Systèmes répartis

Fabrice Rossi

http://apiacoa.org/contact.html.

Université Paris-IX Dauphine

Systèmes répartis

- Définition très large : un système réparti est système informatique dans lequel les ressources ne sont pas centralisées
- Ressources au sens très large :
  - stockage (disques, bases de données)
  - puissance de calcul
  - utilisateurs
- But : permettre à des utilisateurs de manipuler (calcul) leurs données (stockage) sans contrainte sur les localisations respectives des éléments du système
- Généralisation et amélioration du schéma client/serveur :
  - serveurs multiples (équilibrage de charge, redondance)
  - systèmes multi-couches (*tiers*)
  - *peer to peer*
- réparti $\simeq$ distribué

Exemples

DNS	base de données repartie
annuaires X500	base de données repartie
Web	ordinateur réparti : données, traitement, etc
Systèmes multi-couches	architecture classique pour les systèmes mélangeant serveur web, base de données, traitement évolué, etc
peer to peer	système complètement décentralisé, en général pour le stockage réparti
Clusters	calcul réparti
Grid	super-ordinateur pour le calcul et le stockage réparti

Exemple : le DNS

Exemple typique d'un système réparti (1984) :

- DNS : Domain Name System
- le système qui permet de trouver l'adresse IP associée à un nom de machine
- exemple : `www.dauphine.fr` $\mapsto$ 193.49.168.65
- DNS : une base de données répartie

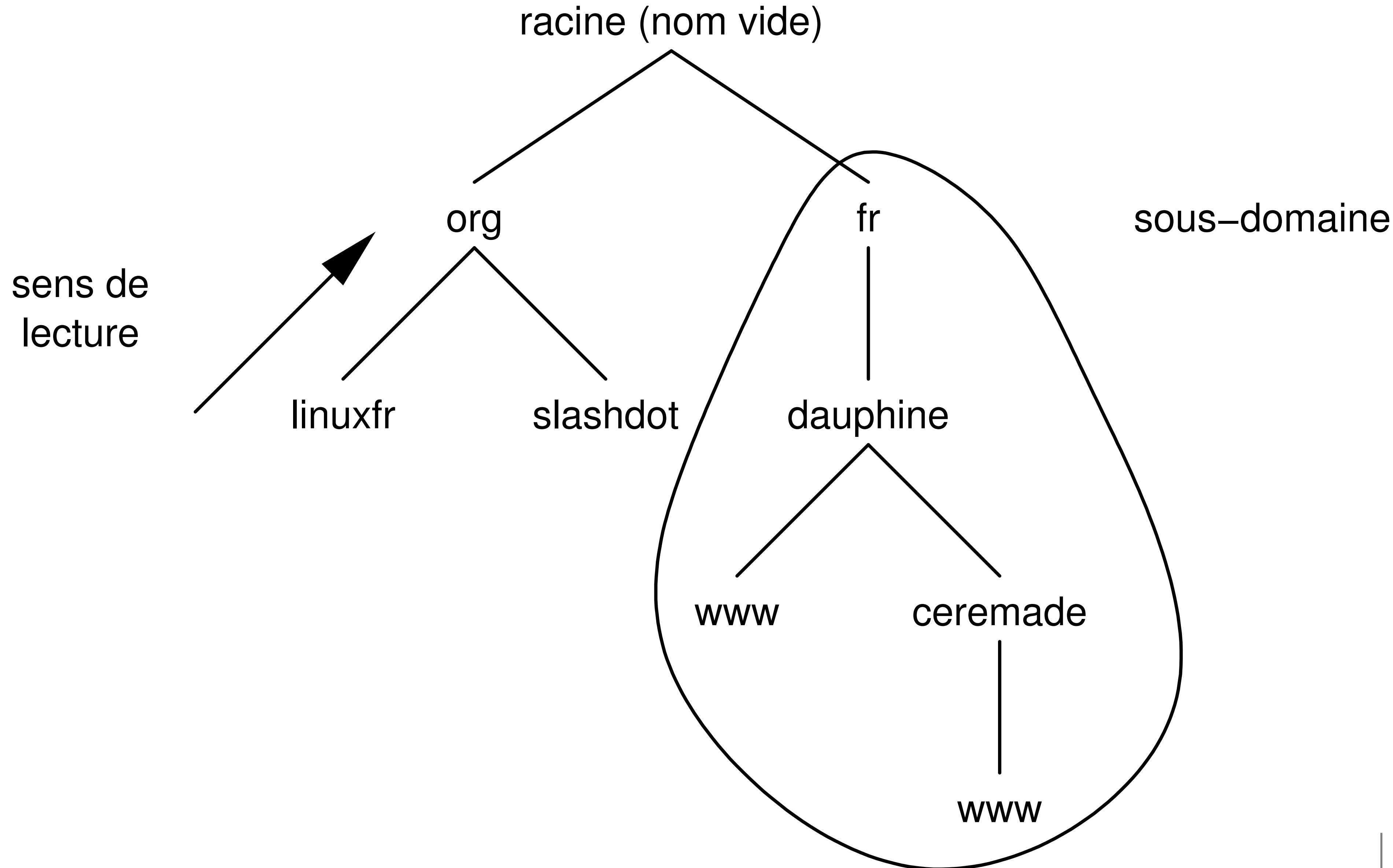
Rappels TCP/IP :

- Internet Protocol (IP) : couche réseau basée entièrement sur l'adresse IP (transmission directe au sein d'un sous-réseau, par routeur en dehors)
- Transmission Control Protocol (TCP) : couche transport qui assure l'arrivée des paquets
- une sorte de système réparti : pas de contrôle central, routes alternatives, etc.

