Architectures distribuées P2P

Dan VODISLAV

Université de Cergy-Pontoise

Master Informatique M2

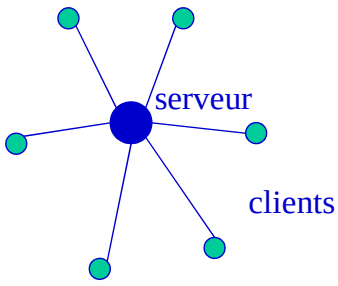
Plan

- Architectures pair-à-pair
  - Caractéristiques
  - Classification
- Exemples de systèmes
  - Architectures non structurées : Napster, Gnutella
  - Architectures structurées: tables de hachage distribuées (DHT)

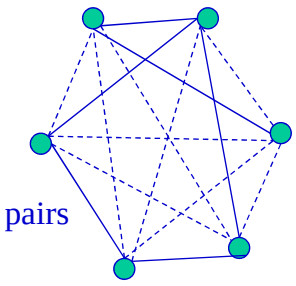
Architectures P2P

- Pair-à-pair (P2P)
  - Architecture distribuée
    - Ressources distribuées sur un ensemble de machines (pairs)
    - Collaboration pour réaliser une fonction d'une manière décentralisée
  - Pas de distinction entre clients et serveurs
  - Caractéristiques:
    - Performances: pas de serveur centralisé, distribution des traitements
    - Autonomie: chaque pair a le contrôle de ses données, connexions ad-hoc
    - Passage à l'échelle: distribution de la charge, réplication des données
    - Dynamique: système ouvert, gestion dynamique de la composition du réseau
    - Uniformité: meilleur support pour anonymat et la confidentialité
  - Difficultés:
    - Coût de la communication
    - Cohérence et qualité des données

P2P et client-serveur



Client-serveur



P2P

Classification

- Selon la topologie du réseau
  - Graphe aléatoire, étoile, arbre, grille, etc.
- Selon le niveau de décentralisation
  - *Centralisé*: un pair central a une fonction privilégiée (Napster)
  - *Hybride*: une partie des pairs (super-pairs) jouent un rôle particulier
  - *Pur*: tous les pairs ont les mêmes fonctionnalités
- Selon la structuration du réseau
  - *Non-structuré*: pas de critère de répartition des données sur les pairs
    - Localisation: demande aux voisins (« flooding »), temps non garanti
  - *Faiblement structuré*: groupement des pairs par caractéristiques communes (« clustering »)
    - Répartition et localisation par groupe, temps partiellement garanti
  - *Structuré*: répartition précise des données (ex. par hachage)
    - Localisation rapide, temps garanti

Applications P2P

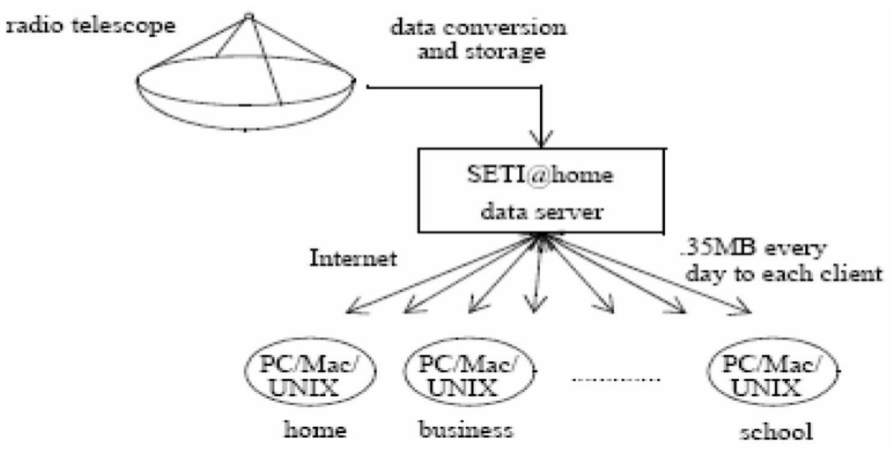
- Communication et collaboration
  - Communication directe entre pairs: chat, messagerie, téléphonie
  - Chat/Irc, MSN Instant Messenger, Jabber, Skype
- Calcul distribué
  - Répartition de parties d'un calcul entre pairs
    - Seti@home, genome@home
- Support aux applications web
  - Allégement de charge d'un serveur (Coral), protection contre attaques
- Systèmes de bases de données
  - Bases de données distribuées: PIER, Piazza, KadoP, Edutella
- Distribution de contenu
  - Échange de fichiers, publication et stockage
  - La plupart des systèmes actuels: Napster, Kazaa, Chord, CAN, …

