Introduction aux systèmes pair-à-pair

Guillaume Raschia

Sources :

librement traduit de CS 347 [H. Garcia-Molina, Stanford]

et combiné avec P2P [B. Defude, GET-Télécom SudParis]

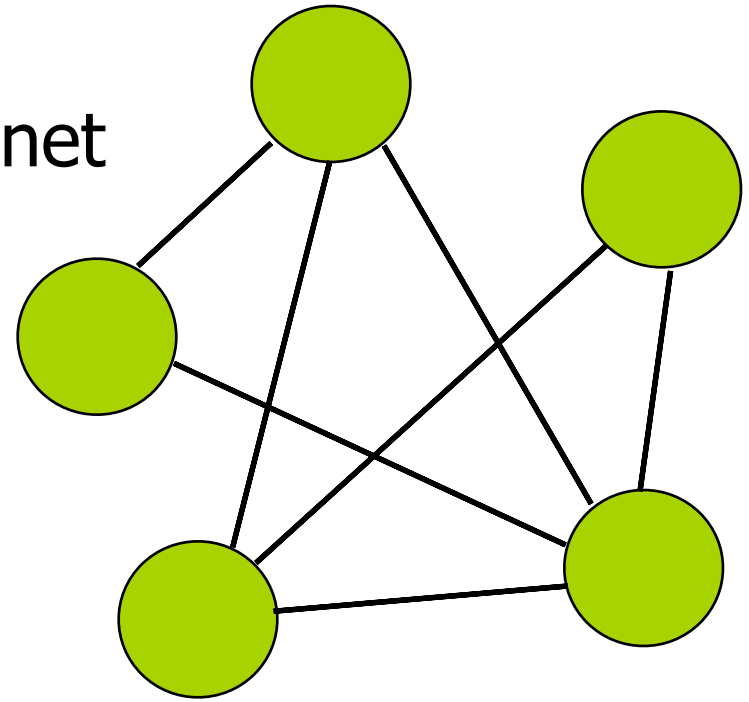
et augmenté avec INF570 [F. Le Fessant, Polytechnique]

et enrichi par Réseaux P2P [A. Benoît, ENS Lyon]

Systèmes Pair-à-Pair

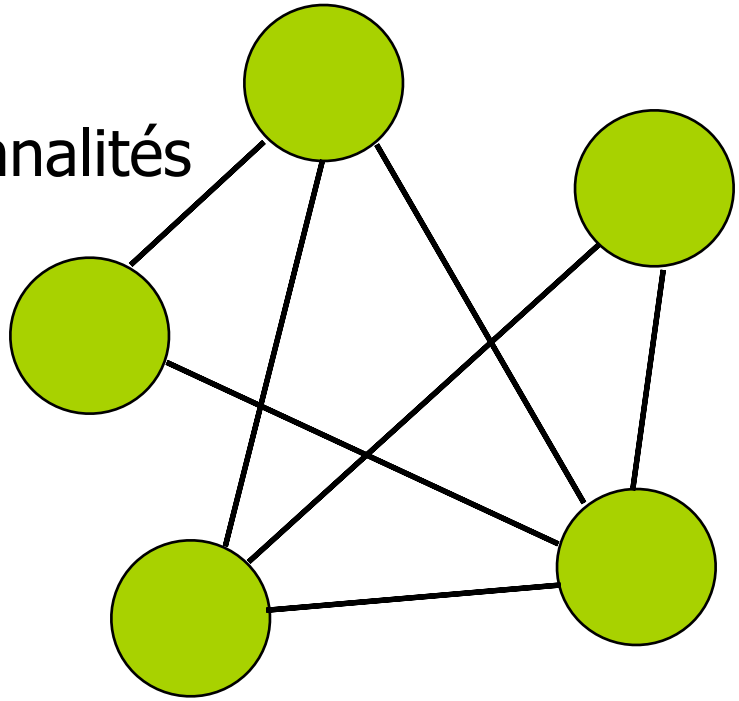
Point de vue sociétal

- Peer-to-Peer = Piratage
- Échange de fichiers sur Internet
- Violation du droit d'auteur et de la propriété intellectuelle



Systèmes Pair-à-Pair

- Applications réparties où les nœuds sont :
  - ✓ Autonomes
  - ✓ Très faiblement couplés
  - ✓ Équivalents en rôle et fonctionnalités
  - ✓ Coopérants par le partage et l'échange de ressources les uns avec les autres



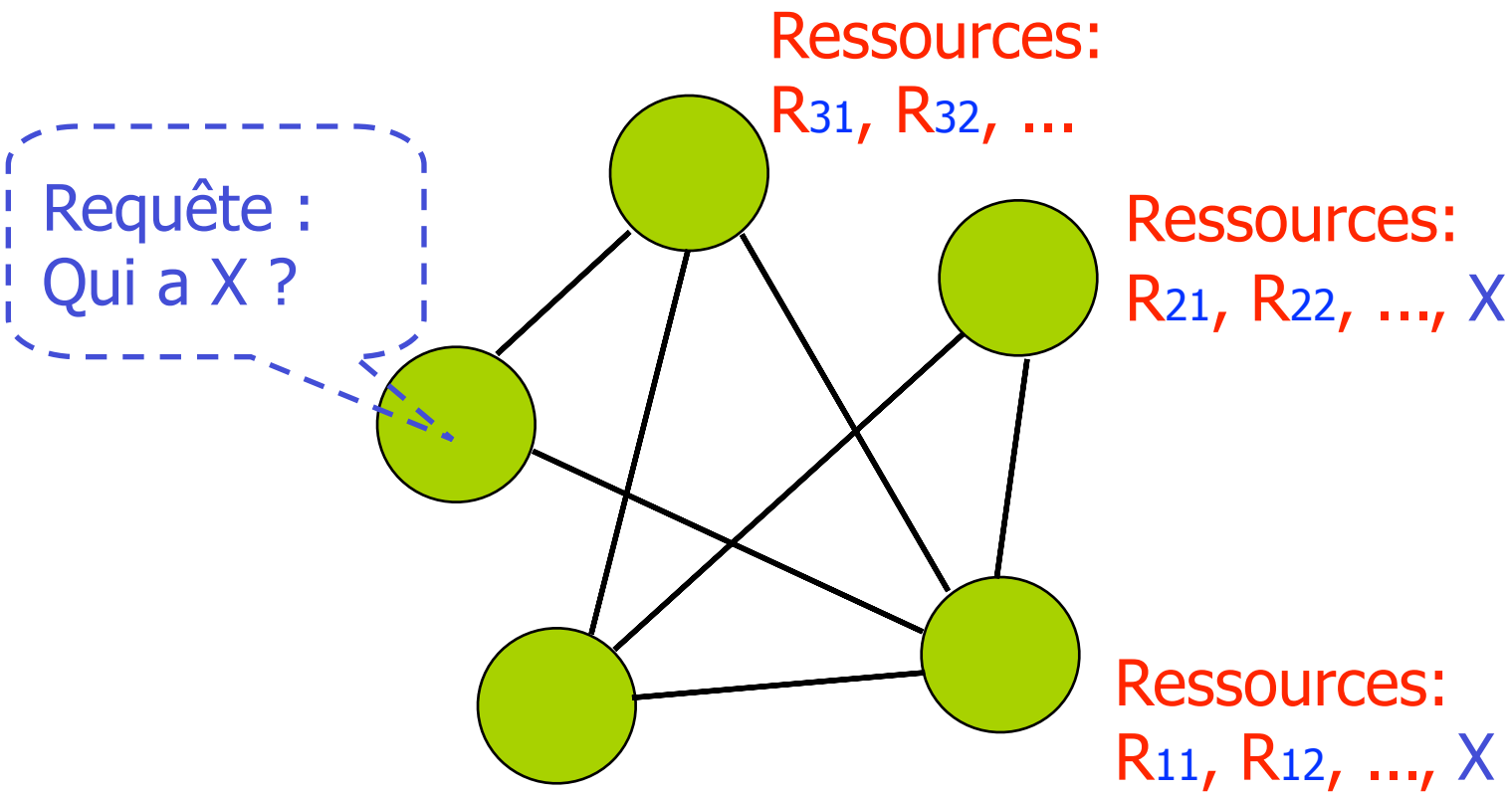
Concepts associés

- Partage de fichiers
  - Ex : Napster, Gnutella, FastTrack, e-Donkey2000, BitTorrent, FreeNet, GnuNet
- Grille de calcul
  - Ex : BOINC, Seti@home
- Système de stockage à grande échelle
  - Ex : OceanStore
- Autres
  - Ex : Skype

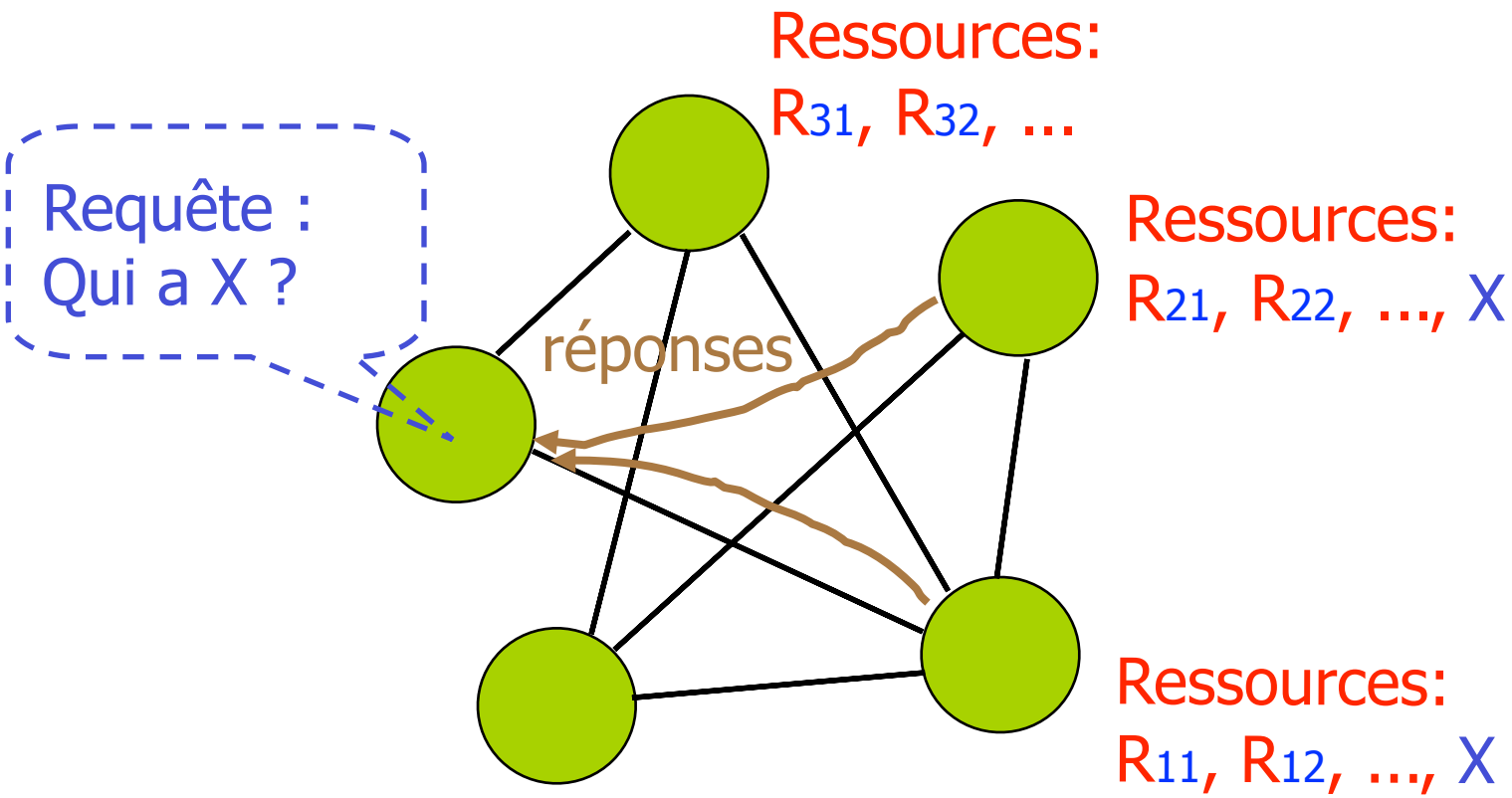
Difficultés

- Fiabilité des réseaux : perte de messages, transmission asynchrone
- Fiabilité des nœuds : arrêt, panne, corruption
- Problème des généraux : attaque concertée de G1 et G2 sur C. Synchronisation par messagers...
- Consensus Distribué impossible

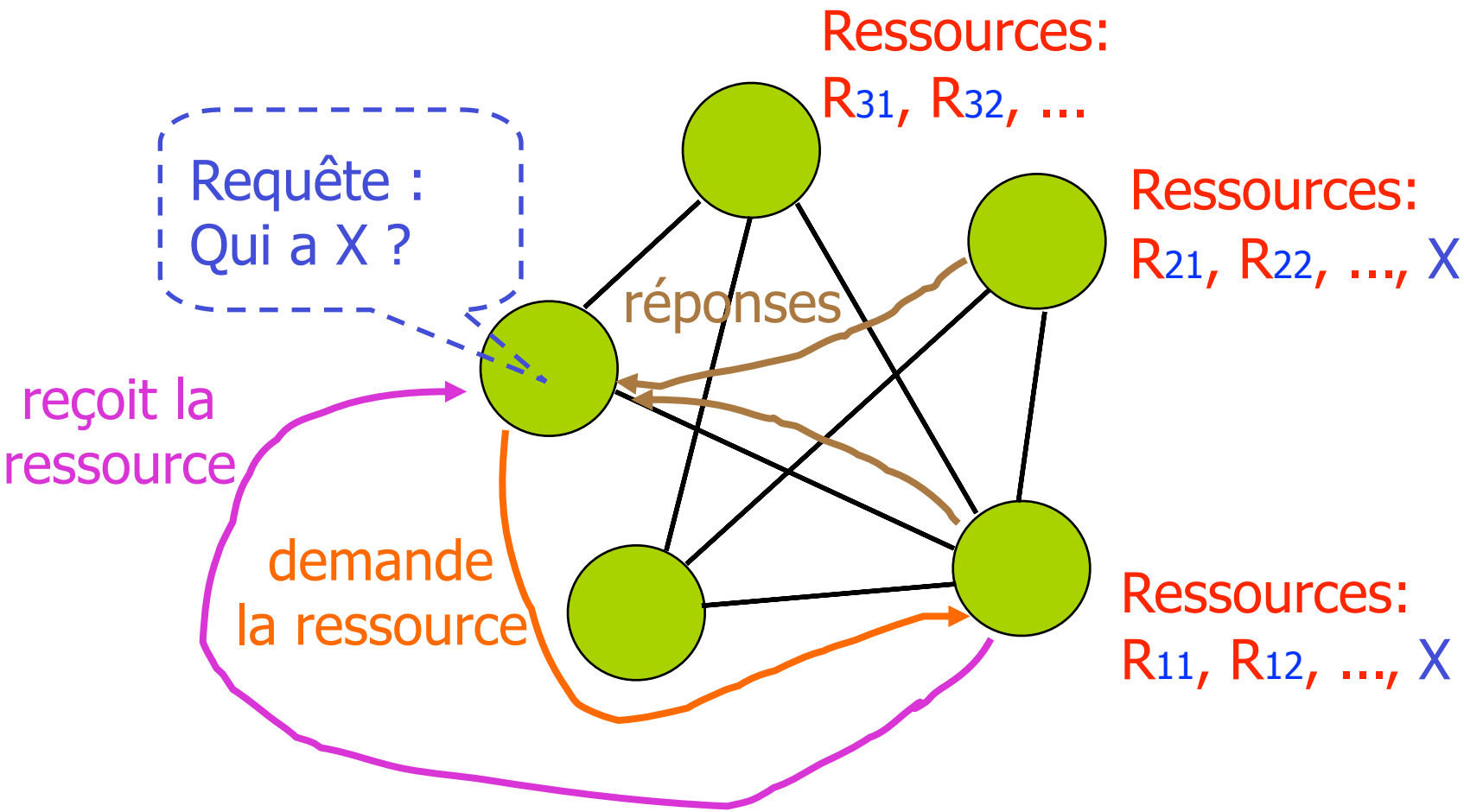
La recherche dans un système P2P



La recherche dans un système P2P



La recherche dans un système P2P



Indexation

- Soit un ensemble de paires ⟨k, v⟩

put(T, k, v) : placer la clé k dans la table T avec comme valeur v

get(T, k) : rechercher les valeurs associées à la clé k dans la table T

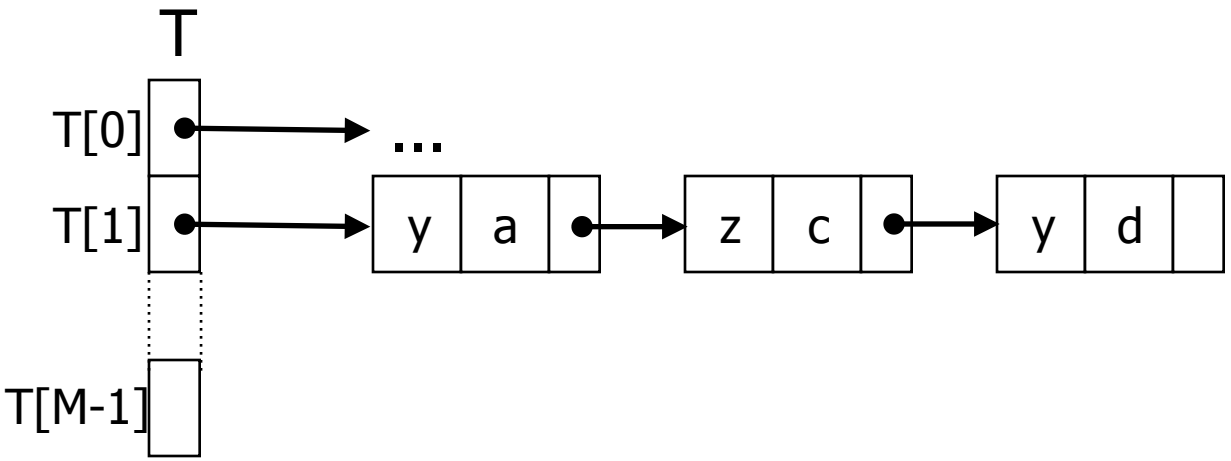
get(T, y) = {a, d}

k	v
x	a
x	b
y	a
z	c
y	d

Tables de hachage

- Fonction H : injection de K dans [0..M)
- Calculer un indice H(k)=i à partir de la clé k
- Rechercher v parmi la liste de collisions à l'entrée T[i]

H(y)=H(z)=1



Tables de hachage : complexité

- n : nombre de paires ⟨k, v⟩ stockées
- Meilleur cas : O(1)
  - M>n
  - Clés uniformément réparties dans [0..M)
- Pire cas : O(n)
  - Même hash-code (i) pour toutes les clés
  - Recherche dans une liste de collisions
  - O(logn) si arbre de recherche (k ordonnées)

Données réparties sur plusieurs nœuds

- N nœuds
- Partitionnement de [0..M) par intervalle sur les N nœuds
- Chaque nœud détient un sous-ensemble de paires ⟨k, v⟩
- On appelle « racine de k » le nœud responsable de la clé k

Notations

- X.f(params) désigne RPC de la procédure f(params) au nœud X
- X.A désigne l'envoi de message à X pour obtenir la valeur de l'objet A
- En l'absence de X, il s'agit d'une procédure locale ou d'une structure de données

nœud Y
données: C, D...

```
...
C := X.A
D := X.f(B)
...
```

nœud X
données: A, B...
fonctions: f...

Hachage distribué

Distributed Hash Table (DHT)

- Chord (2001, MIT/Berkeley Univ.)
- CAN (2001, ICIR/Berkeley Univ.)
- Pastry (2001, Rice Univ.)
- Tapestry (2001, Berkeley Univ.)
- Kademlia (2002, New York Univ.)

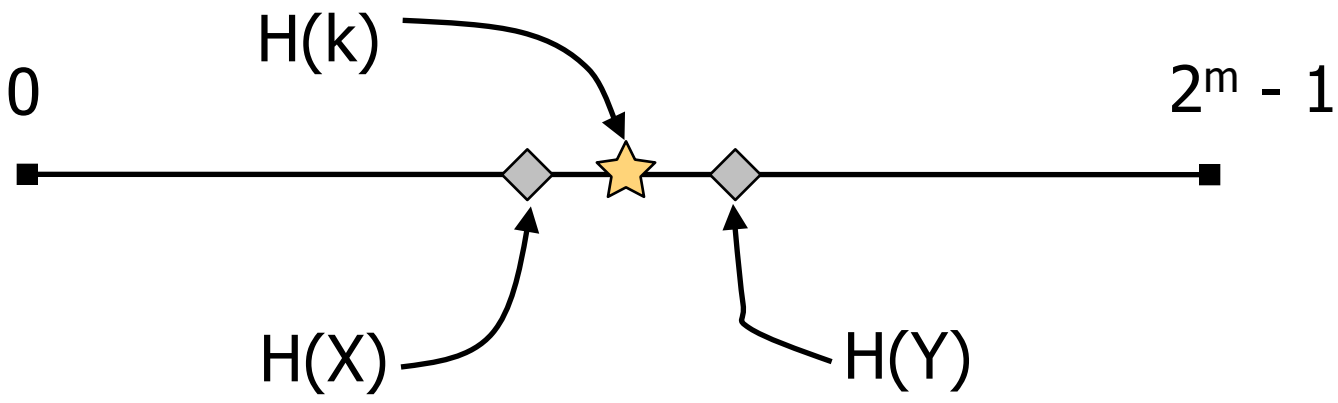
Chord

- La plus populaire scientifiquement
- Peu utilisée en pratique

Article Chord :
Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger,
M. Frans Kaashoek, Frank Dabek, Hari Balakrishnan,
Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications.
IEEE/ACM Transactions on Networking
http://pdos.csail.mit.edu/chord/papers/paper-ton.pdf

Hash-code des objets et des nœuds

- H(k) : entier à m chiffres (k clé d'objet)
- H(X) : entier à m chiffres (X identifiant de nœud)
- La fonction de hachage est « consistante »



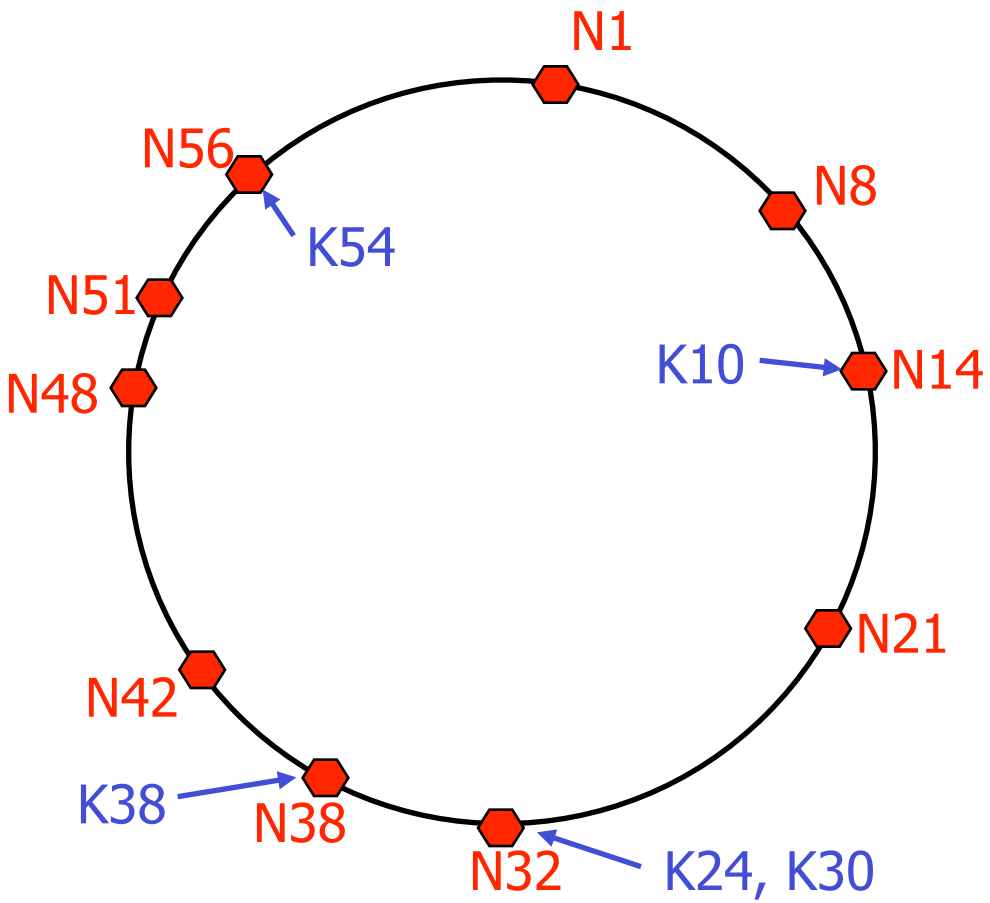
⟨k, v⟩ stocké sur Y

Notion de hachage consistant

- Propriétés :
  - Répartition uniforme des clés
  - Robustesse par ajout/suppression de nœud
- Pour N nœuds et K clés :
  - Nœud responsable d'au plus $(1+\varepsilon)K/N$ clés
  - $O(K/N)$ clés sont relocalisées, et seulement vers (depuis) le nœud qui arrive (part)
- Pour Chord : $\varepsilon = O(\log N)$