DNS : Architecture logique

- système hiérarchique : arbre avec un nom par noeud
- le nom de la racine est vide
- chaque noeud est associé à des enregistrements (`A` pour l'adresse IP, `MX` pour le serveur de mail, etc.)
- domaine : sous-arbre
  - `dauphine` est un sous-domaine du domaine `fr`
  - `ufrmd` sous-domaine de `dauphine`
- nom de domaine : nom d'un noeud (`dauphine`)
- nom complet : suite des noms des noeuds en remontant dans l'arbre, séparés par `.` et sans prendre en compte la racine :
  - `www.dauphine.fr`
  - `www.ufrmd.dauphine.fr`

DNS : exemple



DNS : architecture physique

Tout est basé sur un système de délégation et de réplication :

- un serveur DNS gère un domaine
- chaque domaine est géré par aux moins 2 serveurs
- le protocole DNS propose un mécanisme de réplication (un serveur maître et des esclaves)
- le gestionnaire d'un domaine peut déléguer la gestion d'un sous-domaine à un autre gestionnaire :
  - `fr` délègue la gestion de `dauphine.fr` aux serveurs de dauphine
  - `dauphine.fr` délègue `ufrmd.dauphine.fr` aux serveurs de l'UFR MD, `ceremade.dauphine.fr` à ceux du CEREMADE, etc.
- requêtes :
  - système de cache local (*TTL*)
  - délégation de requête

DNS : traitement d'une requête

Je veux me connecter à `cvs-etud.ufrmd.dauphine.fr` :

1. demande du serveur local à un *root server*
2. réponse du *root server* : il faut demander à un serveur `fr` (délégation)
3. `fr` $\longrightarrow$ `dauphine.fr`
4. `dauphine.fr` $\longrightarrow$ `ufrmd.dauphine.fr`
5. `ufrmd.dauphine.fr` $\longmapsto$ `193.51.89.162`

Dans le cache après cette requête :

- adresses IP des serveurs DNS de `fr`, `dauphine.fr` et `ufrmd.dauphine.fr`
- adresse IP de la machine `cvs-etud.ufrmd.dauphine.fr`

Prochaine requête : IP `www.dauphine.fr` $\longrightarrow$ requête directe à `dauphine.fr`.

Exemple : annuaires X500

Le DNS est un tel succès pratique que les annuaires X500 sont basés sur exactement le même principe :

- annuaire :
  - contient des enregistrements (*entries*) (exemple : une personne)
  - chaque enregistrement contient des attributs typés (exemple : une adresse email)
  - le type d'un enregistrement est précisé (exemple : dans une personne, on doit avoir un email)
  - le système de type est hiérarchique (modèle de l'héritage)
- système réparti avec délégation
- espace de nom global type DNS (hiérarchique)
- recherche, modification, contrôle d'accès, etc.
- implémentation pratique : LDAPv3 (Lightweight Directory Access Protocol)

Exemple : serveur web

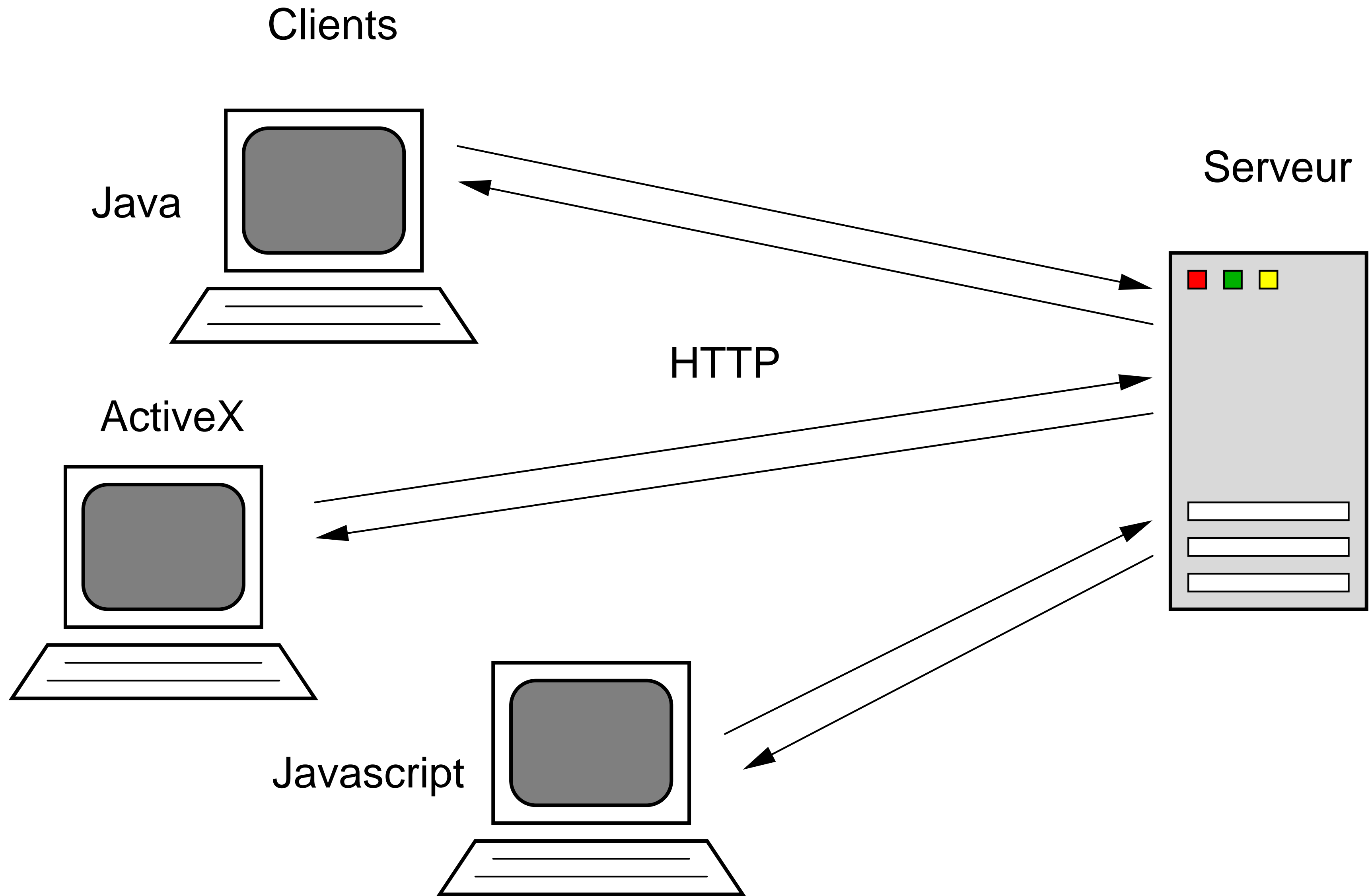
Hyper Text Transfer Protocol (HTTP) :