Types de systèmes P2P

- Calcul distribué : SETI@Home
- Non structurés
  - Centralisé: Napster
  - Distribué: Gnutella
- Structurés
  - Tables de hachage distribuées (DHT): Chord, CAN
  - Topologie hybride en arbre: MediaPeer

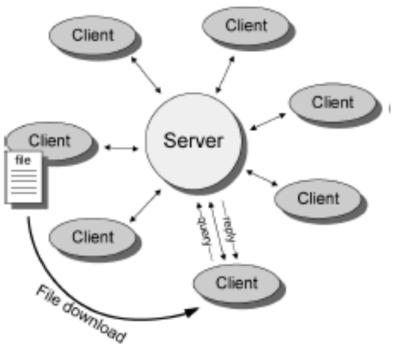
Calcul distribué

- Partage de CPU et des données à traiter
  - On le place plus souvent dans la catégorie « grille de calcul »
- SETI@home : « Search for Extraterrestrial Intelligence »
  - But: découvrir des signaux radio en provenance de l'espace
  - Distribution de fichiers de données à traiter (350K/jour)



Réseau P2P non structuré centralisé

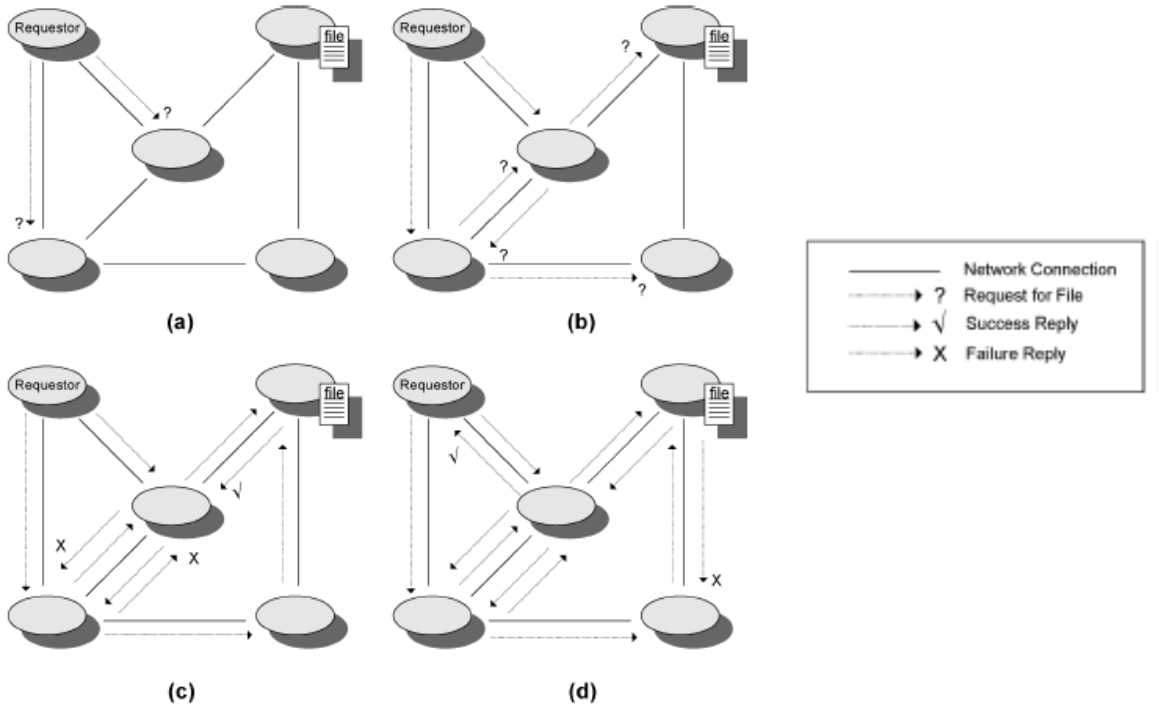
- Napster : répertoire centralisé
  - Étapes:
    1. Les clients publient sur le serveur la liste des noms de leurs fichiers
    2. Le client qui cherche un fichier demande au serveur
    3. Le serveur répond avec une liste de clients
    4. Parmi les pairs cible, le client qui a fait la requête détecte par un « ping » le pair le plus proche
    5. Le client télécharge directement le fichier en provenance du pair choisi