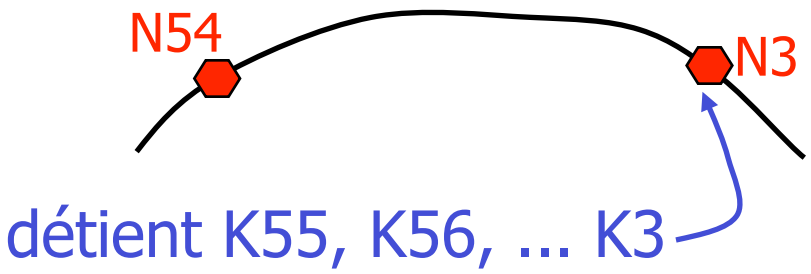
L'anneau Chord

m=6



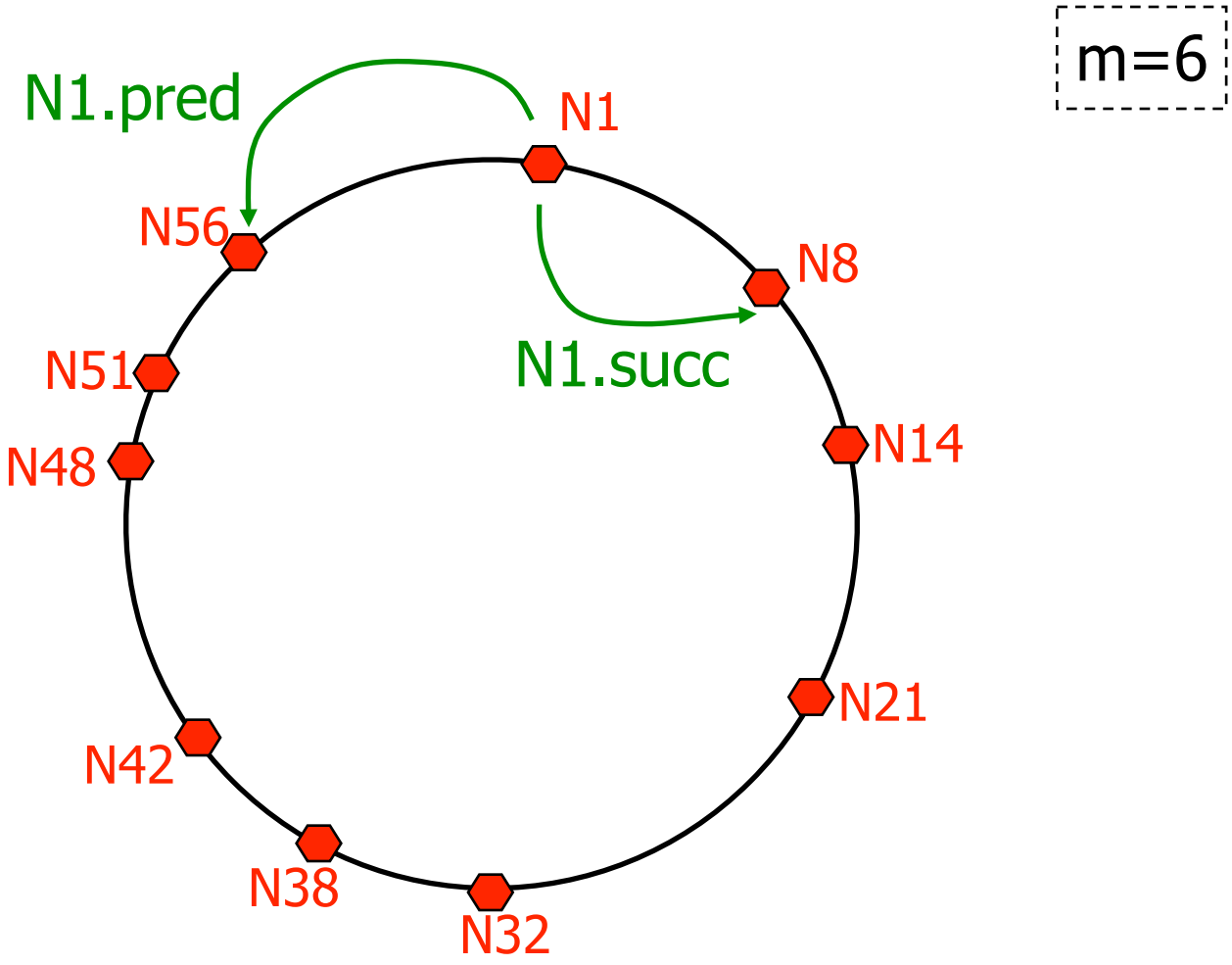
Principe

- On considère les nœuds X, Y tels que Y succède à X (sens des aiguilles d'une montre)
- Le nœud Y est responsable des clés k telles que H(k) ∈ (H(X), H(Y)]



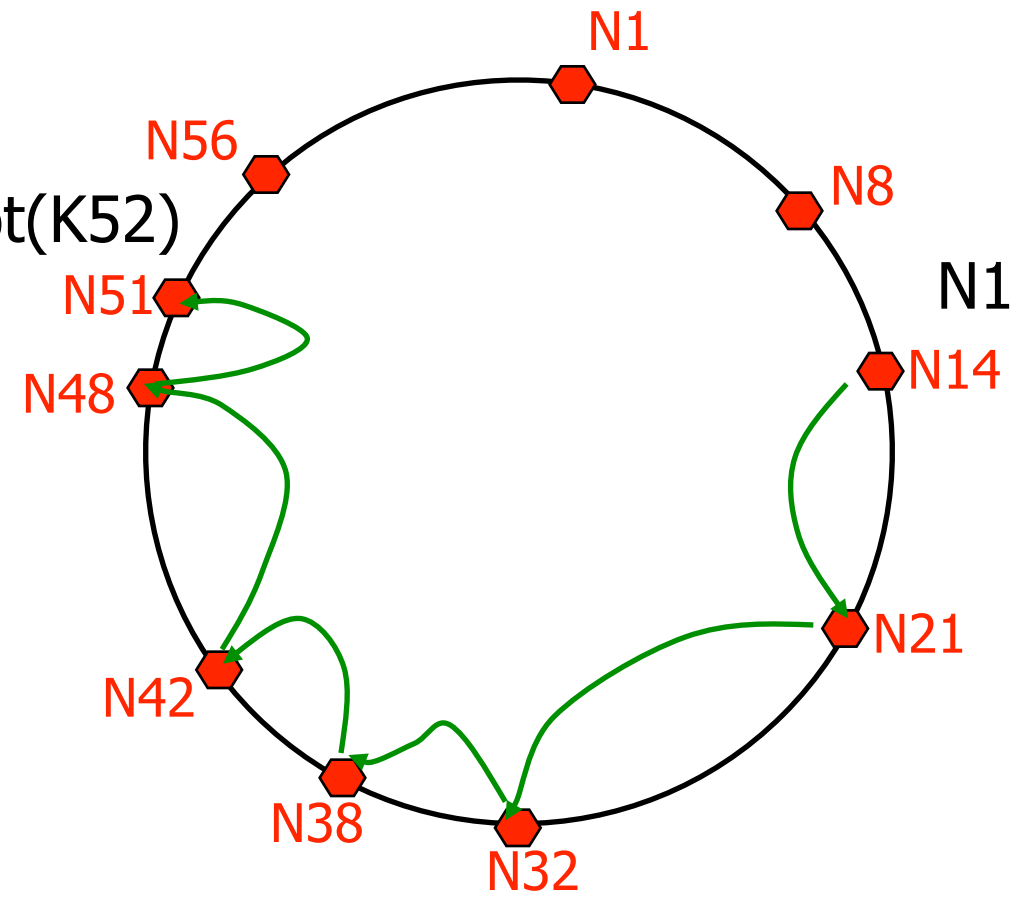
On utilise les valeurs de hash-code...
e.g., N54 est le nœud dont l'identifiant est associé au code 54.

Liens succ et pred



Localisation avec les liens succ

m=6



N51.find_root(K52)
=N56

N14.find_root(K52)

Code

- **X.find_root(k)**

```
if k in (X, succ]
   return succ
else
   return succ.find_root(k);
```

Notation: utilisation des hash-codes

- **X.find_root(k)**

```
if k in (X, succ]
   return succ
else
   return succ.find_root(k);
```

devrait s'écrire

- **X.find_root(k)**

```
if H(k) in (H(X), H(succ)]
   return succ
else
   return succ.find_root(k);
```

Consultation de données

- **X.DHTlookup(k)**

```
Y := find_root(k);
S := Y.lookup(k);
return S;
```

- **Y.lookup(k)**
  renvoie les valeurs locales (sur Y) associées à la clé k

Une autre version

combine la localisation et la consultation; prévient le chaînage de returns

- **X.DHTlookup(k)**

```
X.find_root(k, X);
wait[ans(k, S)]: return S;
```

- **X.find_root(k, Y)**

```
if k in (X, succ]
   Y.ans(k, succ.lookup(k))
else
   succ.find_root(k, Y);
```

Insertion de données

- **`X.DHTinsert(k, v)`**
  ```
  Y := find_root(k);
  Y.insert(k, v);
  ```
- **`Y.insert(k, v)`**
  ajoute ⟨k, v⟩ à la base de données locale

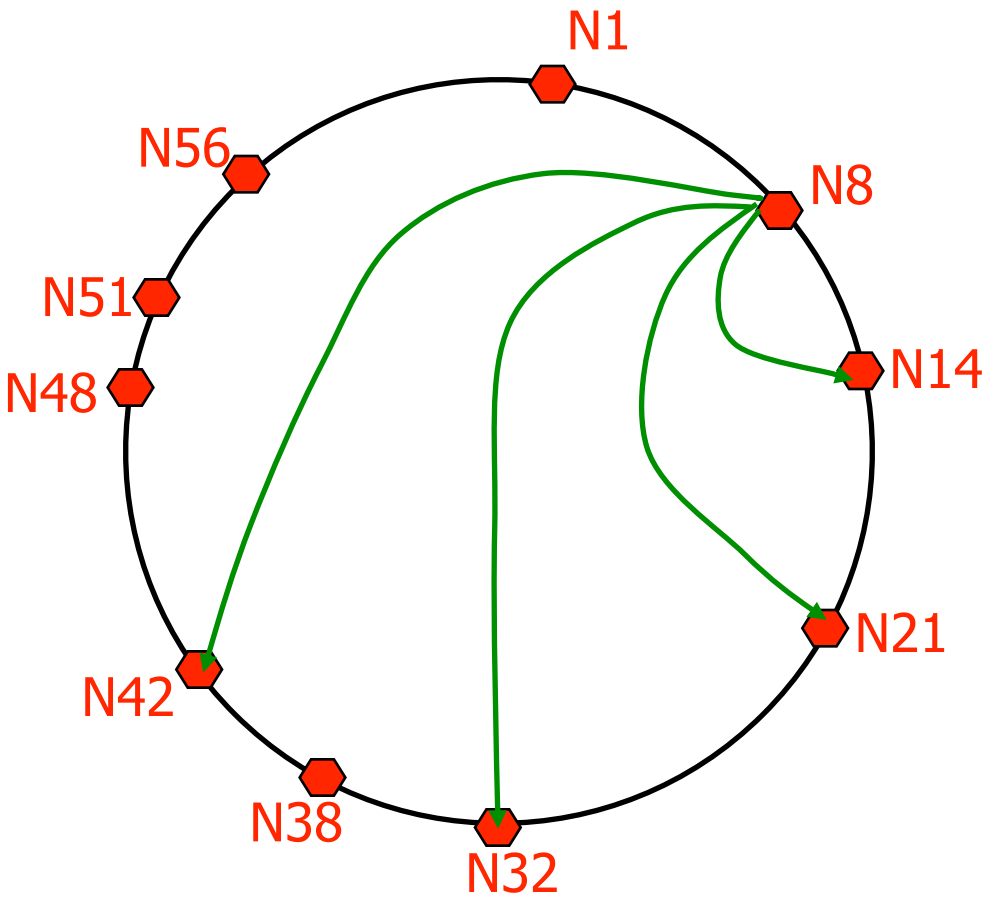
Une autre version (sans chaînage)

- **X.DHTinsert(k, v)**

```
if k in (X, succ]
   succ.insert(k,v)
else
   succ.DHTinsert(k, v);
```

Table des voisins

m=6

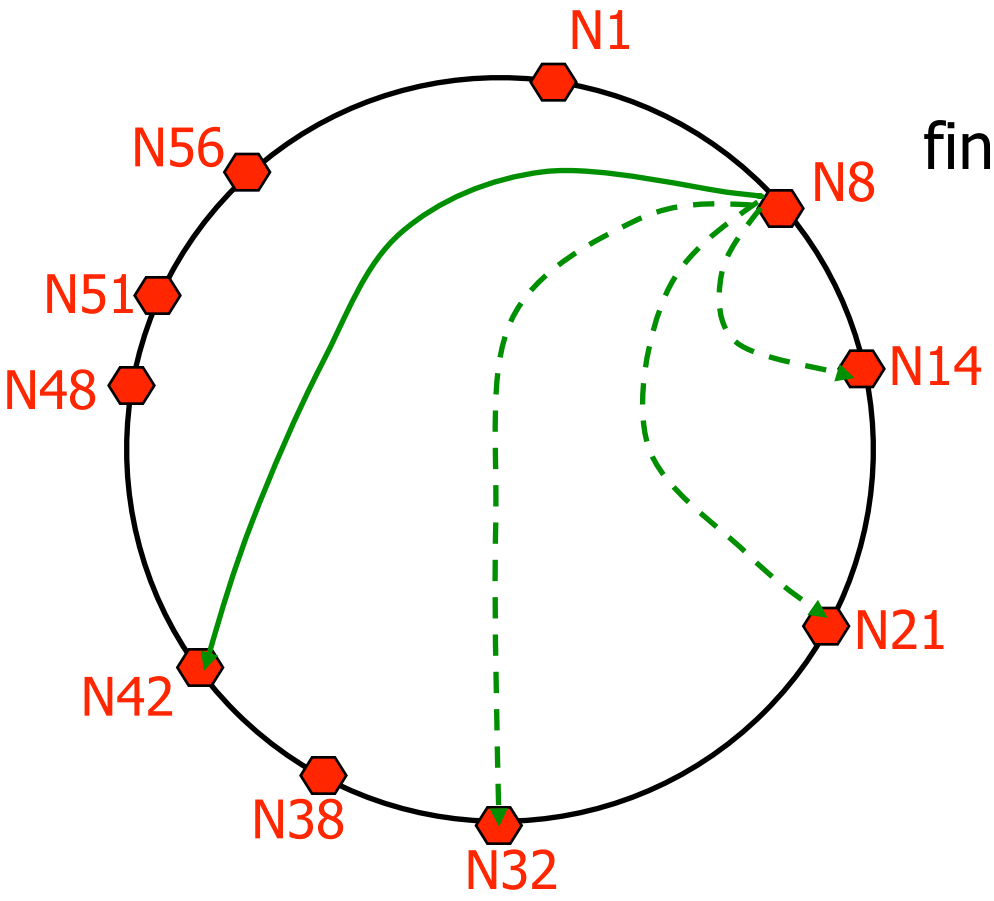


Finger Table pour N8

Indice	Hash-code	Succ
0	$8+2^0=9$	~~N14~~
1	$8+2^1=10$	~~N14~~
2	$8+2^2=12$	N14
3	$8+2^3=16$	N21
4	$8+2^4=24$	N32
5	$8+2^5=40$	N42

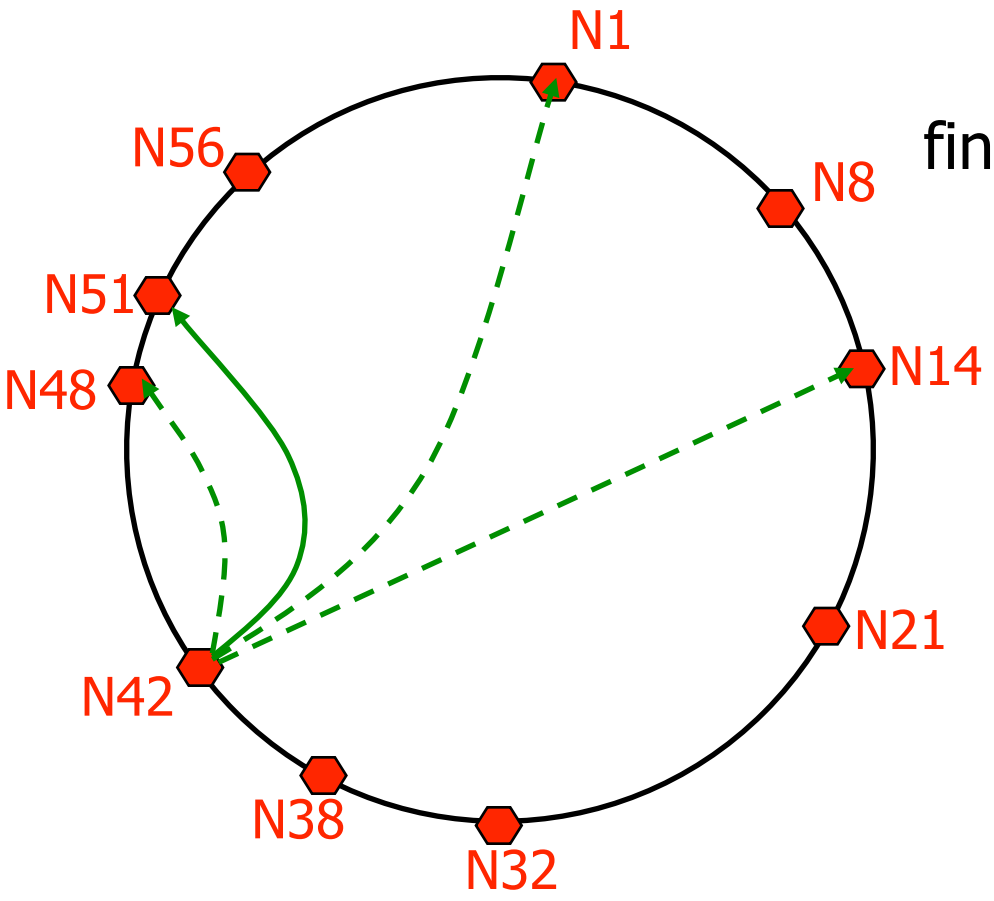
Racine à distance 1/2, 1/4, 1/8, 1/16...

Exemple



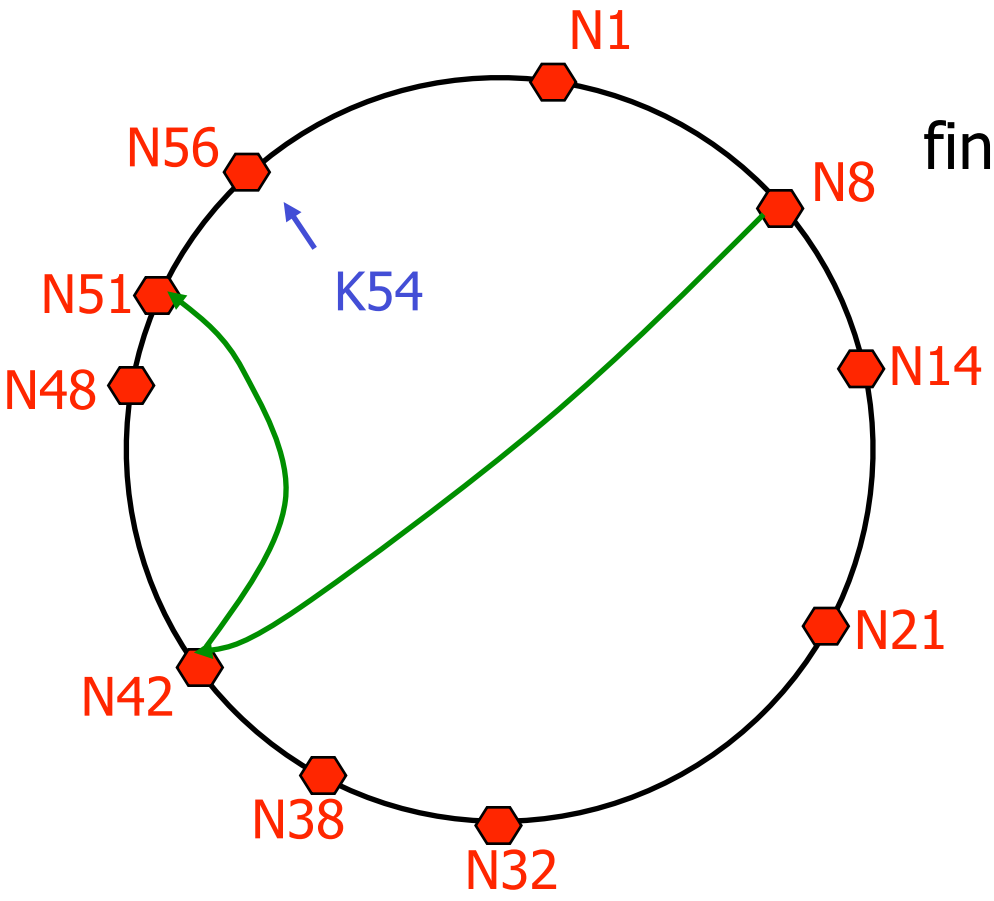
find_root(K54)

Exemple (suite)



find_root(K54)

Exemple (suite)



Code

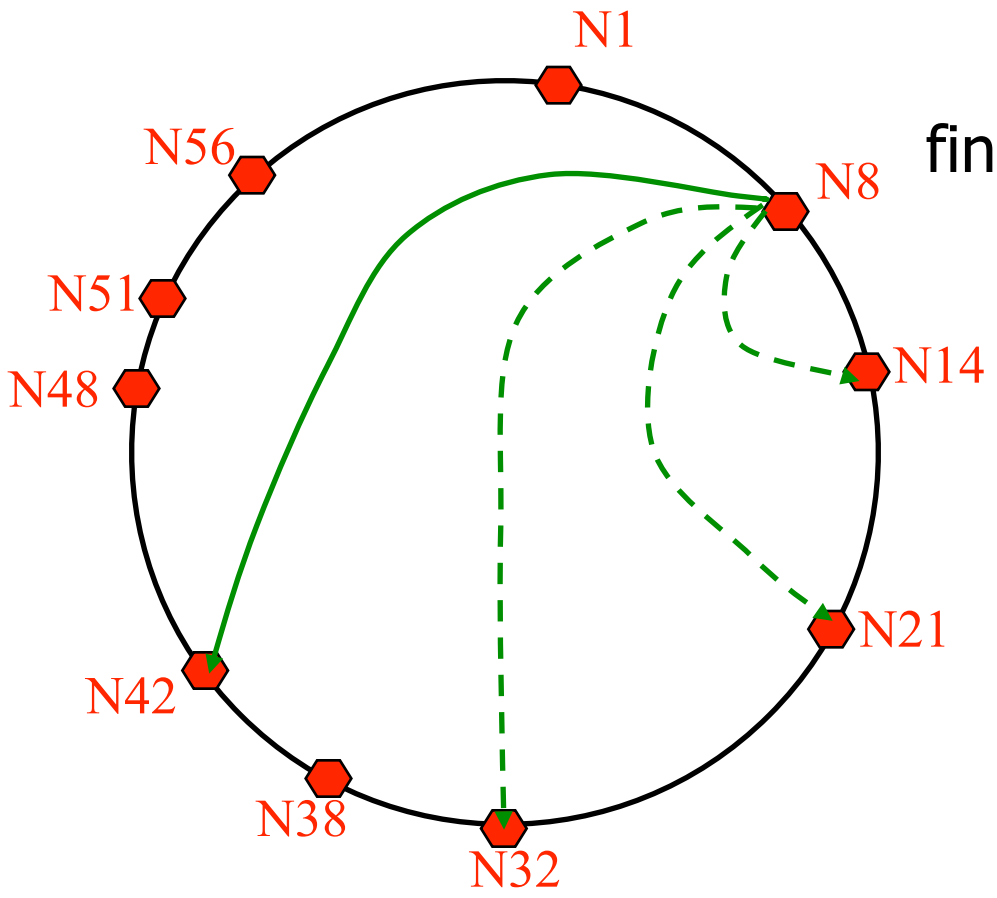
- **X.find_root(id)**

```
if id in (X, succ] return succ
else
    Y := closest_preceed(id);
    return Y.find_root(id);
```