- solution de base de type client/serveur
- intrinsèquement réparti à cause des liens (par exemple, toutes les requêtes cgi peuvent être exécutée sur un serveur différent du serveur principal)

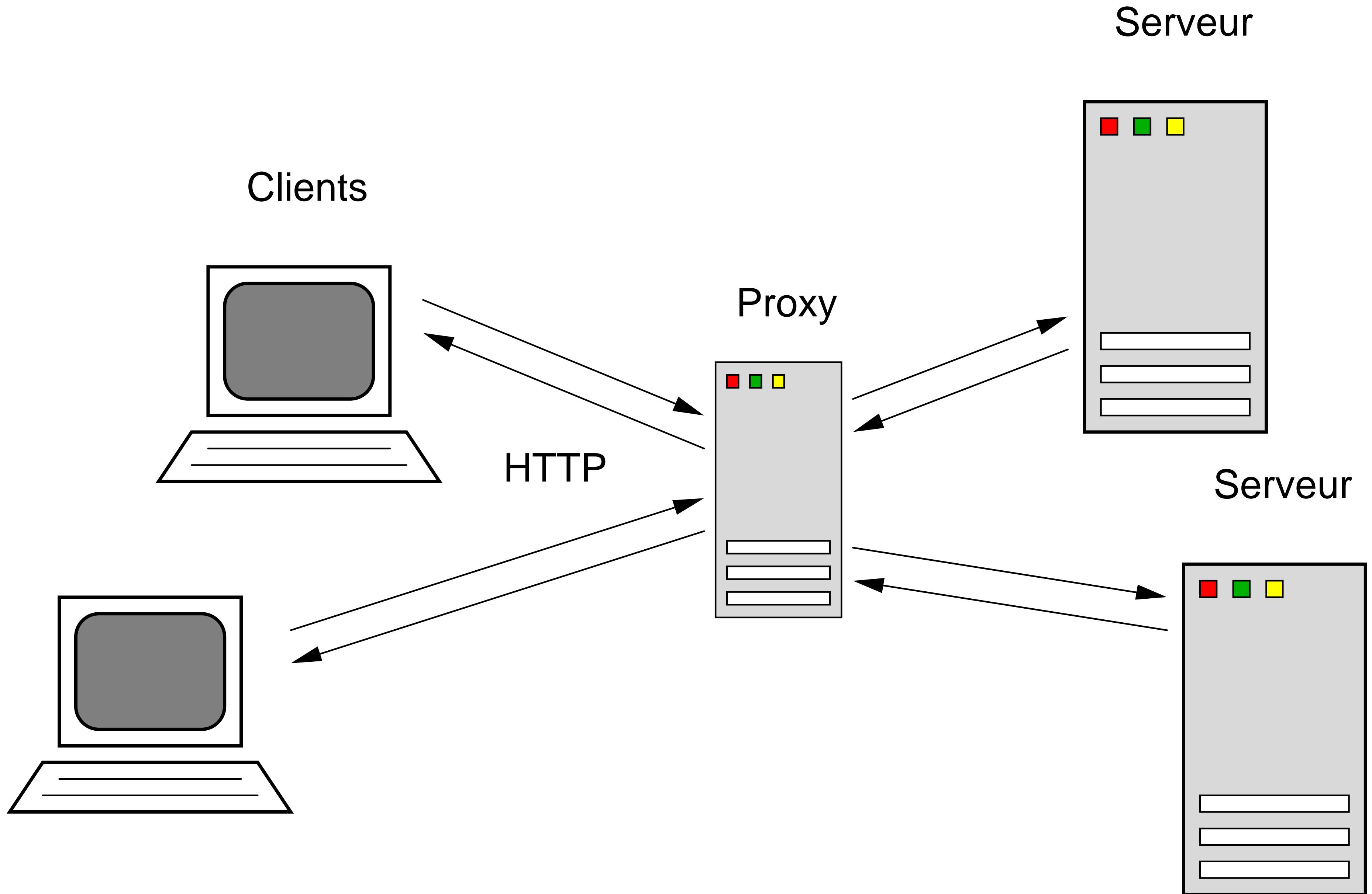
Evolutions :

- traitement sur le client (Javascript, Java, Flash, SVG, etc.) $\rightarrow$ répartition des calculs entre client et serveur
- équilibrage de charge (*load balancing*) :
  - les requêtes arrivent sur un serveur principal
  - elles sont relayées (mécanisme de *proxy*) sur plusieurs serveurs effectifs (grâce à `mod_rewrite` sous apache par exemple)
- programmes sur le serveur : cgi, servlet, langages de script serveur (JSP, PHP, ASP, etc.) $\rightarrow$ base des architectures multi-niveau