Réseau P2P non structuré distribué

- Gnutella
  - Chaque pair gère ses données
  - Chaque pair connaît une liste de voisins
  - Requêtes: vers les voisins (« flooding »)
    - propagées dans un voisinage de rayon limité
  - Joindre le réseau:
    - message « Ping » vers un ensemble de pairs trouvés dans une base de données (http://gnutellahosts.com)
    - Les pairs envoyent un message « Pong » en retour, avec infos sur eux-mêmes et propagent le message « Ping » vers leurs voisins
  - Propriétés
    - Très robuste
    - Ne trouve pas toutes les réponses

Requêtes Gnutella



Réseaux P2P structurées

- Tables de hachage distribuées
  - « Distributed Hash Tables » (DHT)
  - Généralisation des tables de hachage
  - Chaque donnée est identifiée par une clé
- Index par hachage
  - Entrée = couple (clé, valeur)
  - Position de l'entrée dans la table: **h** (clé)
  - Fonction de hachage **h** : distribution uniforme dans la table
- Hachage distribué
  - h (clé) → pair sur lequel le couple clé-valeur sera placé
  - Distribution uniforme des valeurs sur les pairs du réseau
    - Sur un pair: table de hachage locale
  - Fonction **h** : pour *placer* et *retrouver* une valeur dans le réseau

