Département Telecom – 2ème année (T2)

Module RE216: Programmation Réseau

Programmation Réseau

Joachim Bruneau-Queyreix
joachim.bruneau_-_queyreix@bordeaux-inp.fr

Plan

- **Concepts généraux:**
  - **Couche application**
  - **Modèle client-serveur**
  - **Programmation Socket**
  - ⇒ 2 séances de 1h20

- EI-TP-Projet

  => 7 séances de 2h40 en salle machine en 1/3 groupe

Couche application

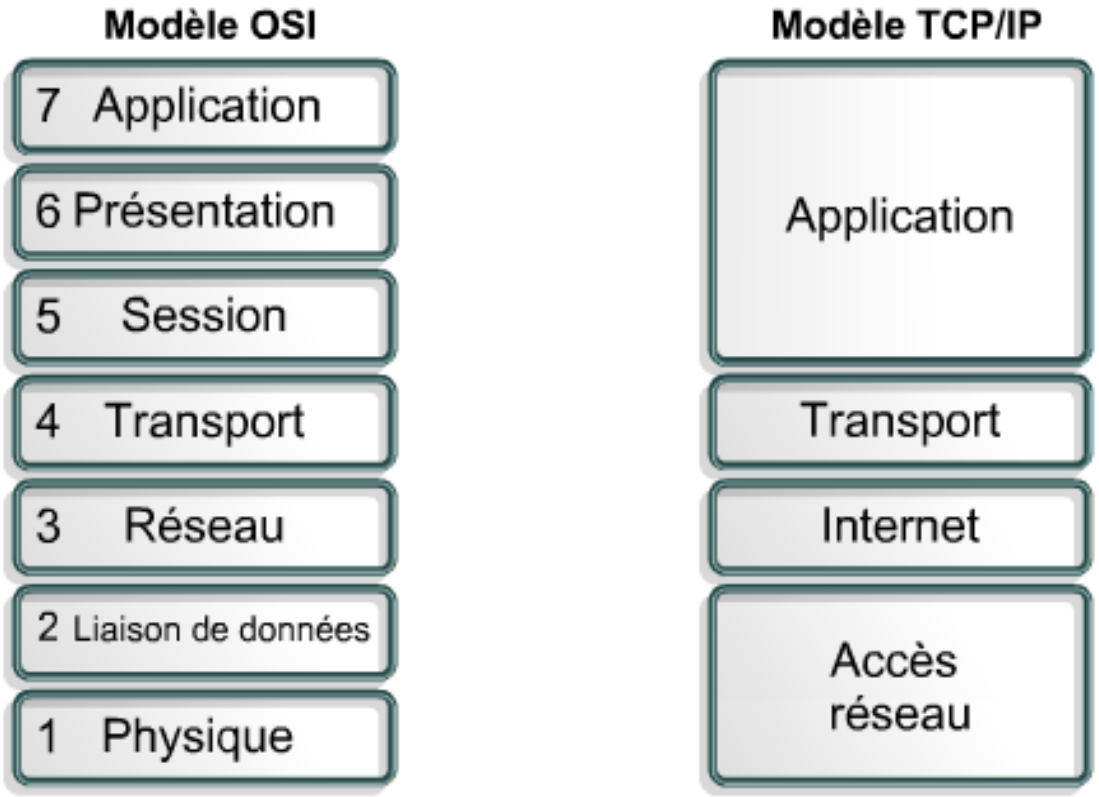
Joachim Bruneau-Queyreix

Applications : Raison d'être d'un réseau

- Partager des ressources
  - Information (web, youtube, …)
  - Ressources physiques: imprimantes, scanner, fichier, etc
- Communication
  - Téléphone, vidéoconférence, emails, messagerie instantanée, etc
- Commerce électronique
- Divertissements (film, jeux interactifs…)
- …

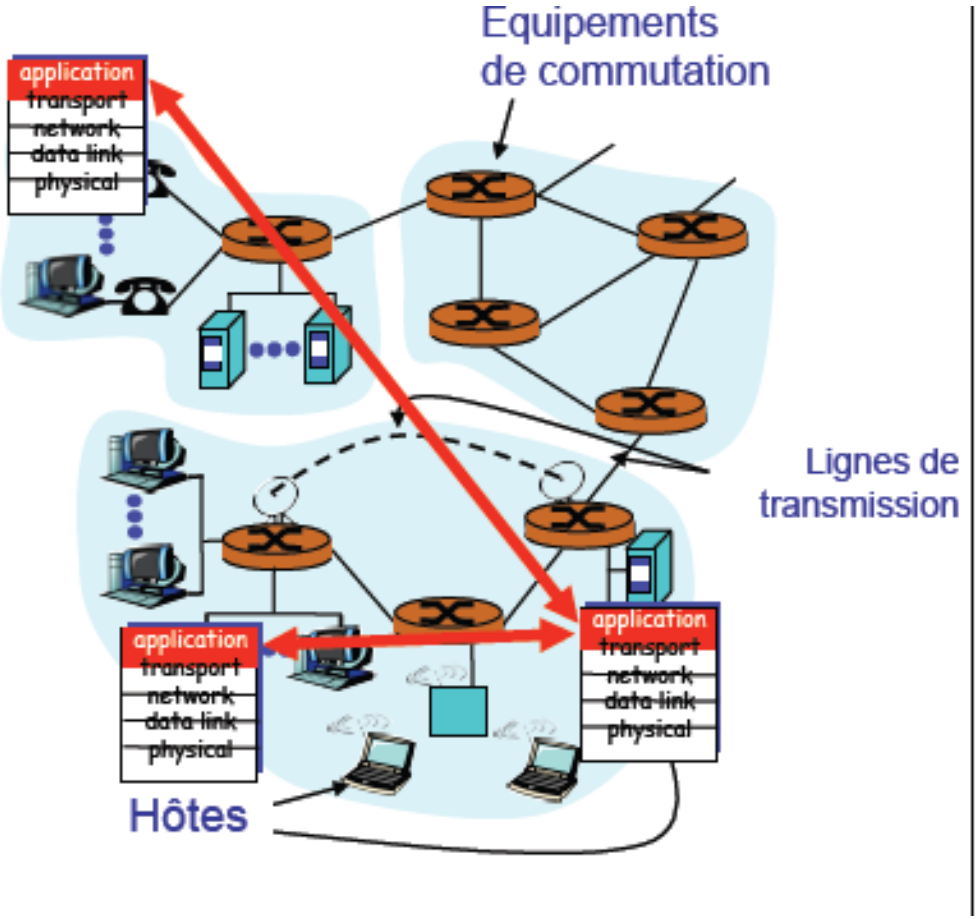
Couche application

- Modèle en couche (OSI vs TCP/IP)



Couche application

- Nombreuses applications créées depuis 30 ans
  - Text-based: accès distant, email, transfert de fichiers, newsgroups, forums de discussion
  - Multimédia: téléphonie sur Internet, vidéoconférence, audio et vidéo à la demande, Live IPTV



Application répartie ou distribuée

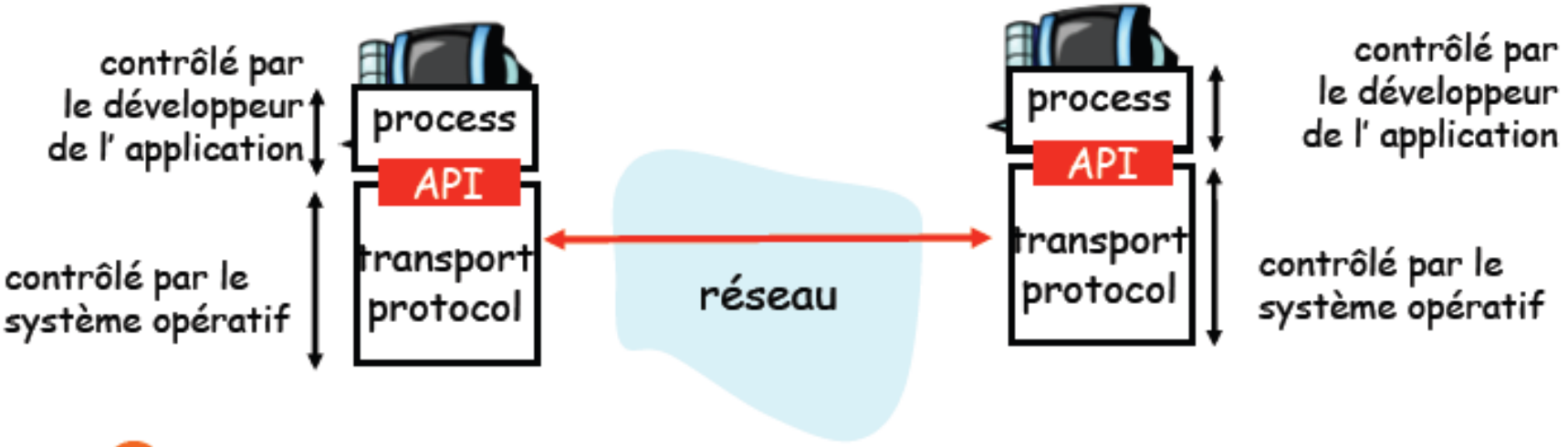
- Application répartie: application comportant plusieurs éléments s'exécutant sur des hôtes différents et coopérant par l'intermédiaire d'un réseau de communication
- Hôtes différents exécutent des programmes constituant une application. Un programme en exécution est dit *processus*
- Un *protocole de couche application* définit comment la communication entre les processus émetteur et récepteur doit être mise en place

Protocoles de couche application

- Certains protocoles applicatifs sont public:
  - Spécifiés dans des RFCs (documents standard IETF, http://www.ietf.org/rfc.html, ex: HTTP [RFC 2068])
  - Interopérabilité entre des logiciels développés indépendamment
- Beaucoup sont propriétaires

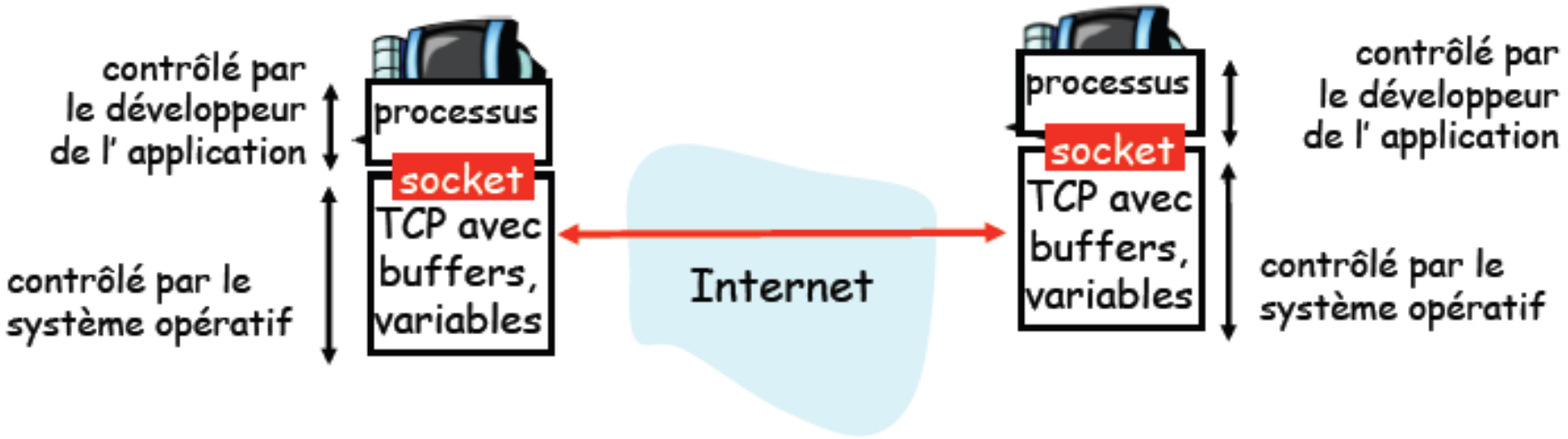
- Exemples:
  - Web: standards pour les formats de documents (HTML), browsers Web, serveurs Web et le protocole applicatif HTTP pour définir l'échange des messages entre les browser et le serveur Web
  - Courrier électronique: serveurs de mails (qui hébergent les boîtes aux lettres), mail readers, standard pour le format des mails et protocole applicatif pour l'échange des messages