- **X.closest_preceed(id)**

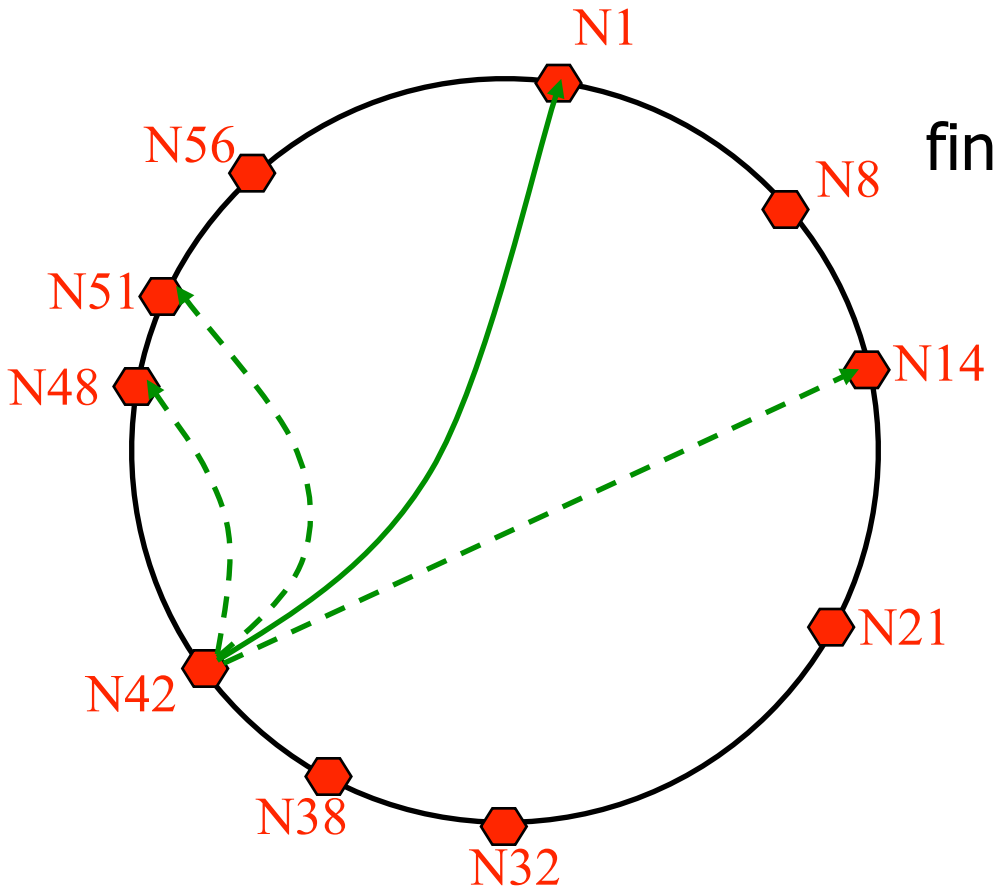
```
for i := m downto 1
    if finger[i] in (X,id)
        return finger[i];
return X;
```

Un autre exemple



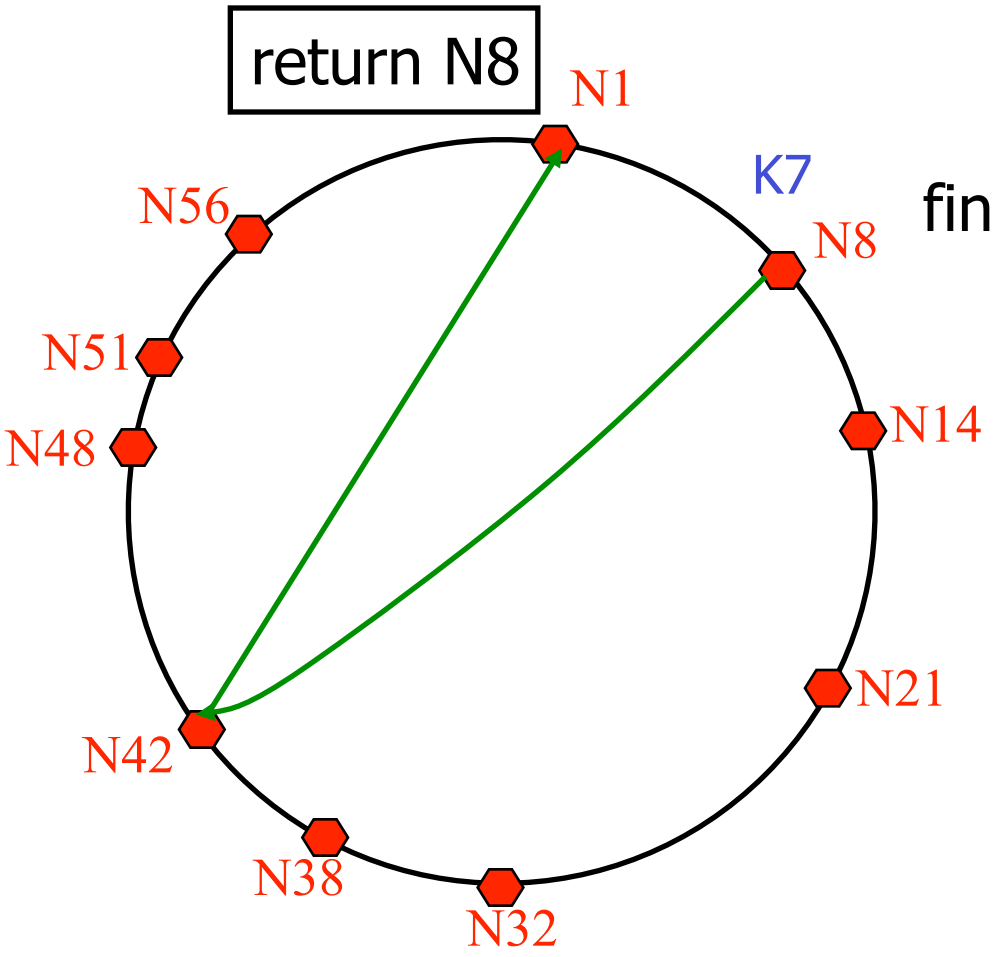
find_root(K7)

Un autre exemple (suite)



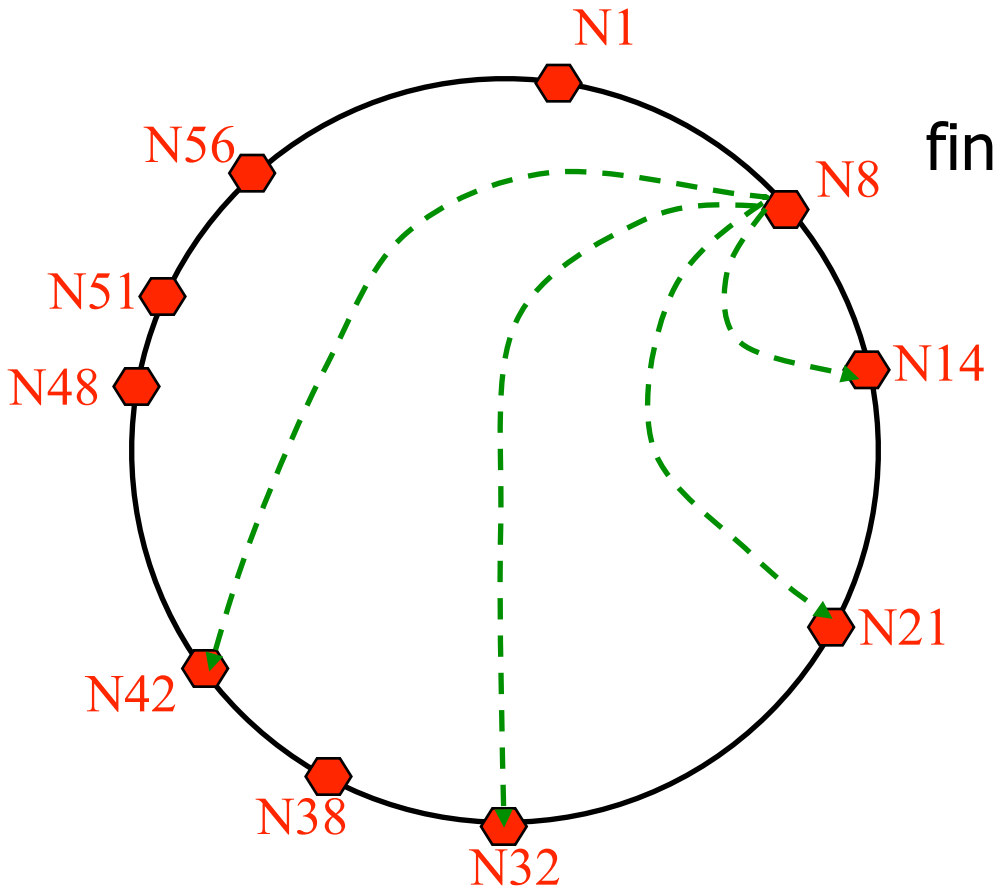
find_root(K7)

Un autre exemple (suite)



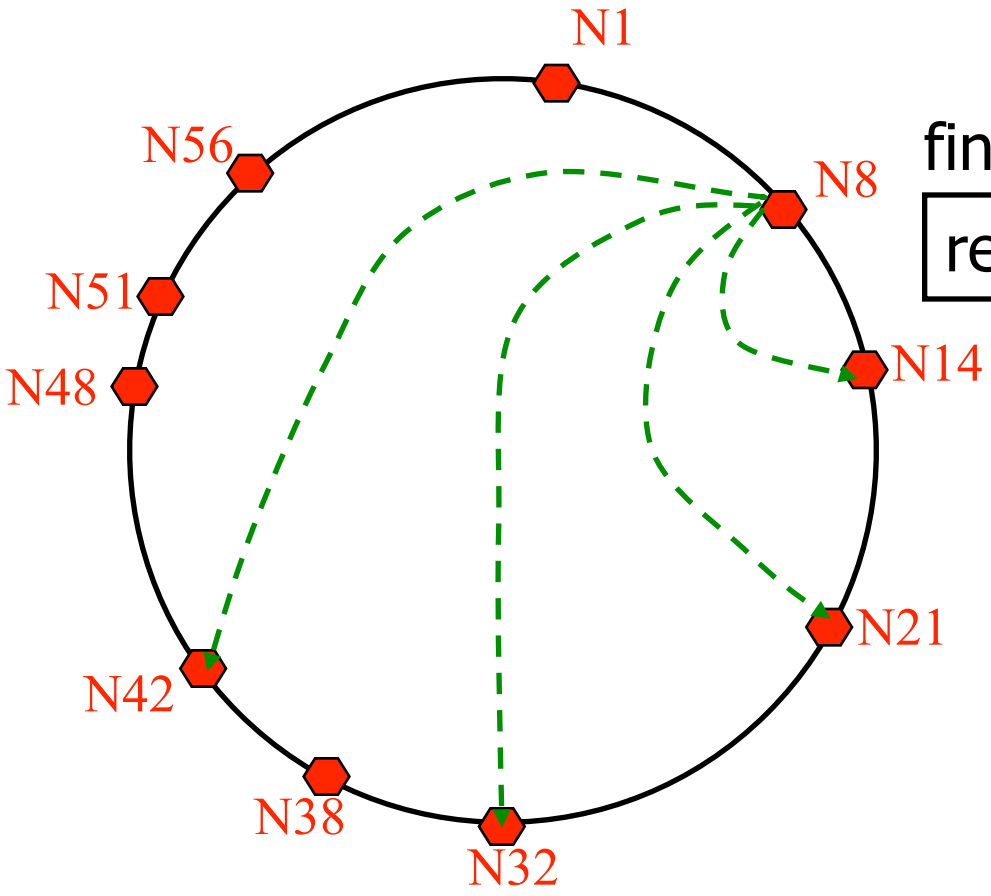
find_root(K7)

Encore un exemple



find_root(K8)

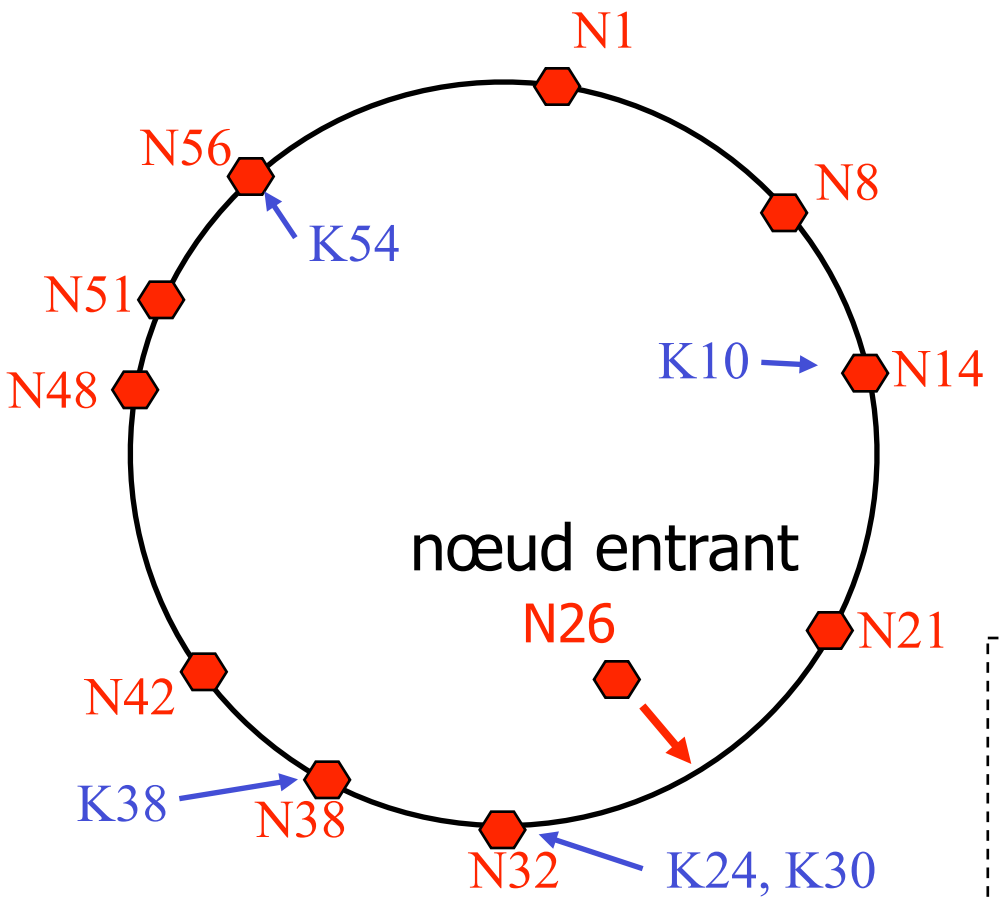
Encore un exemple (suite)



find_root(K8)

return N8

Insertion d'un nœud dans l'anneau



À faire :

1. maj des liens
2. déplacement des données

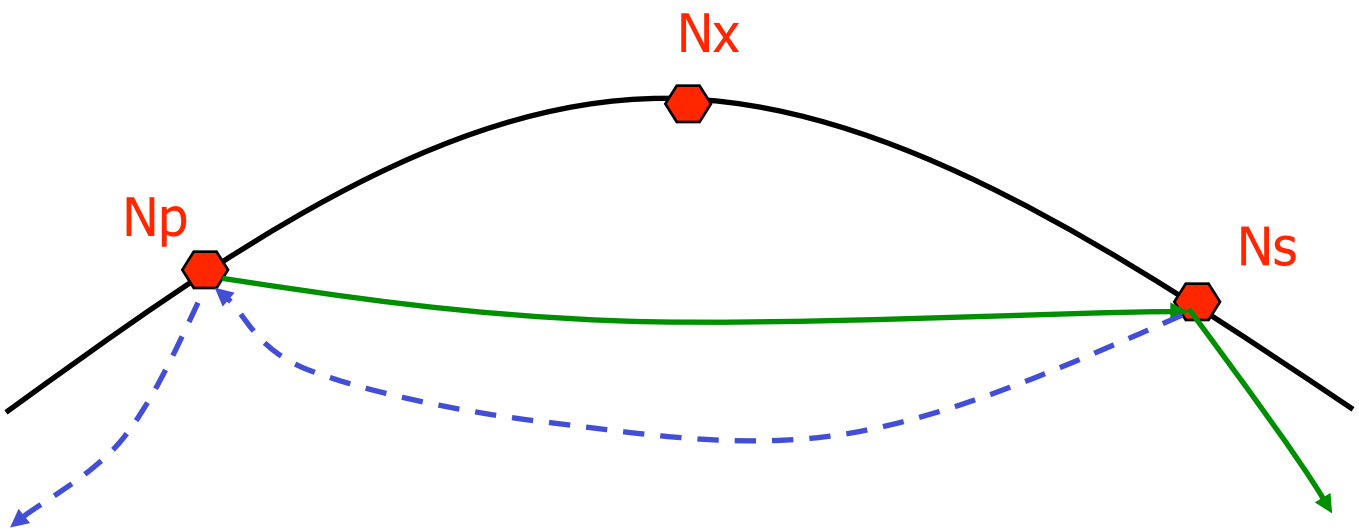
Pour l'heure, on considère que les nœuds ne meurent jamais

Un nouveau nœud X rejoint l'anneau

- Le nœud Y est réputé appartenir à l'anneau
- **X.join(Y)**

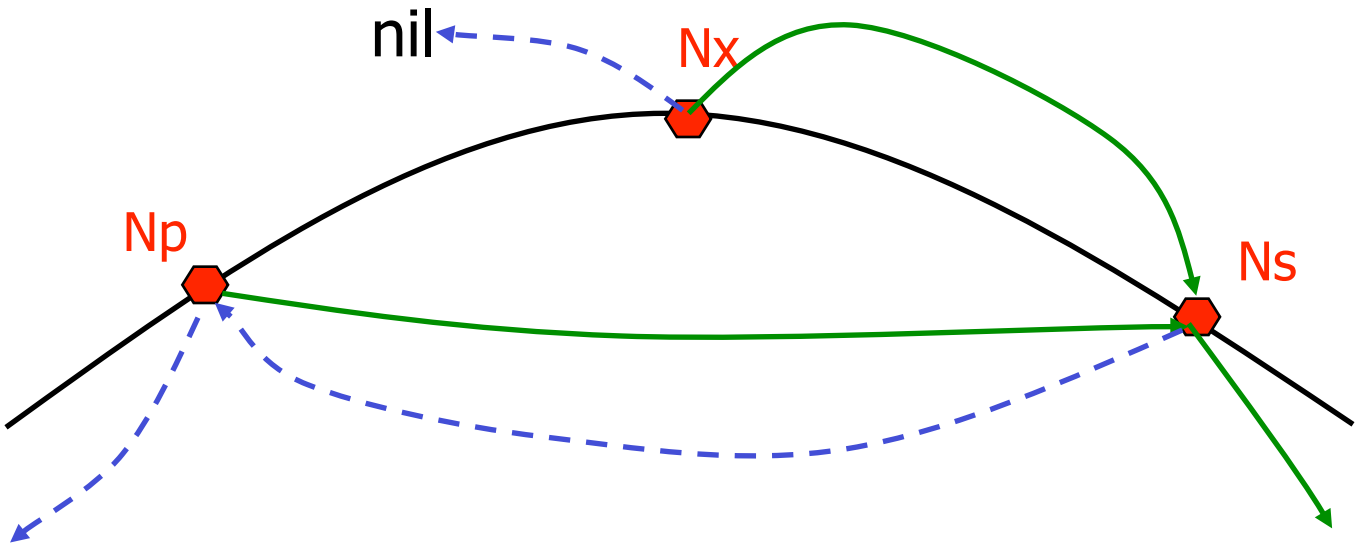
```
pred := nil;
succ := Y.find_root(X);
```

Exemple d'insertion de nœud



Example d'insertion (suite)

après le join :



Stabilisation périodique

- **X.stabilize()**

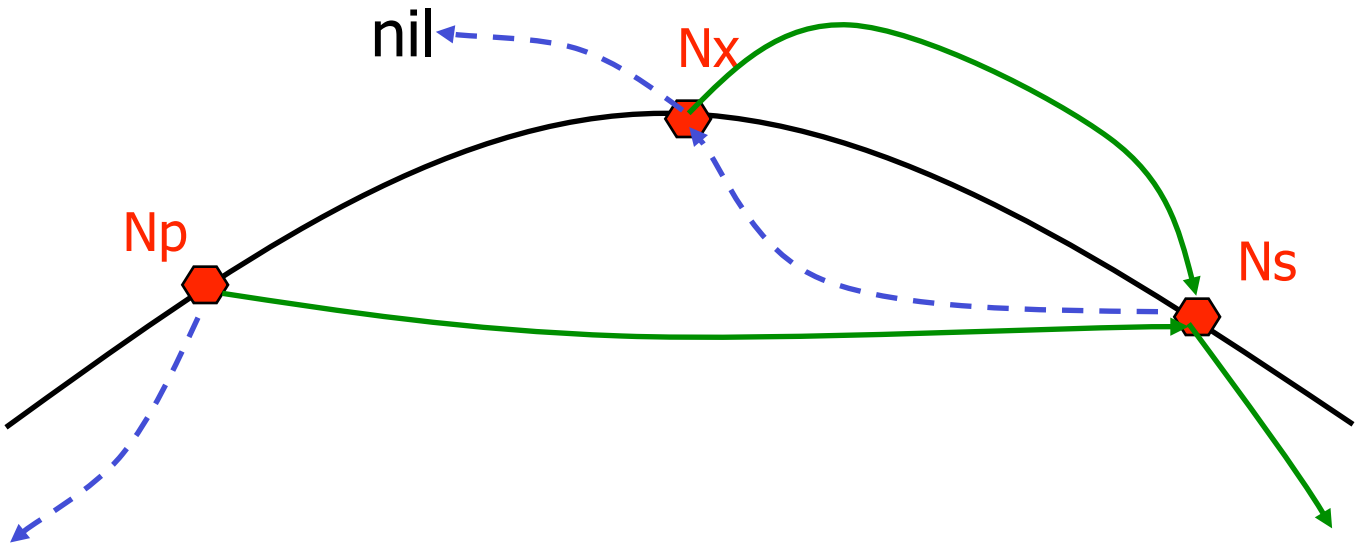
```
Y := succ.pred;
if Y in (X, succ)  succ := Y;
succ.notify(X);
```