Serveur web basique



Equilibrage de charge



Exemple : Systèmes multi-couches

Beaucoup d'applications classiques se découpent en plusieurs couches (ou niveau) :

- **Présentation** : tout ce qui permet l'interaction avec l'utilisateur
- **Application** (*Business* ou Métier) : toute la logique applicative (vérification des entrées de l'utilisateur, etc.)
- **Stockage** : stockage permanent des informations relatives à l'application

Exemple en VPC :

- présentation : site web, catalogue papier (!)
- application : modalité de réduction, frais de port, identification du client, etc.
- stockage : SGBD (catalogue, stock, etc.), facturation, analyse des coûts (ERP)

Systèmes multi-couches : modèles

On peut implémenter une application de nombreuses façons :

- Monolithique : les différents niveaux sont imbriqués dans un programme unique
- Deux couches :
  - presque obligatoire quand on travaille avec un SGBD
  - sur le serveur : le SGBD, avec une partie de la logique applicative (contraintes d'intégrité, procédures stockées, etc.)
  - sur le client : présentation et l'autre partie de l'application
- Trois couches (ou plus) :
  - client : présentation des objets métier
  - objets métier : toute la logique applicative
  - SGBD : stockage des objets métier

Multi-couches : avantages et inconvénients

Avantages :

- découplage :
  - du code : maintenance plus facile
  - des équipes : chacun son métier (maquette, administration du SGBD, etc.)
- répartition et donc équilibrage de la charge :
  - serveur web (présentation)
  - serveur des objets métier
  - SGBD

Inconvénients :

- efficacité
- bande passante
- difficulté de mise en œuvre (conception et programmation délicates)

Couche de présentation

Une technique classique, les extensions sur le serveur :

- cgi (en C, C++, Perl)
- versions embarquées dans le serveur (`mod_perl`), ou dans un container (Servlet en Java)
- relativement lourd : il faut produire le code HTML à la main dans le programme
- équilibrage de charge classique : pages statiques sur un serveur, pages dynamiques sur d'autres (par exemple)
- Servlet : équilibrage intégré (le moteur de servlet Tomcat est indépendant du serveur web)

Problème : comment accéder à la couche métier ?

Couche de présentation (2)

Technique plus facile d'utilisation, les langages “embarqués” :

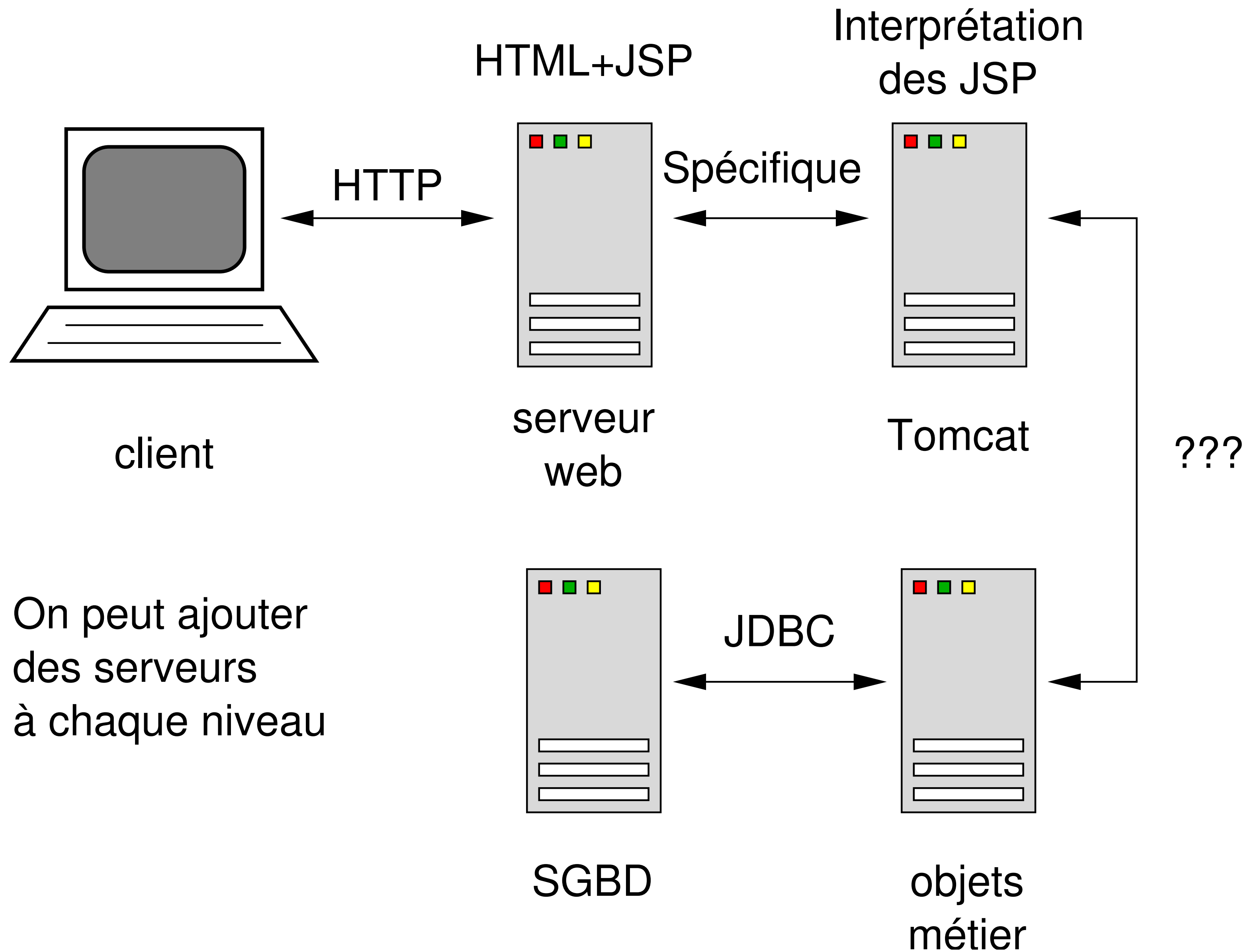
- intégration des commandes dans le code HTML
- JSP (Java), ASP (Visual Basic (!)), Embperl (Perl), PHP (langage spécifique), etc.
- équilibrage parfois très évolué : le moteur JSP Tomcat est indépendant du serveur web

Même problème que les extensions : comment accéder à la couche métier ?