Application Programming Interface (API)



- Définit l'interface entre la couche application et la couche transport
- Socket : API Internet
  - Deux processus communiquent en émettant et recevant des données via les sockets

API Socket



- Porte d'entrée d'un processus : un processus envoie les messages au réseau et en reçoit à travers une socket
- Le développeur contrôle toute la partie application des sockets
- Il n'a que peu de contrôle sur la partie transport : choix du protocole et éventuellement ajustement de quelques paramètres

Adressage des processus

- Pour communiquer avec un processus distant, il faut l'identifier
  - Nom ou adresse de l'hôte distant :
    - adresse IP de l'hôte distant : 32 bits qui identifient de manière unique l'interface qui connecte l'hôte à l'Internet
      - Autre standard d'adressage pour les réseaux ATM
  - Identifiant du processus récepteur chez l'hôte distant
    - «Numéro de port» : permet de différencier les différents processus locaux auxquels le message doit être transmis
    - Les protocoles applicatifs usuels ont des numéros de port réservés (RFC 1700) :
      - 80 pour le processus serveur Web
      - 25 pour le protocole de serveur de mail (utilisant SMTP)
      - Un nouveau numéro de port est affecté à chaque nouvelle application

Applications et Service de Transport

- Socket = interface entre le processus applicatif et le protocole de transport
  - Côté émetteur : l'application envoie des messages par la porte
  - De l'autre côté de la porte, le protocole de transport doit déplacer les messages à travers le réseau, jusqu'à la porte du processus récepteur
- De nombreux réseaux (dont Internet) fournissent plusieurs protocoles de transport

Service de Transport

- Quel est le service de transport nécessaire à une application? Lequel choisir lorsqu'on développe une applications
  - Etude des services fournis par chaque protocole
  - Sélection du protocole qui correspond le mieux aux besoins de l'application
- 3 types de besoins au niveau des applications, en termes de:
  - Perte de données
  - Bande passante
  - Délai

Service de Transport: Perte de données

- Certaines applications nécessitent une fiabilité à 100%:
  - Courrier électronique (SMTP/POP3/IMAP)
  - Transfert de fichier (FTP)
  - Accès distant (Telnet/SSH)
  - Transfert de documents Web (HTTP)
  - Applications financières
- D'autres peuvent tolérer des pertes:
  - Applications multimédia: audio/vidéo

Service de Transport: Bande passante

- Certaines applications (ex : multimédia) requièrent une bande passante minimale:
  - Téléphonie sur Internet : si la voix est codée à 32 Kbps, les données doivent être transmises à ce débit
  - Applications multimédia
- D'autres utilisent la bande passante disponible (applications élastiques)
  - Courrier électronique, transfert de fichiers, accès distant, Web
  - Plus il y a de bande passante, mieux c'est!

Service de Transport: Délai

- Certaines applications nécessitent un délai de bout-en-bout faible (moins de quelques centaines de ms)
  - Applications temps réel interactives :
    - Téléphonie sur Internet
    - Environnements virtuels
    - Téléconférence
    - Jeux en réseau
- Pour les applications non temps réel, un délai court est préférable, mais pas de contrainte forte

Service de Transport: Besoins

Application	Pertes	Bande passante	Sensibilité délai
Transfert de fichiers	sans pertes	élastique	Non
e-mai	sans pertes	élastique	Non
Web	tolérant	élastique	Non
Audio/vidéo temps réel	tolérant	audio: 5Kb - 1Mb vidéo:10Kb - 5Mb	Oui, 100s ms
Audio/vidéo enregistré	tolérant	idem	Oui, quelques s
Jeux interactifs	tolérant	Quelques Kbps	Oui, 100 s ms
Applis financières	sans pertes	élastique	Oui et non

Service de Transport

Service TCP

- Orienté connexion: connexion nécessaire entre le client et le serveur
- Transport fiable entre le processus émetteur et récepteur
- Contrôle de flot: l'émetteur ne submerge pas le récepteur
- Contrôle de Congestion : réduit le débit de l'émetteur quand le réseau est congestionné
- Ne propose pas:
  - de garanties de délai,
  - de bande passante minimale

Service UDP

- Transfert de données non fiable
- Ne propose pas
- de connexion,
- de fiabilité,
- de contrôle de flot,
- de contrôle de congestion,
- de garantie temporelle,
- de bande passante

Protocoles applicatifs et protocoles de transport dans Internet

Application	Protocole applicatif	Protocole de transport
e-mail	SMTP [RFC 821]	TCP
Accès distant	telnet [RFC 854]	TCP
Web	HTTP [RFC 2068]	TCP
Transfert de fichiers	FTP [RFC 959]	TCP
Streaming video	Propriétaire (ex. RealNetworks)	TCP ou UDP
Voix sur IP	propriétaire (ex : Vocaltec)	En général UDP

Agent utilisateur

- Un agent utilisateur est une interface entre l'utilisateur et l'application réseau

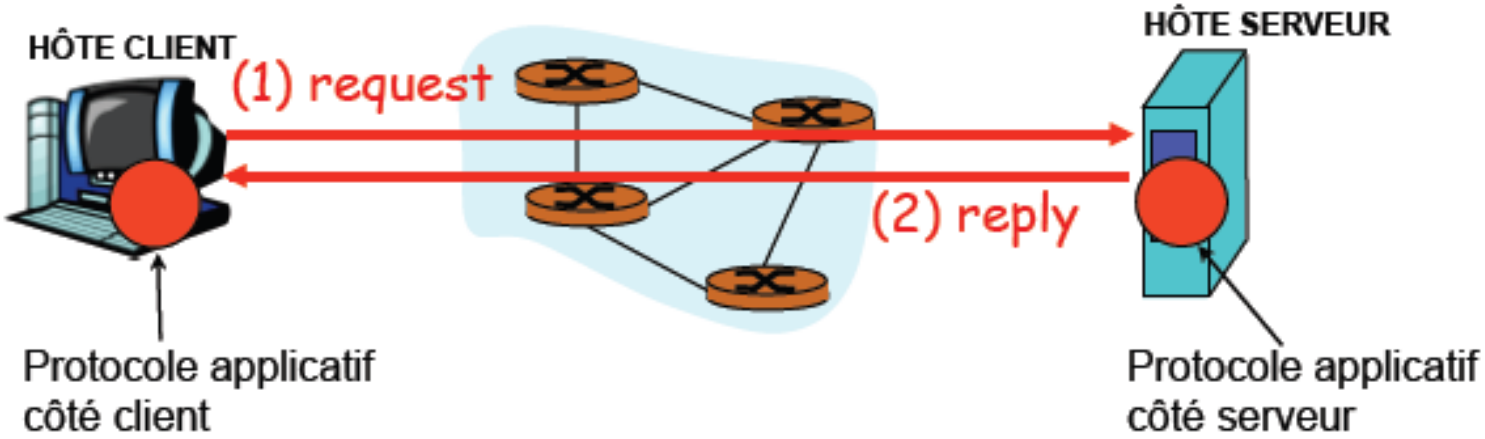
Modèle Client-Serveur

Joachim Bruneau-Queyreix

Modèle Client-Serveur

- Serveur: programme offrant un service accessible par l'intermédiaire d'un réseau de communication
  - Par extension, hôte accueillant un programme serveur
- Client: programme déléguant un travail à un serveur
  - Par extension, hôte accueillant un programme client
- Modèle d'interaction client-serveur:
  - Envoie d'une requête par un client vers un serveur
  - Traitement de la requête par le serveur
    - Validation de la requête (sécurité, validité, etc.)
    - Traitement effectif
- Renvoie d'un résultat vers le client
- Exemple: demande d'une page web

Communication Client-Serveur



Client :

- Initie le contact avec le serveur (il parle en premier)
- Typiquement, il demande un service au serveur

Serveur :

- Propose les services demandés par le client
- généralement exécuté sur un ordinateur puissant
- peut gérer les requêtes de plusieurs clients

Identification d'un serveur

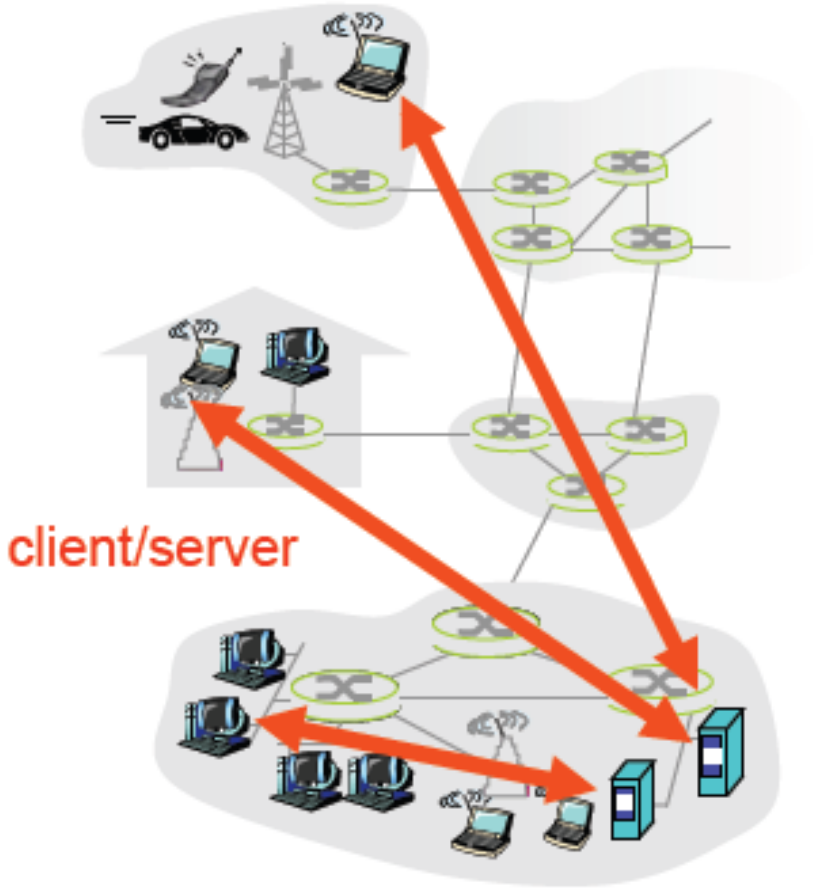
- Identification d'un programme serveur
  1. Identification de l'hôte susceptible de fournir le service recherché:
     - Identification connue: utilisation de son identification
       - Utilisation directe : adresse connue
       - Utilisation indirecte : résolution de l'adresse à l'aide d'un service de nom
     - Identification inconnue: découverte, recherche, diffusion, etc.

