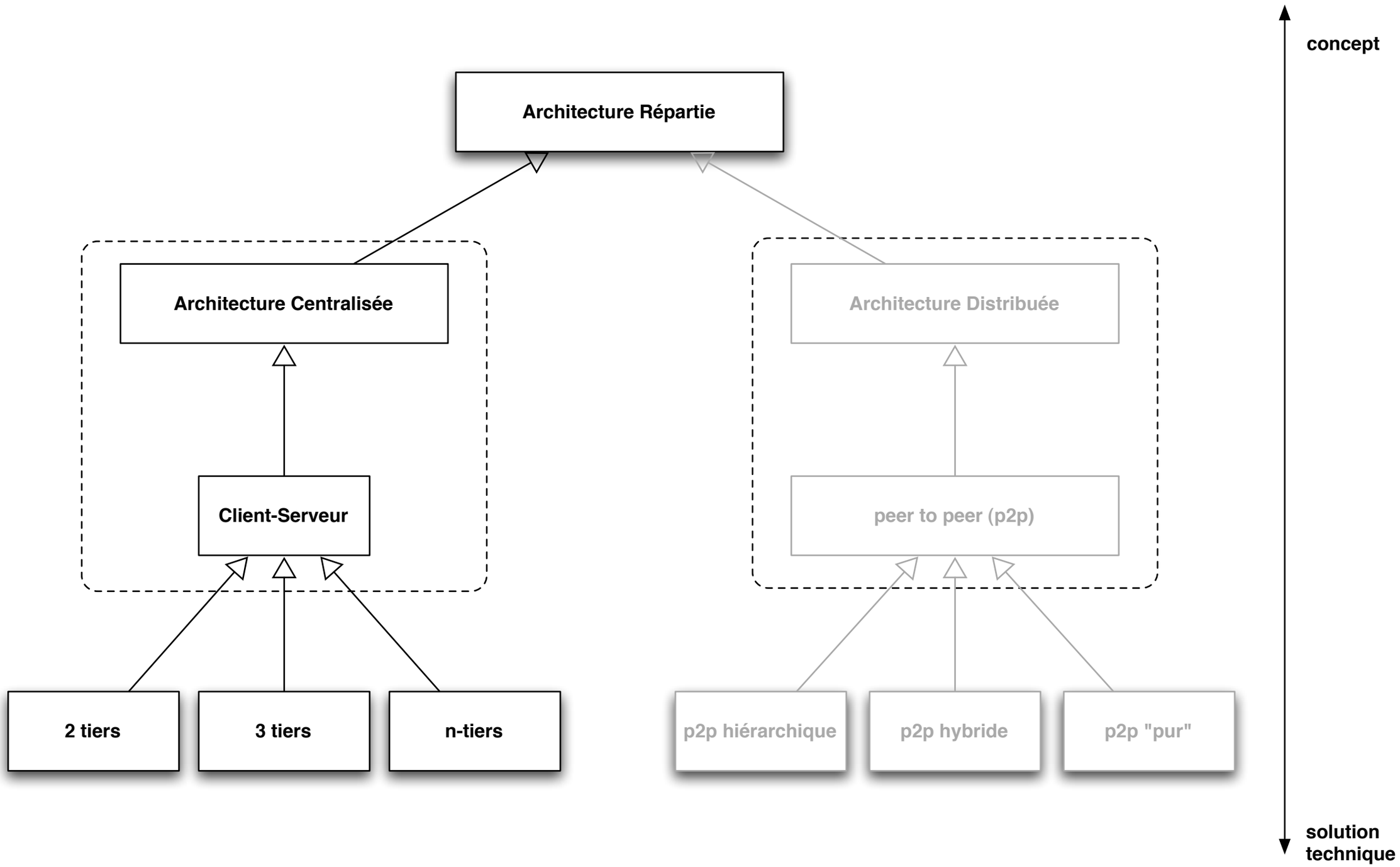
Architectures réparties

Chap #4

Architectures réparties
Centralisées et distribuées

Chap #4.1

Architectures centralisées



Architecture client-serveur

Problématique

- Il s'agit de permettre à des programmes/applications/entités logicielles d'échanger de l'information entre plusieurs machines reliées par un réseau
  - à large échelle (Internet)
  - en local (Intranet)
- **Séparation des rôles** inhérente à la plupart des systèmes informatiques :
  - Certaines entités **fournissent des services** (de calcul, de données,...)
  - Certaines entités **souhaitent accéder à ces services** (pour soumettre des requêtes, des calculs, afficher des données,...)
- Comment cela se traduit-il du point de vue de l'architecture logicielle ?

Architecture client-serveur

Définition

- Le client-serveur est un type d'architecture qui effectue une distinction stricte entre les **rôles** de **client** d'une part et de **serveur** d'autre part.
- On peut l'interpréter d'un **point de vue matériel** (des machines clientes et des machines serveurs) ou **logiciel** (entités clientes et entités serveurs d'une même application).
- Les interprétations matérielles et logicielles sont souvent liées: **une application client-serveur est communément répartie sur plusieurs machines connectées par un réseau.**
- Mode de fonctionnement général :

1. Point de vue client: un client effectue une requête pour un service précis sur un serveur donné et reçoit une réponse.
2. Point de vue serveur: un serveur reçoit une demande de service, la traite, et renvoie la réponse au client.

Architecture client-serveur

Caractéristiques générales

- **Protocoles asymétriques :** un serveur répond aux requêtes de plusieurs clients. Les clients initient le dialogue en demandant un service, alors que les serveurs attendent passivement les requêtes des clients.

- **Encapsulation des services :** le serveur est un spécialiste, lorsqu'il reçoit une requête de service il détermine lui-même comment traiter la demande. Les serveurs peuvent être mis à jours sans que les clients s'en aperçoivent du moment que leur interface publiée reste inchangée.

- **Intégrité : le code et les données d'un serveur sont maintenues de manière centralisée**, ce qui résulte en des coûts de maintenance réduits et une garantie d'intégrité des données. Mais cela implique des problèmes de montée en charge, de part le point de centralisation (serveur) qui constitue une faiblesse.

Architecture client-serveur

Caractéristiques générales client

- Il est actif (ou maître).
- Il envoie des requêtes au(x) serveur(s).
- Il attend (pas forcement) et reçoit les réponses du/des serveur(s).
- Il ne peut être connecté qu’à un petit nombre de serveurs en même temps.
- Il interagit généralement directement avec un utilisateur final en utilisant une IHM.

Architecture client-serveur

Caractéristiques générales serveur

- Il est passif (ou esclave).
- Il est à l'écoute, prêt à répondre aux requêtes envoyées par **plusieurs clients.**
- Dès qu'une requête lui parvient, il la traite et envoie une réponse.
- Il accepte normalement un nombre important de connections de la part des clients.
- Il n'interagit pas directement avec les utilisateurs finaux.

Architecture client-serveur

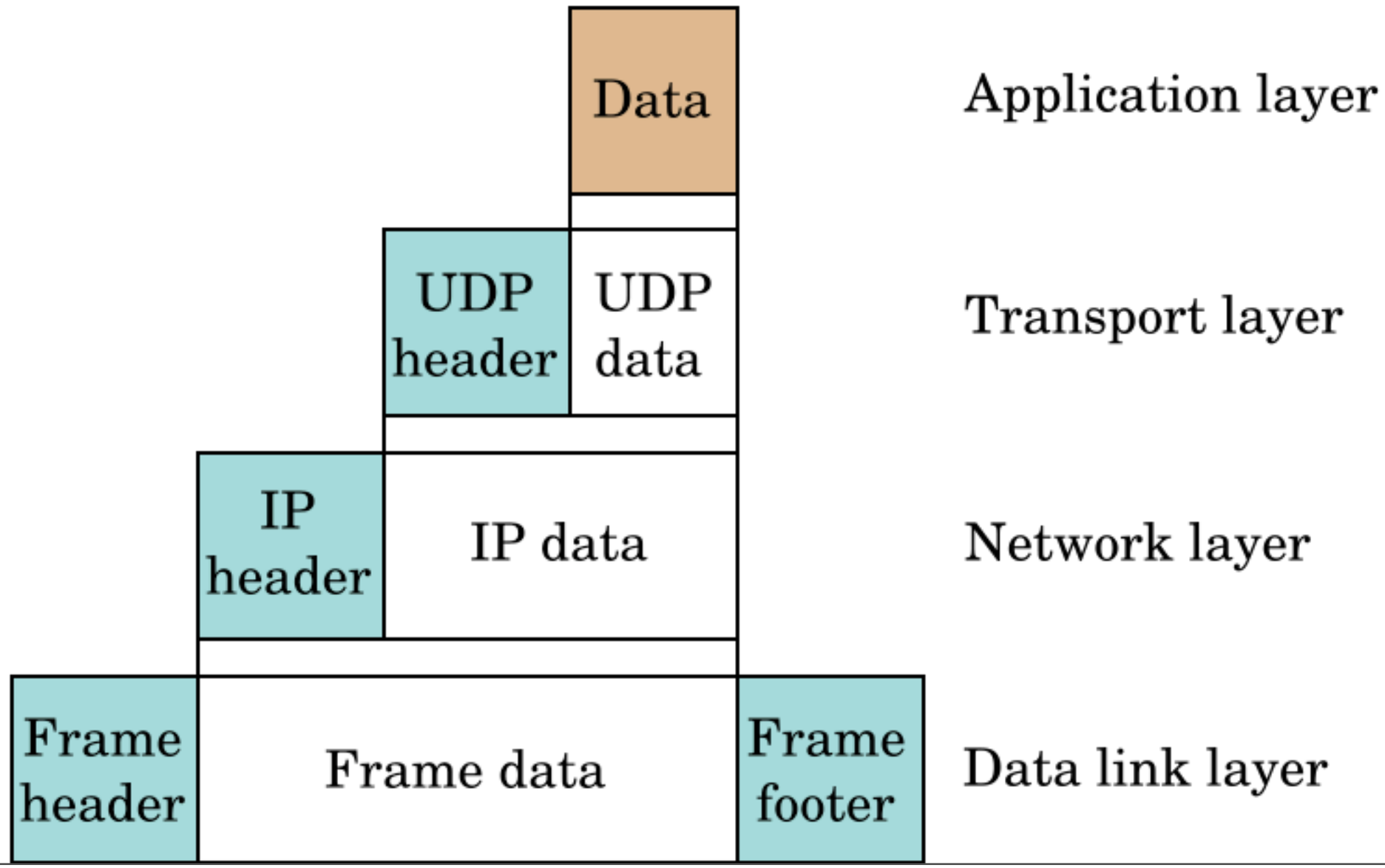
Protocoles client-serveur

- Un protocole client-serveur permet la communication entre clients et serveurs: il définit le format, l'ordre des messages envoyés et reçus via le réseau, et à les actions à entreprendre lors de la réception des messages.
- **On parle ici de protocoles applicatifs de haut niveau, indépendamment de leurs implémentations dans la couche transport inférieure (tcp, udp, ...)**
- Plusieurs paradigmes de communication sont possibles lors de l'implémentation de protocoles client-serveur, par exemple :
  - Passage de messages synchrone.
  - Passage de messages asynchrone.
  - Appel de procédures à distance (RPC).
  - Invocation de méthodes à distance.
  - Interaction événementielle.
  - Interaction par messagerie.

Architecture client-serveur

Empilement des couches et protocoles

- Un exemple avec le protocole de transport UDP : on peut voir que le protocole applicatif peut être développé indépendamment de son encapsulation dans la couche transport de niveau inférieur.