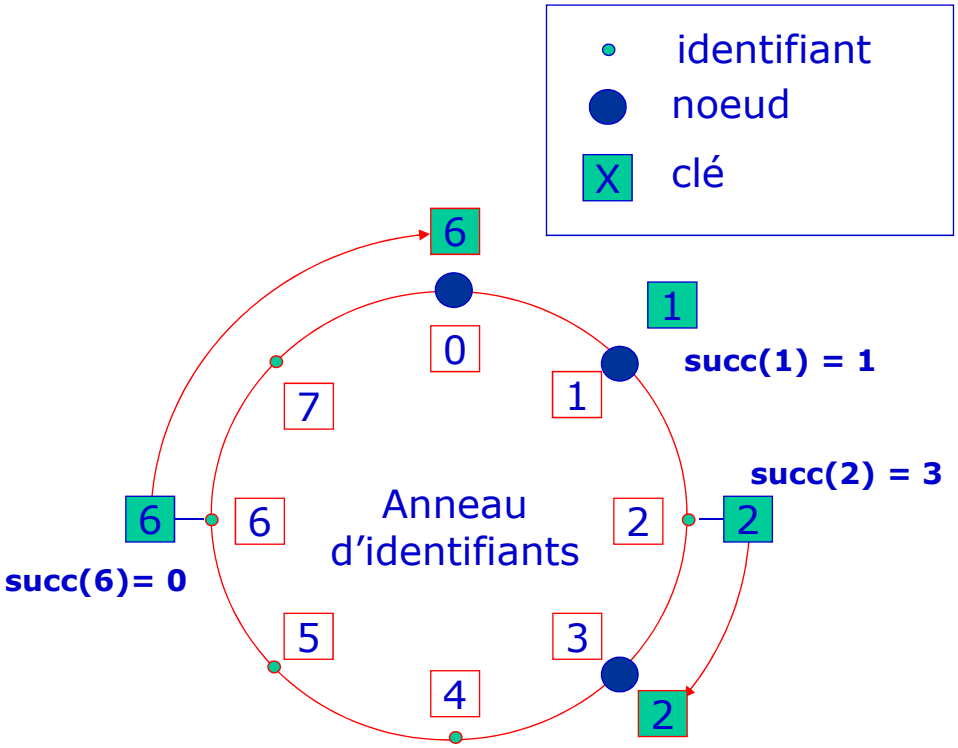
Tables de hachage distribuées

- Commandes
  - Put (clé, valeur)
  - Lookup (clé) → valeur
- Fonctionnement
  - Un pair veut publier/retrouver une donnée *v*, caractérisée par une clé *k*
    - Il calcule l'adresse du pair qui doit stocker *v*, à l'aide de la fonction de hachage **h** (la même pour tous les pairs!) appliquée à *k*
    - Il se connecte au pair cible pour transférer/récupérer la donnée *v*
- Mécanisme général de localisation de ressources distribuées
  - Le modèle clé-valeur est adapté à une large classe d'applications
  - On peut répartir *des données* ou *des index*
    - Distribuer un index : garder le contrôle sur les données
    - Pour un index: la valeur = adresse (liste d'adresses) de données

Chord

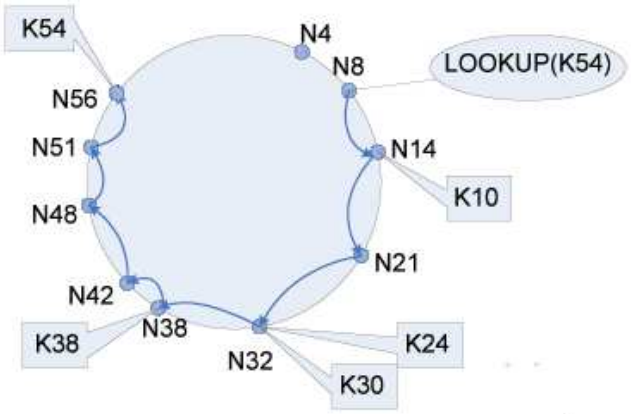
- Table de hachage distribuée
  - Clés: pour les *données* et pour les *pairs* (adresse IP)
  - Fonction de hachage sur *m* bits → valeurs dans l'intervalle $[0..2^m-1]$
  - Espace de hachage ($[0..2^m-1]$) organisé logiquement en *anneau*
- Convention
  - Quand on parle de clé *k* ou d'identifiant de pair *id*, on parle de leur correspondant (par la fonction de hachage) dans l'espace d'adressage
- Les pairs : au maximum 2^m
  - Divisent l'espace d'adressage en intervalles
  - Clé *k* → distribuée sur **succ(*k*)**
    - **succ(k)** = le pair dont l'identifiant est le premier >= *k*
    - **pred(k)** = le pair dont l'identifiant est le premier < *k*
  - Pour chaque pair on peut définir son successeur et son prédécesseur

Chord: exemple



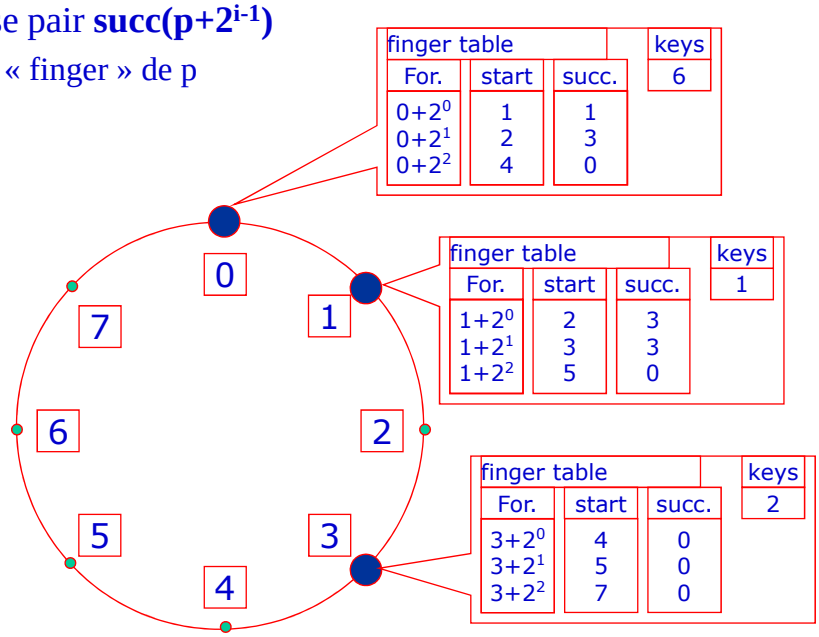
Chord: recherche naïve

- Recherche d'une clé *k* adressée à un pair d'identifiant *p*
  - *lookup(k)*
  - Si *k* est stockée par le pair *p* → il la retourne
  - Chaque pair connaît son successeur
  - Si *k* n'est pas sur *p* → il transmet la requête à son successeur
  - Le pair qui a la clé transmet la réponse directement à celui qui a fait la requête
- Problème: recherche en O(*n*)
  - *n* = nombre de pairs
  - Trop de communication