  2. Identification du programme serveur sur cet hôte

Identification d'un client

- Identification d'un programme client
  - Nécessaire pour l'acheminement d'une réponse
  - En général (implicitement) connue grâce à la requête
  - En général, pas de nécessité particulière d'utilisation d'identificateurs « bien connus »
- Cas de la famille de protocoles internet:
  - Client identifié, en général, par :
    - Un protocole de transport (UDP ou TCP, en général)
    - Un port de transport « dynamique »

Client-server architecture



serveur:

- toujours en exécution
- adresse IP permanent
- Cluster de serveur pour passer à l'échelle
- « infrastructure-intensive »

client:

- communique que avec le serveur
- Pas toujours connecté
- Adresse IP dynamiques possibles

Architecture d'un serveur (1)

- Serveur séquentiel mono-tâche
  - Principe:
    - Ouverture du port « bien connu » d'écoute
    - Attente d'une requête, puis traitement de la requête
    - Retour à l'état d'attente
  - Caractéristiques
    - Mise en oeuvre très simple
    - Traitement strictement séquentiel des requêtes
    - Risque important de rejet des requêtes ne pouvant être traitées pendant l'occupation du serveur

  => « cas d'école » peu utilisé en pratique

Architecture d'un serveur (2)

- Serveur parallèle multi-tâches
  - Principe:
    - Ouverture du port « bien connu » d'écoute
    - Attente d'une requête puis:
      - – création, par le système d'exploitation, d'une prise de communication particulière pour chaque requête acceptée
      - – traitement de chaque requête par un processus dédié, indépendant du processus principal d'attente
    - Retour à l'état d'attente
  - Caractéristiques:
    - Mise en œuvre plus complexe
    - Traitement simultané de plusieurs requêtes possible
    - Risque limité de rejet des requêtes pour cause d'occupation du serveur

Architecture d'un serveur (3)

- Remarques
  - Performances: multiplication des processus pouvant être pénalisante en termes de CPU et de mémoire

  => Utilisation de processus « légers »

  - Sécurité / fiabilité :

  Serveur: programme accessible à des stimuli externes
    - Risques d'intrusion et d'utilisation de failles dans le programme
    - Risque de comportement inattendu face à des requêtes non prévues ou mal formulées

  => Spécification et validation minutieuses

Architecture d'un client

- En général beaucoup plus simple que celle d'un serveur:
  - Architecture d'un programme client :
    - Ouverture d'un port de communication (port « dynamique », par exemple)
    - Envoi d'une requête
    - Attente de la réponse

Exemple application client-serveur: le WEB

- World Wide Web – WWW (1990s)
- Page Web : ensemble d'"objets"
  - Chaque objet est un fichier (page HTML, JPEG image, …)
  - Adressée par une URL (Uniform Resource Locator)
- L'URL a deux composantes (ex www.someSchool.edu/someDept/pic.gif)
  - Nom du serveur qui hôte l'objet
  - Chemin d'accès dans le serveur
- La plupart des pages Web contiennent :
  - Page HTML de base
  - Objets référencés

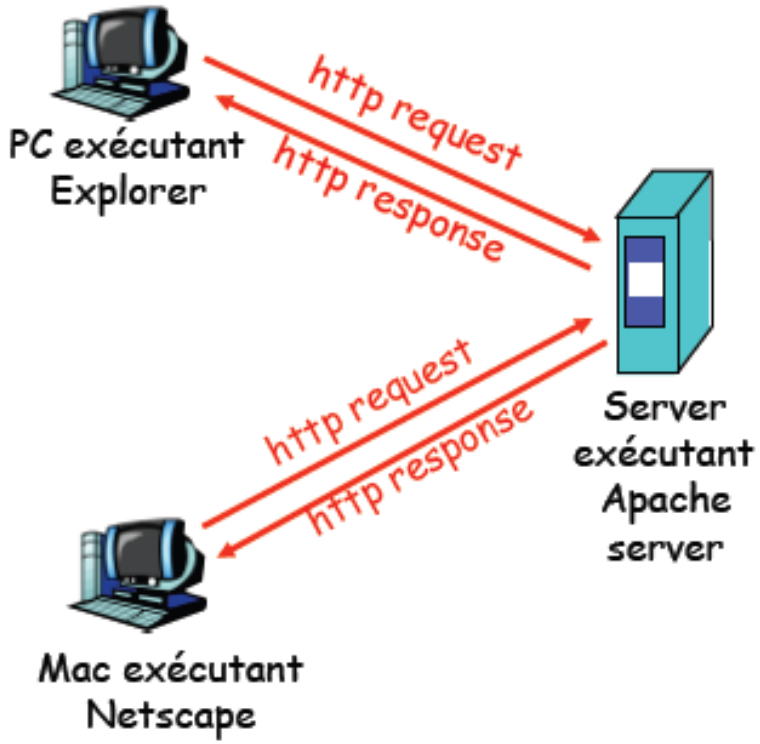
Le Web : côté client et serveur

- L'Agent Utilisateur pour le Web est le browser:
  - Ex: MS Internet Explorer, Firefox, Safari, Chrome, …
  - Visualisation de pages Web
  - Implémentation du côté client du protocole HTTP

- Le serveur Web (ex. MS Internet Information Server)
  - Hôte les objets Web
  - Implémentation du côté serveur du protocole HTTP

Le Web: le protocole HTTP

- Couche applicative Web
- Modèle client/serveur
  - Client : le browser, qui demande, reçoit, affiche les objets Web
  - Serveur : le serveur Web, qui envoie les réponses aux requêtes
- http1.0 : RFC 1945
- http1.1 : RFC 2068



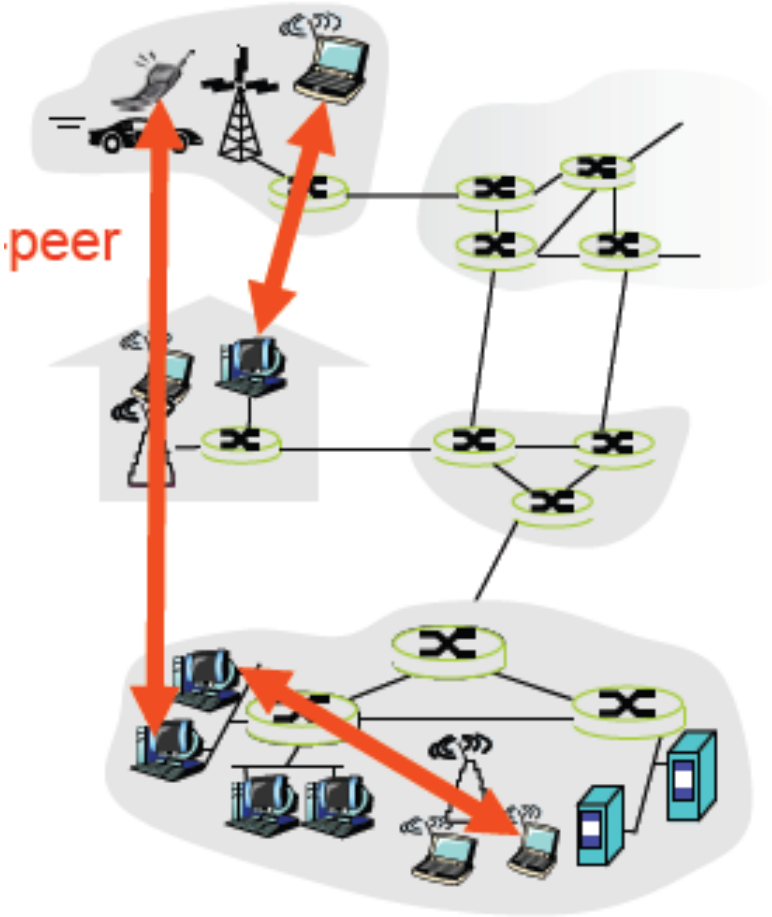
Le Web: le protocole HTTP

- HTTP utilise le service de transport TCP
  - Le client initie une connexion TCP (crée une socket) avec le serveur, port 80
  - Le serveur accepte la connexion TCP du client
  - Les messages HTTP sont échangés entre le browser (client HTTP) et le serveur Web
    - Une fois que le client a envoyé le message dans la socket, c'est le protocole TCP que doit en assurer la réception
  - La connexion TCP est close
  - HTTP est « sans état » : Le serveur ne maintient aucune information au sujet des requêtes précédentes des clients

- Détails de l'exemple HTTP: voir annexe

Modèle alternatif: peer-to-peer (pair à pair)

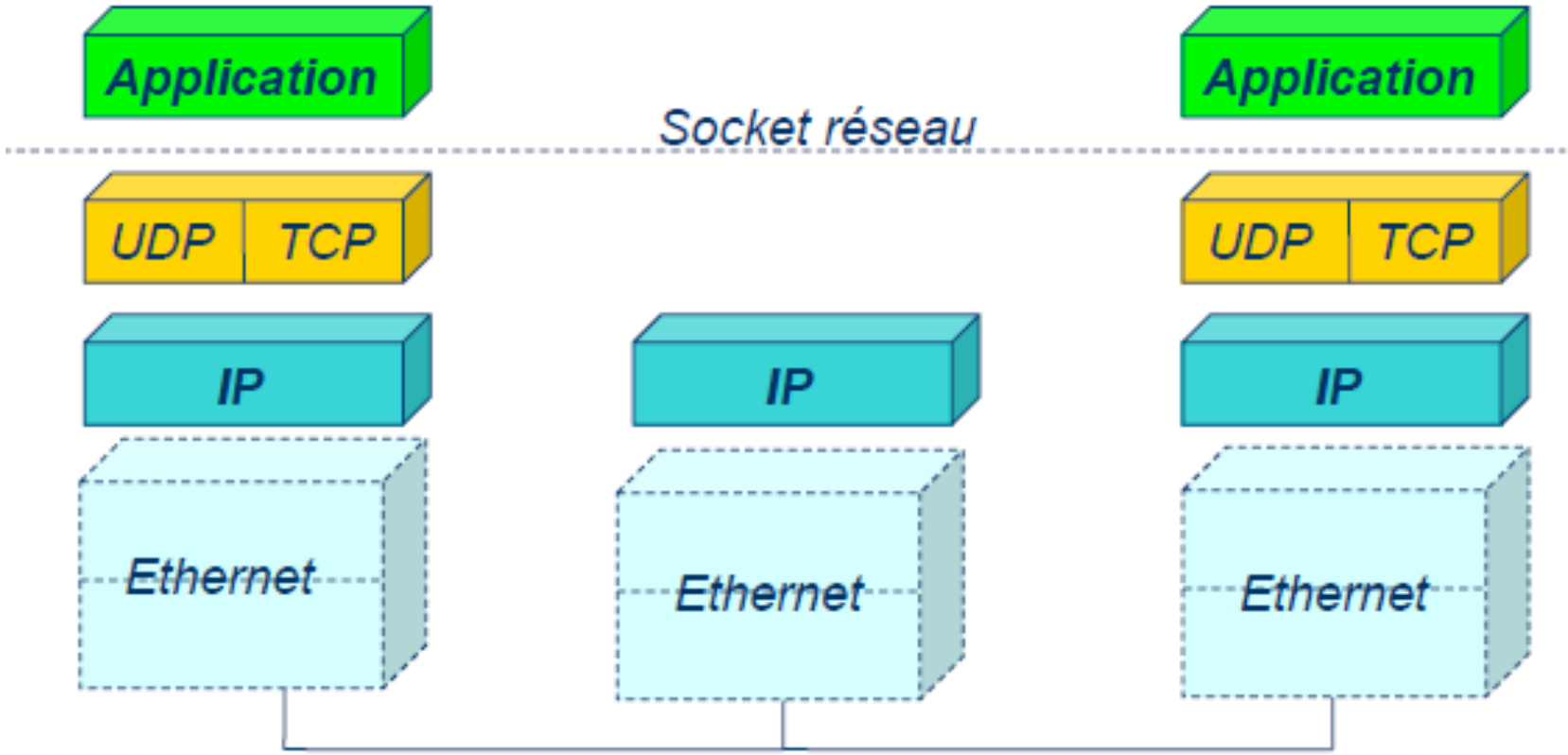
- Pure P2P architecture:
  - *pas* des serveurs toujours en exécution
  - Terminaux quelconques (peers) peuvent communiquer directement
  - Peers sont connectés à intermittence et peuvent changer adresse IP
  - Passe bien à l'échelle mais plusieurs problèmes :
    - Les réseaux d'accès pas pensés pour fonctionner en mode p2p (usage asymétrique de la bande passante)
    - Sécurité
    - Les usagers cherchent à utiliser les services sans partager leurs ressources (ex. bande passante)



API Programmation Socket

Joachim Bruneau-Queyreix

Vue d'ensemble



Les sockets

- Les Sockets
  - Interface de communication introduite par les systèmes Unix pour la communication réseau.
  - Ensemble de primitives assurant ce service,
  - – Générique
    - S'adapte aux différents besoins de communication,
  - **Indépendant** de protocoles et de réseaux particuliers
  - N'utilise pas forcément un réseau
    - Exemple : communication locale (interne à une station) : domaine Unix

Les sockets

- ⇒ Point d'accès aux services de la couche transport, c'est-à-dire TCP ou UDP.
  - ⇒ Point de communication par lequel un processus peut émettre ou recevoir des données

Les sockets

- Peuvent être vues comme un **descripteur identique à un descripteur de fichier** dans lequel on peut lire ou écrire des données.
- La configuration de l'ouverture de connexion permet de spécifier comment seront transmises les données envoyées ou lues dans ce descripteur de fichier.
- API socket s'approche de l'API fichier d'Unix
  - Descripteur de socket dans la table des descripteurs du processus
  - Primitives **read(), write(), close(), ioctl(), fnctl(), ...**
  - **Les descripteurs peuvent être partagés par les processus descendants de son créateur**