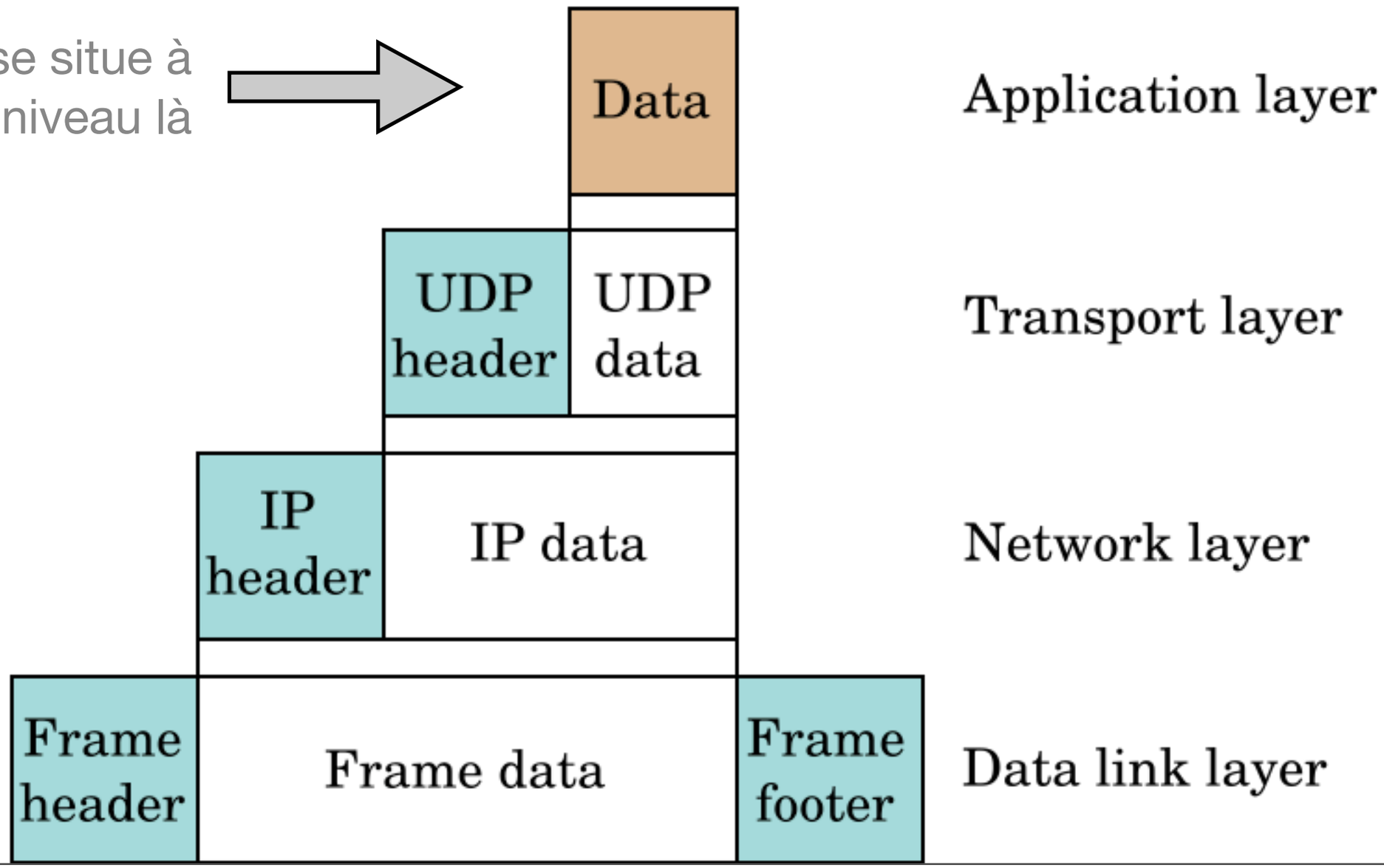


Architecture client-serveur

Empilement des couches et protocoles

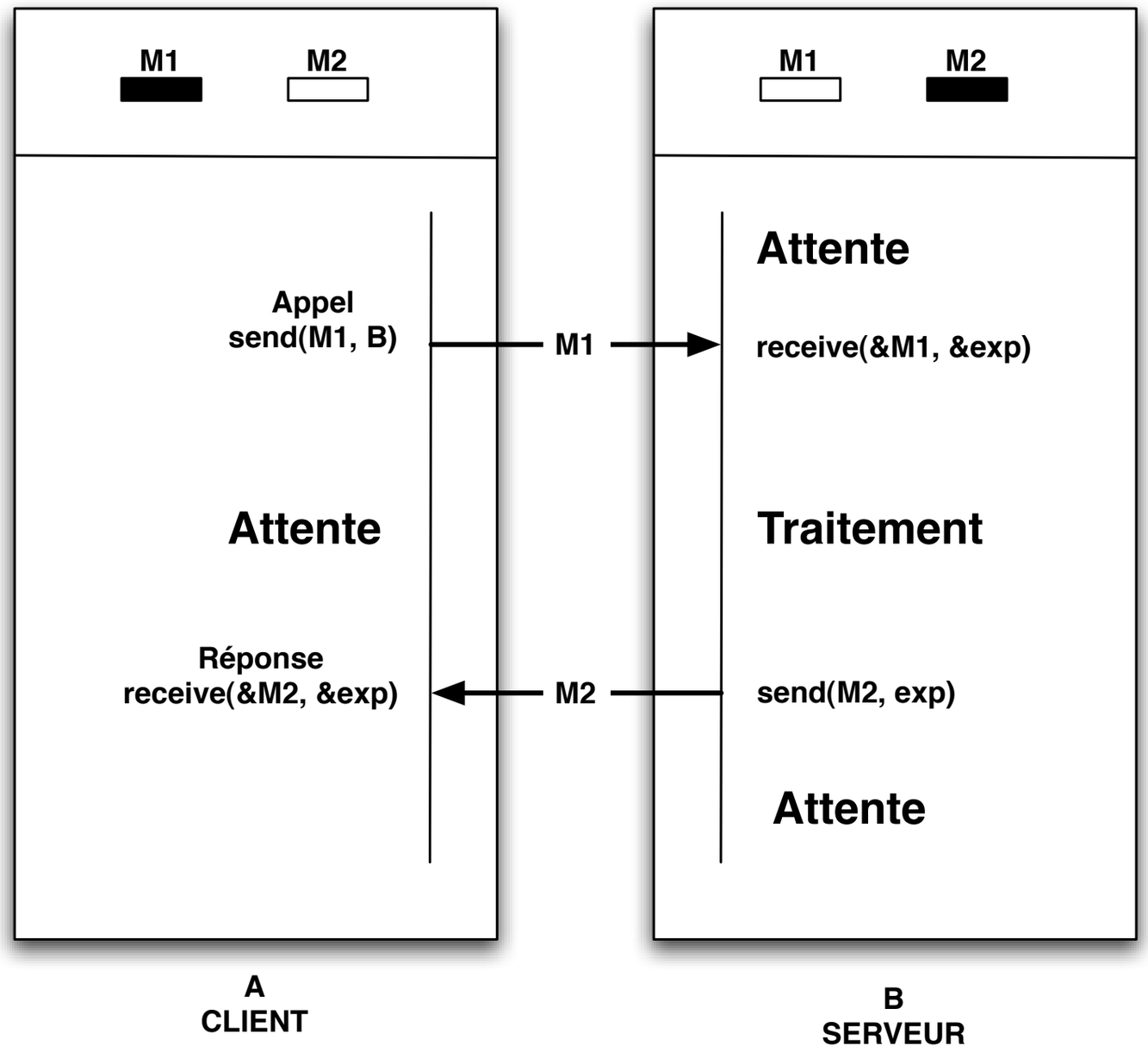
- Un exemple avec le protocole de transport UDP : on peut voir que le protocole applicatif peut être développé indépendamment de son encapsulation dans la couche transport de niveau inférieur.

On se situe à ce niveau là



Architecture client-serveur

Passage de messages synchrone



Architecture client-serveur

Passage de messages synchrone

- **send(message, destinataire)**

1. Recopie du message dans les tampons d'émission systèmes.

2. Envoi du message sur le réseau.

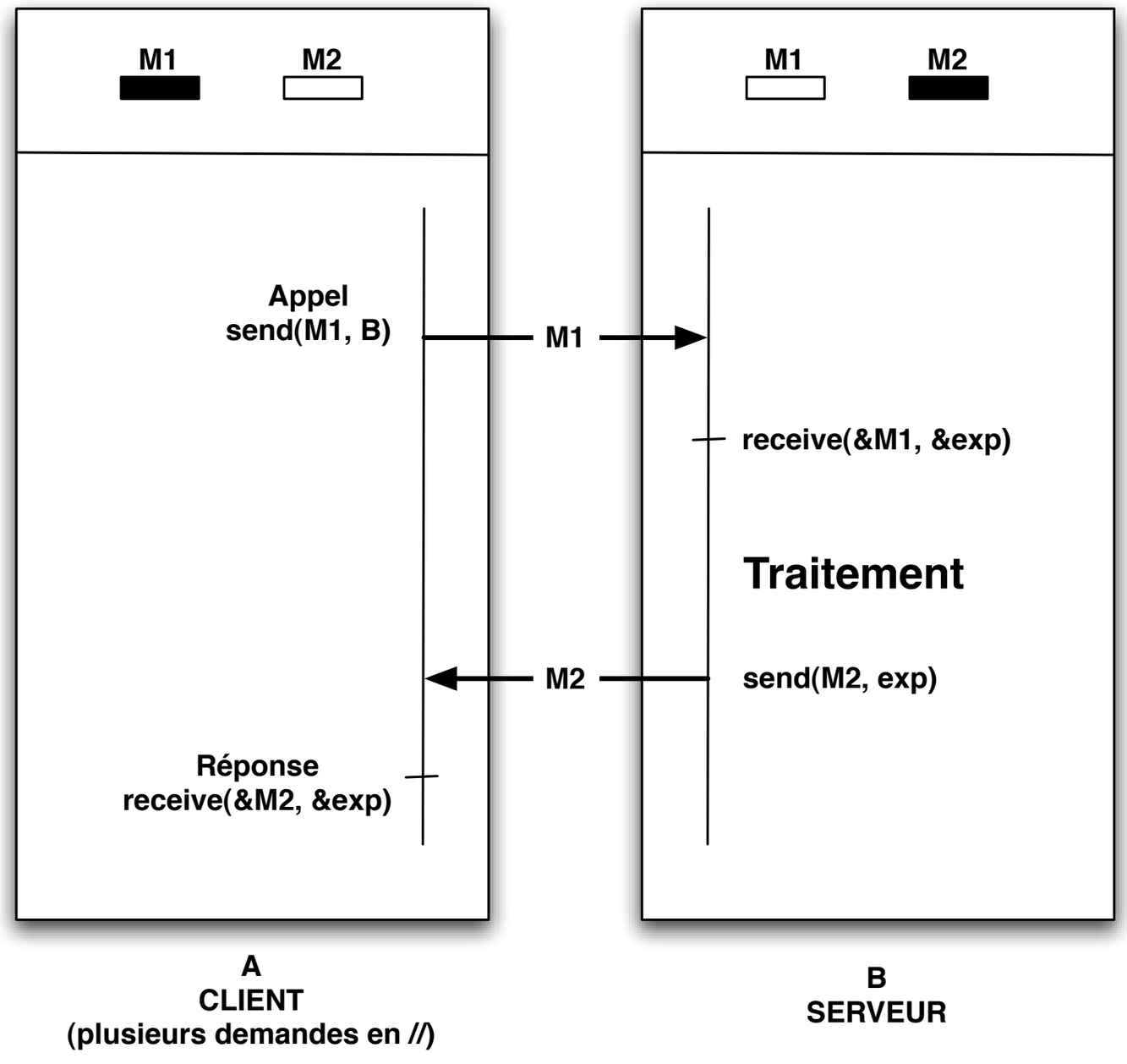
3. La machine destinataire recopie le message dans un tampon de réception.

4. Le message est recopié dans la zone message du processus destinataire; le champ expéditeur est rempli; le processus est réveillé.