- **X.notify(Z)**

```
if pred=nil OR Z in (pred, X)
   pred := Z;
```

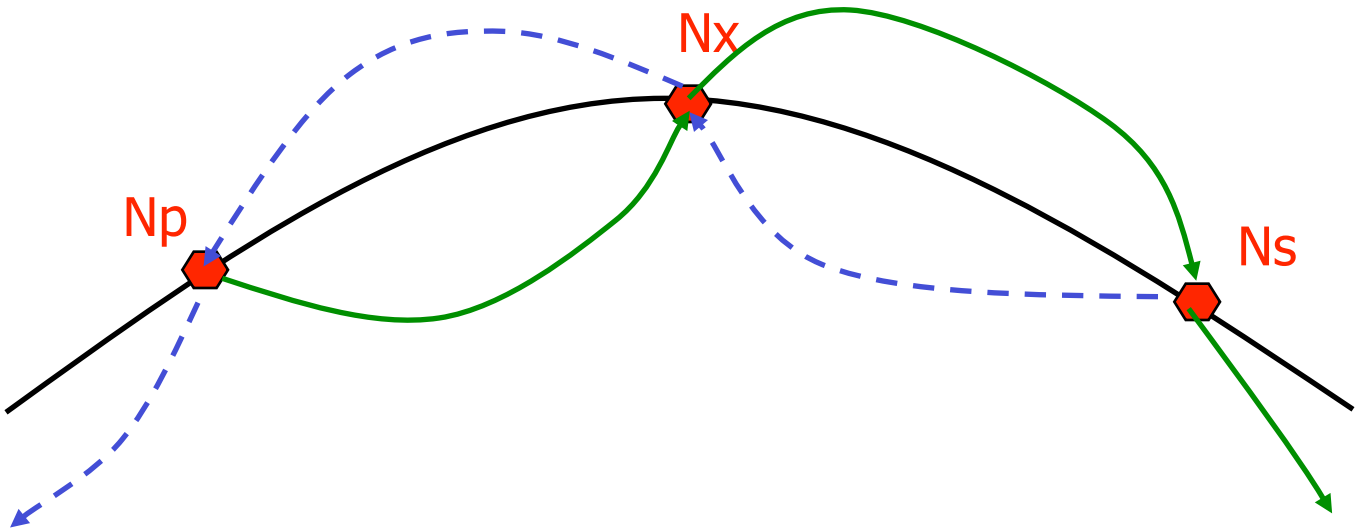
Exemple d'insertion (suite)

après Nx.stabilize : Y= Np



Exemple d'insertion (suite)

après Np.stabilize : Y= Nx

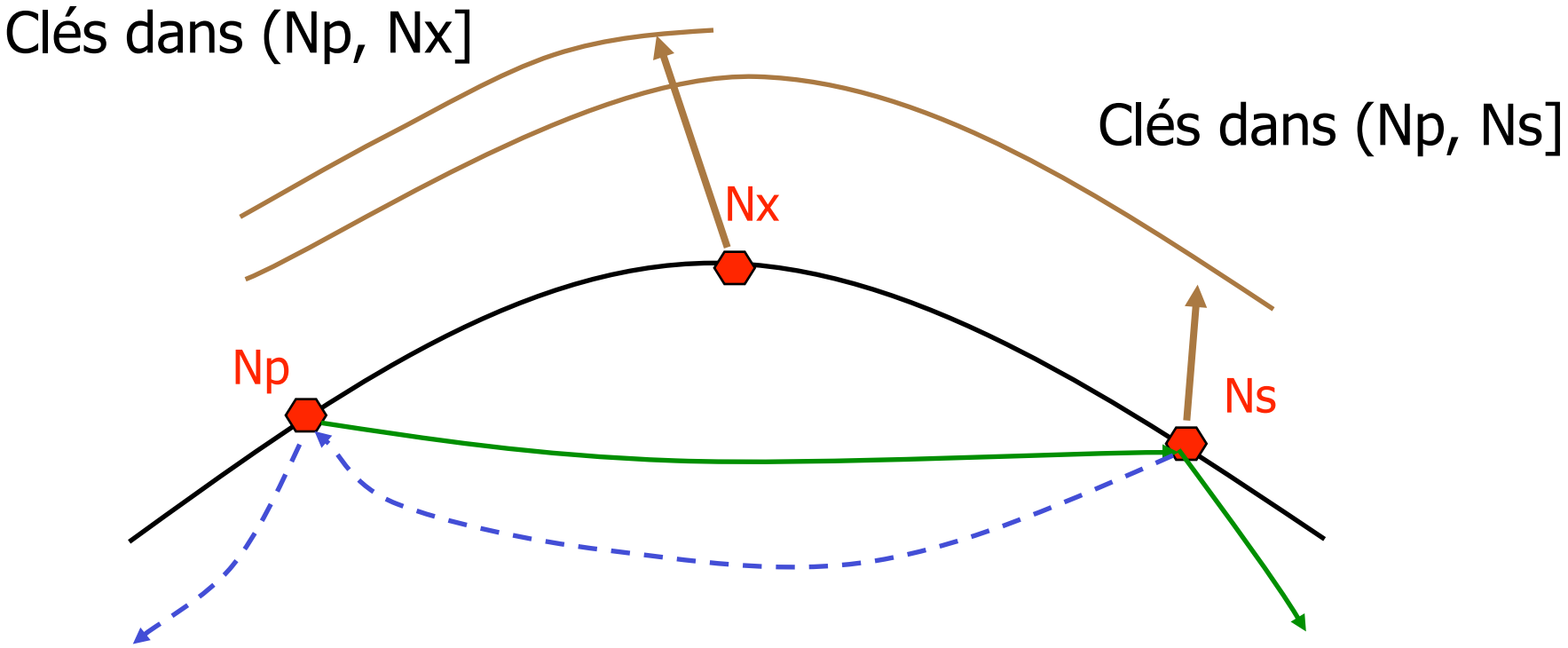


Vérification périodique des voisins

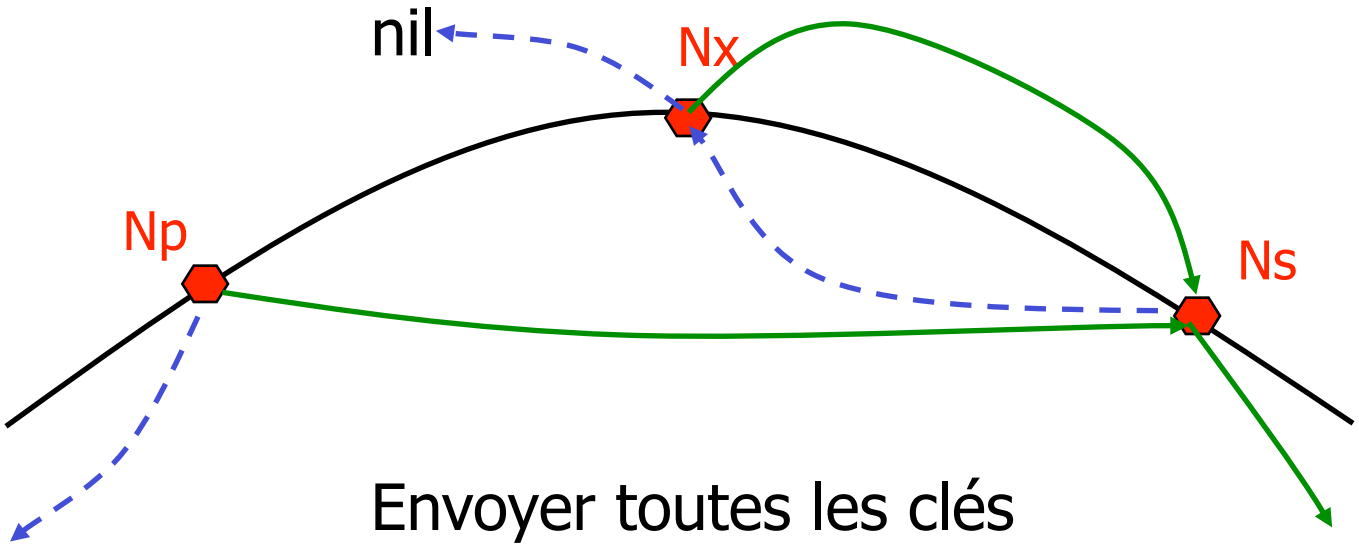
- **X.fix_fingers()**

```
next := next + 1;
if next > m
   next := 1;
finger[next] :=
                find_root(X+2^(next-1))
```

Quand déplacer les données ? ...



… après Ns.notify(Nx)



Envoyer toutes les clés
dans l'intervalle (Np, Nx]
lorsque Ns.pred est mis à jour

Stabilisation périodique

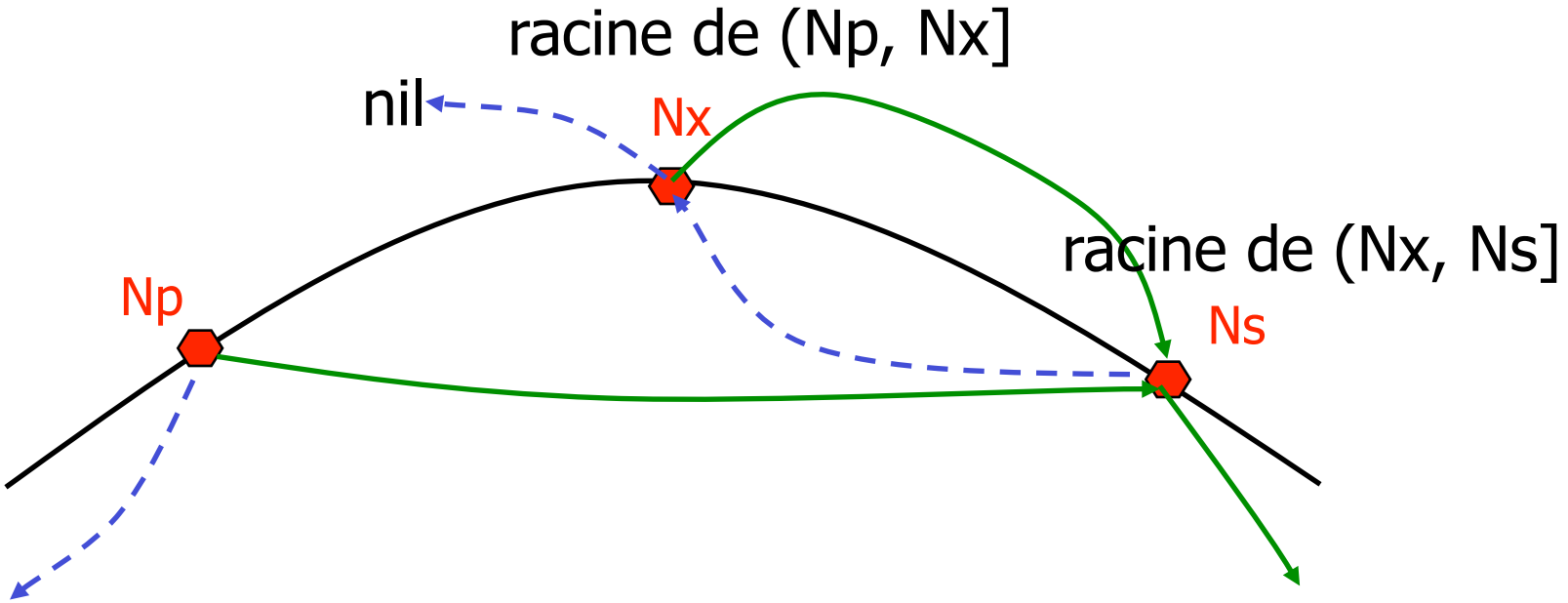
- **X.notify(Z)**

```
if pred=nil OR Z in (pred, X)
   Z.give(data in (pred ,Z] )
pred := Z
   X.remove(data in (pred ,Z] )
```

Question : Quelles données sont déplacées si pred=nil ?

N.B.: on ignore ici les problèmes de concurrence d'accès, e.g., que deviennent les consultations lorsque l'on est en train de déplacer les données ?

La consultation peut être menée sur le mauvais nœud !



Recherche à partir d'une clé k dans (Np, Nx] dirigée vers Ns !

Consultation de données (révisée)

- **X.DHTlookup(k)**

```
ok := false
while not ok do
   Y := find_root(k)
   [ok, S] := Y.lookup(k)
return S
```

- **Y.lookup(k)**

```
if k in (pred, Y]
   return [true, local values
for k]
else return [false, {}]
```

Insertion de données (révisée)

- **X.DHTinsert(k, v)**

```
ok := false
while not ok do
   Y := find_root(k)
   ok := Y.insert(k, v)
```

- **Y.insert(k, v)**

```
if k in (pred, Y]
   insert [k,v] in local storage
   return true
else return false
```

Pourquoi ça marche ?

- Les liens pred, succ sont finalement corrects
- Les données sont stockées sur les bons nœuds

la clé k est bien sur le nœud X, pour k dans l'intervalle (X.prev, X]

- Les routes accélèrent la localisation mais leur correction n'est pas source de problème

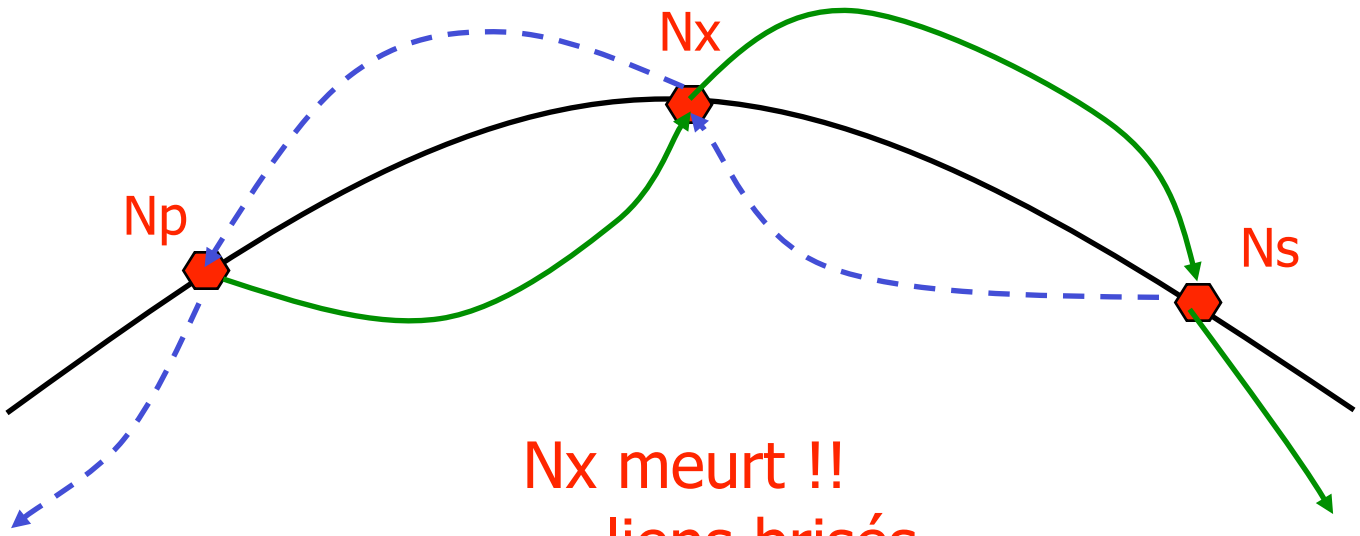
Résultats pour un système à N nœuds

- Avec une très forte probabilité, le nombre de nœuds à contacter pour localiser une racine est **O(log N)**
- Bien que la table des voisins contienne m entrées, seules **O(log N)** entrées sont nécessaires
- Les résultats expérimentaux montrent que le temps moyen de localisation d'une ressource est **(log N)/2**

Défaillance d'un nœud

Données de Nx:

k7	v7
k8	v8
k9	v9



Nx meurt !!
- liens brisés
- données perdues

Pour réparer les liens

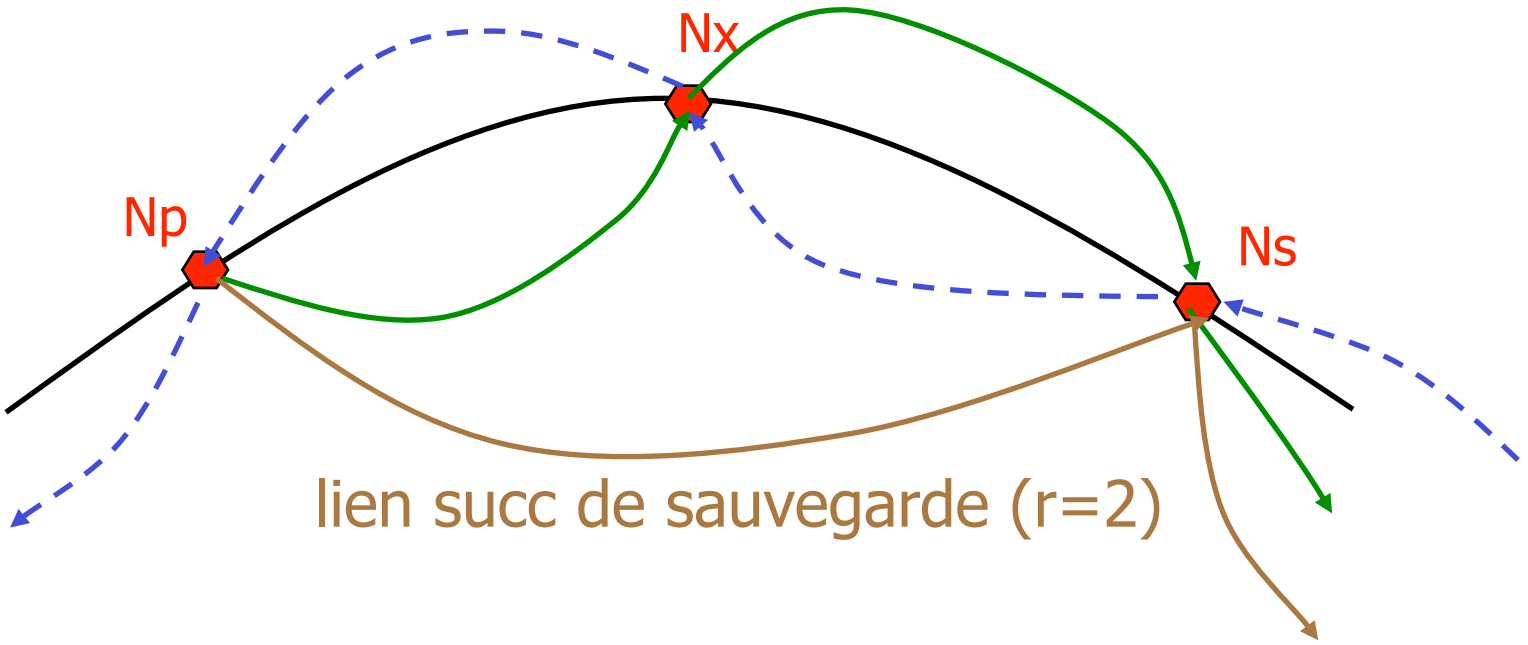
- **`X.check_pred()`**

```
if (pred has failed)
   pred := nil;
```

- Mais également, on maintient les liens vers **r = O(log N)** successeurs dans l'anneau

Exemple de défaillance

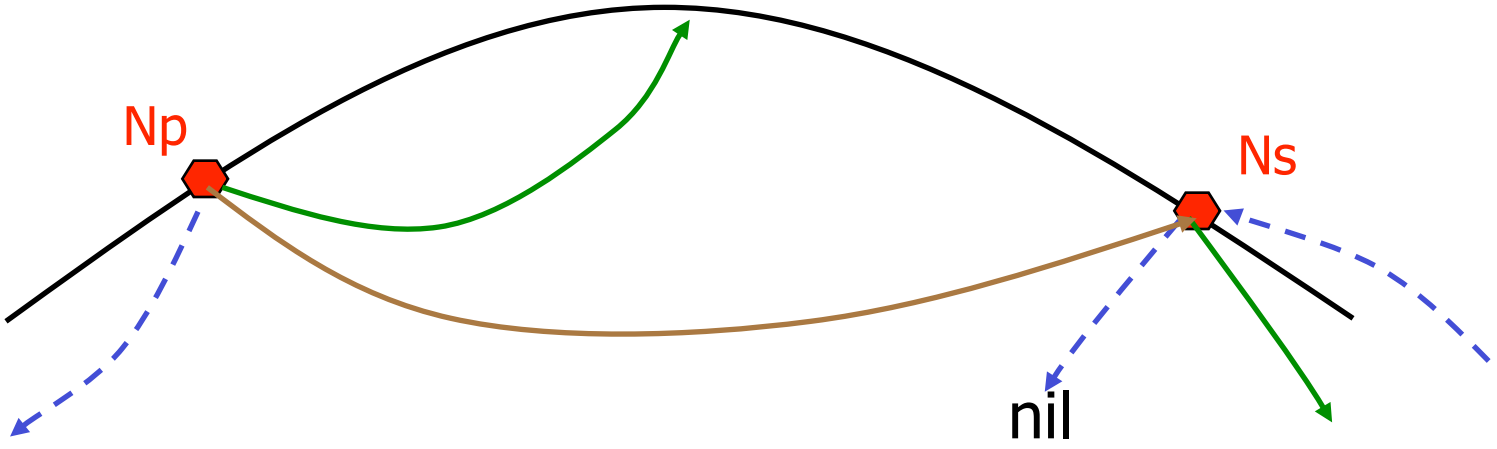
Au début...



Nx meurt...

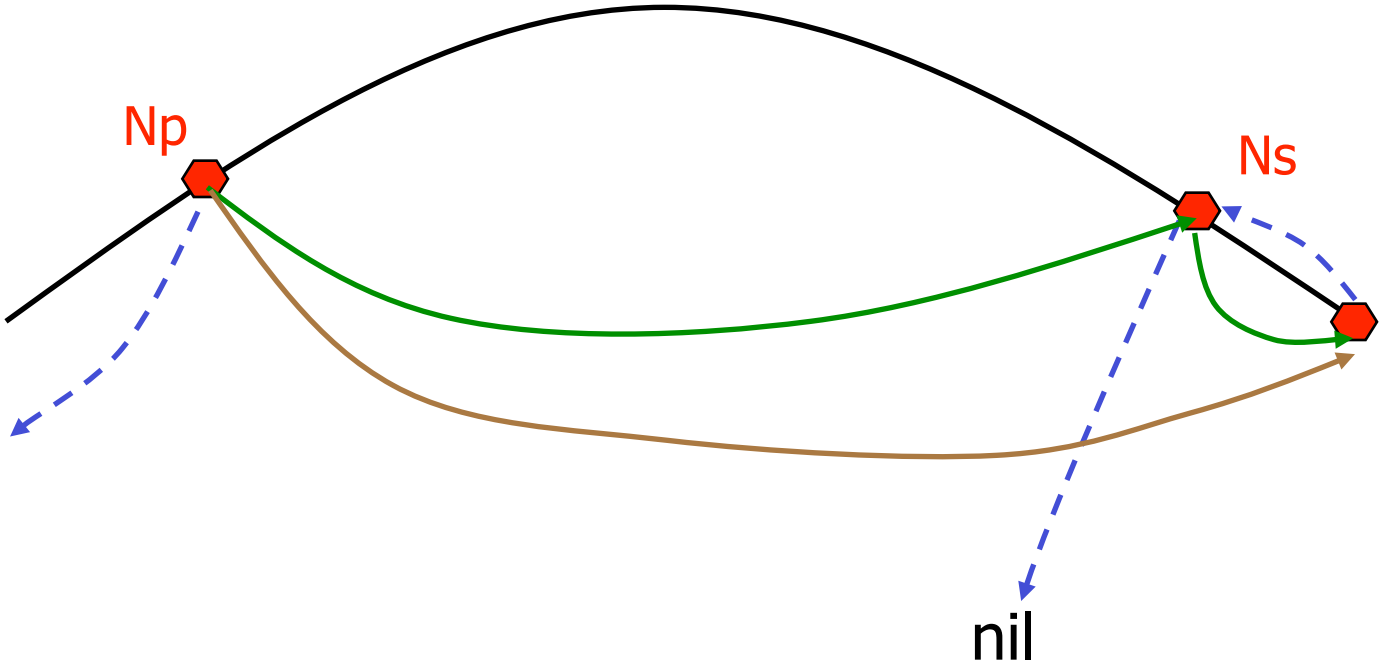
Exemple de défaillance (suite)

après Ns.check_pred...



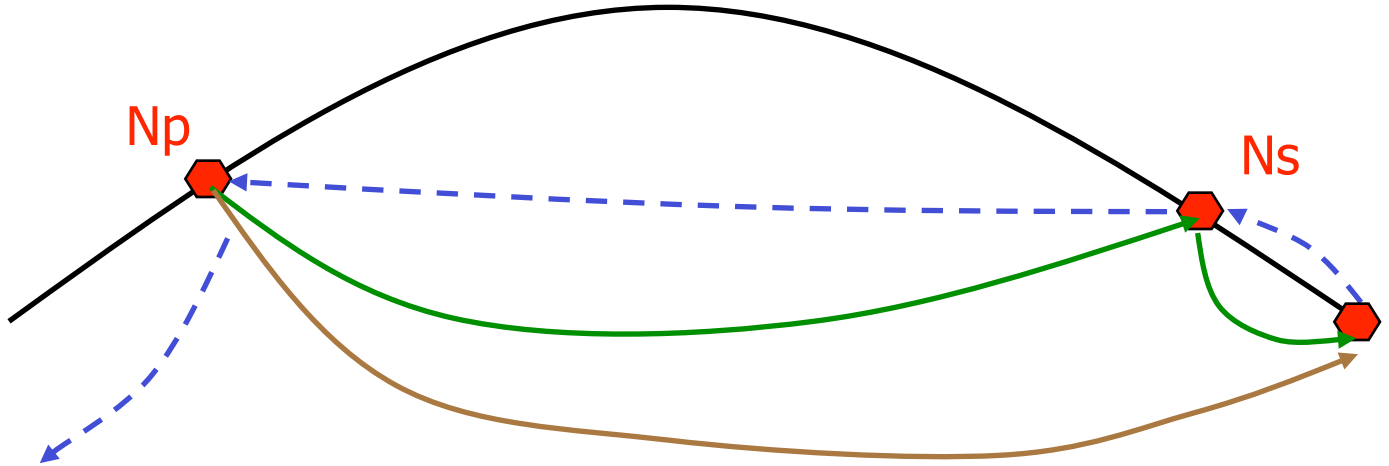
Exemple de défaillance (suite)

après que Np découvre l'absence de Nx...