Les sockets

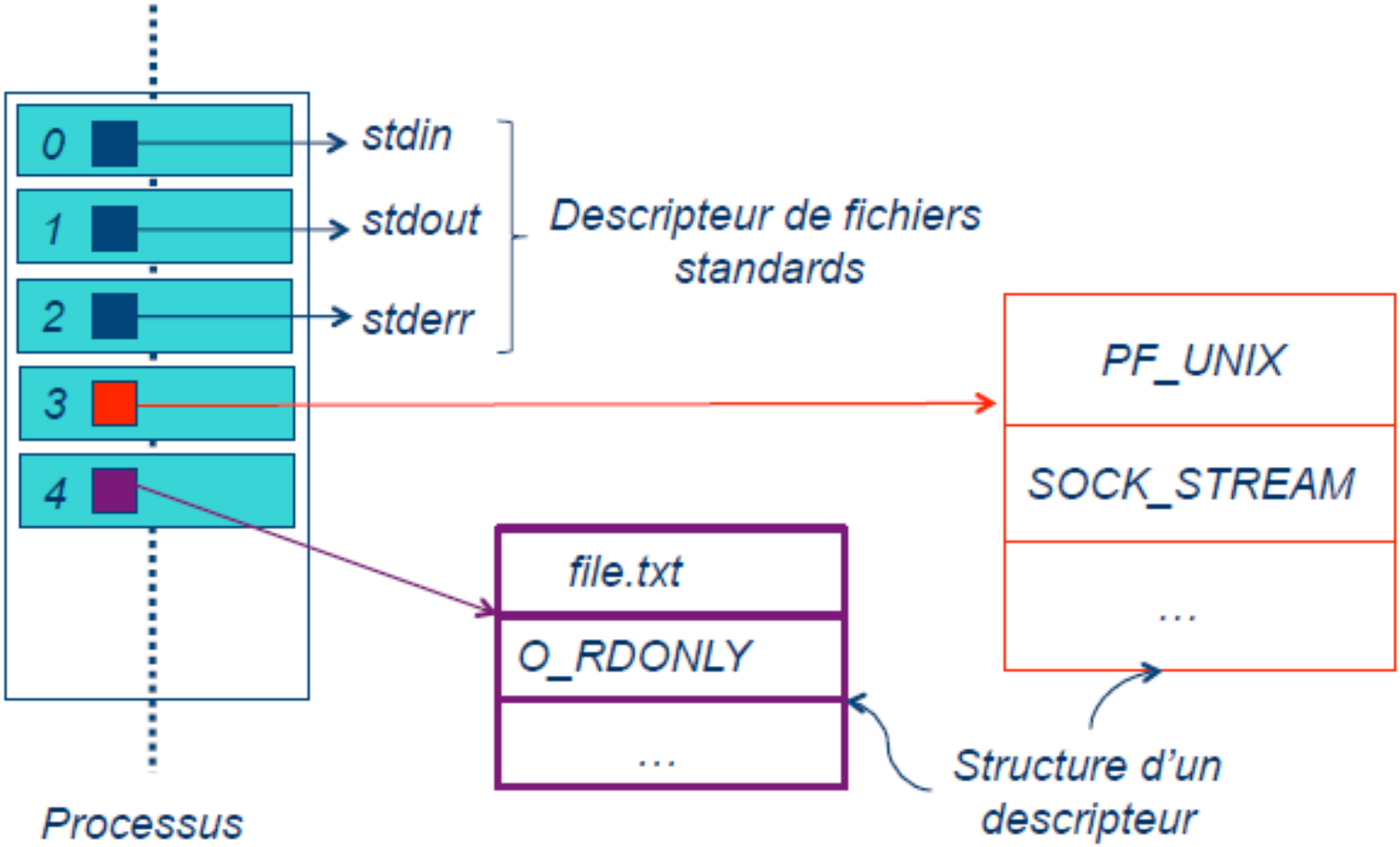
- Manipulation similaire des sockets et des fichiers

Création de la socket

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char ** argv) {
 int sock;
 int file;
 file = open("file.txt", O_RDONLY);
 sock = socket(PF_UNIX, SOCK_STREAM, 0);
 close(sock);
 close(file);
}
```

Descripteur de socket = descripteur de fichier



Socket: modèle client-serveur

- Pour communiquer il faut créer un serveur prêt à recevoir les requêtes d'un client.
- Rappel: Il existe deux modes de connexion :
  - **Connecté:**
    - **Ouverture de connexion**
    - **Envoi des données**
    - **Fermeture de connexion**
  - **Mode non connecté:**
    - **A chaque envoi de données, il faut faire communiquer les deux sockets**

Socket: modèle client-serveur

- **Côté serveur**
  - Création de la socket
    - Mise en attente de message dans le cas d'une communication UDP, ou
    - Mise en attente de connexion dans le cas d'une communication TCP.

    Dans le cas d'une communication TCP, il est généralement profitable de permettre au serveur de gérer plusieurs connexions simultanées ; dans ce cas un nouveau processus sera créé pour chaque connexion.
- **Côté client**
  - Création de la socket
  - Communication se fait tout d'abord en renseignant l'adresse du serveur à contacter.
  - Ensuite envoi de données sans connexions au préalable (UDP), ou demande de connexion (TCP).

Création de socket

- Primitive socket
  - #include <sys/types.h>
  - #include <sys/socket.h>
  - int socket(int domaine, int type, int protocole);

- Appel permettant de créer une structure en mémoire contenant tous les renseignements associés à la socket (buffers, adresse, etc.)
- Renvoie un descripteur de socket permettant d'identifier la socket créée (-1 en cas d'erreur).

Création de socket

- Primitive socket

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domaine, int type, int protocole);
```

Descripteur de la socket — *Domaine de communication* — *Mode de communication* — *Protocole à utiliséer*

Création de socket

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domaine, int type, int protocole);
```

Descripteur de la socket — *Domaine de communication* — *Mode de communication* — *Protocole à utiliséer*

- **Champ Domaine:**
  - Domaine de communication pour le dialogue
  - Sélectionne la famille de protocole à utiliser
  - Défini dans le fichier <sys/socket.h>
  - Prend la valeur:
    - **AF_INET** pour IPv4.
    - **AF_INET6** pour IPv6.
    - **AF_UNIX, AF_LOCAL** pour les communications UNIX en
  - Local sur une même machine.

Création de socket

- Adresse du domaine **AF_INET**
  - Adresse IP + numéro du port

  ⇔ Plusieurs connexions *simultanées*

  ⇔ *Multiplexage : faire passer plusieurs connexions sur une même ligne*
  - Numéros de port < 1024 sont réservés à l'administrateur (root)
  - Sous Unix, faire un *cat /etc/services* pour obtenir la liste des services: Exemple de numéros réservés (côté serveur) :
    - FTP : port 21/tcp
    - TELNET : port 23/tcp
    - ECHO: port 7
    - SSH : port 22
    - HTTP : port 80
  - Du côté client, le numéro de port n'est pas fixe: le système prend en général un port libre > 1024.

Création de socket

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domaine, int type, int protocole);
```

Descripteur de la socket — *Domaine de communication* — *Mode de communication* — *Protocole à utiliséer*

- **Champ Type:**
  - **Fixe la sémantique des données (intégrité des données, type du flux de données, ordre de délivrance)**
  - Prend en général les valeurs suivantes:
    - **SOCK_STREAM**, si on utilise le protocole TCP
    - **SOCK_DGRAM**, si on utilise le protocole UDP
    - **SOCK_RAW,** si on souhaite un accès direct aux données

Création de socket

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domaine, int type, int protocole);
```

Descripteur de la socket — *Domaine de communication* — *Mode de communication* — *Protocole à utiliséer*

- Le **type SOCK_STREAM:**
  - Flux d'octets Full-Duplex
  - *Garantie l'intégrité des données*
  - *Garantie l'ordre d'arrivée des paquets*
  - *Uniquement en mode connecté*
  - Une fois la connexion établie, les données sont transmises par les primitives:
    - **Read()**, **Write()**, **Send(), Recv()**.
  - Pour la fermeture de connexion, on referme la socket avec **close().**

Création de socket

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domaine, int type, int protocole);
```

Descripteur de la socket — *Domaine de communication* — *Mode de communication* — *Protocole à utiliséer*

- **Champ Protocole:**
- **Normalement, un seul protocole par type de socket**
  - 0 si le système choisit le protocole
  - IPPROTO_UDP pour l'UDP (avec SOCK_DGRAM)
  - IPPROTO_TCP pour le TCP (avec SOCK_STREAM)

Création de socket

- **Primitive Socket** conclusions:

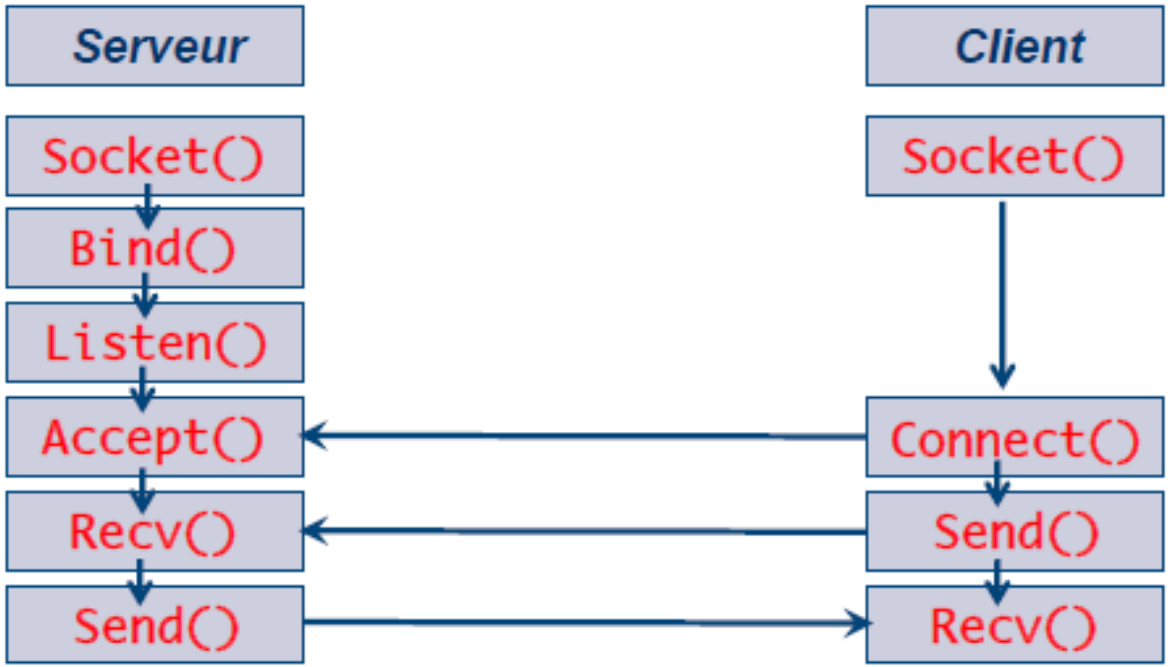
- On utilise les sockets dans les modes suivants:
  - Socket UDP:

```
socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);
```

  - Socket TCP:

```
socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
```

Socket en mode connecté (TCP)



Socket en mode connecté (TCP)

- Côté client:

⇔ demandeur de la connexion ⇔ ACTIF

- socket()
  - Permet de créer une socket
- connect()
  - Permet de se connecter à un couple (adresse IP, port)
- read(), write(), send(), recv()
  - **Pour lire et écrire dans la socket**
- close()
  - Pour fermer la socket

Socket en mode connecté (TCP)

- Côté serveur:

⇔ en attente de connexion ⇔ **PASSIF**

- socket()
  - Pour créer une socket
- bind()
  - Pour associer une adresse à la socket
- listen()
  - Pour se mettre à l'écoute des connections
- accept()
  - Pour accepter une connexion
- read(), write(), send(), recv()
  - Pour lire et écrire sur la socket
- close()
  - Pour fermer la socket

Socket en mode non connecté (UDP)