5. Le processus destinataire peut consommer le message obtenu par receive(&message, &expediteur).

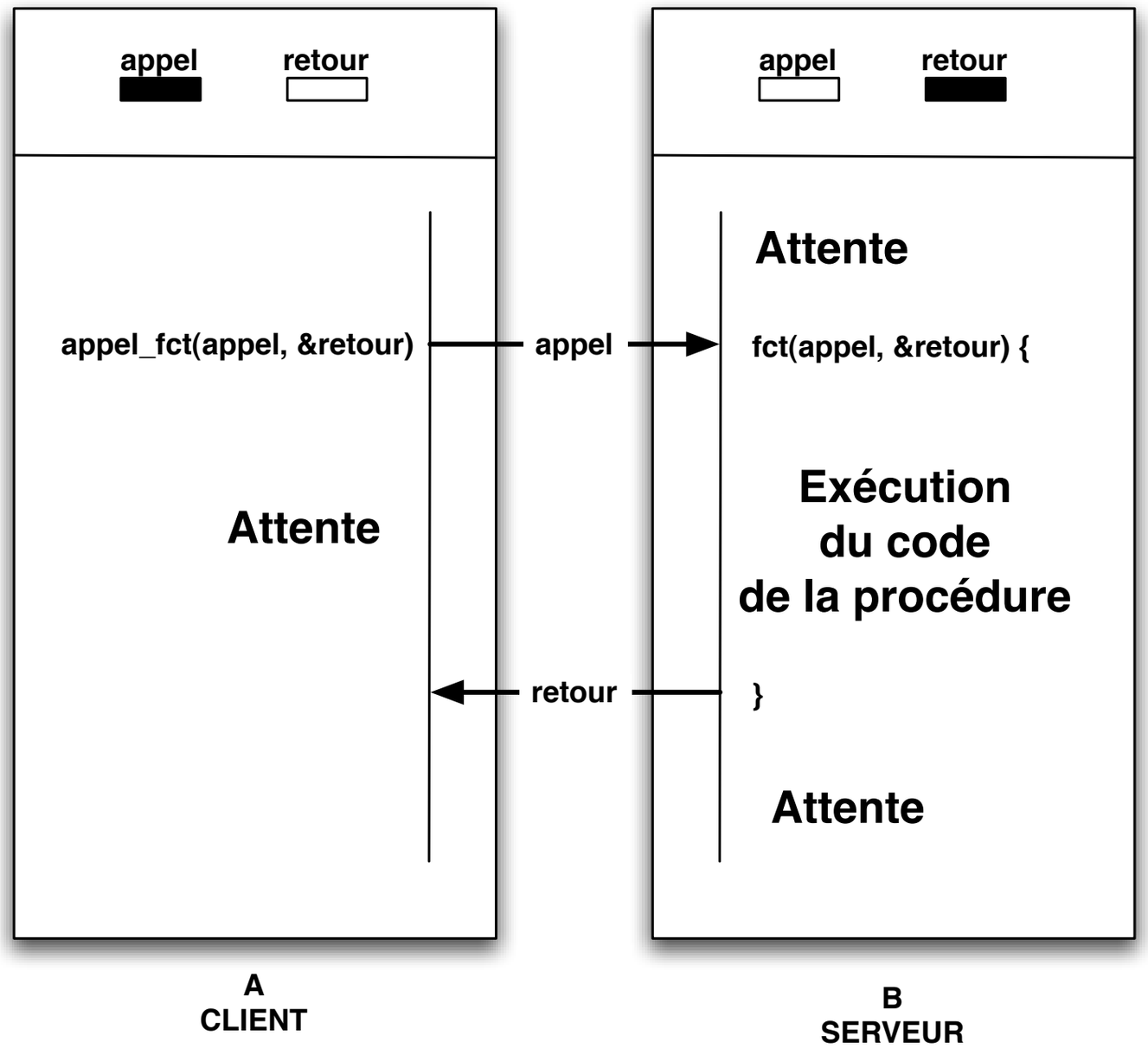
Architecture client-serveur

Passage de messages asynchrone



Architecture client-serveur

Appel de procédures à distance



Architecture client-serveur

Invocation de méthodes à distance

- **Généralisation des RPC au monde objet**: on appel une méthode sur un objet distant (ex. de technologie: Java/RMI)

- Par rapport aux RPC, les invocation de méthodes à distance permettent:

  - **De référencer des objets distants** (il faut précédemment avoir obtenu l'adresse distante de l'objet).

  - **De transférer des objets locaux "non-répartis"**:

    1. Sérialisés puis envoyés comme des données par valeur/copie au destinataire (opération de "marshalling" des données).

    2. L'objet est ensuite reconstitué sur le destinataire (opération de "unmarshalling" des données transmises via le réseau).

Architecture client-serveur

Interaction événementielle

- **Basé sur la notion de publication d'événements :**

  1. Le client s'abonne à un ou plusieurs événements auprès du serveur.

  2. Le serveur enregistre les abonnements, puis à chaque événement correspondant il envoie un message aux abonnés.

- **Notion de Push/Pull :**

  - Push : envoyer l'information à ceux qui en ont besoin.

  - Pull : récupérer l'information quand on en a besoin.

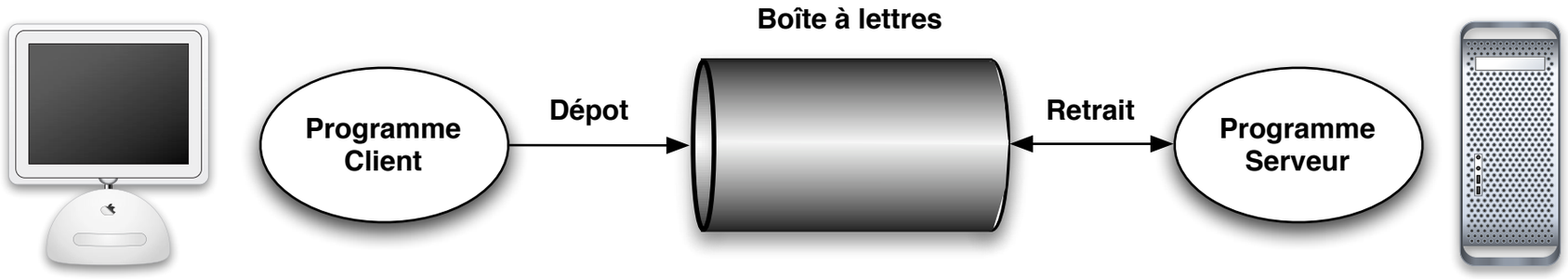
- **Avantages** : évite de surcharger le serveur de messages inutiles (lorsque aucune information est disponible) et évite d'envoyer un message lorsque aucun client n'est abonné.

- **Inconvénients** : le serveur doit conserver la liste des clients en mémoire persistance (cause des pannes).

Architecture client-serveur

Interaction par messagerie

- Consiste à utiliser une “boite à lettres” pour y déposer des messages qui pourront ensuite êtres récupérés par leur destinataire au moment voulu.
- Il s’agit donc d’un mécanisme de communication **asynchrone** (le fonctionnement des clients et des serveurs sont découplés).
- On peut utiliser pour cela un “Message Oriented Middleware” ou MOM.
- **Attention**: ne pas confondre avec l’envoi de messages sur une socket (cf. cours suivant). On parle ici de protocoles applicatifs de haut niveau et non pas de protocoles de transport.



Architecture client-serveur

Exemples

- **La consultation de pages web** fonctionne sur une architecture client/serveur. Un internaute connecté au réseau via son ordinateur et un navigateur Web est le client, le serveur est constitué par le ou les ordinateurs contenant les applications qui délivrent les pages demandées. Dans ce cas, c'est le protocole de communication HTTP qui est utilisé.

- **Les emails** sont envoyés et reçus par des clients et gérés par un serveur de messagerie. Les protocoles utilisés sont le SMTP, et le POP ou l'IMAP.

- **La gestion d'une base de données centralisée** sur un serveur peut se faire à partir de plusieurs postes clients qui permettent de visualiser et saisir des données.

- **Le système X Window (*nix)** fonctionne sur une architecture client/serveur. En général le client tourne sur la même machine que le serveur mais peut être aussi bien lancé sur un autre ordinateur faisant partie du réseau.

Architecture client-serveur

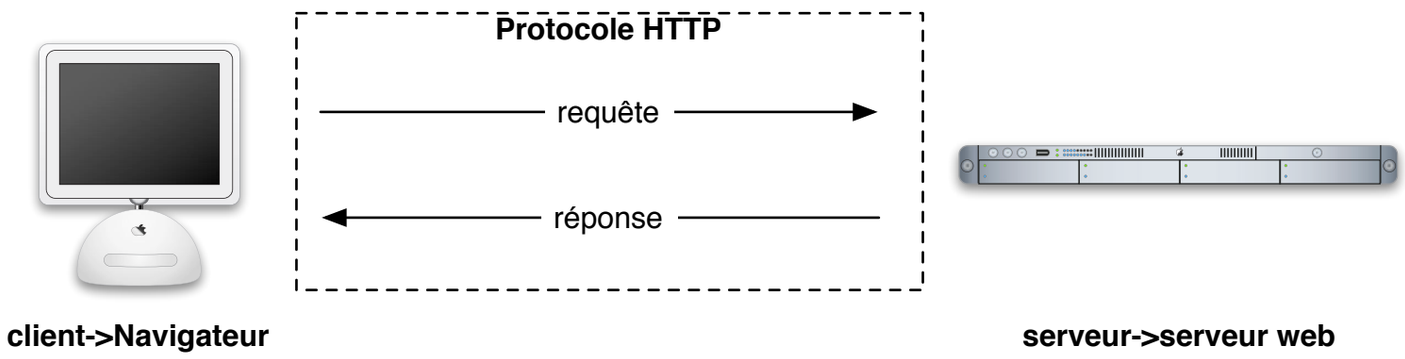
Exemples: le web

- A ne pas confondre avec “internet”, le web est un de ses services constitutifs et historique dans le but est d’afficher de l’information sous forme de pages navigables par des hyper-liens.

- **Le web repose sur un protocole client/serveur: HTTP**

- **Client :** le navigateur (Firefox, Explorer, Opera, Safari,...) va émettre une requête

- **Serveur :** le serveur Web (Apache, IIS,...) répond en fournissant le document demandé ou un message d’erreur si le document n’existe pas



Architecture client-serveur

Couches fonctionnelles

- Toute application client-serveur est composée de 3 “couches” fonctionnelles:
  - **Présentation** : correspondant à l'affichage, la restitution sur le poste de travail, le dialogue avec l'utilisateur (= interface utilisateur).
  - **Traitements métiers** : correspondant à la mise en œuvre de l'ensemble des règles de gestion et de la logique applicative.
  - **Persistance de données**: correspondant aux données qui sont destinées à être conservées sur la durée, voire de manière définitive.
- C’est la répartition de ces couches entre les rôles de clients et serveurs qui permet de distinguer entre les différents types d’architectures client-serveur.

Couches fonctionnelles

Couche Présentation

- Elle correspond à la partie de l'application visible et interactive avec les utilisateurs. On parle d'Interface Homme Machine.

- On conçoit facilement que cette interface peut prendre de multiples facettes sans changer la finalité de l'application. Dans le cas d'un système de distributeurs de billets, l'automate peut être différent d'une banque à l'autre, mais les fonctionnalités offertes sont similaires et les services identiques (fournir des billets, donner un extrait de compte, etc.).

- La couche présentation relaie les requêtes de l'utilisateur à destination de la couche métier, et en retour lui présente les informations renvoyées par les traitements de cette couche. Il s'agit donc ici d'un assemblage de services métiers et applicatifs offerts par la couche inférieure.

Couches fonctionnelles

Couche Traitements métiers

- Elle correspond à la partie fonctionnelle de l'application, celle qui implémente la “logique”, et qui décrit les opérations que l'application opère sur les données en fonction des requêtes des utilisateurs, effectuées au travers de la couche présentation.

- Les différentes règles de gestion et de contrôle du système sont mises en œuvre dans cette couche.