Couche métier

- Accès direct au SGBD :
  - solutions classiques et standardisées : ODBC et JDBC
  - solutions spécifiques : interface SGBD de Perl, de PHP, etc.
  - problème : expertise SQL (transactions par exemple)
- Accès masqué au SGBD :
  - persistance automatique (EJB CMP et JDO pour Java par exemple)
  - solutions relativement nouvelles
  - sûrement l'avenir
- Lien avec la couche de présentation :
  - solution basique : bibliothèque (i.e., le code métier est exécuté par la couche de présentation)
  - **approche distribuée : c'est un des objets de ce cours !**

Exemple multi-couches



Exemple : *peer to peer*

Principe :

- *peer to peer* pur : pas de serveur ; connexions directes entre les participants
- *peer to peer* “pratique” : l’essentiel des communications s’effectue entre les participants (mais il reste des serveurs)

Buts :

- très forte resistance à la panne
- partage de la bande passante

Exemples :

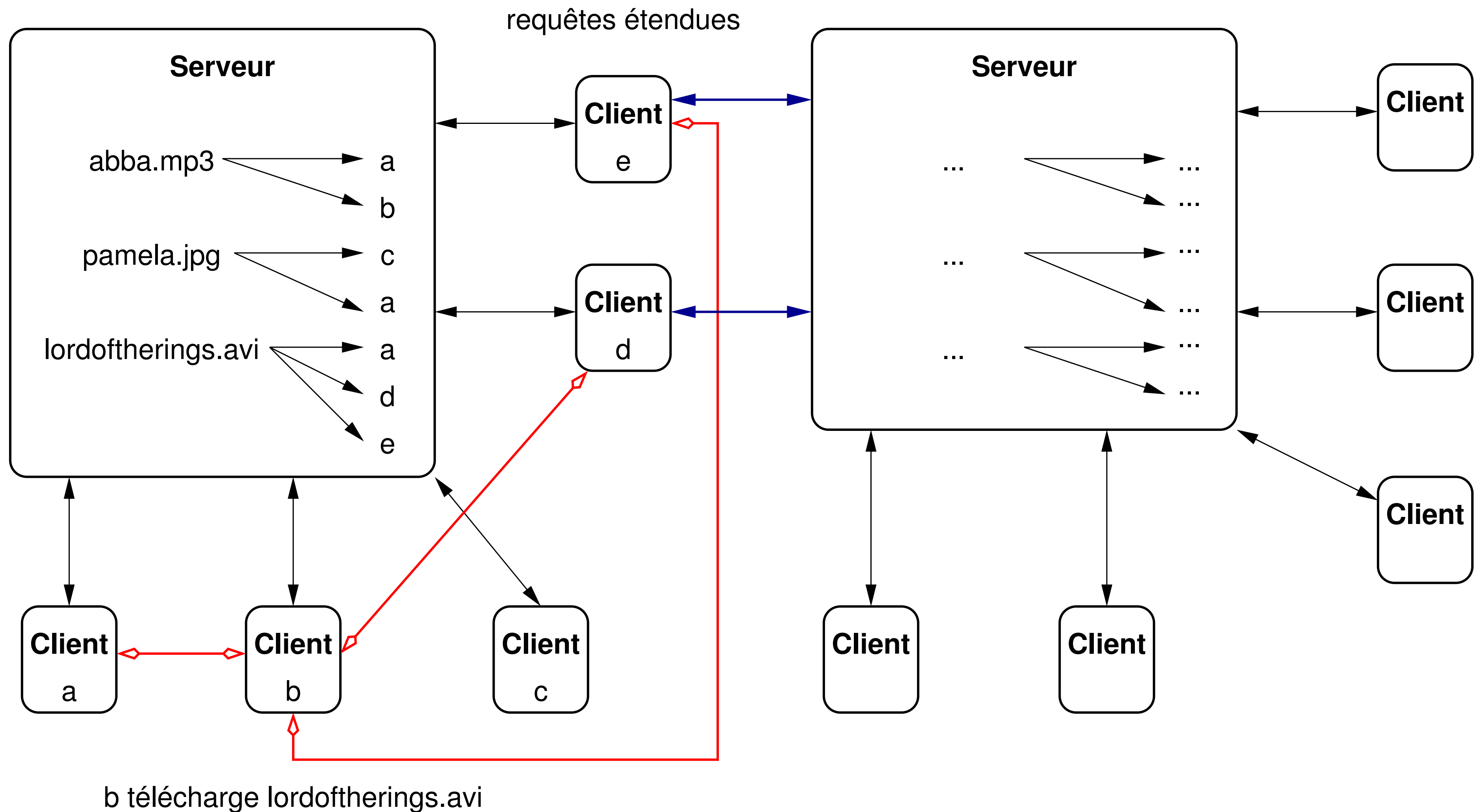
- Napster (serveur central)
- Edonkey 2000 (plusieurs serveurs)
- Freenet et overnet (*peer to peer* pur)

Edonkey 2000

But : piratage (soyons clair !) ou (en langue de bois) partage de documents. Architecture :

- tout le monde peut devenir serveur
- sur le serveur :
  - liste des clients
  - pour chaque client : liste des documents proposés au téléchargement
- communication client vers serveur :
  - le client demande un document
  - le serveur lui renvoie la liste des clients qui possèdent le document
- communication client vers client :
  - téléchargement des documents
  - morceaux par morceaux (un client peut télécharger une partie d'un document d'un client et le reste d'un autre)

Edonkey 2000



Requêtes étendues : depuis un client vers un autre serveur.