Utilisation d'une socket

- Attachement d'une socket

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sock, const struct sockaddr *adr, int adrlen);
```

0 ou -1

Id de la socket

Pointeur sur la Structure contenant les informations sur la machine locale

Taille de la structure précédente

- Une fois la socket créée, il faut lui attacher une adresse qui sera généralement l'adresse locale:
  - Associe l'adresse locale *adr* à la socket décrite par *sock*.

Utilisation d'une socket

- Attachement d'une socket

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sock, const struct sockaddr *adr, int adrlen);
```

0 ou -1 — *Id de la socket* — *Pointeur sur la Structure contenant les informations sur la machine locale* — *Taille de la structure précédente*

- Retourne *SOCKET_ERROR* en cas d'erreur
- *Sock* désigne la socket du serveur avec laquelle on va associer les informations
- *Adr* est un pointeur sur la structure *sockaddr*: permet de spécifier l'IP à laquelle on se connecte
- *Adrlen* est la taille de la structure pointée par *adr*

Utilisation d'une socket

- Sans adresse, une socket ne pourra pas être contactée (il s'agit simplement d'une structure qui ne peut pas être vue de l'extérieur).
- L'attachement permet de préciser l'adresse ainsi que le port de la socket:
  - On attache une adresse à une socket à l'aide de la fonction bind() qui renvoie 0 en cas de succès et -1 sinon.
  - Après sa création, la socket n'est connue que du processus qui l'a créée (et de ses descendants)

Utilisation d'une socket: attachement

- Attachement d'une socket

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sock, const struct sockaddr *adr, int adrlen);
```

0 ou -1

Id de la socket

Pointeur sur la Structure contenant les informations sur la machine locale

Taille de la structure précédente

- La structure générique pour les adresses :
  - /usr/include/sys/socket.h

```
struct sockaddr {
  u_short sa_family; /* address family */
  char sa_data[14]; /* up to 14 bytes of direct address */
};
```

Utilisation d'une socket: structure d'une adresse

```
struct sockaddr {
  u_short sa_family; /* address family */
  char sa_data[14]; /* up to 14 bytes of direct address */
};
```

- sa_family:
  - Peut être de plusieurs types
    - AF_INET pour IPv4
    - AF_INET6 pour IPv6
- sa_data:
  - Contient l'adresse IP et le port de la socket
  - => Pas pratique à manipuler
  - => Utilisation de la structure sockaddr_in pour IPv4

Utilisation d'une socket: structure d'une adresse

```
struct sockaddr_in {
        short sin_family; /* AF_INET */
        u_short sin_port; /* le numéro de port */
        struct in_addr sin_addr; /* l'adresse Internet */
        char sin_zero[8]; /* un champ de 8 zéros */
};

struct in_addr {
    uint32_t s_addr; /* 4 octets */
};
```

- *Structure définie dans <netinet/in.h>*
  - sin_family représente le type de famille AF_INET
  - sin_port représente le port à contacter
  - sin_addr représente l'adresse IP de l'hôte
  - sin_zero[8] contient uniquement des zéros (permet de se caler sur la structure sock_addr)

Utilisation d'une socket: structure d'une adresse

- Pour accéder au champ contenant l'adresse IPv4 :

```
struct sockaddr_in mysock;

mysock.sin_addr.s_addr;
```

Utilisation d'une socket: structure d'une adresse

- Côté serveur
  - Il est nécessaire de spécifier un numéro de port lors de l'appel à la fonction bind(),
  - ⇒ pour que d'autres processus puissent contacter le serveur à travers la socket ouverte.
- Côté client
  - Le numéro de port peut ne pas être spécifié (port=0).
  - => le système choisira lui même un port

- Pour connaître l'@IP source on utilisera INADDR_ANY comme paramètre

Structure d'une adresse: exemple

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int s, const struct sockaddr *adr, int adrlen);
```

- Comme la fonction a besoin d'un pointeur sur structure sockaddr, et que nous disposons que d'une structure sockaddr_in,
  - => faire un cast, pour éviter que le compilateur nous retourne une erreur lors de la compilation.

Structure d'une adresse: exemple

- *Création d'une adresse pour un client*

```
SOCKADDR_IN sin;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_family = AF_INET;
sin.sin_port = 0; //Port alloué dynamiquement
```

- *Création d'une adresse pour un serveur qui écoute sur le port 33000*

```
SOCKADDR_IN sin;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_family = AF_INET;
sin.sin_port = htons(33000); //Port port 33000
```

- *Affectation de l'adresse IP à la socket*

```
bind(sock, (SOCKADDR*)&sin, sizeof(sin)
```

Conversion

```
char * inet_ntoa(const struct in_addr in);
in_addr_t inet_addr(const char *cp);
int inet_aton (const char *cp, struct in_addr *inp);
```

- inet_ntoa() : Convertit la structure binaire in en chaîne de caractères (notation décimale pointée)
- inet_addr() (obsolète): Convertit la chaine cp en binaire et si l'adresse cp est invalide INADDR_NONE est renvoyée
- inet_aton(): Idem que inet_addr mais plus récent

```
struct sockaddr_in myaddr;
char *some_addr;

inet_aton("10.0.0.1", &myaddr.sin_addr);
some_addr = inet_ntoa(myaddr.sin_addr);
printf("%s\n", some_addr);

// and this call is the same as the inet_aton() call, above:
myaddr.sin_addr.s_addr = inet_addr("10.0.0.1");
```

Conversion

```
unsigned long htonl(unsigned long hostlong);
unsigned short htons(unsigned short hostshort);
unsigned long ntohl(unsigned long netlong);
unsigned short ntohs(unsigned short netshort);
```

- Deux méthodes pour enregistrer les entiers
  - Little endian et Big endian
- Les ordinateurs enregistrent dans l'ordre défini par leur processeur (Host Byte Order)
- Exemple:
  - Intel x86, *Host Byte Order* est *Little-Endian*
- L'ordre dans lequel les octets doivent être écrits sur le réseau est Big endian

Utilisation d'une socket: attente de connexions

```
int listen(int socket, int backlog);
```

- Attente de connexions sur une socket
- Listen()
  - Permet de déclarer un service auprès du système
  - socket : la socket créée avec la fonction socket()
  - Backlog : indique le nombre maximum de demandes de connexions mises en attentes
- Renvoi 0 si réussi, ou -1 en cas d'échec

Utilisation d'une socket: établissement de connexions

```
int accept(int socket, struct sockaddr* addr,
                       socklen_t* addrlen);
```

- Réservé aux sockets de type **SOCK_STREAM**
- Appel bloquant en attente d'une connexion
- Si pas d'erreur un nouveau descripteur où des données pourront être lues ou écrites est renvoyé.
- La socket **socket** reste libre pour satisfaire les autres demandes de connexion.
- **Addr** et **addrlen** :
  - Donnent, au retour de l'appel système, l'identité de la machine distante

Exemple: initialisation du serveur

```
struct sockaddr_in saddr_in;
int sock;
```

Déclaration des variables

```
sock = socket(PF_INET, SOCK_STREAM, 0);
if ( sock == -1 ) {
  perror( "socket" ); exit( EXIT_FAILURE );
}
```

Création de la socket

```
memset(&saddr_in, 0,sizeof (saddr_in));
    saddr_in.sin_family = AF_INET;
  saddr_in.sin_port = htons( port );
saddr_in.sin_addr.s_addr = INADDR_ANY;
```

Initialisation de l'adresse

```
if ( bind( sock ,(struct sockaddr *) &saddr_in,sizeof(saddr_in) ) == -1 ){
                    perror( "bind" );
                  exit( EXIT_FAILURE );
                  }
```

Attachement de l'adresse à la socket

```
   if ( listen( sock , SOMAXCONN ) == -1 ) {
     perror( "listen" ); exit( EXIT_FAILURE );
                       }
```

Attente active sur la socket

Utilisation d'une socket: demande de connexion

```
#include <sys/socket.h>

int connect(int socket, struct sockaddr *address, int address_len);
```

- Socket
  - Descripteur de la socket
- Sockaddr
  - Adresse de la socket distante
- Address_len
  - Taille de la structure sockaddr

Exemple: initialisation du client

```
struct sockaddr_in sock_host;
int sock;

sock = socket(AF_INET, SOCK_STREAM, 0);

memset(& sock_host, '\0', sizeof(sock_host));
sock_host.sin_family = AF_INET;
sock_host.sin_port = htons(port);
inet_aton("192.168.1.10", & sock_host.sin_addr);

connect(sock, (struct sockaddr *) & sock_host, sizeof(sock_host));
```

Déclaration des variables

Ouverture de la socket

Création de l'adresse de la machine distante

Connexion à la station distante 192.168.1.10

Connexion établie: **envoi**/réception des données

```
int send(int sockfd, const void *msg, size_t len, int flags);
```

- Sockfd: Socket sur laquelle on souhaite envoyer des données
- Msg: Un pointeur sur les données que l'on souhaite envoyer
- Len: Taille des données en octets que l'on envoie
- Flags: Options, traditionnellement à 0
- send() renvoie le nombre d'octets qui ont été transmis
- Attention : Le nombre d'octets transmis peut être différent de len => C'est au développeur de prendre en charge l'émission des données qui n'ont pas été transmises.
- Fonctionne uniquement sur une socket qui a été préalablement connectée
- -1 peut être envoyé en cas d'erreur et la variable errno est modifiée en conséquence.

Connexion établie: envoi/**réception** des données

```
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

- Sockfd: le descripteur de la socket à partir de laquelle on peut lire les données
- Buf: le buffer dans lequel les données sont inscrites
- Len: la taille du buffer
- Flags: options particulières normalement à 0
- recv(): renvoie le nombre de bits écrits dans le buffer buf
  - Peut renvoyer 0 ⇔ La connexion a été fermée par l'entité distante
  - -1 si une erreur a surgi (variable **errno** mise à jour)

Connexion établie: envoi/réception des données

```
ssize_t write(int fd, const void *buf, size_t count);
int read(int sockfd, char* buf, int len);
```

- Il est également possible d'utiliser les primitives write() et read()

Fermeture de socket

```
int close(int sockfd);
```

- Fermeture d'une socket identique à la fermeture d'un fichier
- Interdit pas la suite toute écriture ou lecture sur la socket
- Toute lecture ou écriture sur la socket génèrera une erreur