- La couche métier offre des services applicatifs et métiers à la couche présentation. Pour fournir ces services, elle s'appuie, sur les données du système, accessibles au travers des services de la couche inférieure. En retour, elle renvoie à la couche présentation les résultats qu'elle a calculés.

Couches fonctionnelles

Couche Persistance de données

- Elle consiste en la partie gérant l'accès aux sources de données du système. Ces données peuvent être propres au système, ou gérées par un autre système. La couche métier n'a pas à s'adapter à ces deux cas, ils sont transparents pour elle, et elle accède aux données de manière uniforme.

  - **Données propres au système:** ces données sont pérennes, car destinées à durer dans le temps, de manière plus ou moins longue, voire définitive. Elles peuvent être stockées indifféremment dans de simples fichiers texte, XML, ou encore dans une base de données.

  - **Données gérées par un autre système:** elles ne sont pas stockées par le système considéré, il s'appuie sur la capacité d'un autre système à fournir ces informations.

    Par exemple, une application de pilotage de l'entreprise peut ne pas sauvegarder des données comptables de haut niveau dont elle a besoin, mais les demander à une application de comptabilité indépendante et pré-existante.

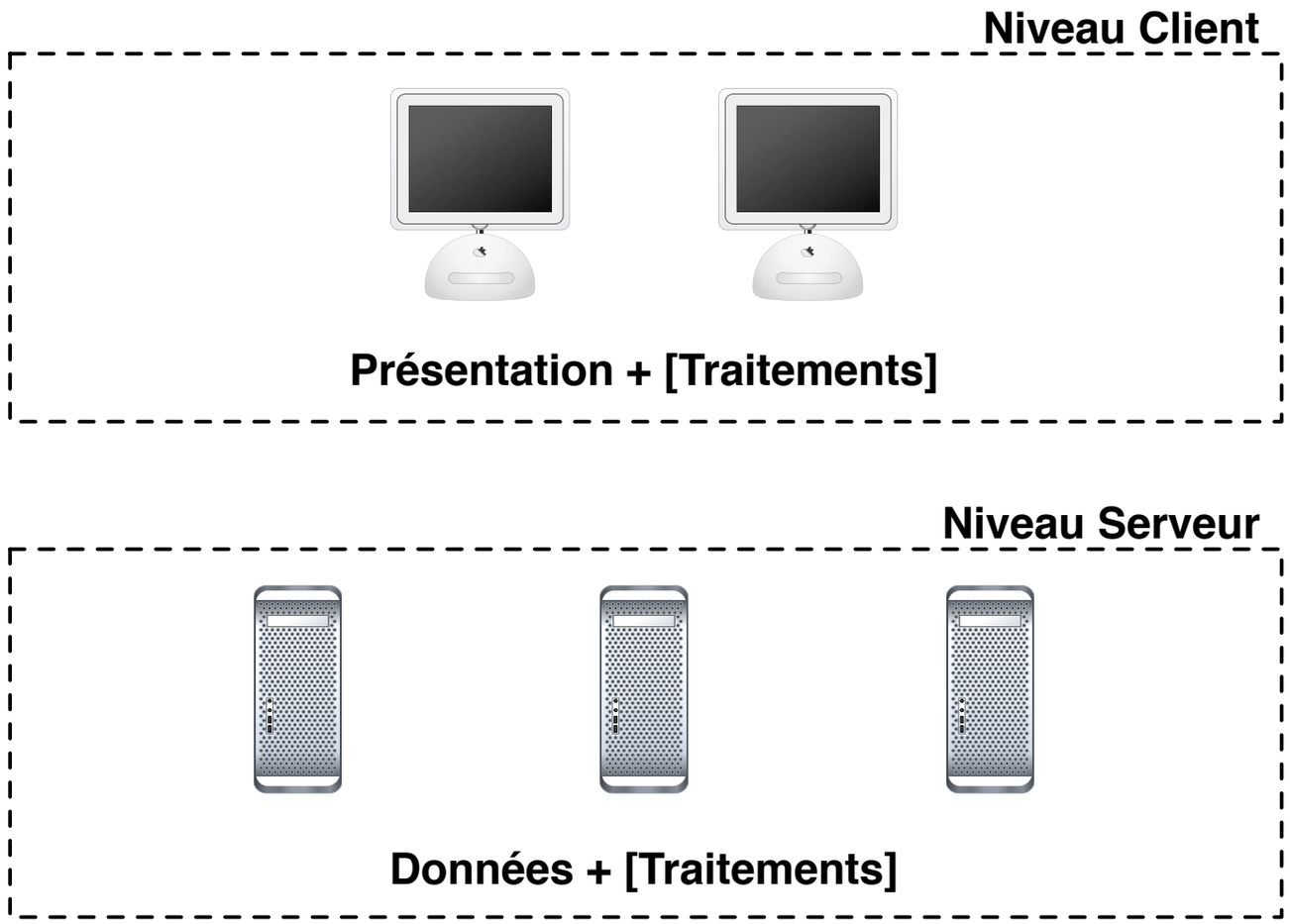
Architecture client-serveur

2-tiers

- C’est le type d’architecture client-serveur le plus basique :
  - La couche présentation est située sur le client
  - La couche donnée est située sur le serveur (par ex. une base de données relationnelles SQL, Oracle,...)
  - La couche traitement peut être située sur le client (au sein même de l’IHM), le serveur (par ex. des procédures stockées sur la base de données), ou partagée entre les deux rôles.
- Il y a une relation directe entre les clients et les serveurs de données.
- (+) simple à mettre en oeuvre
- (-) peu flexible et supporte difficilement la montée en charge
- (-) souvent des solutions propriétaires, économiquement très couteux

Architecture client-serveur

2-tiers



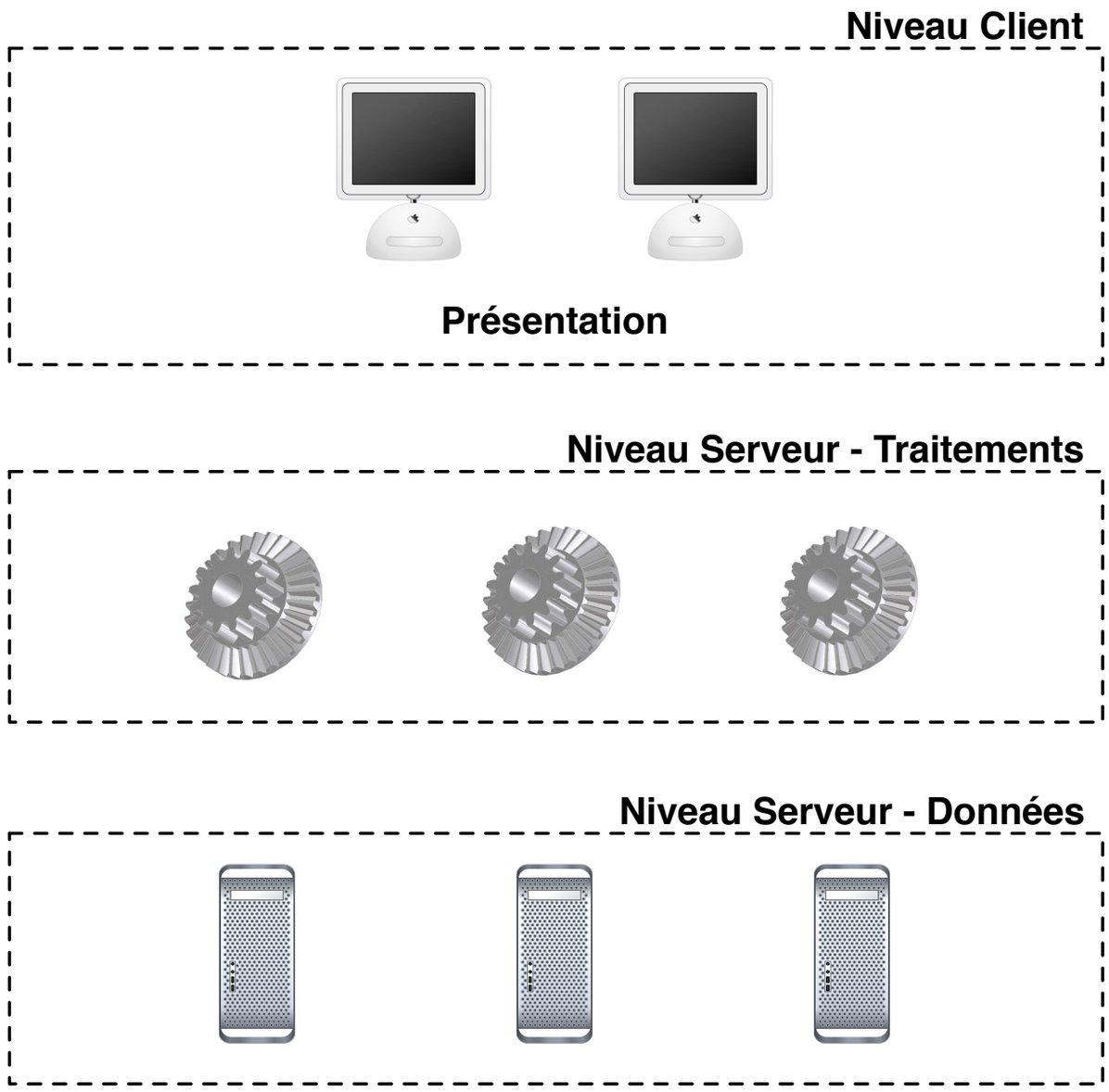
Architecture client-serveur

3-tiers

- L’architecture 3-tiers est une extension du modèle client-serveur qui introduit un rôle spécifique pour la couche de traitements métiers.
- Il y donc décomposition d’un même service sur 2 types de serveurs:
    - Un type de serveur dédié à la gestion des traitements métiers.
    - Un type de serveur dédié à la gestion des données persistantes.
- (+) plus évolutif que le 2-tiers
- (+) meilleure répartition des charges de travail coté serveur
- (+) économiquement moins cher, surtout lors de la montée en charge
- (-) administration et mise en oeuvre plus compliquée que le 2-tiers

