès de `portmap`

Exercices

Lyon 1

Lecture bloquante/non bloquante

- Une application (client ou serveur) veut lire exactement 100 caractères sur une socket (mode connecté)
- Décrire l'algorithme correspondant et donnez les avantages/inconvénients
  - dans le cas d'une lecture complétement bloquante (read retourne quand tout est lu)
  - dans le cas standard des sockets (« au moins 1 »)
  - dans le cas d'une lecture non bloquante (-1 si EWOULDBLOCK)

Exemple de programmation C/S

- 1. Quel est le service proposé par cette application client/serveur ? Combien d'arguments sont nécessaires au lancement du client ? Quels sont-ils ?
- 2. Quel port utilise le serveur ? Aurait-on pu choisir une autre valeur ? Quel port utilise le client ? Comment est-il attribué et par quelle primitive ? S'agit-il d'une connexion en mode connecté ou non et est-ce justifié ?
- 3. A quoi correspondent les constants BUF_SIZE et QUEUE_SIZE ?
- 4. Quand est-ce que le serveur s'arrête ? Que fait le serveur une fois les initialisations terminées (décrire le cas où il y a des connexions pendantes et le cas inverse) ?
- 5. Que se passe t-il si le client est lancé avant que le serveur n'ait démarré ?
- 6. Quand est-ce que le client s'arrête si la connexion a réussi ? Que fait le client une fois la connexion établie ?
- 7. Que pensez-vous de la structure actuelle du serveur ? Peut-il satisfaire un grand nombre de connexions ? Expliquez. Proposez une solution plus adaptée.

Un mini-inetd

- Voici la page man du programme `mini-inetd` ainsi que son code.
- 1. Complétez la partie `DESCRIPTION` de la page man. Représentez à l'aide d'un schéma/diagramme la structure algorithmique du programme.
- 2. Dans le code ci-après, le code de la fonction `tcp_listen()` a volontairement été omis. Quelles sont les paramètres et la valeur de retour de cette fonction ? Quelles sont les opérations qui doivent y être réalisées et où les paramètres interviennent-ils ?
- 3. Commentez le nom du programme. Quelles sont les différences et similitudes entre `mini-inetd` et `inetd` ?
- 4. Comment modifieriez vous la structure donnée à la question 1 pour que `mini-inetd` puisse traiter plusieurs couples (port, program) passés en arguments ?