```
int shutdown(int sockfd, int how);
```

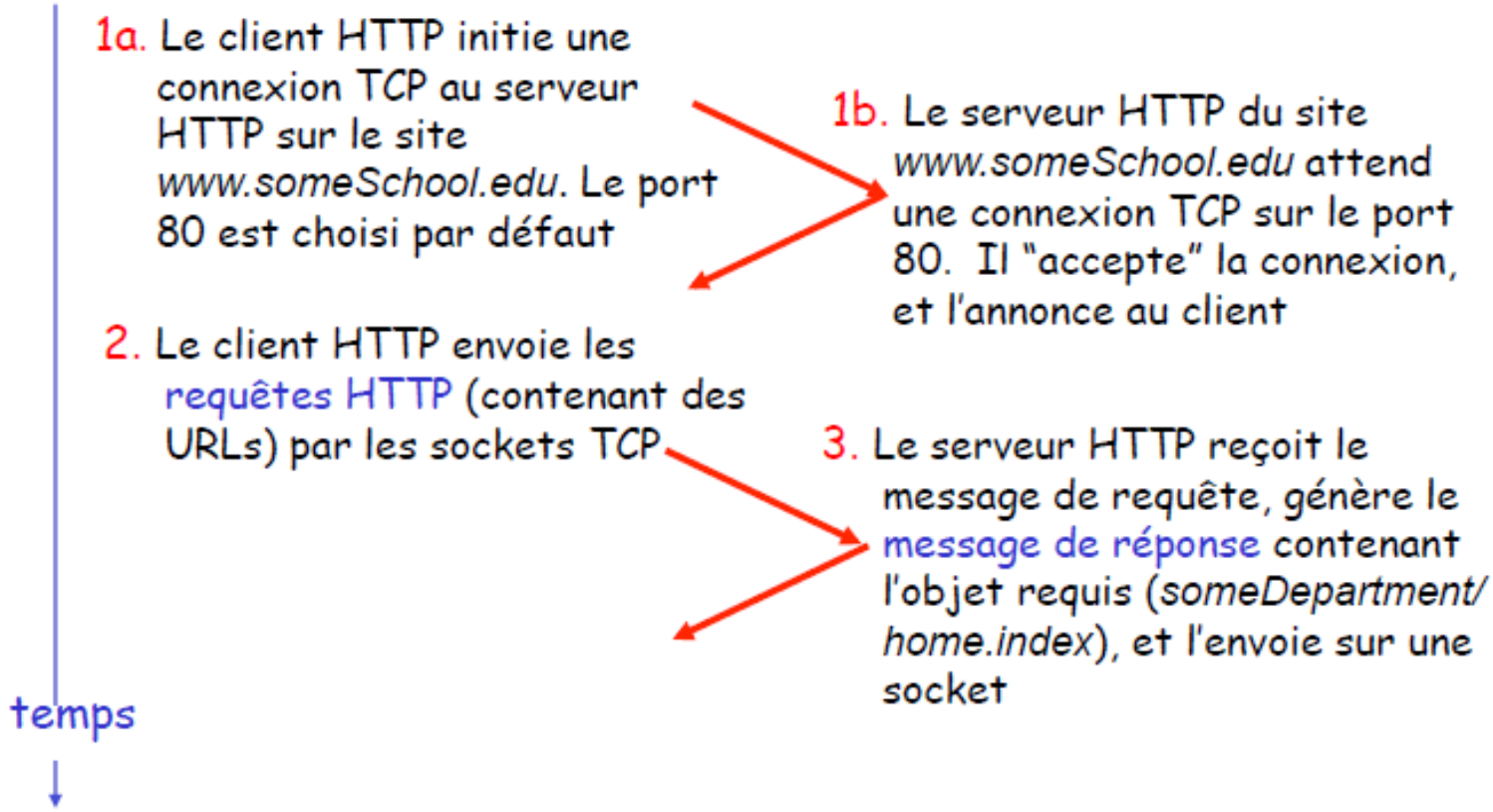
- Permet d'avoir plus de contrôle sur la fermeture de socket:
- how peut prendre une des valeurs suivantes :
  - 0: lecture impossible seulement
  - 1: écriture impossible seulement
  - 2: lecture et écriture impossible ⇔ close()
- Renvoie -1 en cas d'erreur, met à jour la variable errno

Annexe: Exemple du Web (Modèle Client-Serveur)

Joachim Bruneau-Queyreix

Le Web: le protocole HTTP, exemple

- Si un utilisateur entre l'URL : www.someSchool.edu/someDepartment/home.index



Le Web: le protocole HTTP, exemple

4. Le serveur HTTP ferme la connexion TCP

temps

- 5. Le client HTTP reçoit la réponse contenant le fichier HTML file et l'affiche.

  En décodant le fichier, le browser trouve les URLs référencées

6. Les étapes 1-5 sont répétées pour chaque URL référencée

Le Web: le protocole HTTP, connexions

- Connexions non-Persistantes
  - HTTP/1.0
  - Après avoir servi une requête, le serveur ferme la connexion TCP
  - 2 Round Trip Times sont nécessaires pour lire chaque objet
  - Chaque transfert doit supporter le slow-start
  - La plupart des navigateurs de version 1.0 utilisent des connexions parallèles

Le Web: le protocole HTTP, connexions

- Connexions Persistantes
  - Par défaut (avec pipeline) dans HTTP/1.1
  - 1 connexion TCP est ouverte vers le serveur
    - Le client envoie la requête de tous les objets requis dès qu'ils sont référencés dans la page HTML en utilisant la même connexion TCP/IP
    - En général, le serveur ferme la connexion dès que la connexion n'a pas été utilisée pour une certaine période (qui souvent peut être configurée)
  - Deux versions :

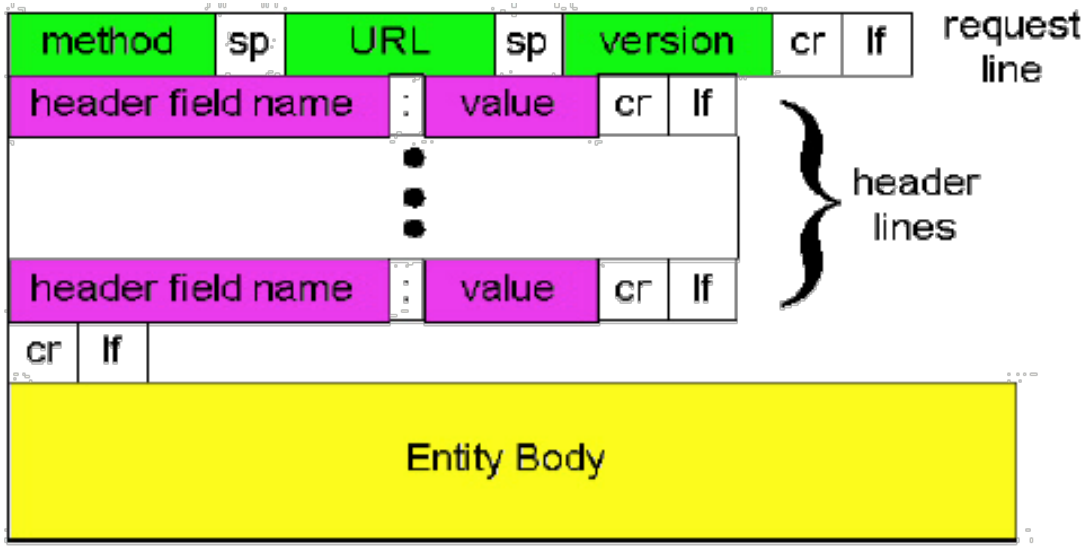
1. Sans pipeline ⇨ le client envoie la prochaine requête seulement quand la précédente a été servie
2. Avec pipeline ⇨ une requête est envoyée dès que le processus client trouve une référence

Le Web: le protocole HTTP, Format

- Format de message HTTP: requête
- Deux types de messages http : requête et réponse
  - Message de requête http :

Le Web: le protocole HTTP, Format

- Format de message HTTP: requête

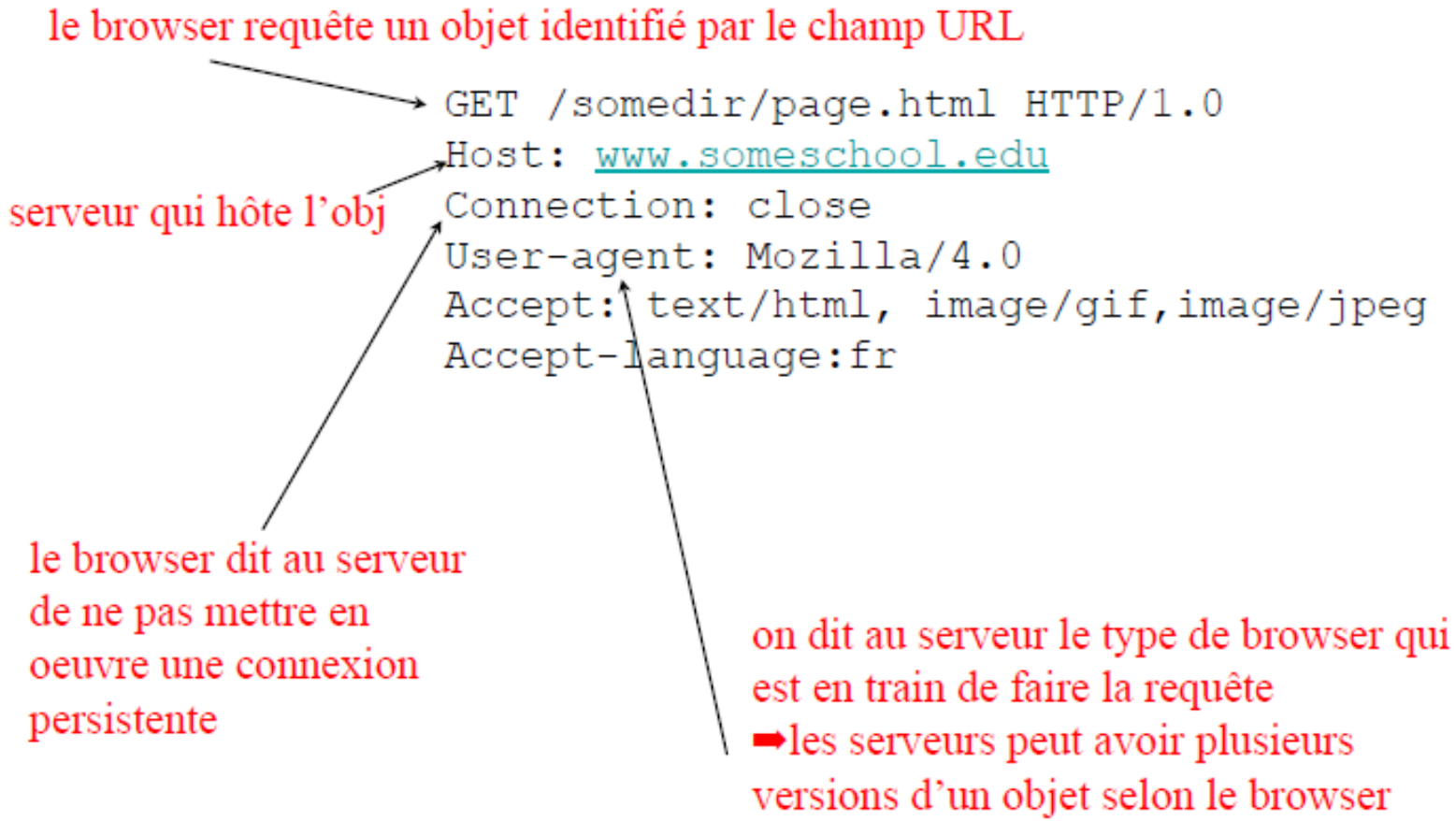


Quelques méthodes:

- GET pour envoyer la requête d'un objet
- HEAD comme GET mais le serveur ne doit pas envoyer l'objet: sert à vérifier la validité, l'accessibilité, … de nouveaux URL
- POST utilise le champ «Body» et sert à demander au serveur de considérer les données contenues comme une partie de l'objet spécifié dans l'URL

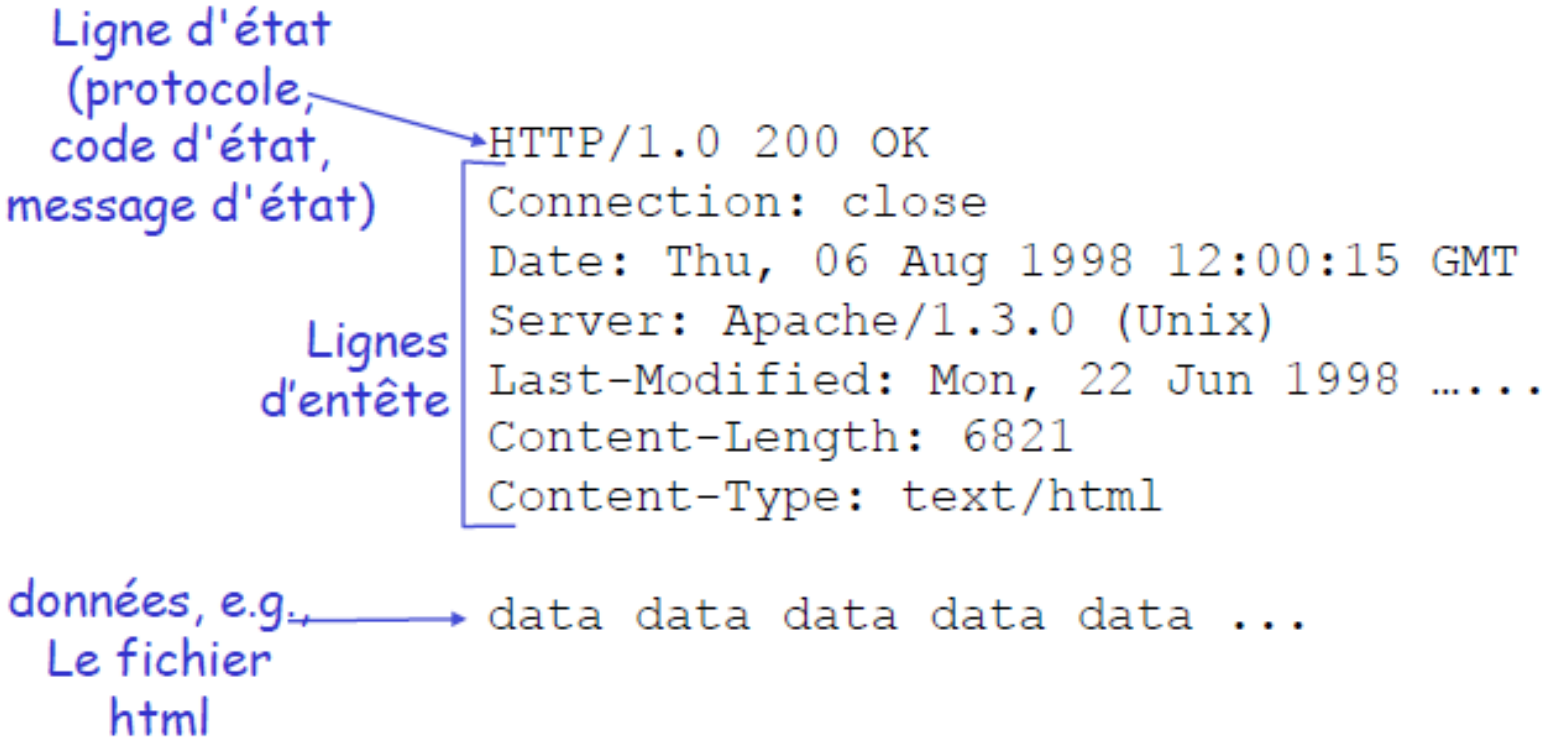
Le Web: le protocole HTTP, Format

- Format de message HTTP: requête



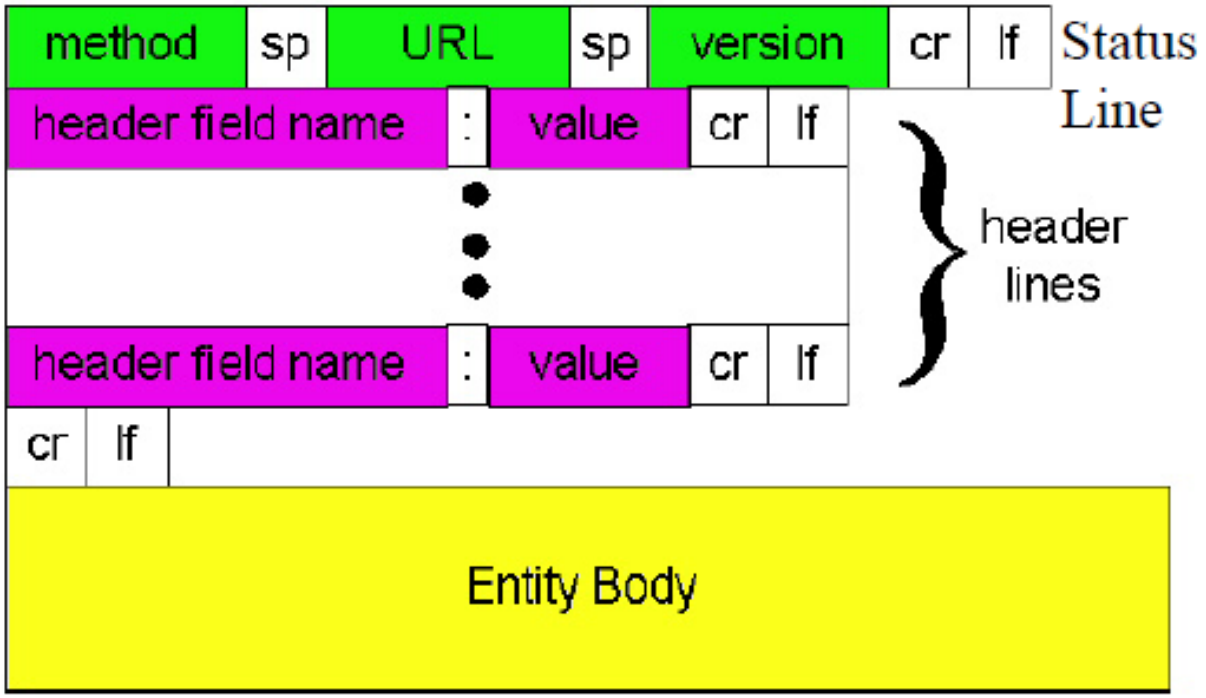
Le Web: le protocole HTTP, Format

- Format de message HTTP: réponse



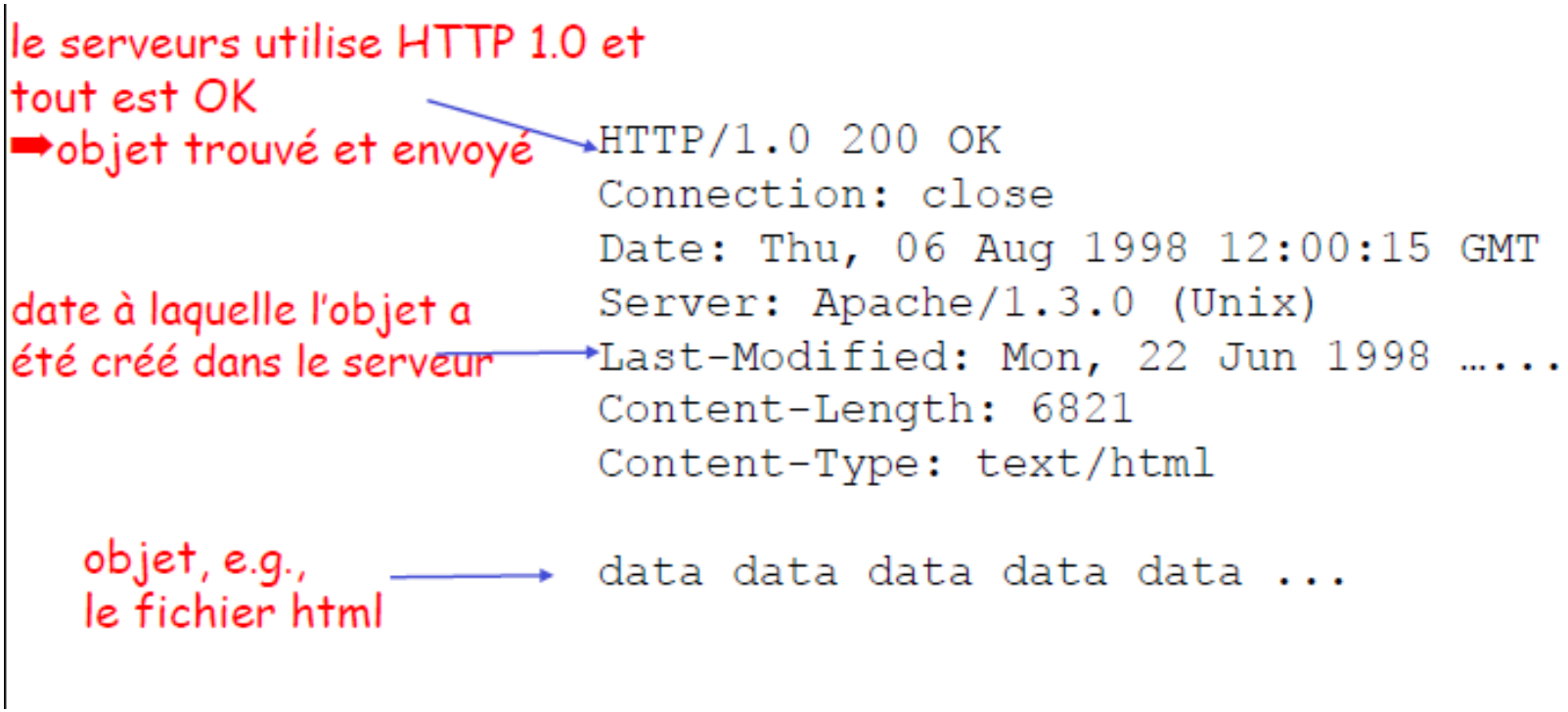
Le Web: le protocole HTTP, Format

- Format de message HTTP: réponse



Le Web: le protocole HTTP, Format

- Format de message HTTP: réponse

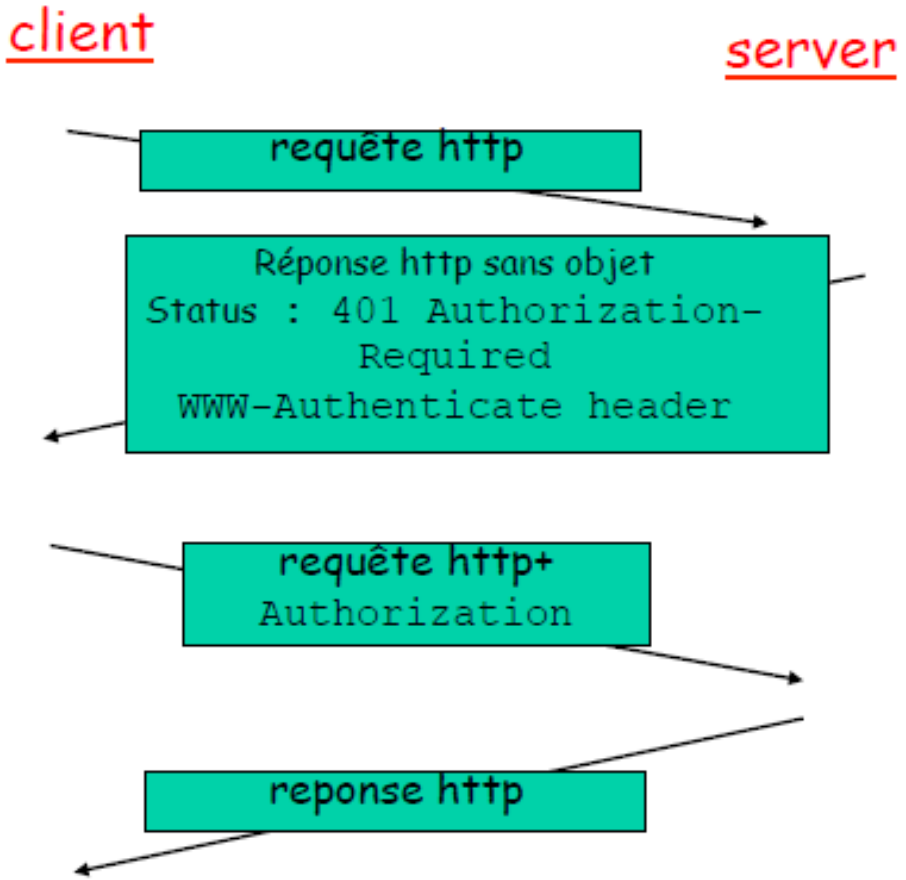


Le Web: le protocole HTTP, Format

- Format de message HTTP: réponse