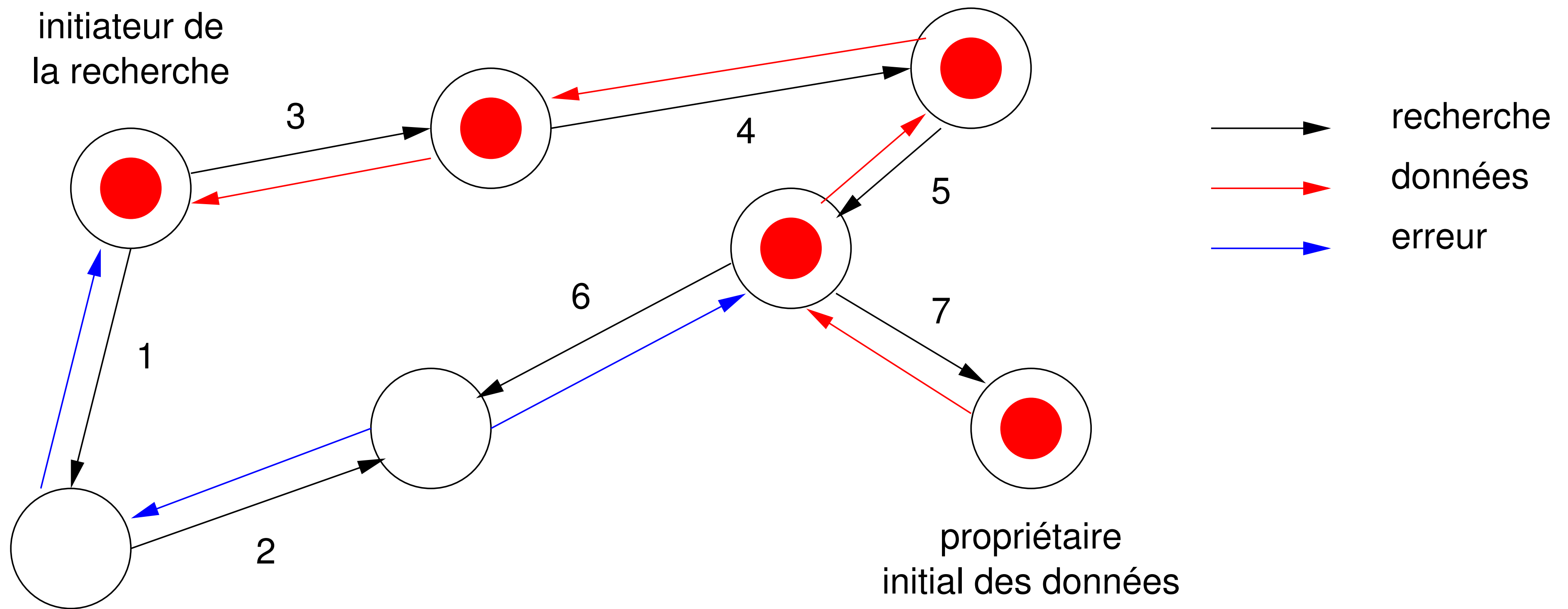
Freenet

Mêmes buts ! Architecture :

- véritable *peer to peer*
- aucun serveur
- recherche d'un document :
  - un client demande un document à ses “amis” (autres clients)
  - quand un client reçoit une requête de documents à laquelle il ne peut pas répondre, il propage cette requête à d'autres clients
  - la réponse suit le chemin inverse
- confidentialité forte : tout est crypté

Dans les deux cas (Edonkey 2000 et Freenet), on obtient une sorte de système de fichiers distribué.

Freenet



Dessin d'après les documents de conception de freenet.

- pas de boucle
- routage basé sur les clés des documents

Exemple : les *clusters*

La puissance de calcul d'un microprocesseur est souvent insuffisante :

- simulations en physique, chimie, ingénierie, etc.
- effets spéciaux et images de synthèse
- intelligence artificielle (jeu d'échecs, etc.)
- ...

Solution “simple”, les systèmes multi-processeurs symétriques (SMP) :

- plusieurs processeurs dans un même ordinateur
- mémoire partagée
- *threads*

Quand c'est encore insuffisant, les *clusters* :

- plusieurs ordinateurs (éventuellement SMP)
- connexion haut débits (ethernet 100Mb, plusieurs cartes par machine)

Les *clusters*

En général (Beowulf, Mosix, etc.) :

- processus, i.e., mémoire non partagée → passage de message
- transparence réseau totale : deux processus communiquent comme si ils étaient sur la même machine
- migration de processus
- équilibrage dynamique de la charge

Problèmes :

- programmation parallèle très délicate
- pas de vrai standard
- pas de mémoire partagée répartie

Les *Grid*

Quand un *cluster* ne suffit pas, on passe au *grid* :

- plusieurs ordinateurs (beaucoup !)
- pas de contrainte de géographie (un *grid* peut mettre en commun des ordinateurs répartis dans plusieurs pays)
- architectures physiques et logicielles hétérogènes

Exemples :

- Seti@Home et toutes ses variantes
- supers calculateurs (NASA, NSF, etc.)