Chord: recherche optimisée

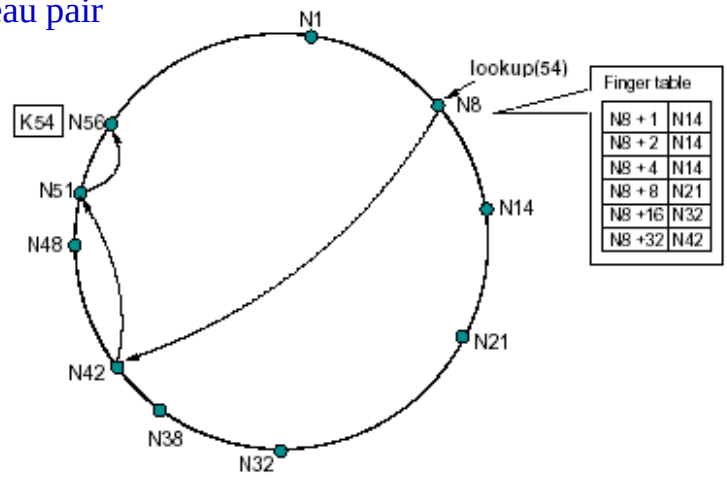
- Utilisation de tables de routage « fingers »
  - Pair *p*, table de taille *m*
  - Entrée *i* = adresse pair **succ(p+2^{i-1})**
    - Entrée i = i^{ème} « finger » de p



Chord: recherche optimisée (suite)

- **lookup(*k*)** sur un pair *p*
  - Chaque nœud connaît ses successeurs en puissances de 2 sur l'anneau
  - On prend dans la table « finger » l'entrée la plus proche avant *k*
    - *p* + 2^i est le plus proche possible de *k*, sans le dépasser
    - Au pire on tombe à moitié de l'intervalle [*p*, *k*]
  - On continue avec le nouveau pair
- Dichotomie
  - Recherche en O(log *n*)

Exemple
m = 6 (N0 – N63)
lookup(54) sur N8



Chord: ajout et retrait d'un pair

- Ajout pair *p*
  - **S = succ(*p*)** est trouvé (par lookup(*p*)) et le pointeur succ de *p* est initialisé à S
  - Les clés <= *p* de S sont déplacées sur *p*
  - Le pointeur succ de pred(*p*) est mis à *p*
- Retrait d'un pair *p*
  - Ses clés → déplacées vers **succ(*p*)**
  - Le pointeur succ de **pred(*p*)** est mis à **succ(*p*)**
- Processus de stabilisation
  - Processus indépendant qui s'exécute périodiquement et maintient la cohérence du réseau
  - Mise à jour des tables « finger »
  - Mise à jour des pointeurs « succ » en cas de panne d'un pair
    - Seuls des pointeurs « succ » corrects garantissent le fonctionnement correct
    - Maintien d'une liste de plusieurs successeurs (pas seulement le premier)