Architecture client-serveur

3-tiers



Architecture client-serveur

3-tiers

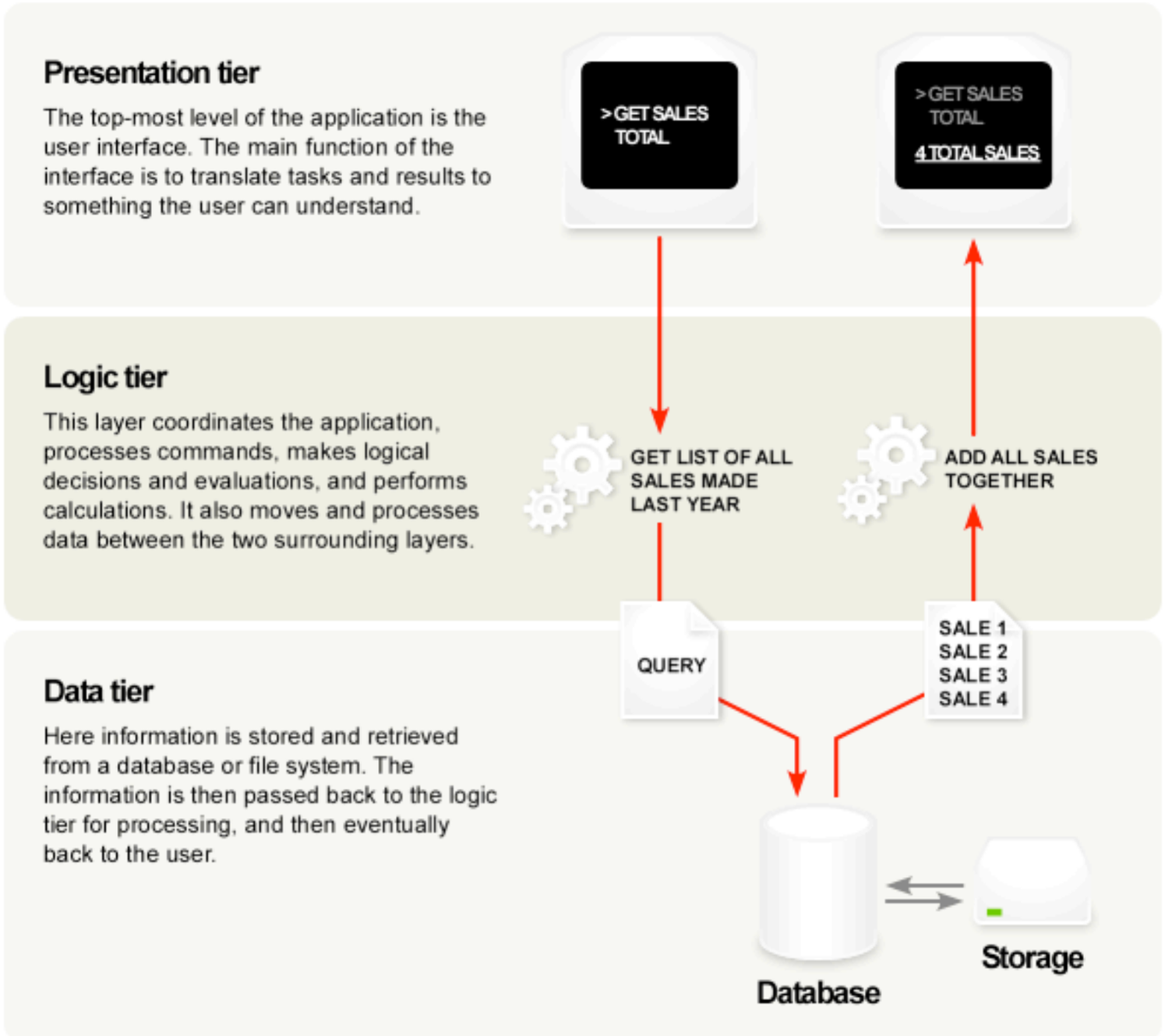


Image from wikipedia

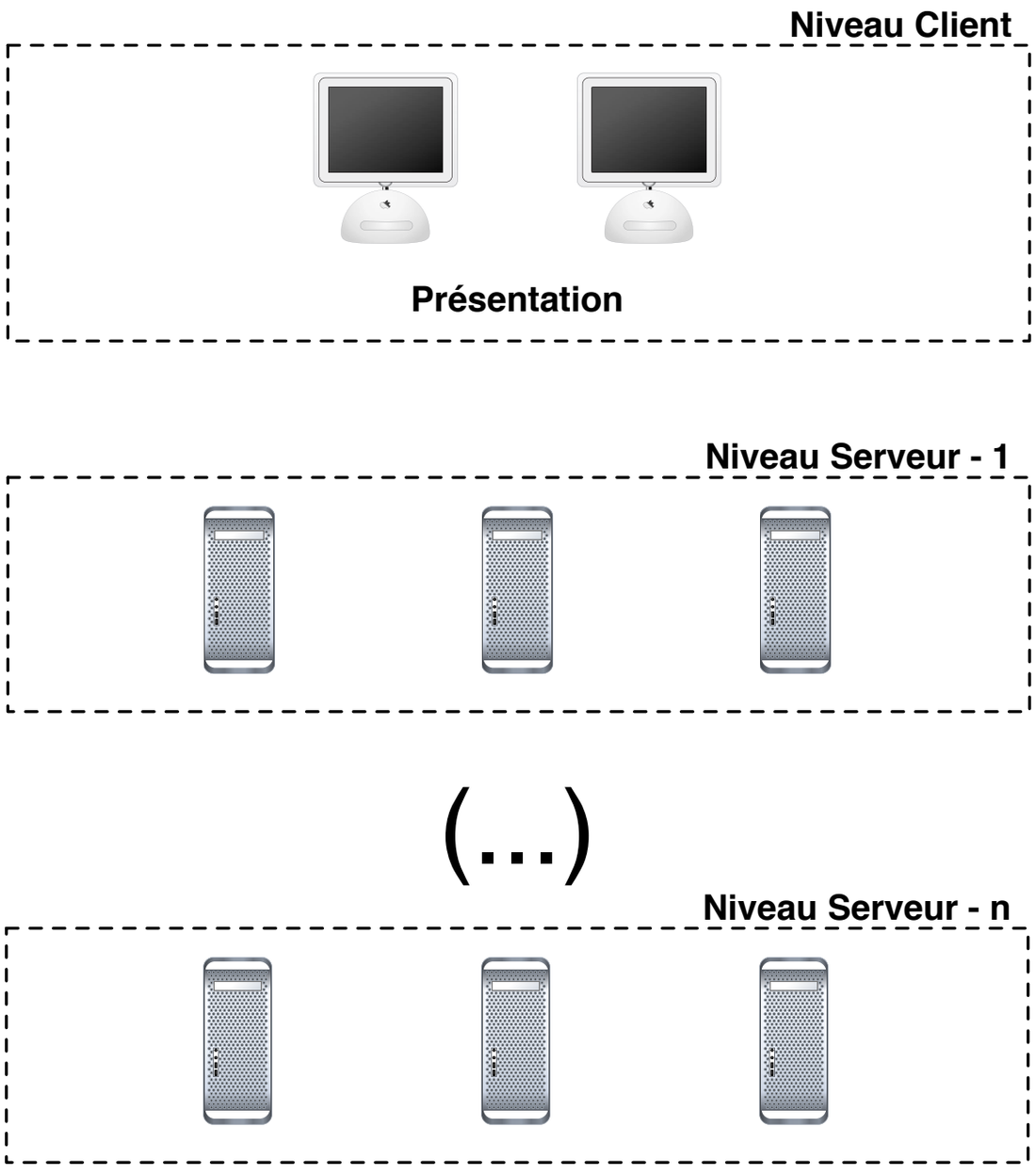
Architecture client-serveur

n-tiers

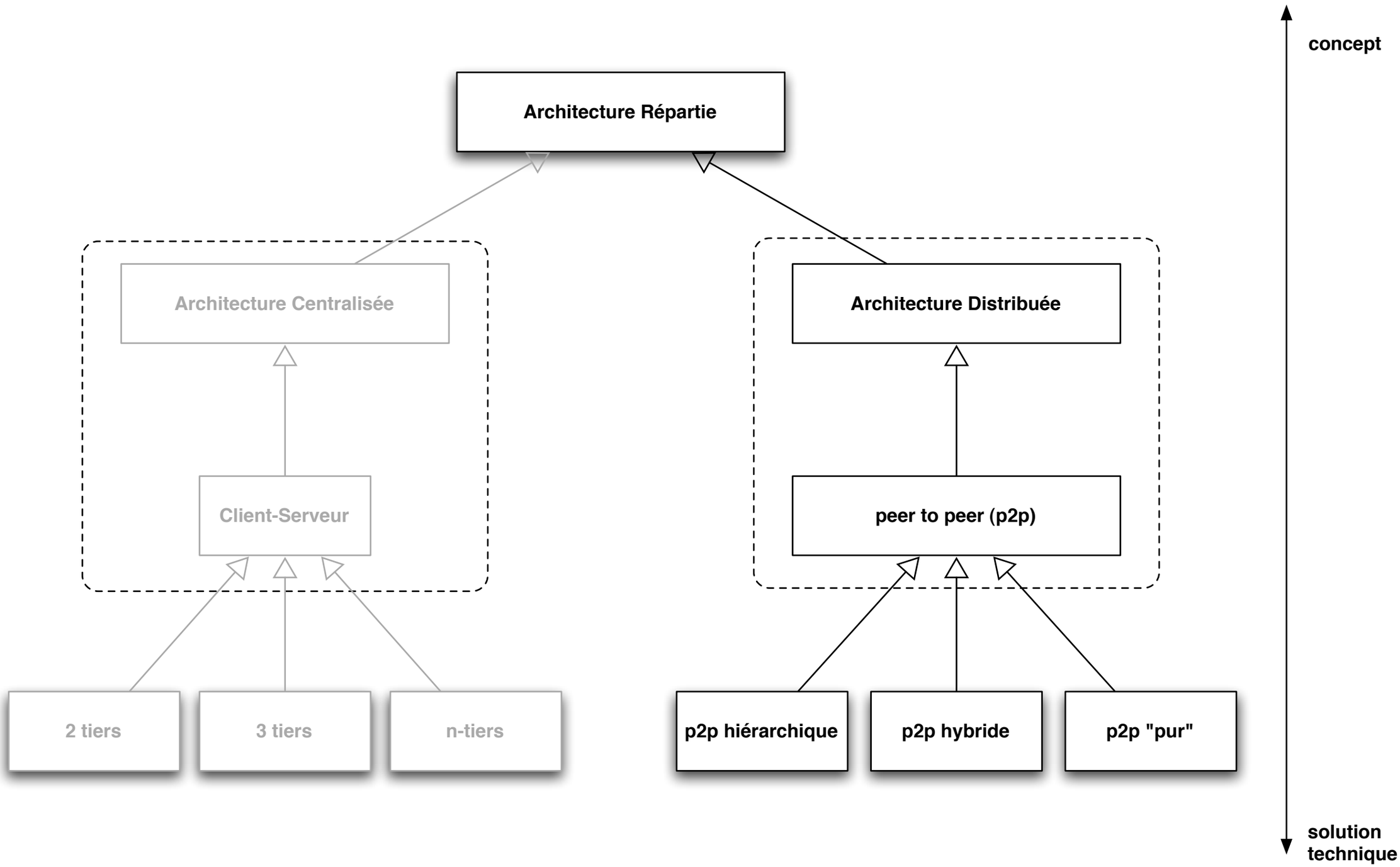
- Généralisation de l’architecture 3-tiers
- La couche serveur peut être divisée en autant de sous-rôles que voulus
- Sensiblement les mêmes avantages et inconvénients que la couche 3-tiers.

Architecture client-serveur

n-tiers



Architectures distribuées



Architectures distribuées

- Dans le cadre des architectures réparties, on distingue :
  - **Les architectures centralisées** (type client-serveur) que nous venons de parcourir.
  - **Les architectures distribuées** où les problématiques sont différentes de part l'absence d'un décideur.

- La différence fondamentale entre ces deux type d'architecture se situe dans la **distribution des rôles** entre les noeuds du réseau :
  - On parle d'architecture centralisée car les noeuds jouant le rôle de serveur centralisent une grande partie de l'information et des communications (ex. n clients pour 1 serveur) et sont **distincts** des noeuds clients.
  - A contrario, dans les architectures distribuées, tous les noeuds du réseau partagent théoriquement les mêmes rôles : ils sont simultanément clients et serveurs et disposent de tout ou partie de l'information répartie.

Architectures distribuées

- Dans les architectures distribuées les noeuds du réseau sont en relation d'égal à égal.
- **Les caractéristique habituelles** de ces architectures sont les suivantes :
  - Pas de connaissance globale du réseau.
  - Pas de coordination globale des noeuds.
  - Chaque noeud ne connaît que les noeuds constituant son voisinage.
  - Toutes les données sont accessibles à partir de n'importe quel noeud.
  - Les noeuds sont très volatiles (ils peuvent disparaître ou apparaître à tout moment).

Architectures distribuées

- Les technologies Peer-to-Peer (P2P, ou Pair-à-Pair) constituent le pendant technologique des architectures distribuées.
- La parfaite homogénéité des rôles vue précédemment correspond rarement à la réalité des réseau P2P déployés de part le monde. De ce fait, on procède à leur classification en différentes grandes familles (cf. transparent suivant).

Architecture centralisée
(client-serveur en étoile)

Architecture distribuée
(Peer-to-Peer pur)

Architectures distribuées

- **P2P “pur”** (par ex. Gnutella) :
  - Respecte les caractéristiques précédentes :
    - Les pairs sont égaux et fusionnent les rôles de client et de serveur.
    - Il n’y a pas de serveur central pour gérer le réseau.
    - Il n’y a pas de routeur central.
- **P2P hybride** (par ex. Napster) :
  - Données distribuées mais index centralisé :
    - On dispose d’un serveur central qui conserve des informations sur les pairs et peut répondre à des requêtes sur cette information (il joue souvent le rôle d’un annuaire de clients et de fichiers).
    - Les pairs sont responsables de l’hébergement des ressources partagées mais doivent indiquer leur disponibilité au serveur central.