Commentaires :

- l'avenir des super-calculateurs
- pas encore de norme
- programmation très délicate
- problèmes de sécurité

Les problèmes

Ces exemples illustrent les problèmes classiques des systèmes répartis :

- communication :
  - l'ossature de tout système réparti
  - la zone de compromis entre efficacité et facilité de programmation
- hétérogénéité :
  - portabilité
  - interopérabilité
- sécurité :
  - authentification
  - droits d'accès (politiques de sécurité)
- déploiement
- administration

Communication dans un système réparti

Plusieurs niveaux d'abstraction :

- bas niveau : *socket*
- niveau intermédiaire : transparence réseau pour les appels de fonction (*Remote Procedure Call*), systèmes à base de messages (PVM, MPI, SOAP, etc.)
- haut niveau : transparence réseau pour les objets (*Object Request Broker*)
- très haut niveau : objets répartis et objets mobiles

Actuellement :

- protocoles spécialisés : bas niveau (DNS, LDAP, etc.)
- applications multi-couches : RPC ou ORB
- calcul distribué (*cluster* et *grid*) : PVM et MPI
- *web services* : SOAP

Avenir : ? ? ?

Sockets

Rappels :

- connexion stable (TCP) bi-directionnelle entre deux programmes
- fonctionnent comme des fichiers : *file descriptor* en C, *stream* en Java
- très bas niveau :
  - il faut connaître l'adresse ip et le numéro de port du serveur → pas de transparence réseau
  - gestion par le programmeur du **protocole** (langue parlée par le serveur)
  - souple
  - efficace
  - interopérable avec un bon protocole (difficile)
- on peut faire encore plus efficace : *User Datagram Protocol* (UDP). Encore plus bas niveau (pas de connexion).

Sockets (2)

Beaucoup de solutions actuelles sont basées sur TCP ou même sur UDP :

- client/serveur TCP : *Simple Mail Transfer Protocol*, *File Transfer Protocol*, *Hyper Text Transfer Protocol*, etc.
- systèmes répartis :
  - *Domain Name System* (UDP pour les requêtes, TCP pour le reste)
  - *Lightweight Directory Access Protocol* TCP
  - Edonkey 2000 (UDP et TCP)
  - freenet (implémentable au dessus de diverses méthodes de transport, mais en général TCP)
  - jeux en réseaux
  - etc.

Sockets (3)

Mise en œuvre délicate car il faut tout faire :

- mettre au point un protocole :
  - mise en place de la connexion (sécurité)
  - commandes reconnues par le serveur (paramètres et résultats)
  - codes d'erreur
  - etc.
- programmer le "serveur"
- fournir une API pour le client (sinon le client doit "parler" directement au serveur)
- fournir un minimum de transparence réseau
- être interopérable (exemple *bid endian* et *little endian*)

Difficile et surtout **très redondant** $\Rightarrow$ automatiser ce processus.

Vers le haut niveau

Que peut-on automatiser ?

- représentation portable des données
- passage de messages
- appel de fonctions à distance
- objets distants
- objets répartis
- objets mobiles
- ? ? ?

eXternal Data Representation (XDR)

Une grosse difficulté redondante : la représentation interne des données dépend dans la machine (*bid endian* et *little endian* par exemple). Une solution XDR :

- norme issue de SUN (RFC 1832) et de l'ONC
- système de représentation des données indépendant du hardware : précise l'ordre des octets (*bid endian*)
- orienté C (int, float, struct, etc.)
- langage de description des types très inspiré du C. Exemple :

```
1  const MAXNAME = 20;
2  struct person {
3    string firstname<MAXNAME>;
4    string lastname<MAXNAME>;
5  };
```

Représentation des données : autres solutions

De très nombreuses autres solutions sont disponibles :

- *Network Data Representation*
  - Même principe que XDR, mais proposé par l'Open Group (ex *Open Software Foundation*)
  - Elément de *Distributed Computing Environement*
  - Relativement répandu sous Unix mais pas sous Linux
- *XML-RPC*
  - Contient entre autre une représentation de données structurées en XML
  - A la mode...
- *Simple Object Access Protocol*
  - Evolution de XML-RPC définie par le W3
  - Représentation de données structurées en XML (modèle des types des schémas)
  - Très à la mode...

Systèmes à messages : *PVM*

Le standard des *clusters* est *Parallel Virtual Machine* :

- cible : *cluster* de machines (hétérogène)
- un programme PVM est un ensemble de processus
- transparence réseau totale : chaque processus est identifié par un *id* qui est utilisé le contrôle et la communication
- message : bloc de mémoire rendu indépendant de la machine grâce à XDR
- communication optimisée : mémoire partagée sur une même machine, TCP sinon
- API C (C++) et fortran

Grand pas en avant : optimisation automatique, transparence réseau, portabilité des données et du code, interopérabilité.

Systèmes à messages : *MPI*

Message Passing Interface est l'autre standard des *clusters* et super-ordinateurs :

- cible : machines parallèles
- plus de primitives de communication que PVM (broadcast, etc.)
- en général plus efficace que PVM sur les machines parallèles
- portable

Contrairement à PVM, il faut ajouter une sur-couche (MPICH-G2 version *Grid* de MPI) pour obtenir la transparence réseau, l'interopérabilité, la tolérance à la panne, etc.

Systèmes à messages

Avantages :

- portabilité
- transparence réseau
- interopérabilité
- efficacité

Inconvénients :

- messages de bas niveau (au mieux struct)
- technique de programmation spécifique (ni procédurale, ni objet)

Autres solutions basées sur des messages, beaucoup moins orientées *clusters* :

- SOAP
- Java Message Service

Ce sont des solutions de plus haut niveau, en général moins efficaces mais plus accessibles.