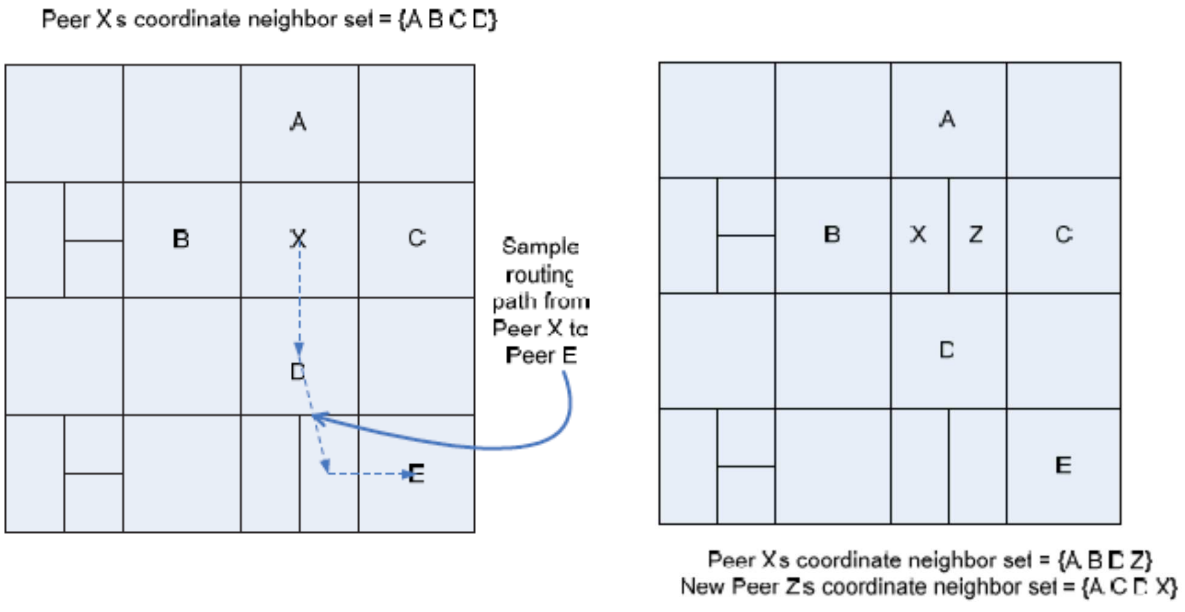
Chord: conclusions

- Recherche rapide, en log(n)
- Algorithmes simples et robustes
- Fiabilité par maintenance des successeurs (liste)
  - Résistance en cas de panne d'un pair
  - Combinée avec de la réplication sur les voisins
- Coût de maintenance
  - Déplacement de clés
  - Calcul et maintenance de successeurs
  - Processus de stabilisation
- Problème: mapper l'anneau virtuel sur le réseau réel
  - Prise en compte de la distance réseau

CAN

- Table de hachage distribuée
  - Espace de hachage: espace d-dimensionnel en coordonnées cartésiennes
  - Voisinage entre les zones extrêmes sur chaque axe (tore)
  - L'espace est divisé entre les pairs: chaque pair a une zone bien déterminée
    - Division par dichotomie sur un axe
  - Fonction de hachage uniforme clé → espace d-dimensionnel
- Routage
  - Chaque pair a des voisins (ceux des zones voisines)
  - Il connaît sa zone (peut décider si une clé lui revient ou non) et ses voisins
  - lookup(*k*) sur pair *p*:
    - Si k est dans la zone de p → trouvée
    - Sinon, parmi les voisins de *p*, *un seul* est le plus proche de la zone de *k* !
    - Routage de *lookup(k)* vers ce voisin et recherche récursive

CAN : exemple dans un espace 2D



CAN : ajout et retrait d'un pair

- Ajout d'un pair
  - Choix aléatoire d'un point *P* dans l'espace
  - Recherche de la zone et du pair *X* qui est responsable du point *P*
  - Division de la zone de *X* selon l'un des axes
  - Affectation d'une des moitiés au nouveau pair et transfert des clés
  - Création de la liste de voisins du nouveau pair à partir des voisins de *X*
  - Pour chaque voisin du nouveau pair (dont *X*), mise à jour de la liste de voisins pour tenir compte du nouveau pair
- Retrait d'un pair
  - Un des voisins prend en charge la zone désertée et les clés
  - Mise à jour des voisinages

CAN: performances

- Routage pour *lookup*
  - *d* dimensions, *n* zones (pairs) → $O(d * n^{1/d})$
    - Ex. d = 2 → $O(2\sqrt{n})$
  - Taille table routage pour chaque pair: 2*d* (indépendant de *n*)
- Avantage
  - Si un pair dans le chemin est en panne, un nouveau chemin optimal existe
  - Adaptation automatique en choisissant le meilleur voisin disponible
- Comparaison avec Chord
  - Moins bien en temps de routage
  - Meilleure localité → plus flexible, moins d'information de routage