Architectures distribuées

- **P2P Structuré** (Chord, P-Grid, CAN, …) :
  - Index distribué et stocké par DHT (Distributed Hash Tables).
- **P2P Hiérarchique** (Super-Peer, Kazaa,…) :
  - Couplage entre les architectures Client-Serveur et le P2P.
- **P2P Sémantique** (SON, Routing Indices, …) :
  - P2P Pur avec routage enrichit de critères sémantiques.

Architectures distribuées

Avantages - Inconvénients

- **Avantages :**
  - Tous les pairs fournissent des ressources (bande passante, stockage, puissance de calcul,...). On obtient ainsi **des architectures qui supportent beaucoup mieux la montée en charge** (“scalability”) que les architectures centralisées.
  - La distribution augmente la robustesse du réseau dans le cas d’une panne par la réplication des données sur plusieurs pairs. Et, dans le cas d’un P2P pur, il n’y pas de point central de vulnérabilité (l’annuaire).
- **Inconvénients :**
  - Mauvaise réputation du “P2P”, invariablement associé au téléchargement musical → faible adoption par l’industrie.
  - Les architectures distribuées amènent leur lot de problématiques spécifiques (concurrence entre les pairs, fragmentation des données, ...).
  - Elles ne permettent pas un contrôle avancé des échanges d’information entre les pairs (inconvénient ou avantage ?).

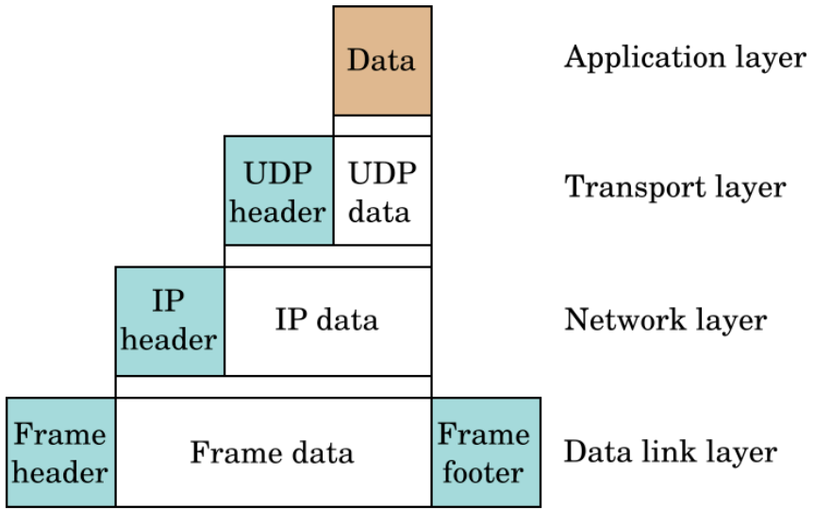
Architectures réparties

Java pour la programmation C-S

Chap #4.2

Protocoles de transport

- Nous avons précédemment présentés différents paradigmes de communication pour les protocoles de niveau applicatif: passage de messages, appel de procédures à distances, ...
- Les protocoles de niveau applicatif reposent sur les protocoles de transport réseaux.
- Le fonctionnement d'un protocole de niveau applicatif est indépendant du protocole de transport (dans une certaines mesure).
- Lui même s'appuie sur des protocoles de niveau inférieur pour le routage, le transfert noeud à noeud, ...
- → On parle de pile des protocoles :



Protocoles de transport

- Dans ce cours nous allons aborder l'implémentation standard Java des protocoles de transport suivants:
  - **TCP**: permet l'utilisation de flux bi-directionnels de communication
  - **UDP**: permet l'envoi asynchrone de messages
  - **Multicast-IP**: permet l'envoi de message à un groupe de destinataires
- On dispose alors d'APIs distinctes pour pouvoir envoyer et recevoir des données pour chacun de ces protocoles de transport.
- **Manipulation de ces API en TP !**

Protocoles de transport

- L'utilisation de ce niveau primitif de communication permet la communication dans les architectures simples de type client-serveur 2-tiers.
- Mais il n'est pas recommandé de s'appuyer directement sur cette couche réseau pour l'implémentation d'une application répartie plus complexe.
- En effet il est nécessaire de :

1. Définir le format des messages réseau.
2. Localiser le serveur.
3. Emballer ("marshall") les informations émises par le client pour le réseau.
4. Déballer ("unmarshall") les informatiions émises par le réseau pour le serveur.
5. Gérer la requête.
6. Emballer/déballer la/les valeur(s) de retour pour le client.

Protocoles de transport

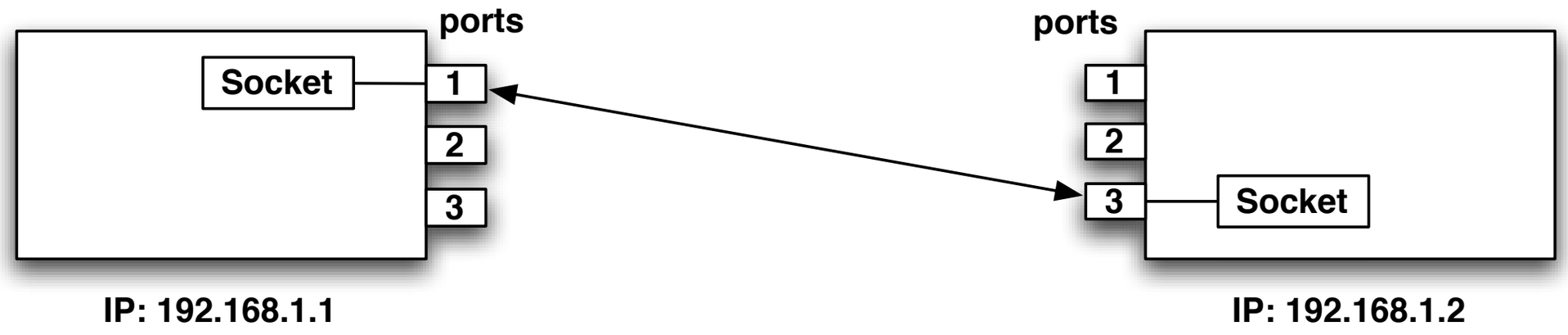
- Pour chacun de ces protocoles, on dispose de deux **primitives de communication** :
  - **send** :
    envoi d'un message dans un buffer (zone tampon) distant
  - **receive** :
    lecture d'un message à partir d'un buffer local
- Différentes propriétés sont associées à ces primitives et changent en fonction du protocole de transport retenu :
  - **Fiabilité** : est-ce que les messages sont garantis sans erreurs ?
  - **Ordre** : est-ce que les messages arrivent dans le même ordre que celui de leur émission ?
  - **Contrôle de flux** : est-ce que la vitesse d'émission est contrôlée
  - **Connexion** : les échanges de données sont ils organisés en connexions ?

Protocoles de transport

- De plus, on distingue deux modes de fonctionnement pour ces primitives :
  - **synchrone** : les primitives sont bloquantes
  - **asynchrone** : les primitives sont non-bloquantes
- Par exemple :
  - un *send* synchrone va rester bloqué jusqu'à l'envoi complet du message,
  - de même un *receive* synchrone va rester bloqué jusqu'à ce qu'il y ait un message à lire.
- L'utilisation de primitives asynchrones est plus souple que celui de primitives synchrones.
- Mais il est plus simple d'implémenter un programme avec des primitives synchrones qu'asynchrones (pas besoin de gérer toutes les problématiques de concurrence).

Adressage

- Que ce soit on TCP ou en UDP, on dispose de la notion de socket.
- Une socket correspond à l'abstraction des extrémités servant à communiquer.
- Chaque socket est associé à une adresse IP et un # de port local.
  - 1 récepteur pour chaque couple (ip, port)
  - Eventuellement plusieurs émetteurs vers le même couple (ip, port)



Adressage

- **Notion d'adresse réseau**: classe `java.net.InetAddress`
  Cette classe est utilisée par les API TCP/UDP.
  - `static InetAddress getByName(String host)`
    Determines the IP address of a host, given the host's name.
  - `static InetAddress[] getAllByName(String host)`
    Given the name of a host, returns an array of its IP addresses, based on the configured name service on the system.
- `static InetAddress getLocalHost()`
  Returns the local host.
- `String getHostName()`
  Gets the host name for this IP address.
- `String getHostAddress()`
  Returns the IP address string in textual presentation.
- (...)

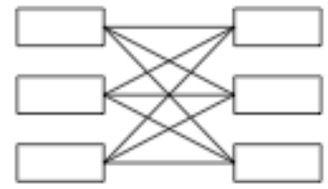
Connexion

- **Problématique**: les communications entre un client et un serveur sont elles précédées d'une ouverture de connexion (entre le client et le serveur) et suivies d'une fermeture ?
  - **Mode non connecté** (le plus simple): NON
    → les messages sont envoyés librement
  - **Mode connecté**: OUI
    → facilite la gestion d'état et fournis un meilleur contrôle des clients (on peut suivre les arrivées et les départs).

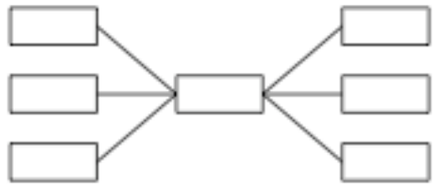
	Niveau Transport	**Niveau Applicatif**
Connecté	TCP	FTP, Telnet, SMTP, POP, JDBC
Non connecté	UDP	NFS, DNS, TFTP

Représentation des données

- **Problématique**: communication entre machines avec des formats de représentation de données différents.
  - Pas le même codage (big endian vs little endian)
  - Pas la même façon de stocker les types (entiers 32 bits vs 64 bits, ...)
- **Solutions :**
  - On prévoit tous les cas de conversion possibles
    → n^2 convertisseurs à écrire



  - On prévoit un format pivot et on effectue 2 conversions pour chaque formats de représentation (vers le format pivot, depuis le format pivot)
    → 2*n convertisseurs à écrire



- Il existe déjà de nombreux formats pivots: ASN.1, Sun XDR, **Sérialisation Java**, CORBA CDR, ...

TCP
Propriétés

- Taille des messages
  - Quelconque.
  - Envoi en général bufférisé.
  - Vidage explicite des tampons possible (dépend de l'OS).
- **Fiabilité** - perte de messages
  - Acquittements (masqués au programmeur) des messages envoyés.
  - Timeout de ré-émission des messages en cas de non réception de l'**acquittement** (mécanisme de fenêtre).
- **Contrôle de flux**
  - Eviter qu'un émetteur trop rapide fasse "déborder" le buffer du récepteur.
  - Blocage de l'émetteur si nécessaire.
- **Ordre des messages**
  - Garantie que *ordre d'émission = ordre de réception*.
  - Garantie de non duplication des messages.

TCP
Propriétés

- **Connexions TCP**
  - Demande d'ouverture par un client.
  - Acception explicite de la demande par le serveur :
    - Ensuite échange en mode bi-directionnel.
    - Ensuite distinction rôle client/serveur “artificielle” → toujours valable au niveau applicatif !
  - Fermeture de connexion à l'initiative du client ou du serveur.
    Vis-à-vis notifié de la fermeture.
- Pas de mécanisme de gestion de panne :
  trop de pertes de messages ou réseau trop encombré → connexion perdue.
- Utilisation de TCP par de nombreux protocoles applicatifs tels que HTTP, FTP, Telnet, SMTP, POP, ...