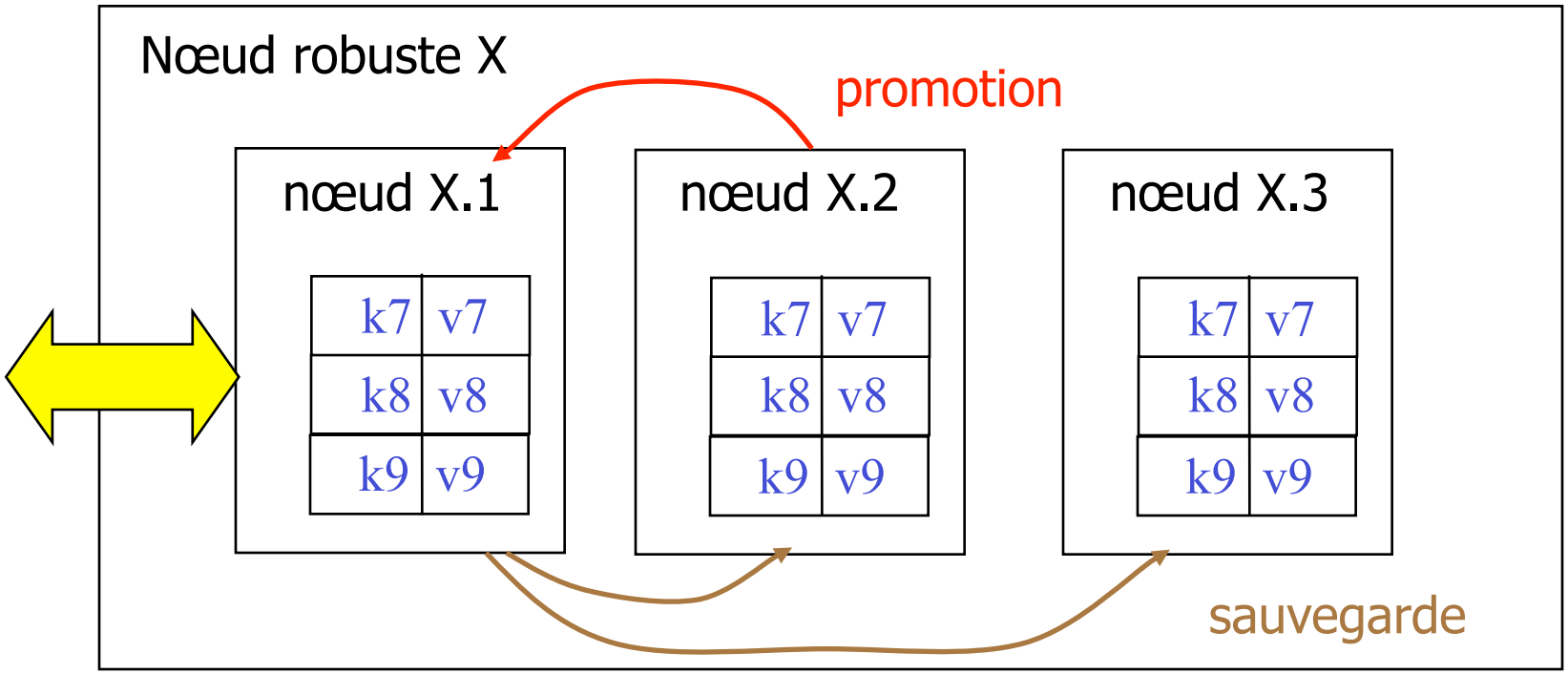
Exemple de défaillance (suite)

après stabilisation...



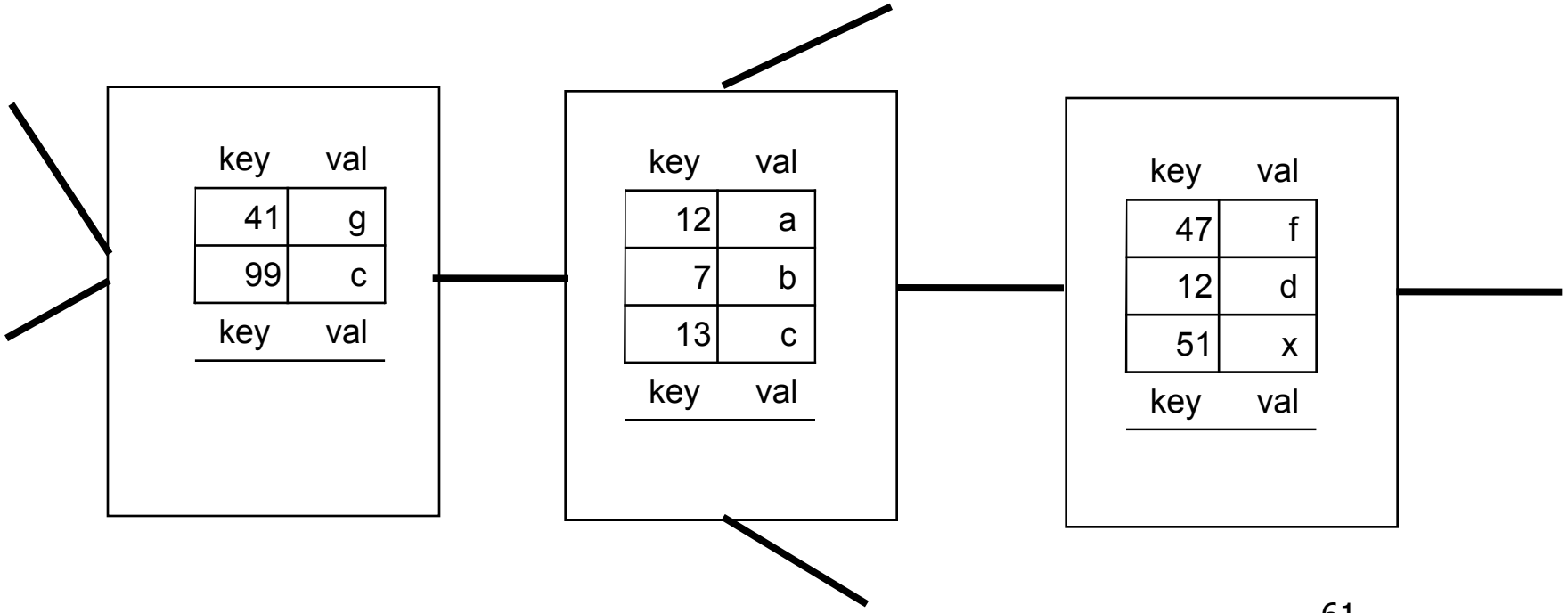
Prévention contre la perte de données

- Une idée : nœud robuste (réplication)



Recherche par voisinage (Gnutella)

- Chaque nœud « Servent » stocke ses propres données et recherche dans son voisinage (par inondation « flooding »)



Stockage des données

- **`X.DTinsert(k, v)`**
  insertion de (k,v) localement sur X

Consultation

- **X.DTlookup(k)**

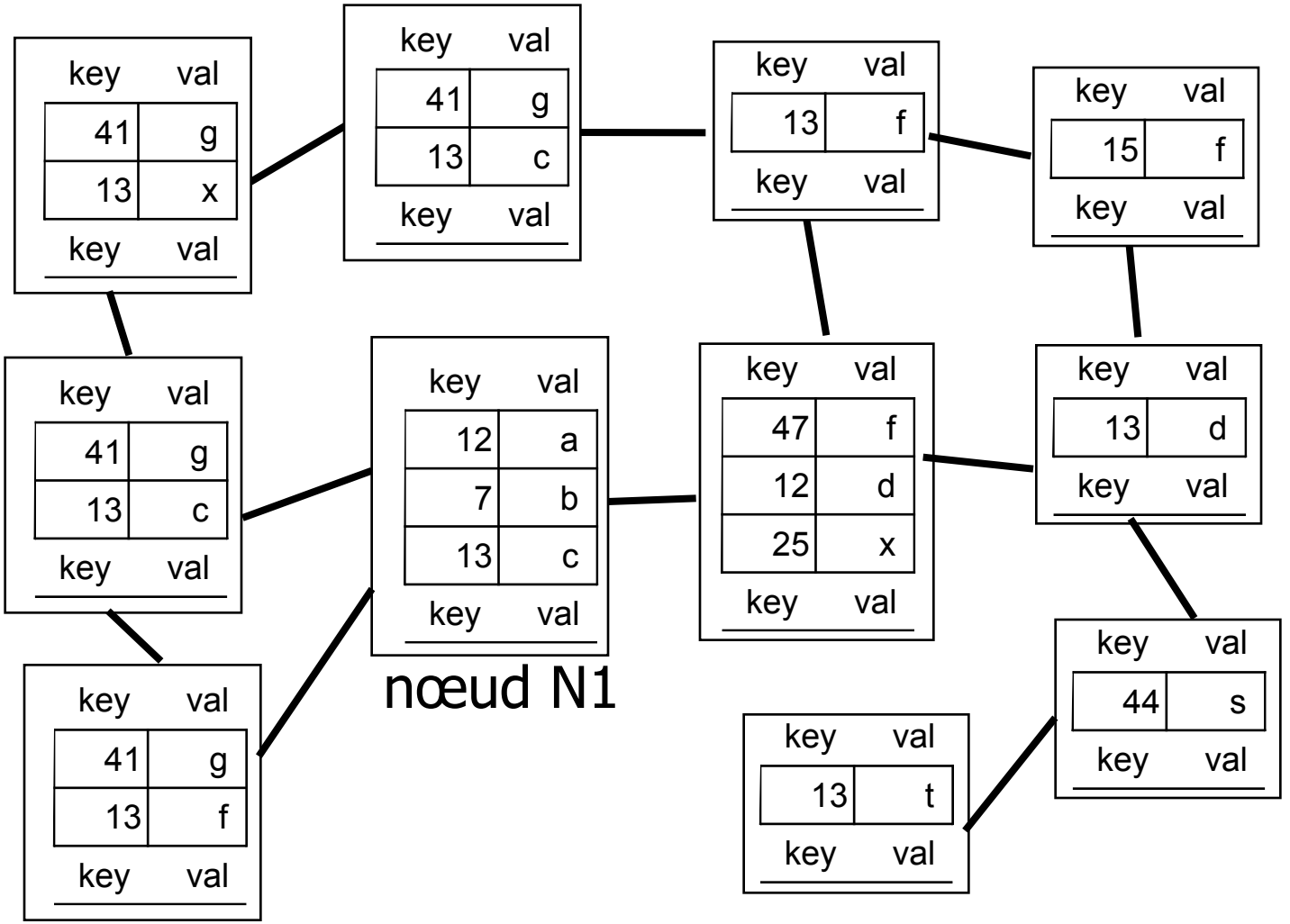
```
TTL := valeur choisie
return( X.find(k, TTL, X) )
```

- **X.find(k, TTL, Z)**

```
TTL := TTL - 1
S := paires <k,v_i>
if TTL > 0 then
   for all Y≠Z in X.neighbors do
        S:=S U Y.find(k, TTL, X)
return(S)
```

Exemple

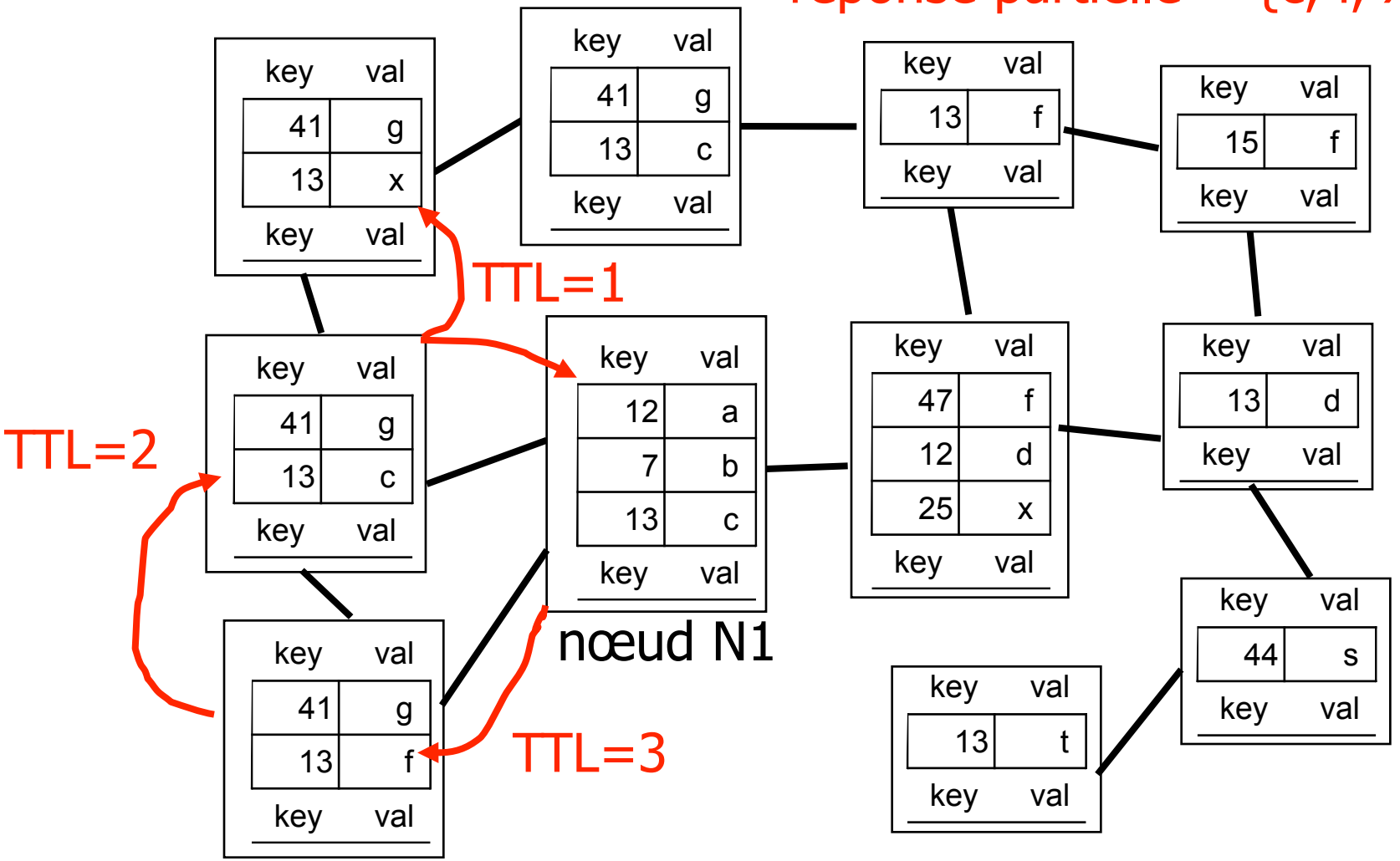
N1.DTlookup(13), TTL = 4



Exemple

N1.DTlookup(13), TTL = 4

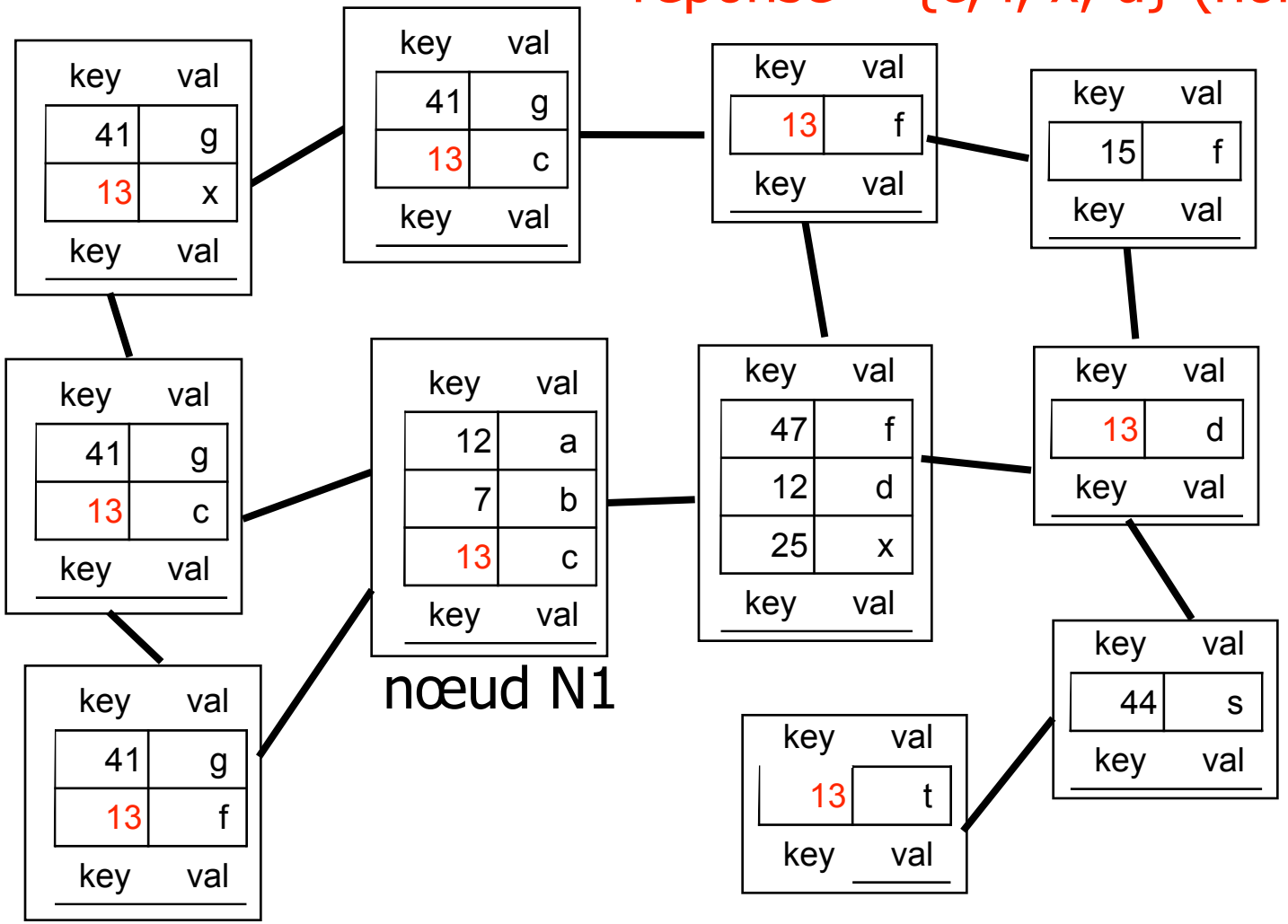
réponse partielle = {c, f, x}



Exemple

N1.DTlookup(13), TTL = 4

réponse = {c, f, x, d} (non t!)



Coût d'une diffusion dans Gnutella

- Np pairs, Nc liens par pair (v0.4=5)
- TTL=7 (v0.4)
- M = #messages propagés
  - $M=Nc + Nc^2 + Nc^3 + ... + Nc^{TTL}$

Nc	5	6	7	8	9	10
M	97K	336K	960K	2,5m	5,3m	11m

Coût d'un PING dans Gnutella /1

- Recherche de voisins
- Horizon = #pairs atteignables
  - Dans l'idéal : Horizon=M pairs
  - En réalité : Horizon << M (clustering)
- Émission du PONG :
  - 1er rang : Nc messages
  - 2ème rang : $Nc^2 \times 2$
  - $Mp = Nc + Nc^2 \times 2 + \ldots + Nc^{TTL} \times TTL$

Coût d'un PING dans Gnutella /2

- Coût total = M + Mp (PING & PONG)

Nc	5	6	7	8	9	10
M	97K	336K	960K	2,5m	5,3m	11m
Mp	660k	2,2m	6,5m	16m	36m	76m
M+Mp	756k	2,6m	7,5m	18,8m	42m	87m

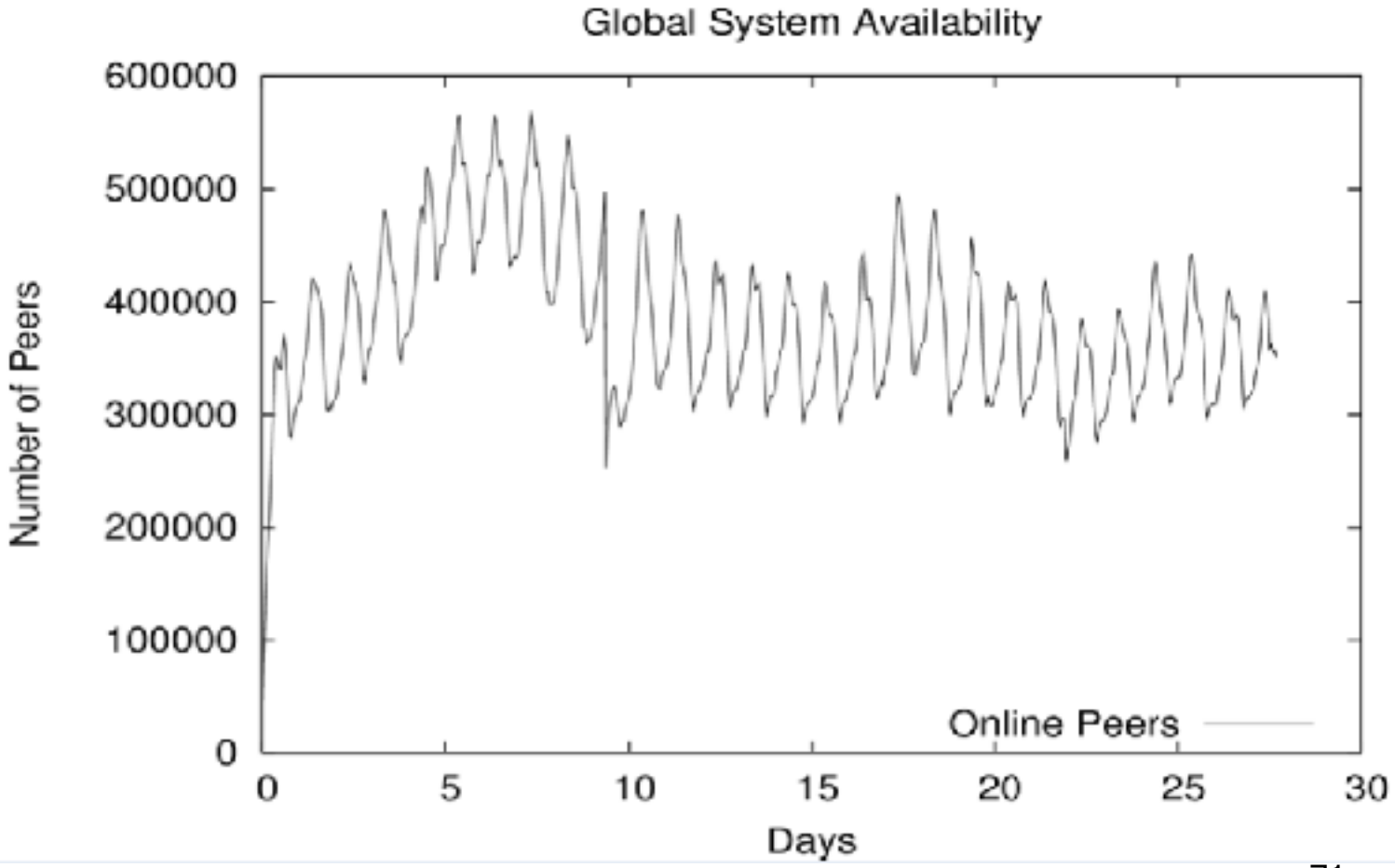
Coût en émission

- Hypothèses
  - 1 PING par pair/j : M+Mp
  - 3 QUERY par pair/j : 3 x M
  - Message = 100 octets
  - Distribution uniforme des messages dans le temps

Nc	5	6	7	8	9	10
Mo/jour	100	345	992	2481	5580	11538
kbps	9,7	33,6	96	240	541	1120
Horizon	756k	2,6m	7,5m	18,8m	42m	87m