- Dans la 1ere ligne de la réponse serveur -> client
  - 200 OK: la requête a réussi et l'objet demandé est à la suite
  - 301 Moved Permanently: l'objet demandé a changé définitivement de place, son nouvel emplacement est donné dans la suite du message
  - 400 Bad Request: la requête est erronée, ne peut pas être comprise par le serveur
  - 404 Not Found: le document demandé n'est pas disponible sur le serveur
  - 505 HTTP Version Not Supported

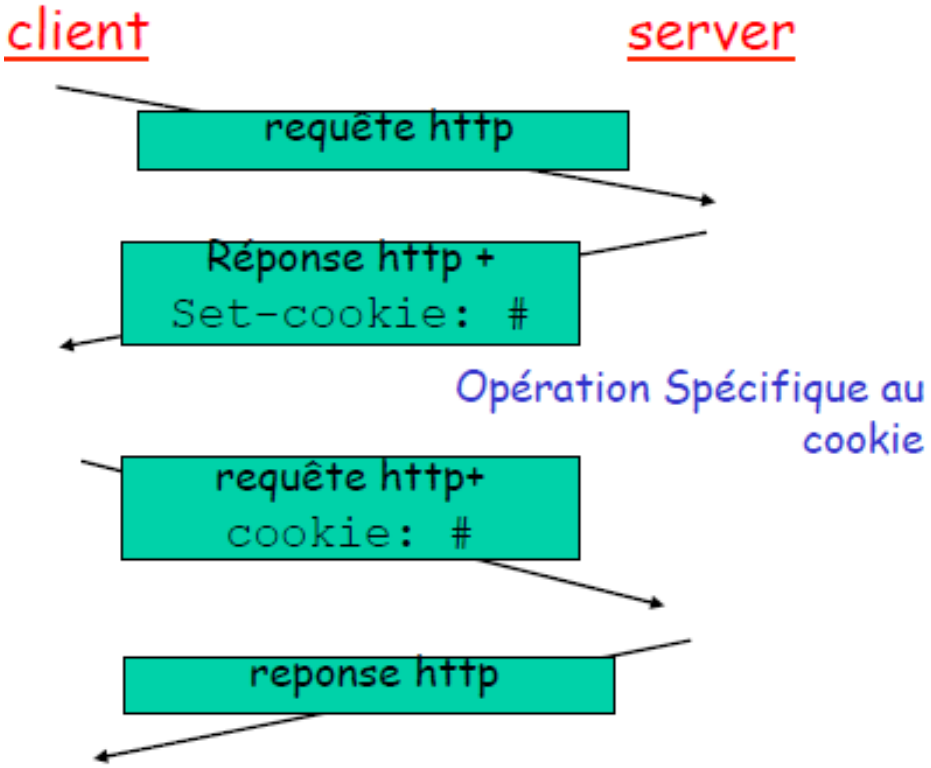
Interaction entre le client et le serveur: Authentification

- Authentification :
  - De nombreux sites demandent un identifiant et un mot de passe
  - HTTP fournit des codes et des entêtes d'état pour permettre l'authentification
  - WWW-Authenticate header définit pour s'authentifier



Interaction entre le client et le serveur: Cookies (RFC 2109)

- Le serveur envoie un “cookie” vers le client dans la reponse
  - Set-cookie: 1678453
- Le client présente le cookie dans les requêtes suivantes
  - cookie: 1678453
- Le serveur vérifie le cookie avec ces informations enregistrées

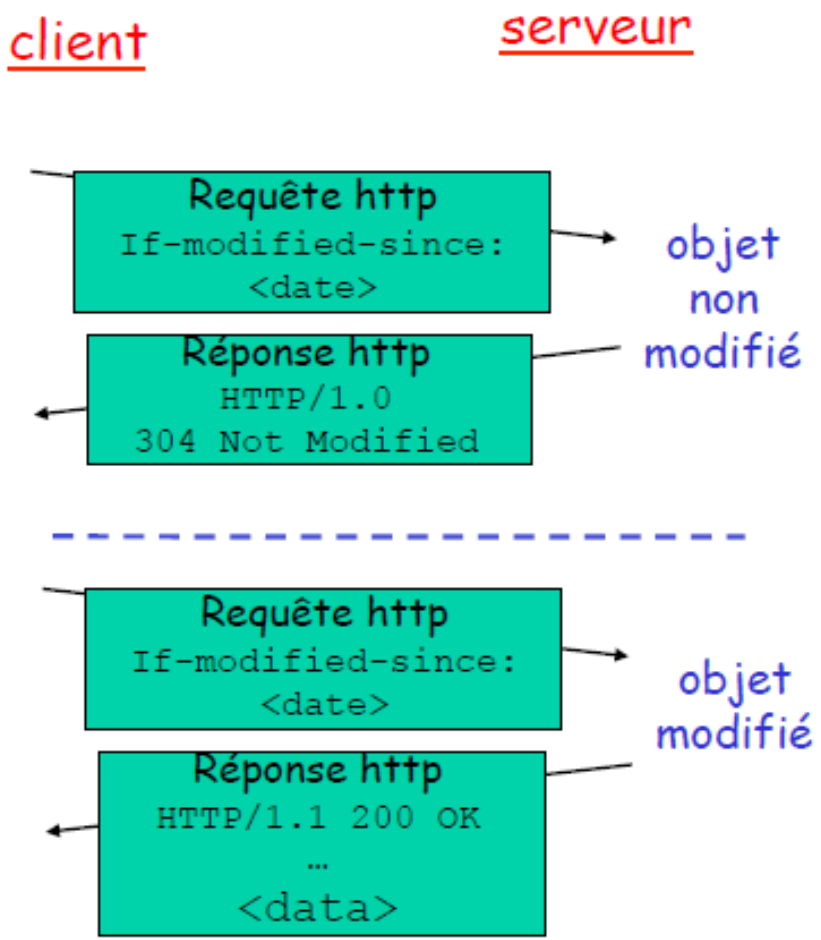


Utilité des cookies

- Serveur nécessitant une authentification, sans demander systématiquement un identifiant et un mot de passe
- Trace des préférences de l'utilisateur, par exemple pour faire de la publicité ciblée
- Garder une trace des achats de l'utilisateur lors d'achats en ligne
- …
- Problème : utilisateurs nomades accédant à un même site depuis différentes machines

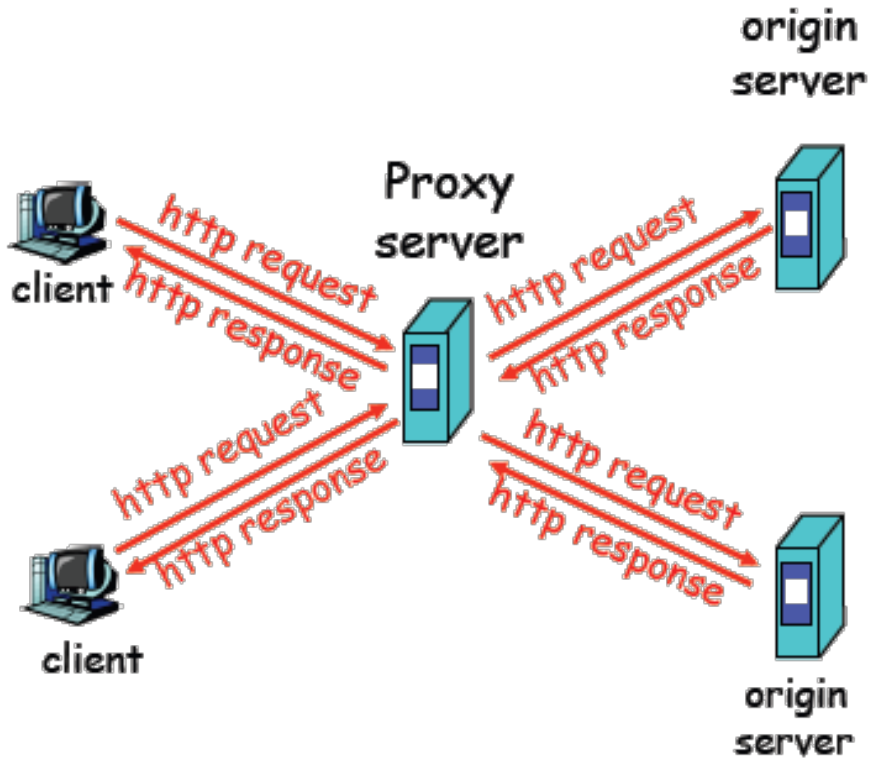
GET conditionnel ET caching

- Objectif : ne pas envoyer un objet que le client a déjà dans son cache
- Problème : les objets contenus dans le cache peuvent être obsolètes
  - Client: spécifie la date de la copie cachée dans la requête http
    - If-modified-since: <date>
  - Serveur: la réponse est vide si la copie cachée est à jour
    - HTTP/1.0 304 Not Modified



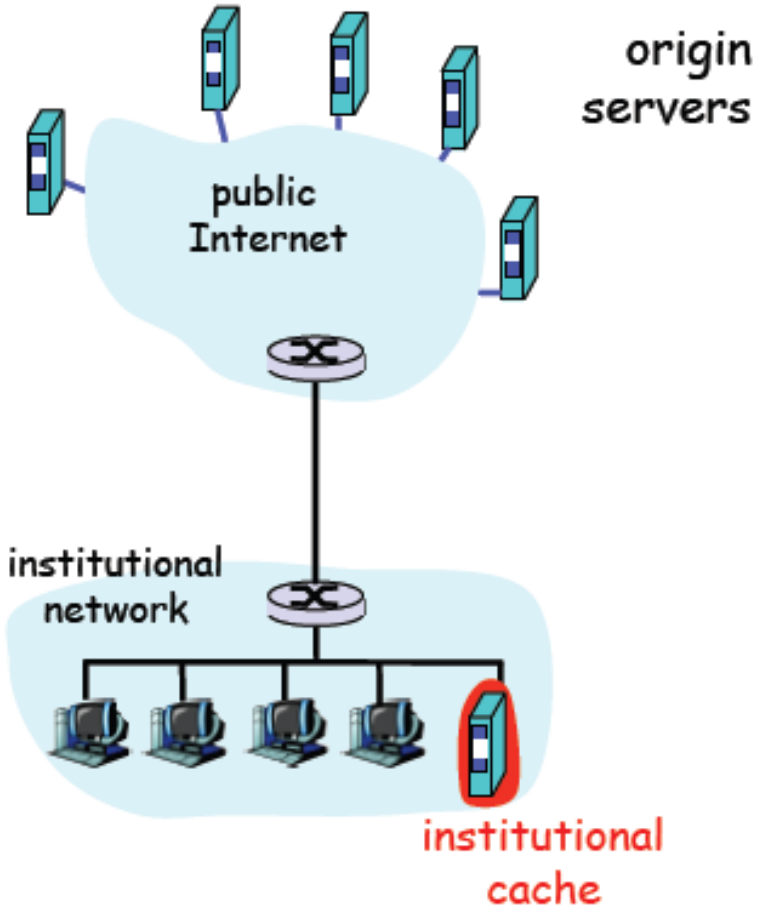
Cache Web / Server Proxy

- Objectif : satisfaire rapidement la requête du client
- Configuration du browser pour qu'il pointe vers le cache
  - Le client envoie toutes ses requêtes HTTP vers le cache Web
  - Si l'objet est dans le cache, on le renvoie
  - Sinon le proxy demande au serveur initial et il répond ensuite à la requête



Intérêt du cache Web

- Hypothèse : le cache est proche du client
  - Réduction du temps de réponse
  - Réduction du trafic vers les serveurs distants



Obtention d'une @ à partir du nom

- int gethostname(char *nom, size_t lg)
  - Obtient dans le tableau nom, le nom de la machine locale à partir de laquelle la fonction est appelée
  - lg contient la taille alloué pour nom
  - Renvoi 0 en cas de succès ou -1 dans le cas contraire

- struct hostent *gethostbyname(char *nom)
  - Renvoie
    - Les informations relatives à la machine nom, par exemple, www.yahoo.fr
    - ou NULL si elle n'est pas identifiable
  - Le pointeur renvoyé pointe en zone statique. Il faut donc copier le résultat

Obtention d'une @ à partir du nom