TCP

Fonctionnement

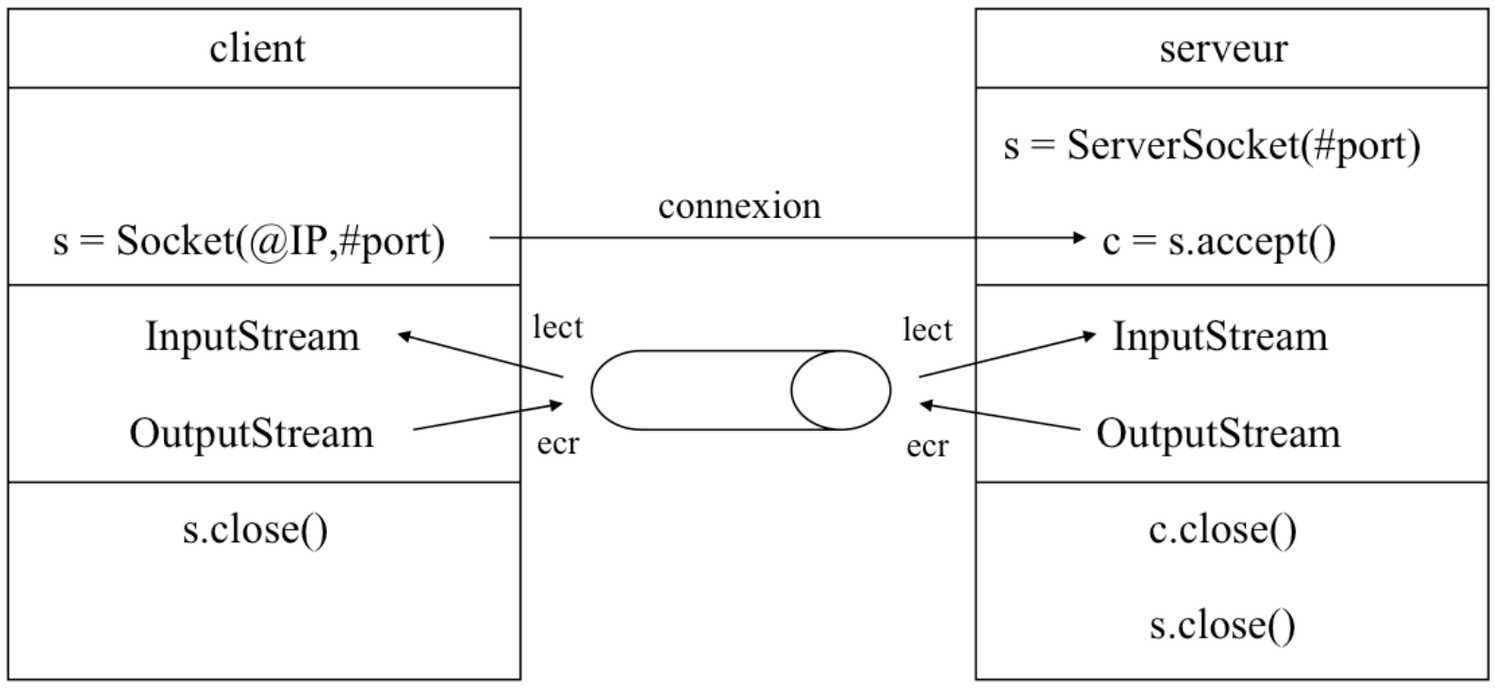
1.Le serveur crée une socket et attend une demande de connexion

2.Le client envoie une demande de connexion

3.Le serveur accepte la connexion

4.Etablissement du dialogue entre client et serveur en mode flux

5.Fermeture de connexion à l'initiative du client ou du serveur



TCP
API Java

- **Cotés client et serveur :** Classe `java.net.Socket`
  - Constructeur prend en argument l'adresse du serveur et son # de port
    `Socket(InetAddress address, int port)`
    "Creates a stream socket and connects it to the specified port number at the specified IP address."
  - Méthodes principales :
    - adresse IP : `InetAddress getInetAddress(), getLocalAddress()`
    - port : `int getPort(), getLocalPort()`
    - **flux in** : `InputStream getInputStream()`
    - **flux out** : `OutputStream getOutputStream()`
    - fermeture : `close()`

TCP

API Java

- **Cotés client et serveur :** Classe `java.net.Socket`
    - `public void setSoTimeout(int timeout)`
      Permet de spécifier le délais (en milli-secondes) à partir duquel le blocage sur la méthode `read()` du flux de donné associé à cette socket est levé. (timeout = 0 → attente infinie)

      Passé ce délai, exception SocketTimeoutException levée (mais la socket reste opérationnelle).

- Un ensemble d'options TCP:
    - *get/set*SoTimeout*()* (cf. si dessus)
    - *get/set*SoLinger*()*
    - *get/set*TcpNoDelay*()*
    - *get/set*KeepAlive*()*

TCP
API Java

- **Coté serveur :** Classe `java.net.ServerSocket`
  - Constructeur prend en argument le # de port servis
    `ServerSocket(int port)`
    “Creates a server socket, bound to the specified port.”
  - Méthodes principales :
    - adresse IP : `InetAddress getInetAddress()`
    - port : `int getLocalPort()`
    - **attente de connexion :** `Socket accept()`
      Nota: Une fois la connexion d’un client acceptée, la méthode `accept()` renvoie une instance de `Socket`. On travaille ensuite avec cette instance.
    - fermeture : `void close()`

TCP
API Java

- **Coté serveur :** Classe `java.net.ServerSocket`
    - `public void setSoTimeout(int timeout)`
      Permet de spécifier le délais (en milli-secondes) à partir duquel le blocage sur la méthode `accept()` d'attente de connextion est levé. (timeout = 0 → attente infinie)

      Passé ce délai, exception SocketTimeoutException levée (mais la socket reste opérationnelle).

- Un ensemble d'options TCP:
    - *get/set*SoTimeout*()*
    - *get/set*ReceiveBufferSize*()*
    - *get/set*ReuseAddress*()*

TCP
API Java

- **Coté serveur :** Classe `java.net.ServerSocket`
  - Méthode `accept()` bloquante par défaut, ce n'est pas forcement un inconvénient : beaucoup de serveurs implémentent un schéma *dispatcheur* :
    1. Une thread "dispatcheuse" écoute sur un port donné et ne fait que ça
    2. Dès qu'une connexion arrive, le travail est délégué à une autre thread
    3. La thread "dispatcheuse" se remet en attente sur `accept()`. (go to 1.)

2. Il existe actuellement deux solutions pour obtenir une méthode `accept()` non-bloquante:
   - `java.net.ServerSocket: void setSoTimeout(int timeout)`
   - `java.net.ServerSocket: public ServerSocketChannel getChannel()`

UDP
Propriétés

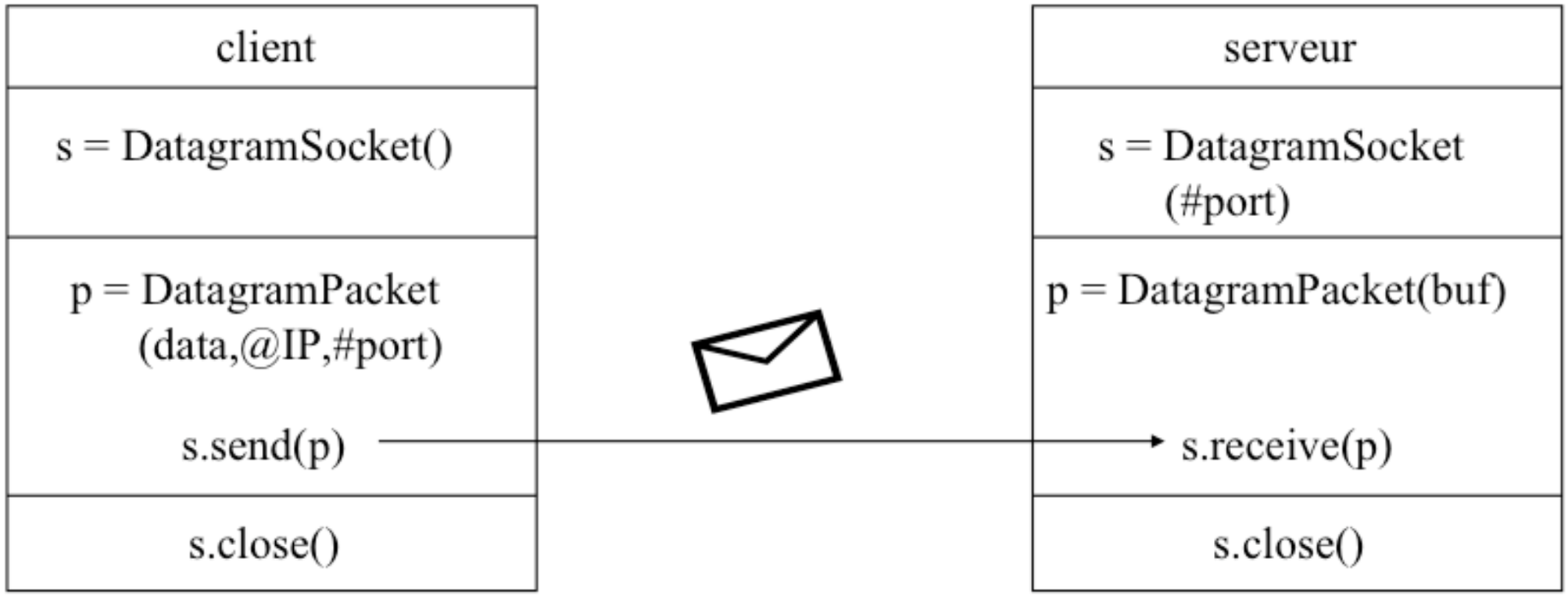
- Taille des messages
  - limitée par l'implantation d'IP sous-jacente (en général 64 K)
- **Fiabilité** - Perte de messages
  - possible
  - Nota: pas problématique pour certaines applications (streaming audio, vidéo, ...)
- **Contrôle de flux** : non
- **Ordre des messages** : non garanti
- **Connexion** : non

- **Avantage** : plus performant que TCP
- **Inconvénient** : moins de qualité de service (peut être masqué au niveau du protocole applicatif, la charge est reportée sur le niveau supérieur).
- Utilisation : NFS, DNS, TFTP, streaming, jeux en réseau.

UDP

Fonctionnement

1. Le serveur crée une socket UDP.
2. Le serveur attend la réception d'un message.
3. Le client envoie un message.



UDP
API Java

- **Cotés client et serveur :** Classe `java.net.DatagramSocket`
  - Constructeur prend en argument l'adresse du serveur [et son # de port]
    `DatagramSocket()`
    `DatagramSocket(int port)`
  - Méthodes principales :
    - envoi de données : `void send(DatagramPacket p)`
    - réception de données : `void receive(DatagramPacket p)`
- Options UDP : get/setTimeout(), Broadcast, ReceiveBufferSize, ...
- Mêmes solutions pour `receive()` non bloquant qu'avec les sockets TCP.
- Nota: possibilité de simuler une connexion à un couple (ip, port) via la méthode `public void connect(InetAddress address, int port)`
  → pas une réelle connexion, juste un contrôle pour restreindre les send/receive.