À propos de distribution

- Présence des pairs sur un réseau eDonkey

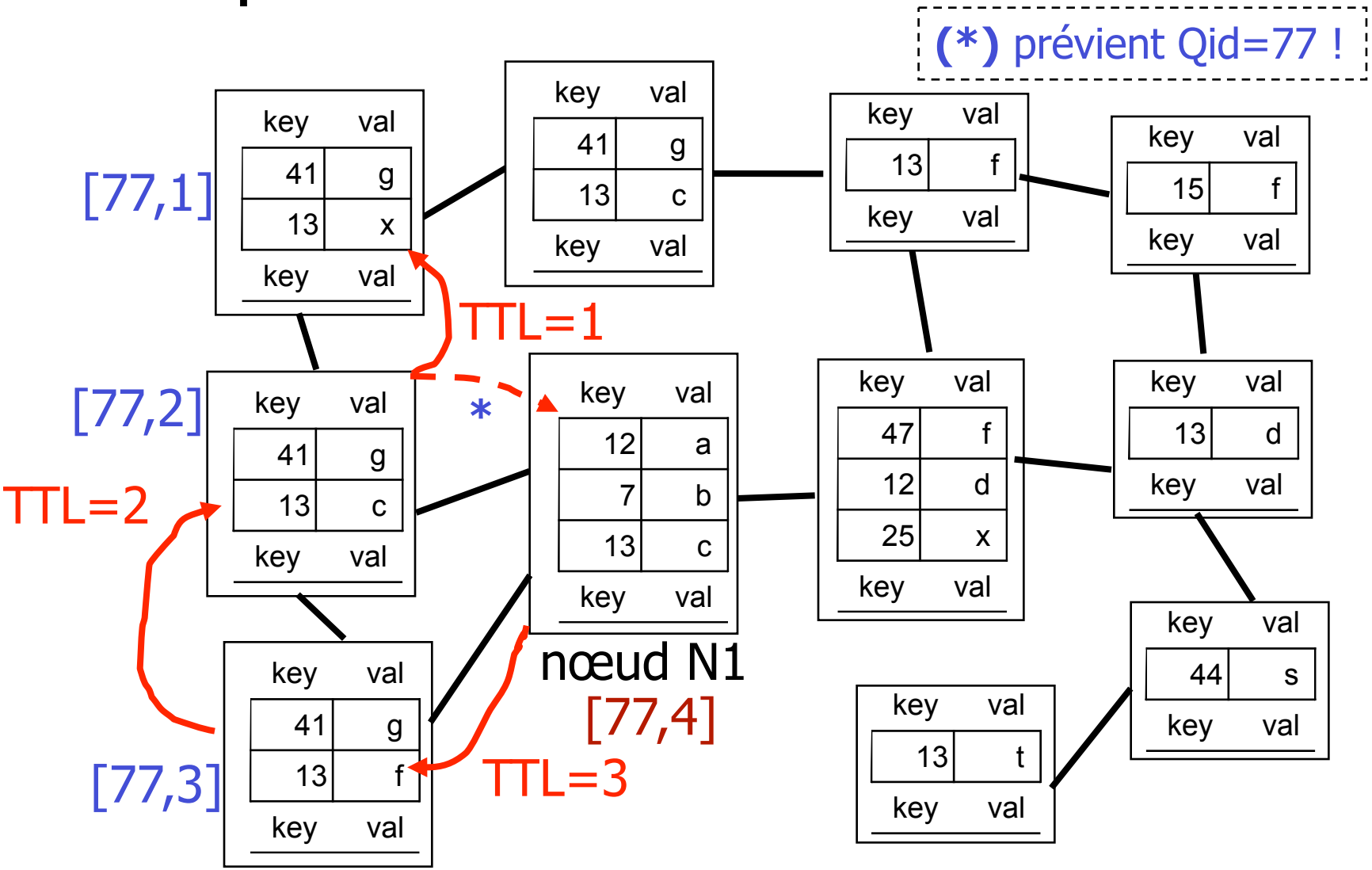


Optimisations

- Terminaison adaptative (Expanding Ring)
  - Condition d'arrêt : #réponses
- #Messages émis limité
  - Marche aléatoire (Random Walk) : Nw < Nc
- Cache de requêtes
  - Chaque requête a un identifiant unique Qid
  - Les nœuds conservent en cache les requêtes récentes (Qid + TTL)

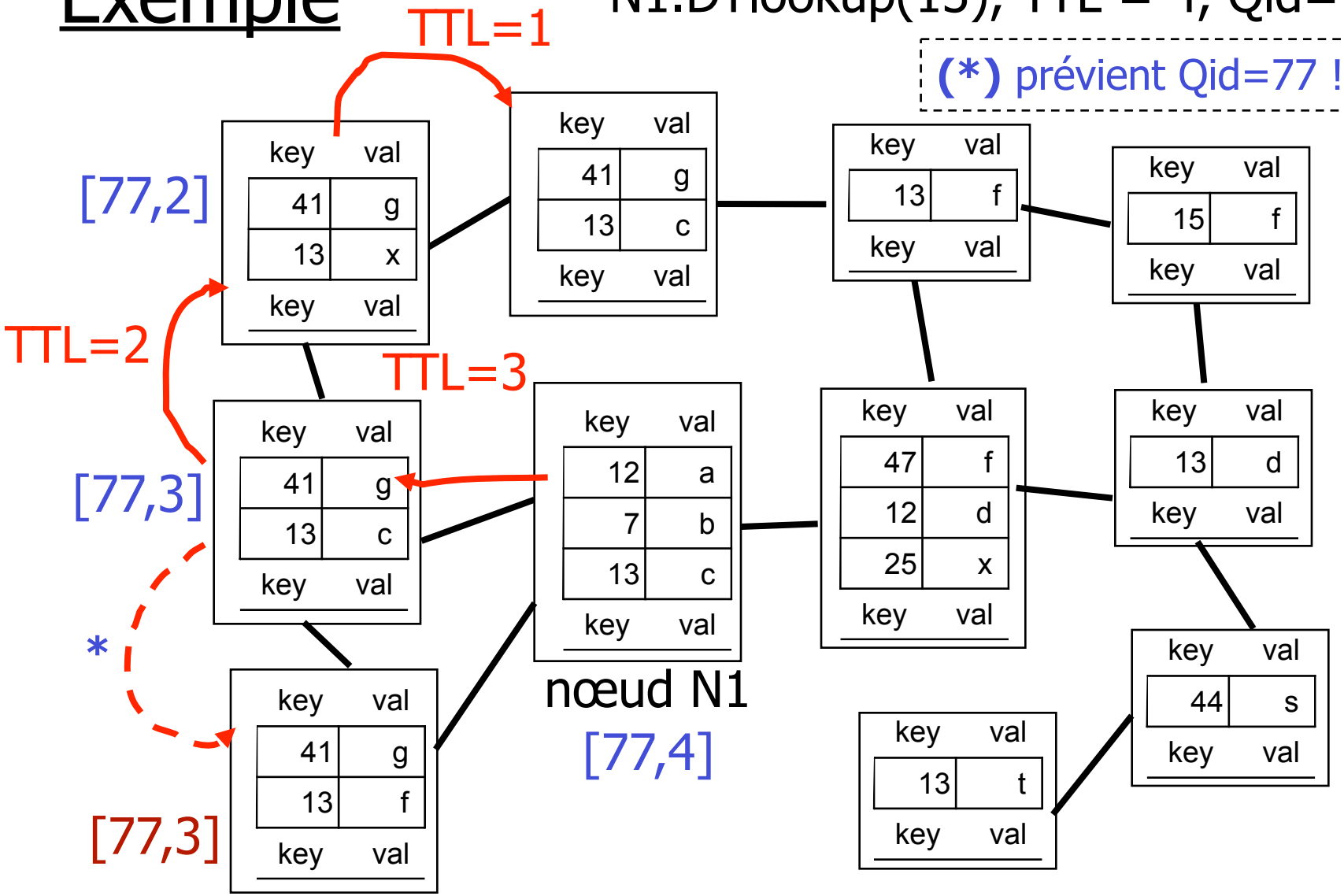
Exemple

N1.DTlookup(13), TTL = 4, Qid=77



Exemple

N1.DTlookup(13), TTL = 4, Qid=77



Insertion d'un nœud

- **X.join**

```
neighbors := {}
cand:= nœuds recommandés
Z:= serveur initial connu
cand:= cand U Z.getNodes
for Y in cand do
   ok := Y.wantMe(X)
   if ok then
       neighbors:=neighborsU{Y}
   if |neighbors| > max, return
```

Insertion d'un noeud (suite)

- **Y.wantMe(X)**

```
if Y veut de X comme voisin then
   neighbors := neighbors U {X}
   return (true)
return (false)
```

Quel est le rapport entre...

Réseaux pair-à-pair « purs »

Graphe du Web

Wikis

Réseaux sociaux

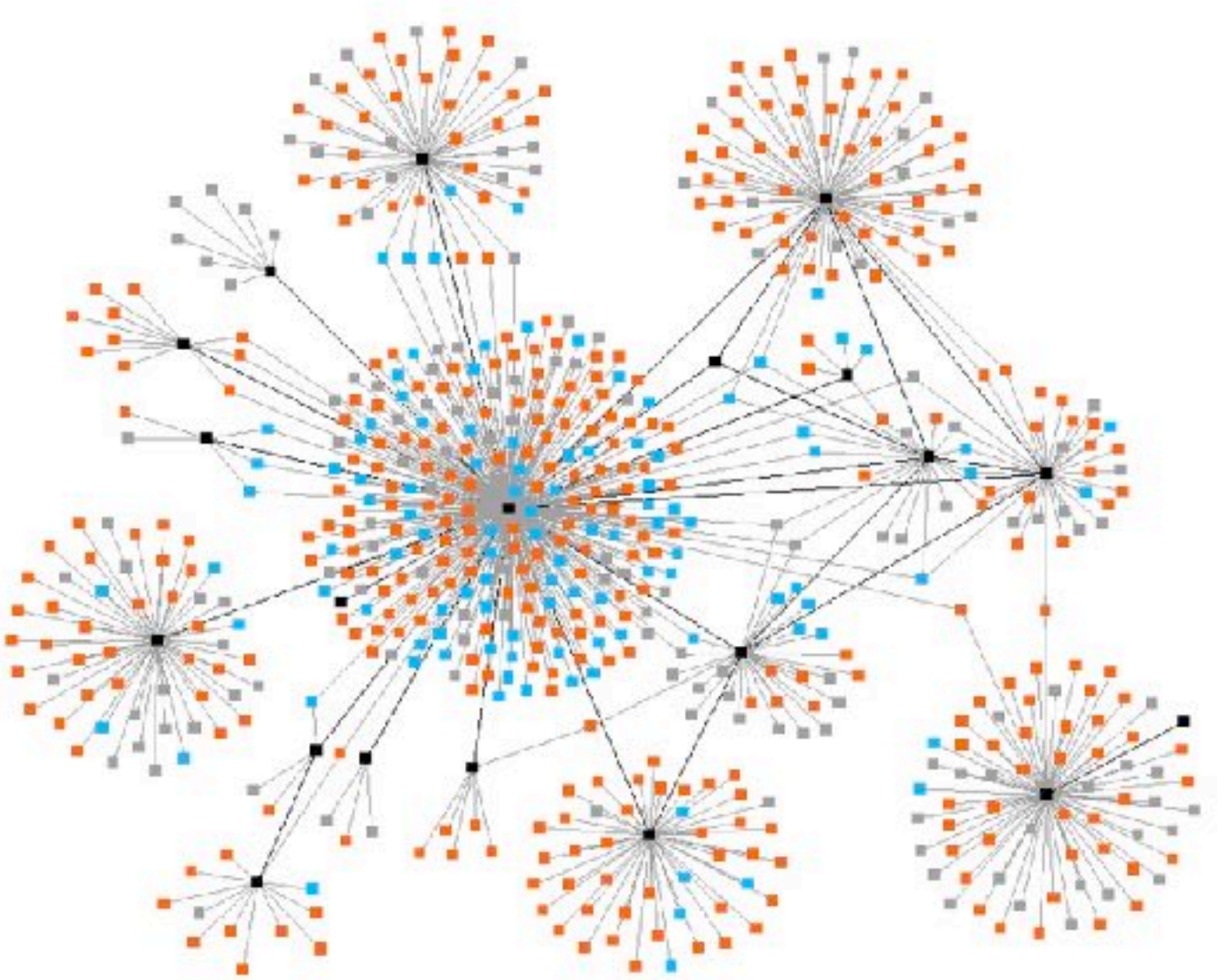
Réseaux d'interactions protéine-protéine

Chaîne alimentaire

Réseaux de transport aérien

Ontologies lexicales et sémantiques

...



scienceoftheinvisible.blogspot.com/search/label/SmallWorlds

Les graphes Petit-Monde

- Phénomènes observés :
  - Petit diamètre : longueur du plus court chemin réduite $O(\log(n)^k)$
  - Fort cœfficient de clustering : « les amis de mes amis sont mes amis »
- Propriétés dérivées :
  - Densité globale faible = degré moyen faible
  - Quelques nœuds fortement connectés : les hubs
  - Robuste aux suppressions aléatoires de nœuds
- Pas de définition formelle

Zoom sur le cœfficient de clustering

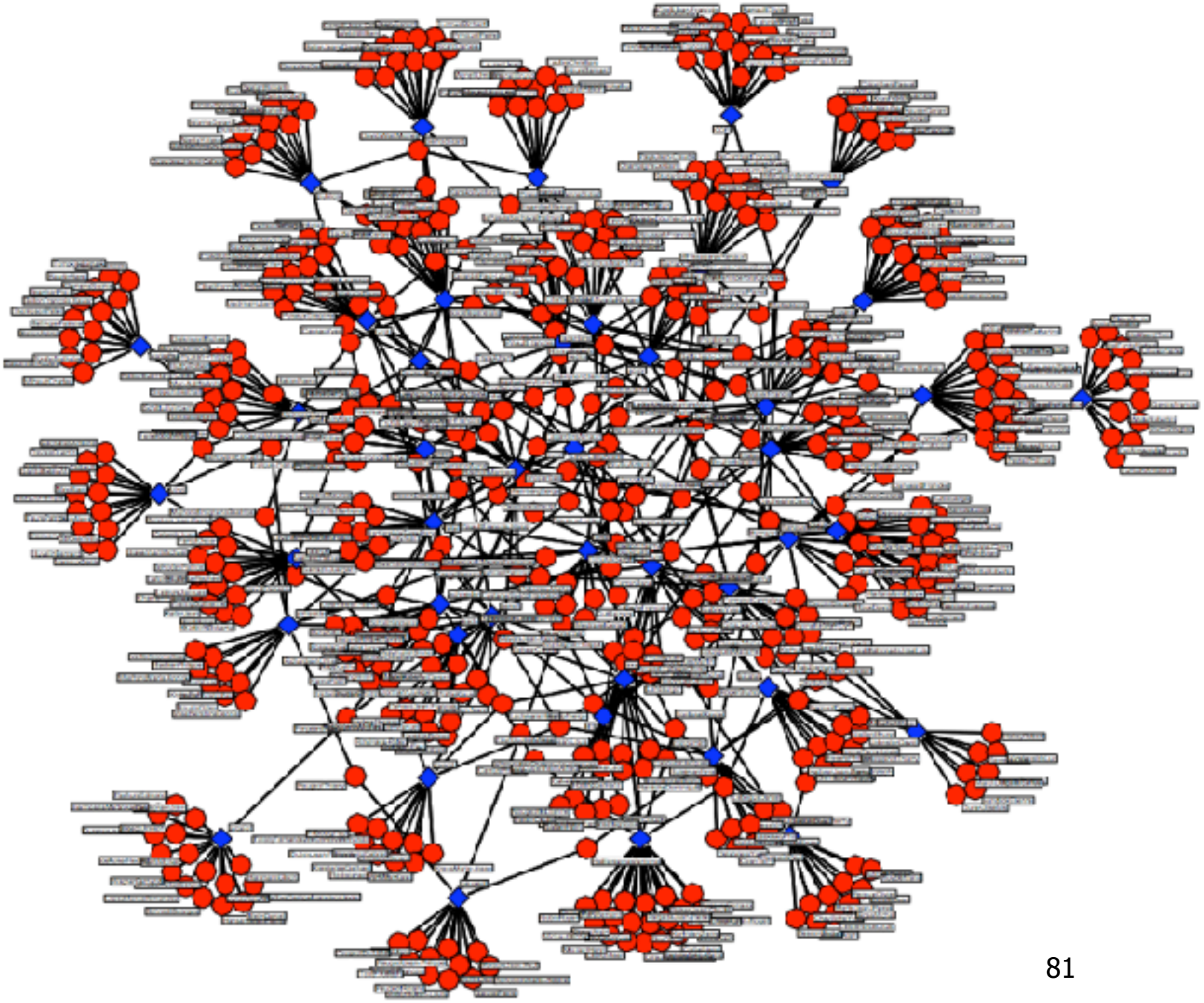
- CC(p) =

$$\frac{\text{\# liens entre les voisins de p}}{\text{\# liens possibles entre les voisins de p}}$$

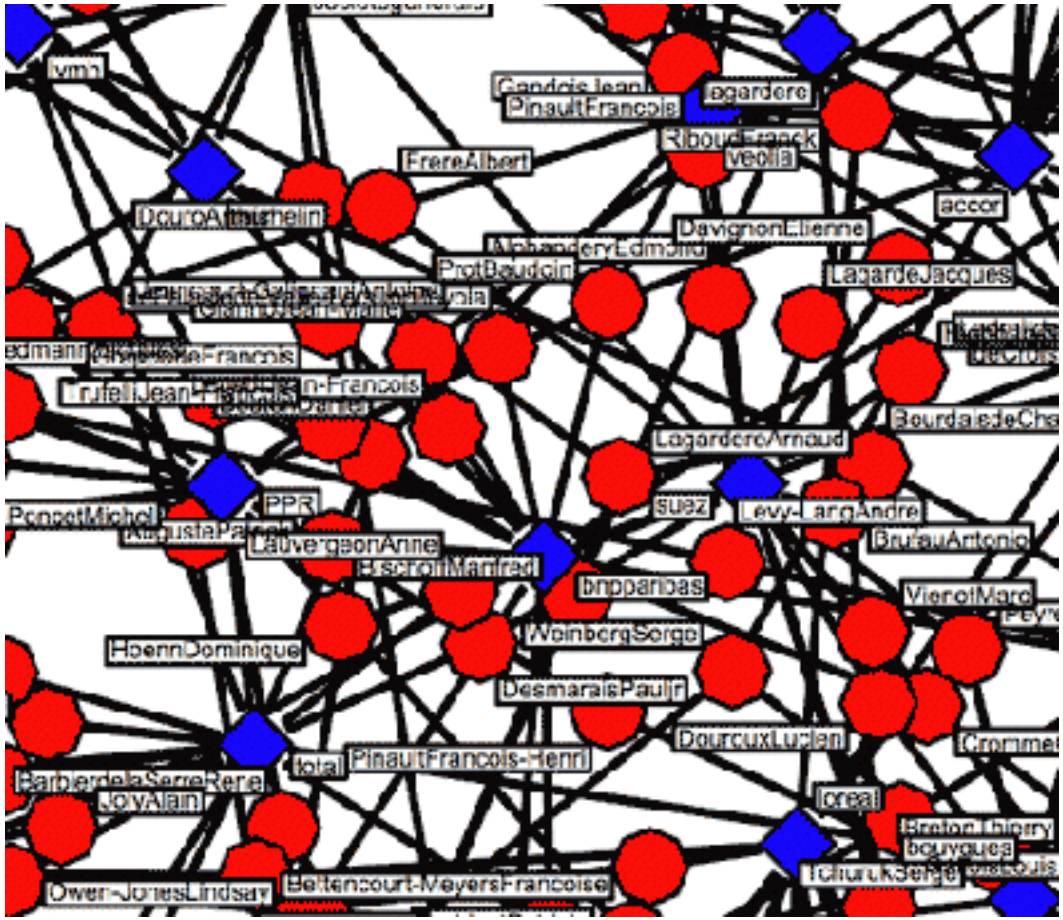
- Idée : les voisins se connaissent entre eux
- CC(G) = moyenne { CC(p), p $\in$ G}
- Exemples :

  CC(graphe complet) = 1

  CC(arbre) = 0



Un graphe Petit-Monde

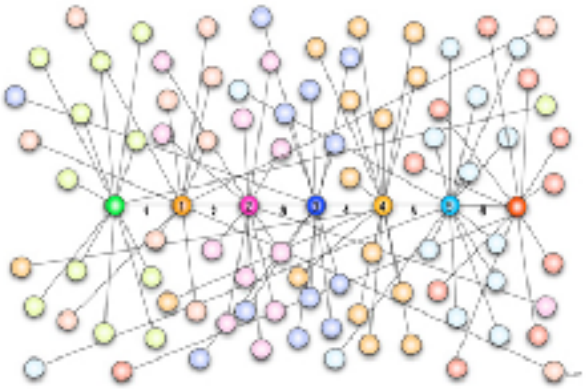


Bleu : entreprises du CAC40
Rouge : administrateur

Données Opesc 2005

Baptiste Coulmont, "Les réseaux du CAC 40", créé le 28/08/2009

Faux exemples célèbres



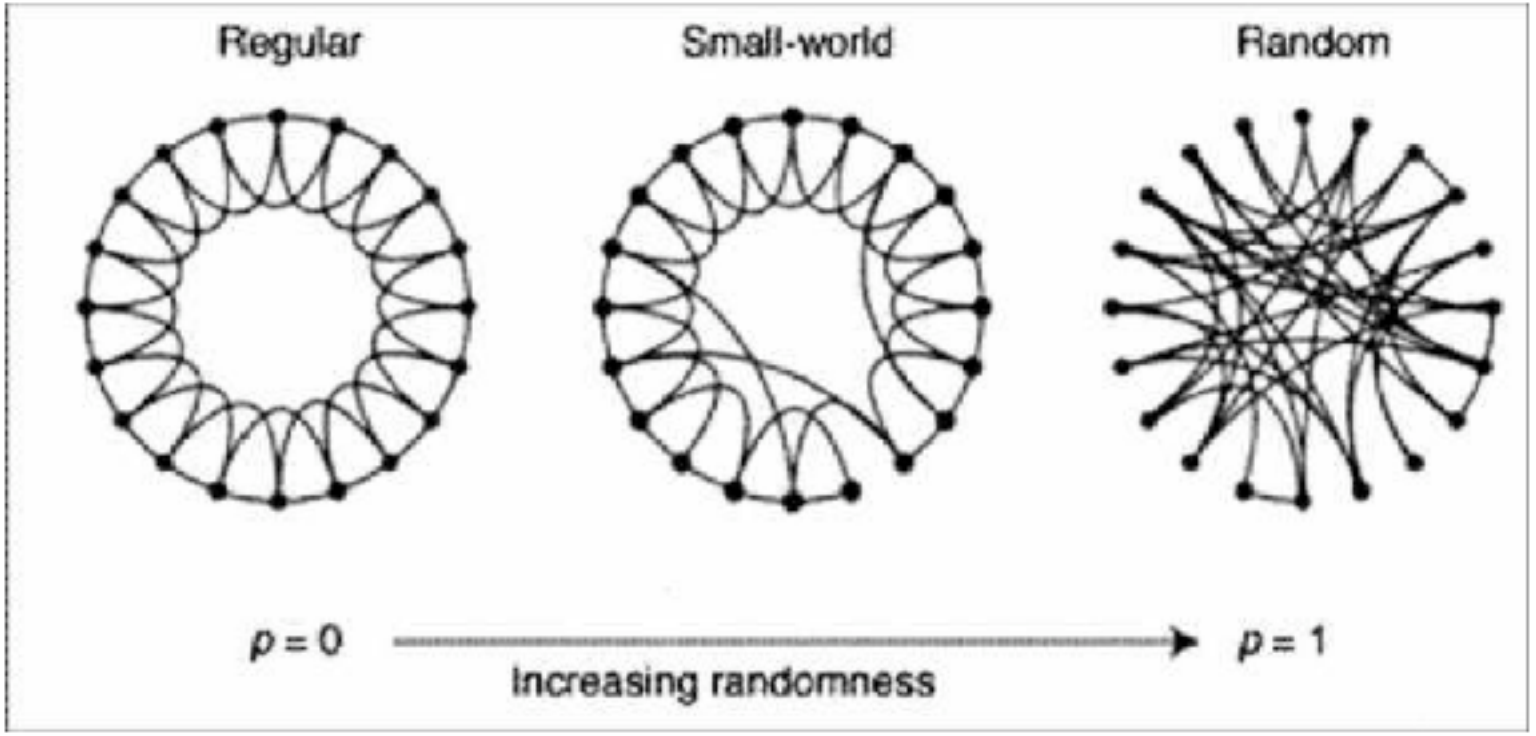
- Collaborations scientifiques
  - Erdös Number (5)
- Du Nebraska à Boston (Milgram, 1967)
  - degré de séparation = 6
  - Graphe de Hollywood (Kevin Bacon)

- Hypothèse de localité spatiale et temporelle
  - Contemporanéité, mobilité

Modèle générateur

- Watts & Strogatz, Nature (1998)

  à partir des graphes aléatoires



Réseau à invariance d'échelle

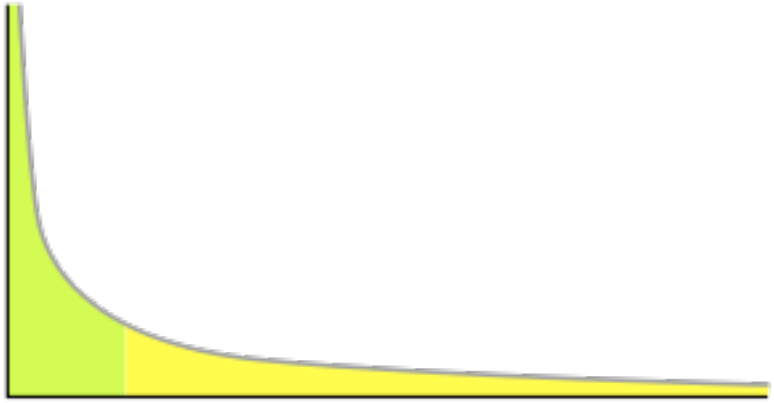
- « Scale-free network »
- Loi de probabilité des degrés k des nœuds :

$$P(k) \sim c \cdot k^{-Y}$$

- Coefficient Y compris dans $2<Y<3$
- Topologie de réseaux Petit-Monde !