```
struct hostent *gethostbyname(char *nom)
```

- **Retourne un pointeur sur une structure du type :**

```
struct hostent {
        char * h_name; /* nom de la machine */
        char **h_aliases; /* tableau des autres noms */
        int h_addrtype; /*  type d'adresse : AF_INET */
        int h_length; /*  la longeur d'une adresse (4 ipv4) */
        char ** h_addr_list;
        /* tableau des adresses de types struct in_addr */
}
```

- **Obtention de l'@ à partir du nom:**

```
struct hostent* res;
struct in_addr* addr;
res = gethostbyname("machine");
addr = (struct in_addr*) res->h_addr_list[0];
printf("L\'adresse est : %s\n", inet_ntoa(*addr));
```

Transfert de données **sans connexion (UDP)**

```
int sendto(int sockfd, const void *msg, int len,
       unsigned int flags, const struct sockaddr *to,
                                        socklen_t tolen);
```

- Sendto(): Similaire à send() avec deux paramètres supplémentaires:
  - To: Pointeur sur la structure sockaddr
  - Tolen: ⇔ sizeof(to)

Transfert de données **sans connexion (UDP)**

```
int recvfrom(int sockfd, void *buf, int len,
unsigned int flags,struct sockaddr *from, int *fromlen);
```

- Recvfrom(): Similaire à recv() avec deux paramètres supplémentaires:
  - From: Pointeur sur la structure sockaddr
  - Fromlen: ⇔ sizeof(from)