UDP

API Java

- **Cotés client et serveur :** Classe `java.net.DatagramPacket`
  - Cette classe effectue l'encapsulation des données à envoyer ou recevoir via les méthodes `receive()`/`send()`.
  - Une instance de DatagramPacket contient les informations suivantes :
    - Données à envoyer.
    - Longueur des données contenues.
    - Adresse IP de l'hôte distant.
    - Le port utilisé sur l'hôte distant.
- Méthodes principales associées aux informations ci dessus : get/setPort(), get/setAddress(), get/setData(), get/setLenght()

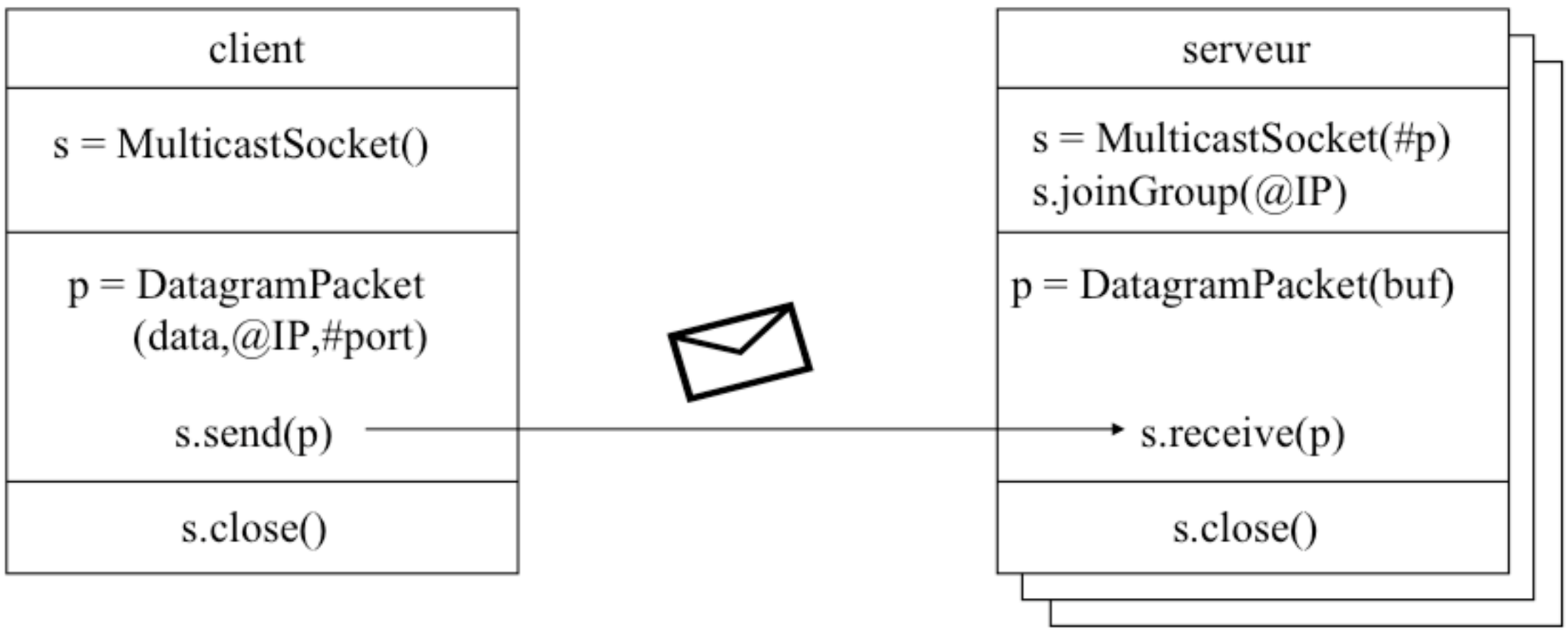
Multicast IP
Propriétés

- Le multicast IP sert à diffuser des messages vers un groupe de destinataires :
  - Les messages sont émis vers une adresse de classe D (224.0.0.1 à 239.255.255.255) indépendante de la localisation physique des émetteurs et récepteurs.
  - Les messages sont alors recus par tous les récepteurs “écoutant” sur cette adresse.
    - Plusieurs émetteurs/récepteurs possibles vers/sur la même adresse.
    - Les récepteurs peuvent quitter ou rejoindre un groupe à tout instant.
- **Il dispose des mêmes propriétés que UDP.**

Multicast IP

Fonctionnement

1. Le serveur crée une socket MulticastIP.
2. Le serveur rejoint le groupe de diffusion.
3. Un client envoie un message.



Multicast IP

API Java

- **Cotés client et serveur :** Classe `java.net.MulticastSocket`
  - Constructeur avec un # de port précis ou quelconque.
    `MulticastSocket()`
    “Create a multicast socket”
    `MulticastSocket(int port)`
    “Create a multicast socket and bind it to a specific port”
- Méthodes principales :
  - envoi de données : `void send(DatagramPacket p)`
  - réception de données : `void receive(DatagramPacket p)`
  - joindre un groupe : `void joinGroup(InetAddress mcastaddr)`
  - quitter un groupe : `void leaveGroup(InetAddress mcastaddr)`

Entrées-Sorties

- Nous avons vu dans les transparents précédents que l'implémentation Java des **sockets TCP utilise la notion de flux** (Input/Output stream) pour la lecture et l'écriture des données.

- Ce chapitre constitue une référence rapide pour les entrées/sorties (I/O) Java.

- Les I/O Java permettent de lire et d'écrire des données à partir de fichiers, de la mémoire, du réseau, ...

- Elles sont gérées à l'aide des classes du package `java.io`

- Au niveau de l'API, on effectue la distinction entre :

  - **Les entrées/sorties en mode binaire.**

  - **Les entrées/sorties en mode caractère.**

Entrées-Sorties binaires

- En mode binaire, on dispose des classes **abstraites** `java.io.InputStream` et java.io.`OutputStream` :

	InputStream	**OutputStream**
Rôle	Lecture de données	Ecriture de données
Méthode principale	Définie une méthode abstraite pour la lecture: `abstract int read()` - lecture d'un octet - **bloquant** - retourne -1 si fin du flux	Définie une méthode abstraite pour l'écriture: `abstract void write(int b)` - écriture d'un octet - close() pour fermer le flux
Sous classes possibles (types concrets)	AudioInputStream, ByteArrayInputStream, **FileInputStream**, FilterInputStream, InputStream, ObjectInputStream, PipedInputStream, SequenceInputStream, StringBufferInputStream, **BufferedInputStream**, ...	ByteArrayOutputStream, **FileOutputStream**, FilterOutputStream, ObjectOutputStream, OutputStream, PipedOutputStream, **BufferedOutputStream**, ...

Entrées-Sorties binaires

Exemples d'utilisation

- Utilisation de InputStream :

```
InputStream is = //instanciation d'un type concret
int i = is.read();
while ( i != -1 ) {
   byte b = (byte) i;
   ...
   i = is.read();
}
is.close
```

- Utilisation de OutputStream :

```
OutputStream os = //instanciation d'un type concret
os.write(123);
os.write(456);
...
is.close();
```

Entrées-Sorties binaires

InputStream

- On dispose d'autres méthodes pour lire des données :
  - `int read(byte[] b)`
    "Reads some number of bytes from the input stream and stores them into the buffer array b."
  - `int read(byte[] b, int off, int len)`
    "Reads up to len bytes of data from the input stream into an array of bytes."
  - Le tableau doit être alloué avant l'appel à la méthode `read(...)`.
  - Lecture de au maximum `length` octets, écrits à partir de `offset` dans `b`.
  - **Lecture bloquante.**
  - Les méthodes retournent le nombre d'octets lus.
- Plus encore d'autres méthodes : skip(), available(), close(), ...

Entrées-Sorties binaires

OutputStream

- On dispose d'autres méthodes pour écrire des données :
  - `void write(byte[] b)`
    "Writes b.length bytes from the specified byte array to this output stream."
  - `void write(byte[] b, int off, int len)`
    "Writes len bytes from the specified byte array starting at offset off to this output stream."
    - Le tableau doit être alloué avant l'appel à la méthode `write(...)`.
    - Ecriture de `length` octets, écrits à partir de `offset` dans `b`.
- Plus encore d'autres méthodes : flush(), close()

Entrées-Sorties binaires

FileInputStream

- Cette sous-classes de InputStream permet la lecture de données **stockées dans un fichier** :
  - Elle fournit une implémentation pour la méthode read().
  - Elle fournit les mêmes méthodes que InputStream.

```
InputStream is = new FileInputStream(“/Users/pierre/Desktop/test.txt”);
int i = is.read();
while ( i != -1 ) {
   byte b = (byte) i;
   ...
   i = is.read();
}
is.close
```

- Principe similaire pour FileOutputStream.

Entrées-Sorties binaires

BufferedInputStream

- Cette sous-classes s'utilise après l'instanciation d'un InputStream. Elle permet la lecture **bufferisée** de données.
  - Elle permet d'effectuer des lectures par blocs (de taille fixe) de données qui seront ensuite stockées dans un buffer.
  - Cela permet d'éviter les lectures octet à octet sur le support de stockage qui sont potentiellement coûteuses (par ex. le délais de négociation et d'acheminement via un réseau).
  - Constructeurs:
    `BufferedInputStream(InputStream in)`
    `BufferedInputStream(InputStream in, int size)`
  - On dispose des même méthodes que sur InputStream() (← sous-classe) :

```
InputStream is = new FileInputStream("/Users/pierre/Desktop/test.txt");
BufferedInputStream bis = new BufferedInputStream(is);
int i = bis.read();
while ( i != -1 ) { ... }
```

- Principe similaire pour BufferedOutputStream.

Entrées-Sorties caractère

- En mode caractère, on dispose des classes **abstraites** `java.io.Reader` et java.io.`Writer` :

	Reader	**Writer**
Rôle	Lecture de données	Ecriture de données
Méthode principale	`int read()` - lecture d'un caractère - bloquant - retourne -1 si fin du flux	`abstract void write(int c)` - écriture d'un caractère (les 8 bits inférieurs du int) - close() pour fermer le flux
Sous classes possibles (types concrets)	**BufferedReader**, CharArrayReader, FilterReader, **InputStreamReader**, PipedReader, StringReader, **FileReader**, ...	**BufferedWriter**, CharArrayWriter, FilterWriter, **OutputStreamWriter**, PipedWriter, PrintWriter, StringWriter, **FileWriter**, ...

Entrées-Sorties caractère

Exemples d'utilisation

- Utilisation de Reader :

```
Reader r = //instanciation d'un type concret
int i = r.read();
while ( i != -1 ) {
   char c = (char) i;
   ...
   i = r.read();
}
r.close
```

- Utilisation de Writer :

```
Writer w = //instanciation d'un type concret
w.write(41);
w.write(42);
...
w.close();
```

Entrées-Sorties caractère

Reader

- On dispose d'autres méthodes pour lire des données :
  - `int read(char[] cbuf)`
    "Read characters into an array".
  - `abstract int read(char[] cbuf, int off, int len)`
    "Read characters into a portion of an array."
    - Le tableau doit être alloué avant l'appel à la méthode `read(...)`.
    - Lecture de au maximum `length` caractères, écrits à partir de `offset` dans `cbuf`.
    - **Lecture bloquante.**
    - Les méthodes retournent le nombre de caractères lus.
- Plus encore d'autres méthodes : skip(long n), close(), ...

Entrées-Sorties caractère

Writer

- On dispose d'autres méthodes pour écrire des données :
  - `void write(char[] cbuf)`
    Write an array of characters.
  - `abstract void write(char[] cbuf, int off, int len)`
    Write a portion of an array of characters.
  - `void write(String str)`
    Write a string.
  - `void write(String str, int off, int len)`
    Write a portion of a string.
  - Le tableau doit être alloué avant l'appel à la méthode `write(...)`.
  - Ecriture de `length` caractères, écrits à partir de `offset` dans `cbuf`.
- Plus encore d'autres méthodes : flush(), close()

Entrées-Sorties caractère

Lecture de données au clavier

- On souhaite pouvoir lire une ligne entière de **caractères**.
- Pour cela il faut avoir un flux de caractères **bufferisé**.
- On utilise l'entrée standard Java (`System.in` de type `InputStream`) qui est un flux **binaire** de données **non-bufferisé**.

1. Il faut faire passer System.in du mode binaire en mode caractère :
```
InputStreamReader isr = new InputStreamReader(System.in)
```

2. Il faut ensuite bufferiser le flux de caractères obtenu :
```
BufferedReader br = new BufferedReader(isr);
```

3. Puis lecture des données dans une String:
```
String s;
while( (s=br.readLine()) != null ) {
   ...
}
```

Bibliographie

- La meilleure source d'information à ce sujet: l'API Java
  http://java.sun.com/j2se/1.5.0/docs/api/

Fin du cours #5