Loi de puissance

$y = a \cdot x^k$



- Loi de Pareto ou règle des 80-20
- Effet « longue traîne »
- Échelle Log-Log : $y = k \cdot x + a$
  - linéaire sur 3 ordres de grandeur

Modèle générateur

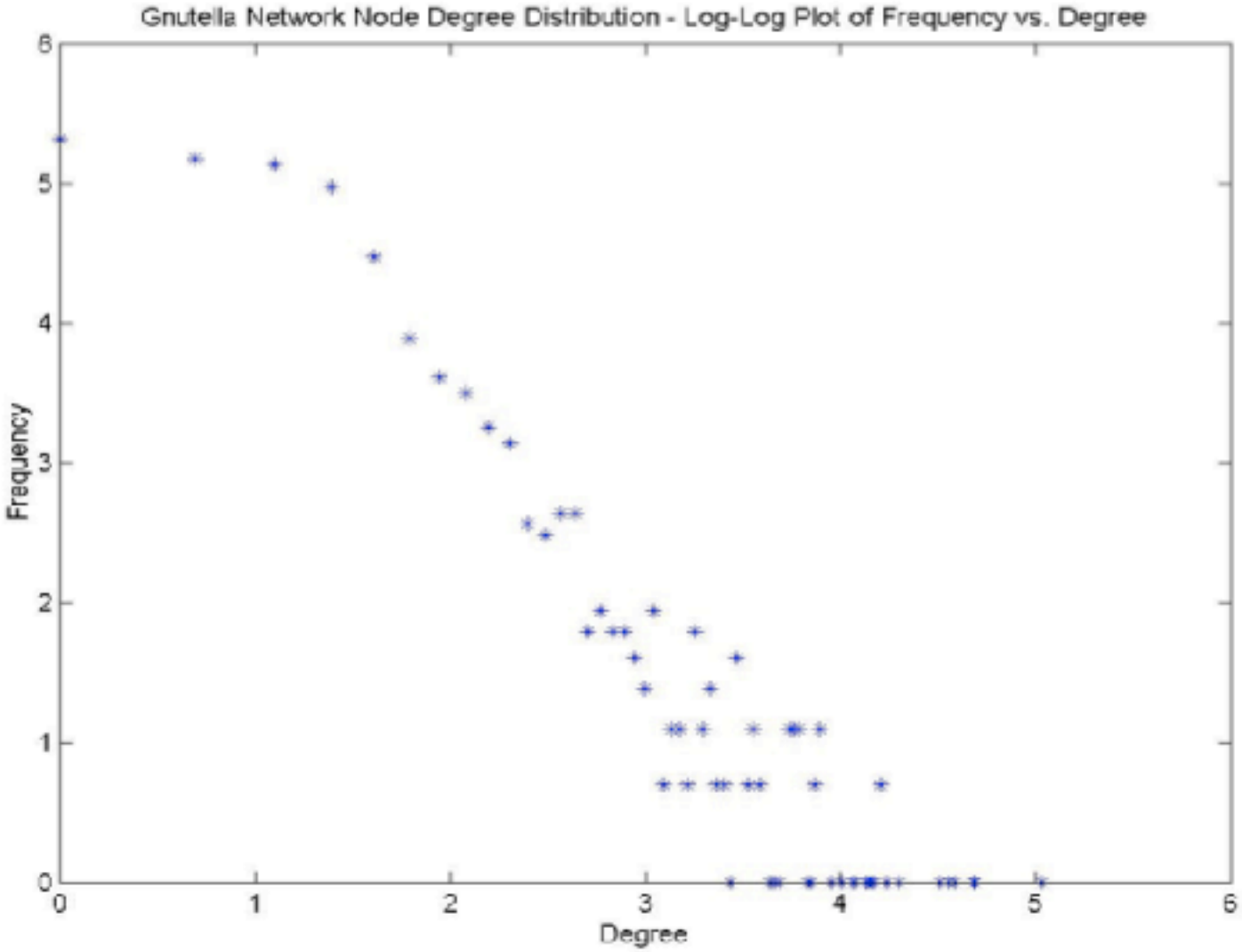
- Croissance et attachement préférentiel
  - « avantage cumulatif » Price (1976)
  - appliqué au WWW : Barabàsi & Albert (1999)

  La probabilité de connexion d'un nouveau nœud à un nœud n est proportionnelle au degré de n.

- Connexions fréquentes aux hubs
  - Les riches s'enrichissent !

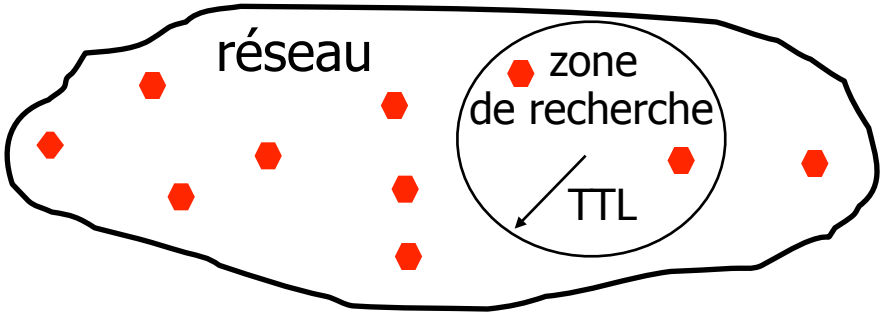
Retour aux réseaux P2P...

Topologie du réseau Gnutella



Pourquoi Gnutella c'est bien ?

- Requêtes complexes (filtres)
- Algorithme simple et robuste
- Fonctionne bien si les données sont fortement répliquées (Q populaires)



sites contenant la distro Ubuntu Intrepid Ibex

La recherche par voisinage : problèmes

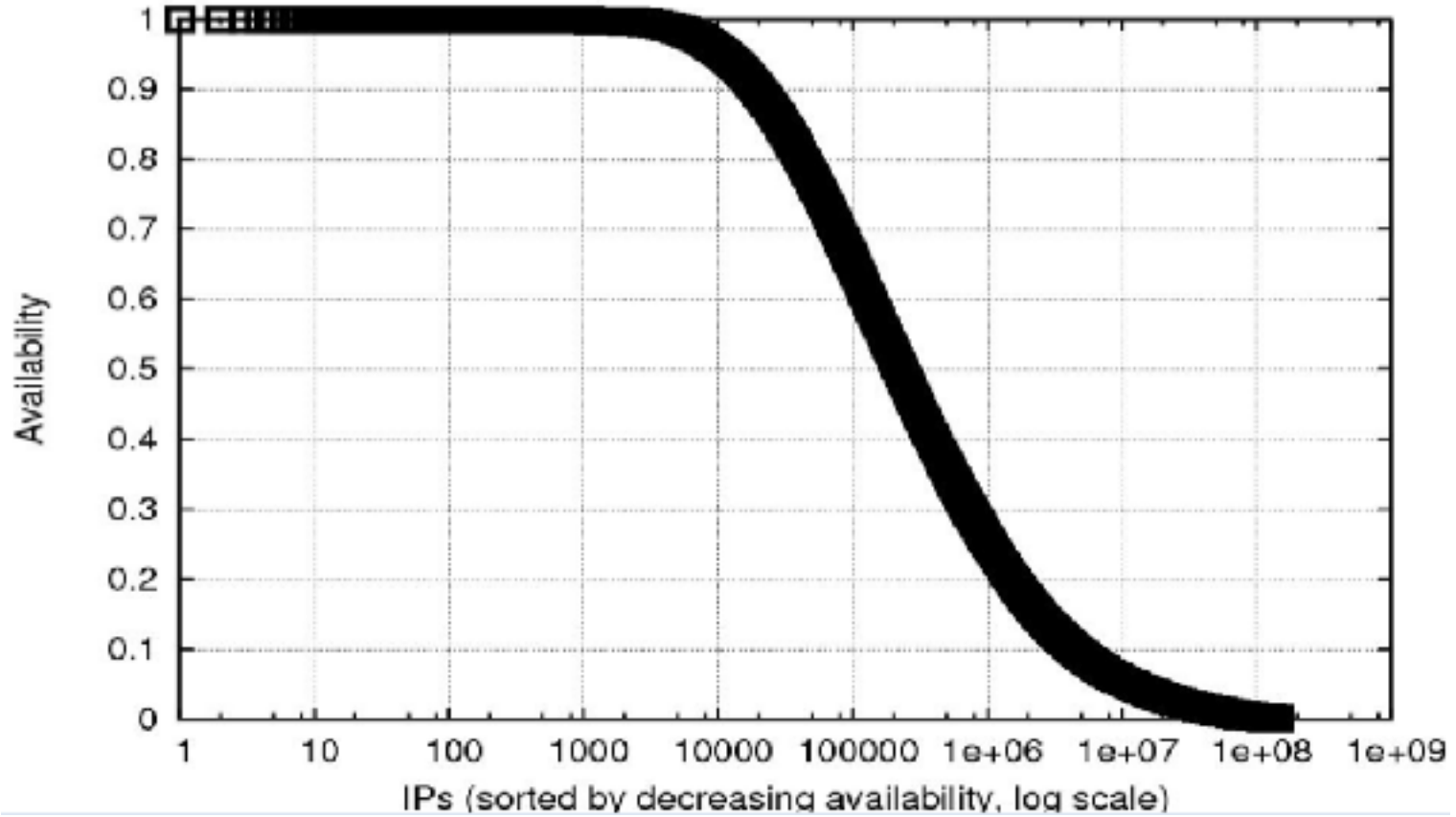
- Messages redondants (« Small World »)
  - Horizon << M (messages émis)
- Charge importante et trafic élevé
  - Exemple : si les nœuds ont Nc voisins, le nombre de messages émis est $O(Nc^{TTL})$
- Les nœuds de faible capacité sont des goulots d'étranglement
- Réponses partielles

Gnutella v0.6 ou les ultra-pairs

- Passage à l'échelle
- Des pairs « plus égaux que les autres »
  - Pair feuille (leaf node) : degré 3
  - Ultra-pair : degré 32
- Les critères de choix :
  - Disponibilité (temps passé sur le réseau)
  - Bande passante
  - Capacité de calcul (CPU et RAM)

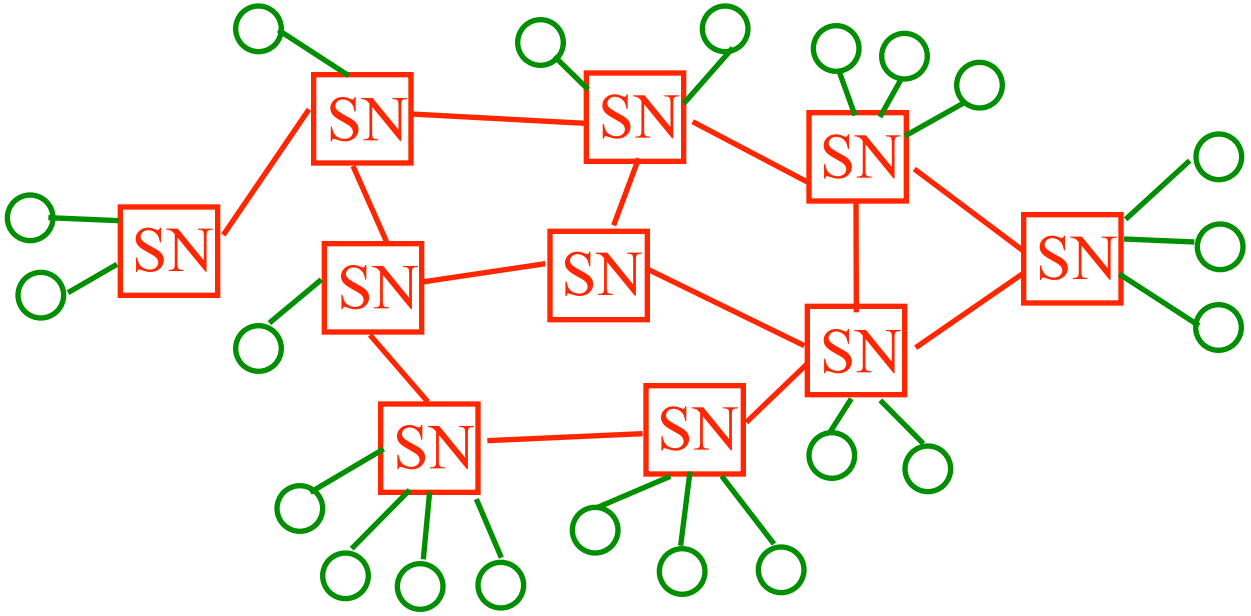
Disponibilité des pairs

- Hétérogénéité (loi de puissance)



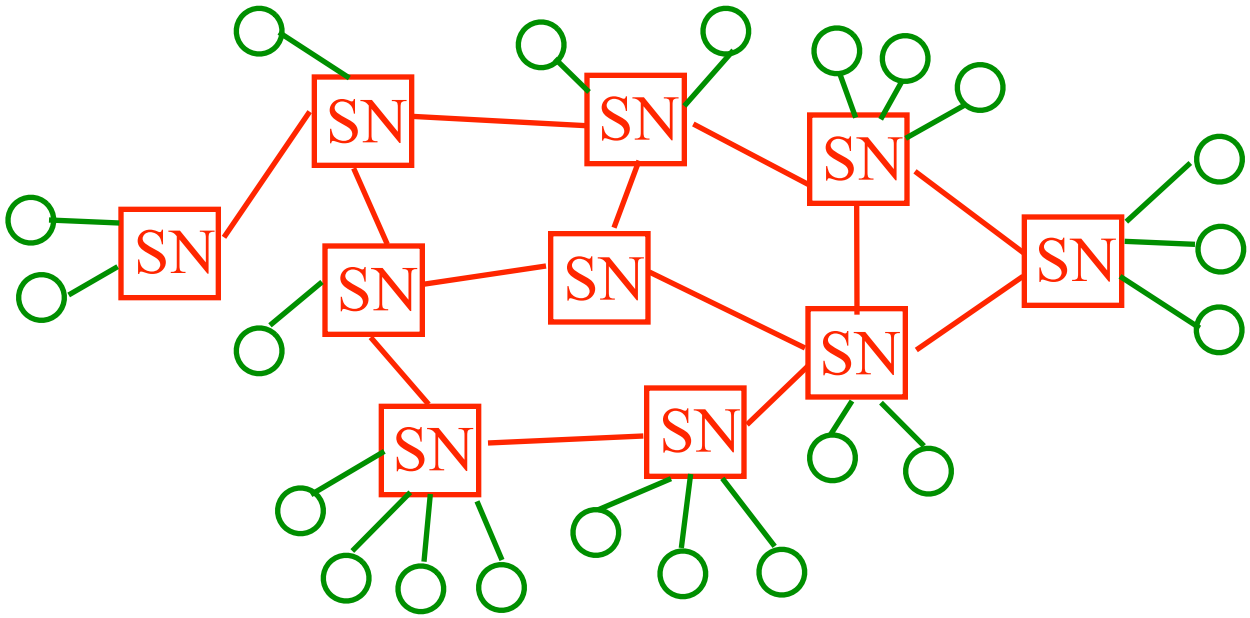
Super-nœuds (KaZaA)

- Les nœuds terminaux indexent leurs données sur les super-nœuds (C/S)
- Les super-nœuds exécutent les recherches de voisinage (P2P)



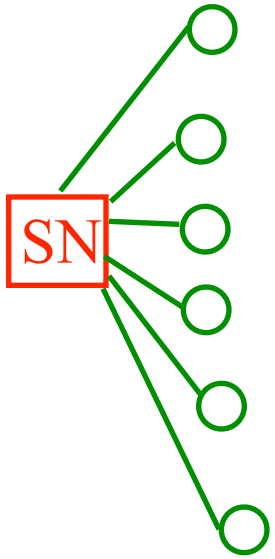
Motivation pour les super-nœuds

- Tirer profit des nœuds puissants
- Parcourir un seul gros index vaut mieux que beaucoup de petits



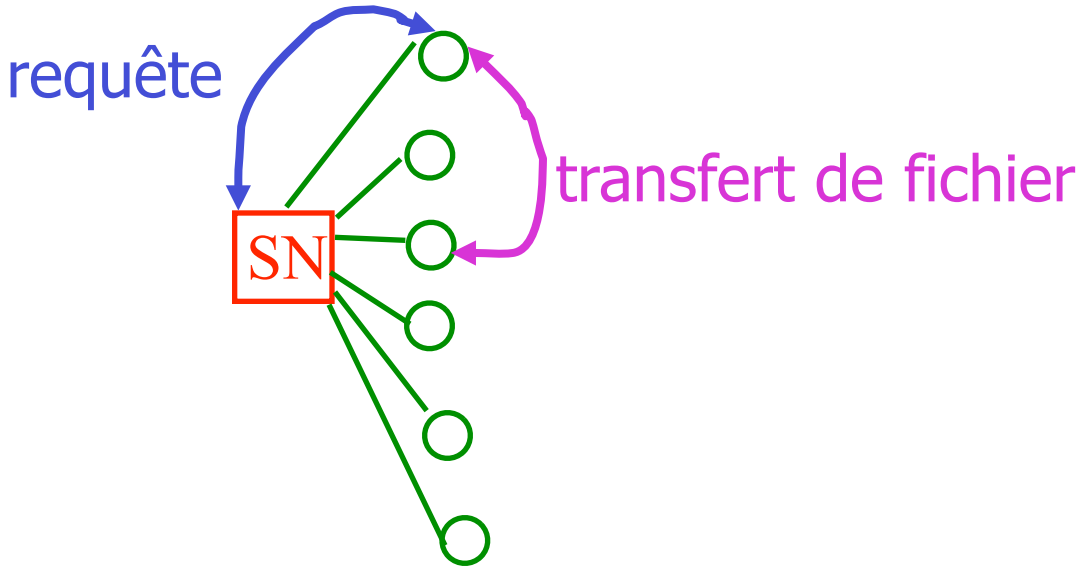
Napster (version originale)

- Un unique super-nœud



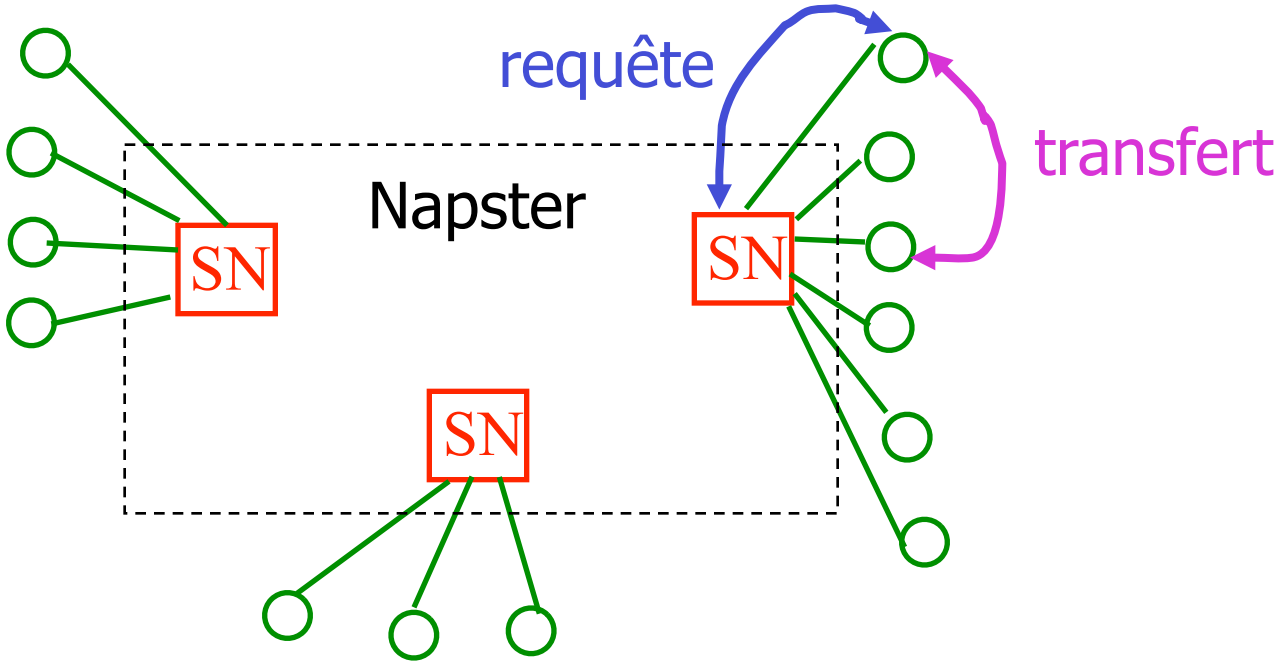
Napster (version originale)

- Un unique super-nœud



Napster

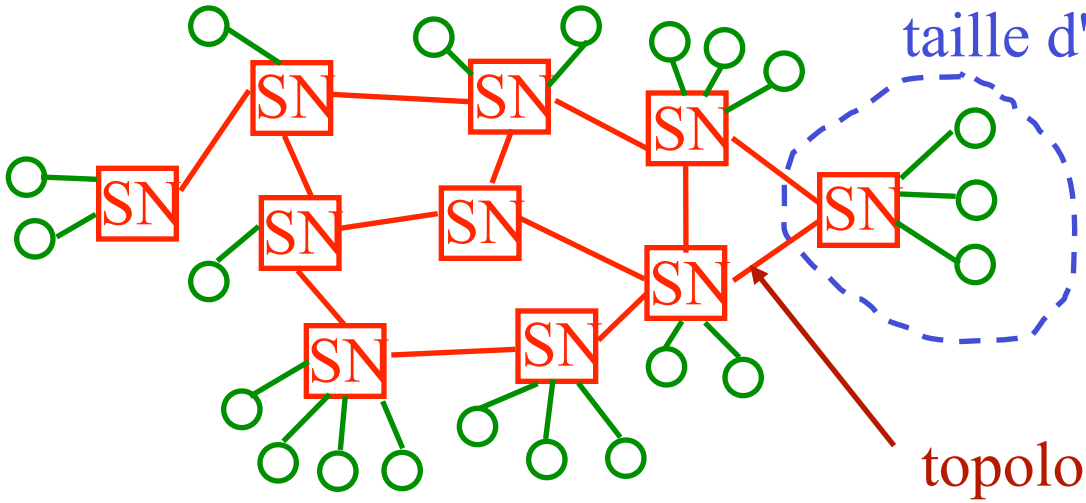
- En pratique, Napster dispose de plusieurs super-nœuds non connectés



Évaluation de performance

- B. Yang and H. Garcia-Molina. Designing a Super-peer Network, IEEE ICDE, 2003.