Appel de fonctions à distance (*RPC*)

Idée de base : proposer au programmeur la transparence réseau pour les appels de fonctions. Quelques solutions :

- ONC *Remote Procedure Call* (C, Sun)
- DCE *Remote Procedure Call* (C, Open Group)
- *Remote Methode Invocation* (Java, Sun)
- XML *RPC* (tout langage, Userland)
- SOAP (tout langage, W3C)

Principe de base :

- le client appelle une fonction “normale”
- un “code” local transforme l’appel de fonction en un message (au sens large), envoyé au “serveur”
- un “code” sur le “serveur” transforme le message en un appel de fonction
- le serveur exécute l’appel de fonction
- même chose en sens inverse pour la réponse du serveur

Les *RPC*

Vocabulaire :

- souche (*stub*) : représentant local du serveur $\longrightarrow$ traduction de l'appel local en message
- squelette (*skeleton*) : équivalent de la souche sur le serveur $\longrightarrow$ traduction du message en appel local
- empaquetage (*marshalling*) : représentation des données de l'appel (ou du résultat) sous forme d'un message
- dépaquetage (*unmarshalling*) : opération inverse

Première brique du *RPC*, la représentation des données interopérable :

- XDR pour les RPC ONC
- NDR pour les RPC DCE
- XML pour XML RPC et son “successeur” SOAP
- Sérialisation Java pour RMI (pas seulement)

Les *RPC* ONC et DCE

RPC classiques :

- orientés programmation procédurale (C)
- limitent le travail du programmeur :
  - un langage permet de décrire la fonction (extension de XDR par exemple)
  - un programme adapté engendre à partir du langage le *stub* et le *skeleton*
  - il reste à programmer :
    - la fonction à intégrer au serveur
    - le client (connexion au serveur et appel de la fonction)
- fonctionnent essentiellement avec TCP et UDP
- exemple incontournable : *Network File System*
- DCE $\simeq$ ONC + sécurité + *threads*

Les *RPC* basés sur XML

Principe de base, représentation des données en XML :

- XML RPC :
  - très simple
  - la spécification donne seulement la représentation d'un appel de fonction en XML
  - ne dépend pas du langage source
  - le transport est assuré par HTTP
- SOAP :
  - beaucoup plus ambitieux
  - format de message au sens général
  - indépendant du transport (HTTP, Mail, etc.)
- les outils proposés autour de XML RPC et de SOAP sont intéopérables mais non standard
- relativement haut niveau

Remote Methode Invocation (RMI)

Technologie spécifique Java :

- RPC en Java
- orienté objet
- très intégré à Java : pas de langage extérieur par exemple
- travail minimal pour le programmeur

Même s'il s'agit d'une technique RPC, elle est trop orienté objet pour être vraiment comparable avec les autres techniques.

Voir la suite pour les liens avec CORBA.

Appel de fonctions à distance

Avantages des RPC ONC et DCE :

- transparence réseau
- portabilité grâce aux standards
- interopérabilité (idem)
- indépendant du langage
- bonnes performances en général pour les RPC ONC et DCE

Inconvénient : orienté procédure (dépassé).

Avantages des RPC XML :

- haut niveau (sans être objet)
- intégration internet très forte (*Web services*)

Inconvénients :

- technologies très récentes
- performances en retrait

Objects distants

Appel de **méthodes** à distance, *RPC* objets :

- idée de base : apporter les bénéfices de l'objet aux systèmes répartis. Par exemple :
  - encapsulation
  - polymorphisme
  - etc.
- plusieurs grands standards :
  - CORBA (Common Object Request Broker Architecture)
  - Java RMI
  - Microsoft COM+/DCOM (Distributed Component Object Model)
- tous basés sur la notion d'interface et sur des mécanismes de bas niveau semblables à ceux utilisés par les RPC

CORBA

A la fois l'ancêtre et le standard :

- norme de l'Object Management Group
- version actuelle 3.0.2 (décembre 2002)
- basé sur un bus logiciel, l'Objet Request Broker (ORB)
- totalement interopérable :
  - les objets sont décrits en IDL (Interface Definition Language)
  - implémentés dans n'importe quel langage (Java, C++, C, etc.)
  - les ORB sont des plus en plus compatibles, même au niveau source
- nombreuses implémentations, commerciales et open source, mais pas toujours en accord avec la norme (en général, la norme 2.4 est supportée)

Java RMI

Les RPC objets version Java. Deux versions :

- Java Remote Method Protocol
- Internet InterORB Protocol (celui de CORBA)

JRMP est spécifique Java et donc très intégré à la plate-forme :

- programmation minimale
- concepts Java en version répartie (par exemple *Garbage Collecting* réparti)
- portabilité au sens Java
- **ne peut communiquer qu'avec une autre JVM**

RMI-IIOP est basé sur le protocole de CORBA :

- programmation plus complexe (mais pas besoin d'IDL)
- intégration moins bonne
- interopérable

COM+/DCOM

Solution Microsoft basée sur COM+ et sur les RPC DCE :

- propriétaire Microsoft (des portages commerciaux existent)
- multi-langage (IDL)
- *Garbage Collecting* réparti
- programmation plus lourde que CORBA (très verbeux)
- accès aux services de Windows

Intérêts limités par rapport à CORBA, sauf en environnement

100% Microsoft.

Très haut niveau

Diffusion plus confidentielle et outils plus expérimentaux :

- objets répartis :
  - l'espace de stockage alloué à l'objet est effectivement réparti sur les différents clients
  - gestion des répliques, contrôle de modification, etc.
- objets mobiles :
  - objet (i.e., données et code) libre de passer d'une machine à une autre
  - l'objet bouge mais n'est pas accessible à distance
  - exemple évolué : agent capable d'enchérir sur un site de type *ebay*