#noeuds = 10,000



Métriques:
- Bande passante
- Capacité de calcul

Topologie du réseau de super-nœuds

- 2 hypothèses :
  - Situation idéale

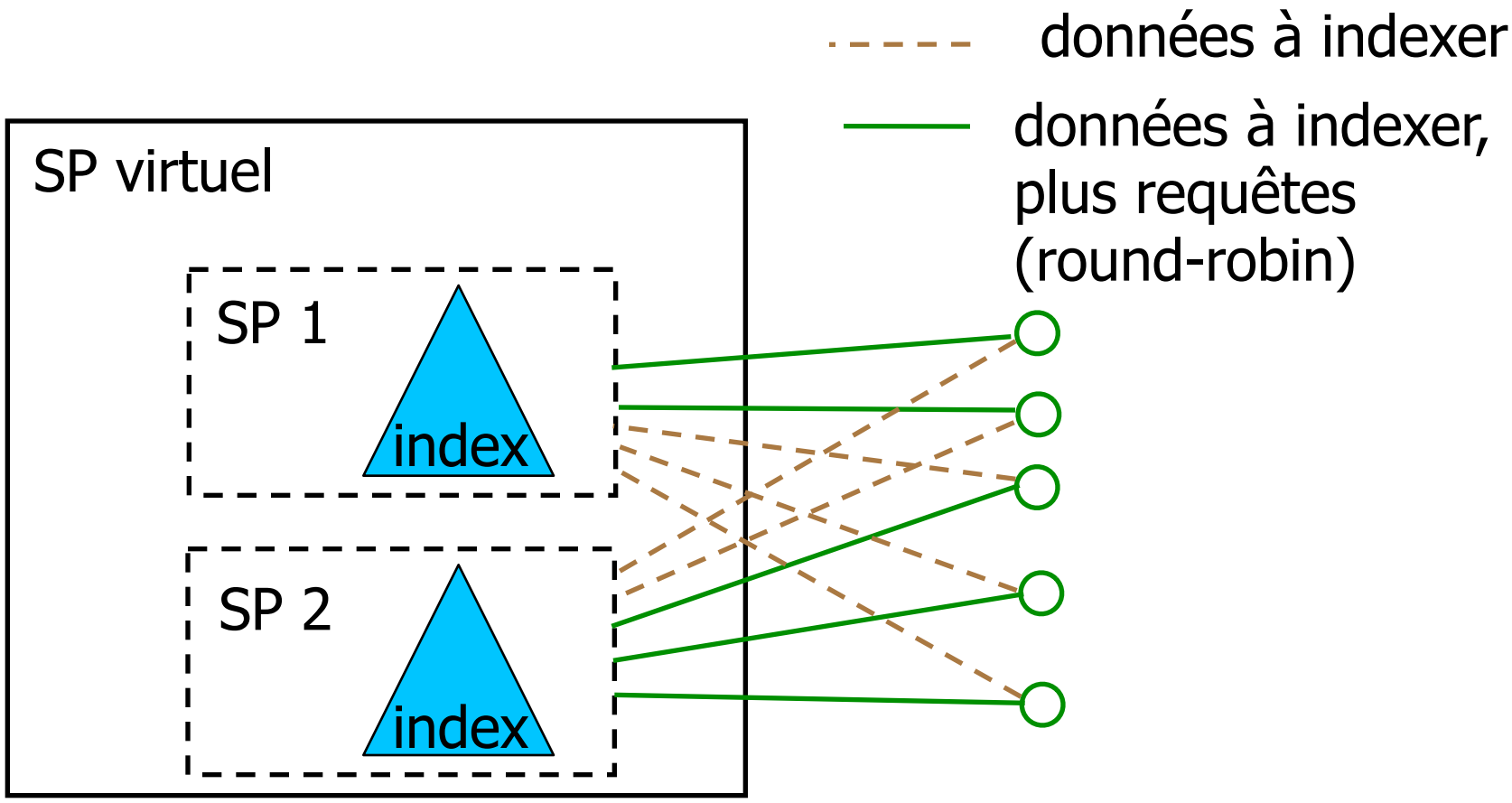
    Graphe fortement connecté

    TTL=1
  - Gnutella

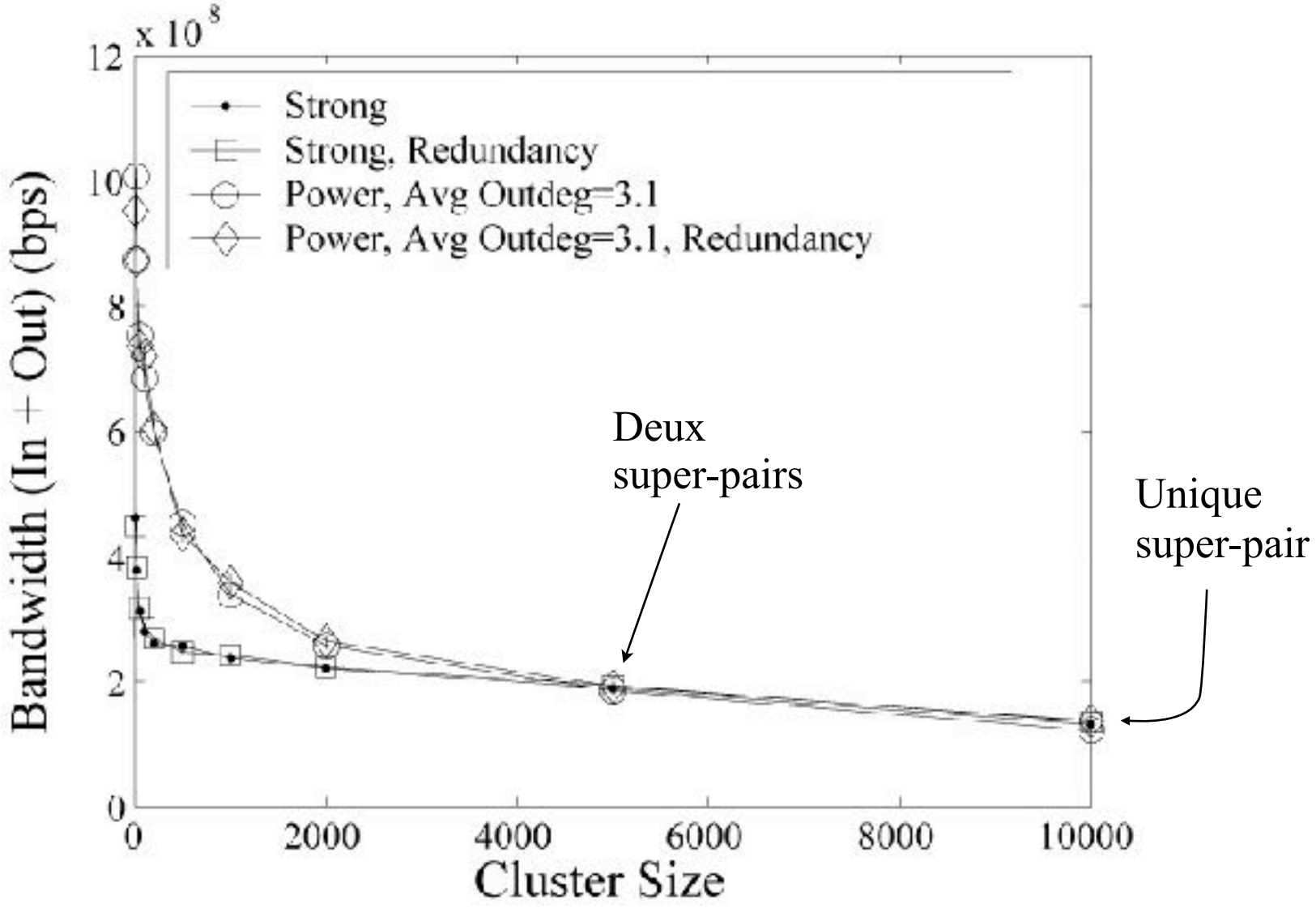
    Graphe Petit-Monde

    TTL=7, Degré SN=3,1
- Avec ou sans redondance

Redondance des super-nœuds

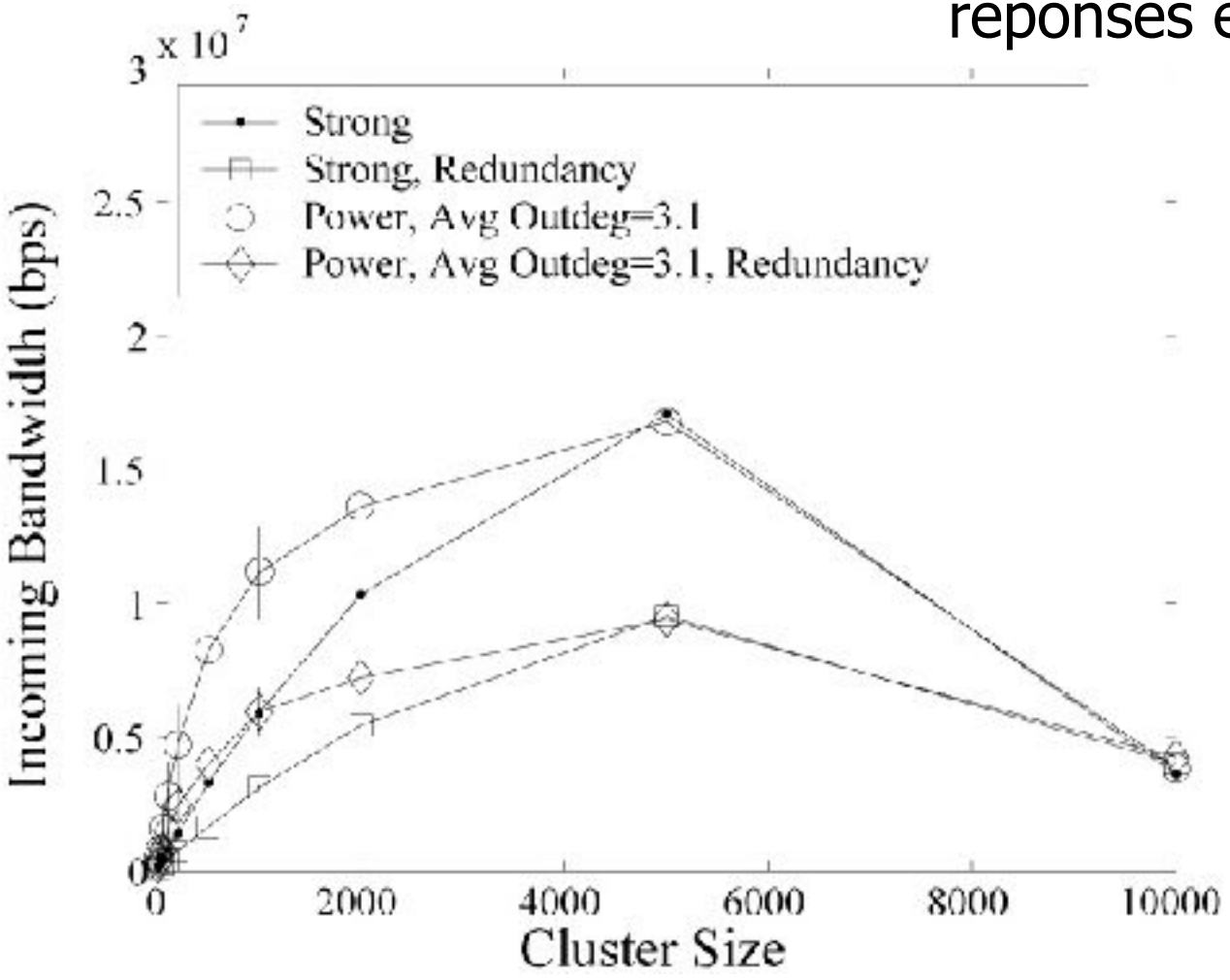


Débit cumulé

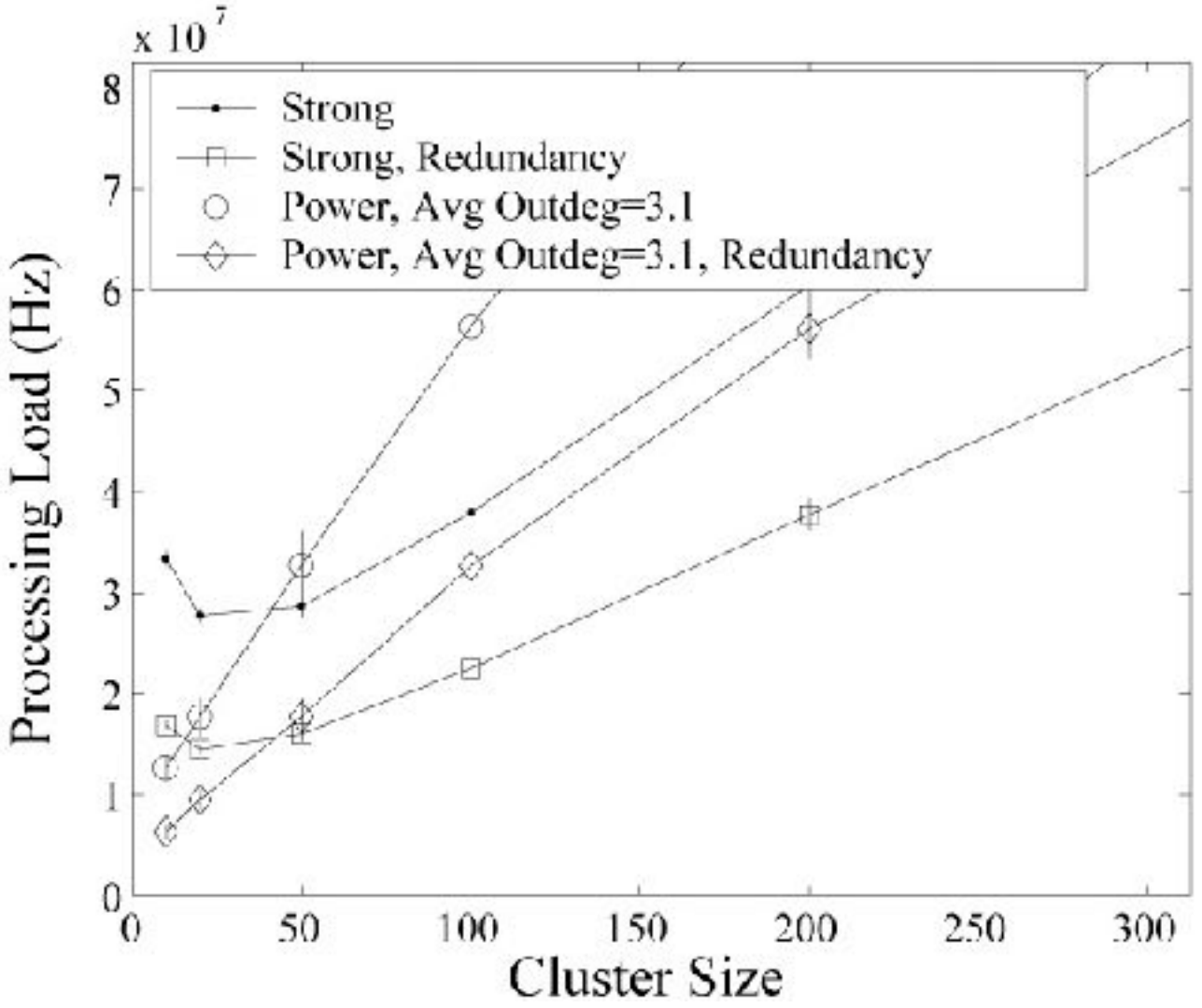


Débit individuel entrant

f=taille cluster/Np
réponses entrantes ~ f·(1-f)
max : f=1/2
min : 0 et 1



Charge individuelle de calcul



Règles de conception d'un super-nœud

1. Augmenter la taille des clusters diminue la charge cumulée mais augmente la charge individuelle
2. La redondance des super-nœuds est encouragée
3. Le degré des super-nœuds est maximisé
4. Le TTL est minimisé

Communication entre les ultra-pairs

- Réseau de super-nœuds de Gnutella v0.6
- Graphe Petit-Monde
- Au-delà de l'inondation : Filtres de Bloom

  Limite la diffusion des messages, et

  Indique un résultat possible pour une requête

Filtre de Bloom

Coder un ensemble E de n éléments $x_1, \ldots, x_n$

- Vecteur B de m bits (0/1), m grand
- Fonction de hachage $h: \Sigma \rightarrow [0..m]$
- Création du filtre B de taille m :

  $B[h(x_i)] = 1, i \in [1..n]$

  0 sinon.

Usage du filtre de Bloom

Pour tester si x appartient à E :

- Si le bit h(x) vaut 0, alors non
- Si le bit h(x) vaut 1, alors oui, c'est possible

- Probabilité de « faux positif ».

Combattre les faux positifs

- Augmenter m, la taille du vecteur B
- Plusieurs fonctions de hachage : h_j, $j \in [1..p]$

  $\forall xi \in E, \forall h_j \; F[h_j(x_i)]=1$

- Pour tester l'appartenance de x à E :

  Si l'un des bits en position $h_1(x),..., h_p(x)$ est à 0, alors x n'est pas dans E

- Faible impact des faux positifs :

  Pour m=8n bits, la probabilité de faux positifs est de 0,02.

Diffusion optimisée

- Chaque ultra-pair code sa liste de clés des ressources qu'il indexe dans un filtre de Bloom B_0
- Chaque ultra-pair échange son filtre avec ses voisins
- Une requête de TTL 1 n'est transmise qu'aux voisins dont le filtre B_0 indique la présence possible des mots de la requête

Avec un horizon élargi

- Chaque ultra-pair calcule le filtre de Bloom B_1 de ses mots et de ceux de ses voisins et l'échange avec ses voisins
- Une requête de TTL 2 n'est transmise qu'aux voisins dont le filtre B_1 indique la présence possible des mots de la requête
- Ainsi de suite pour B_2, B_3,...

Implémentation dans Gnutella v0.6

- Une seule fonction de hachage :

$$h(x)= \underset{0\le i<len(x)}{XOR}\ x[i] << (8\cdot(i \bmod 4))$$

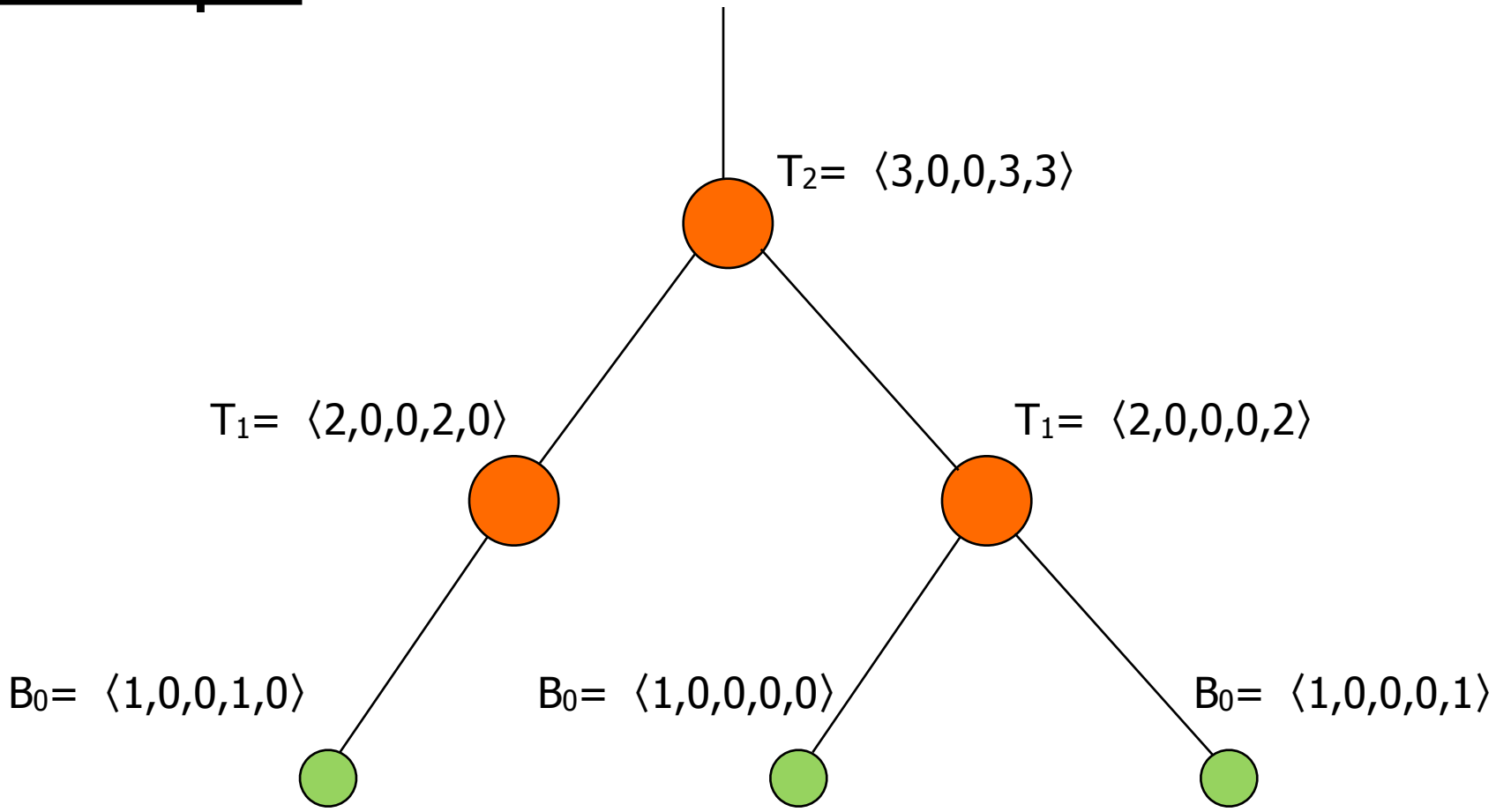
XOR	0	1
0	0	1
1	1	0

- Exemples :
  - h(«bonjour»)=1780226573
  - h(«a»)=97
- Taille du filtre : $M=2^{20}$ bits
- h(x,M)=h(x) mod M

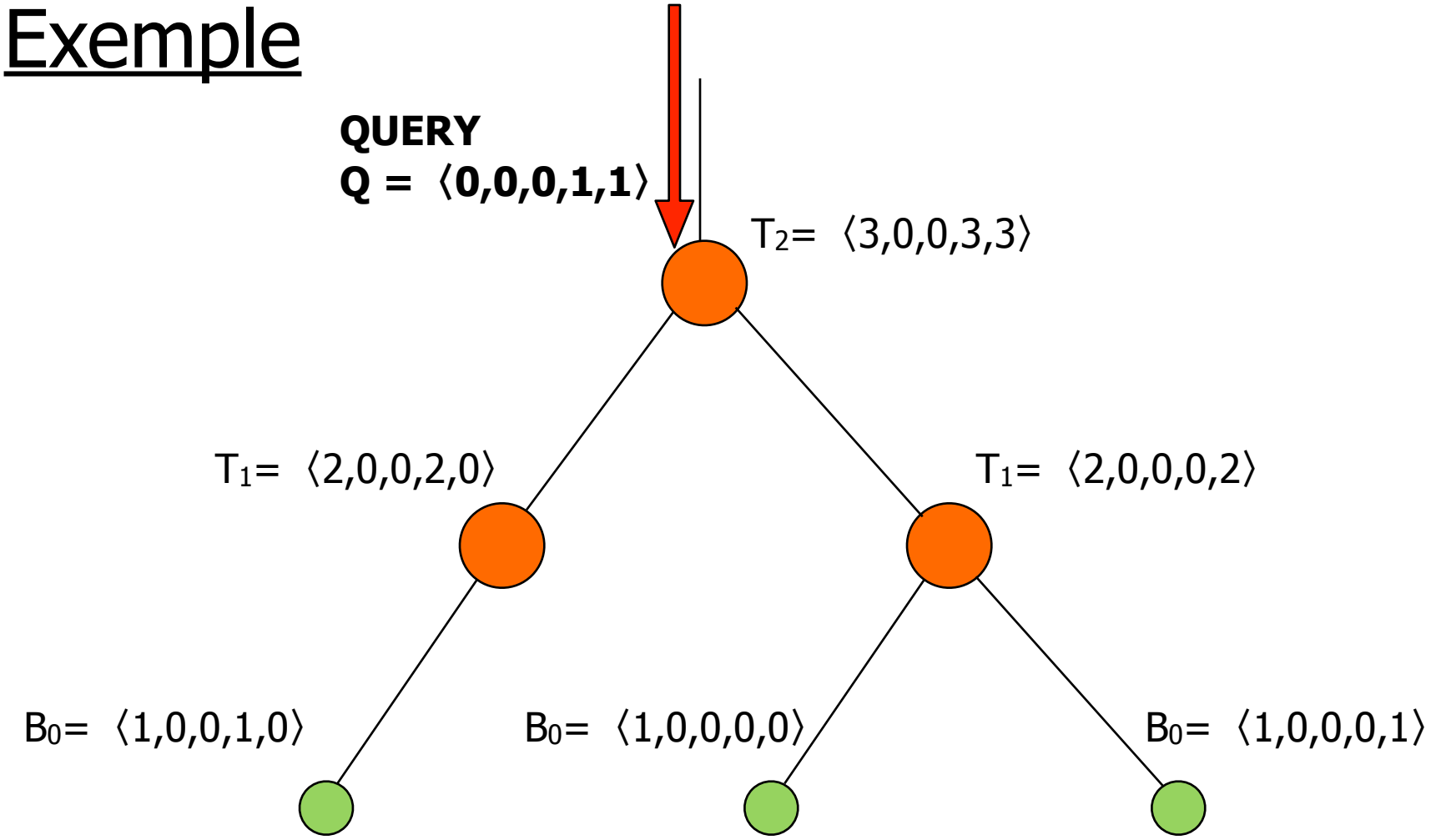
Dans Gnutella v0.6

- M >> #clés, donc beaucoup de bits à 0
  - Compression efficace du filtre
- Chaque feuille envoie son filtre B_0 compressé à son ultra-pair immédiat
- Les ultra-pairs s'échangent des tableaux d'octets T de taille M
  - Chaque octet T[i] renseigne sur le bit B[i] d'un filtre de Bloom
  - T[i]$\in$[0..7] indique une distance à la ressource

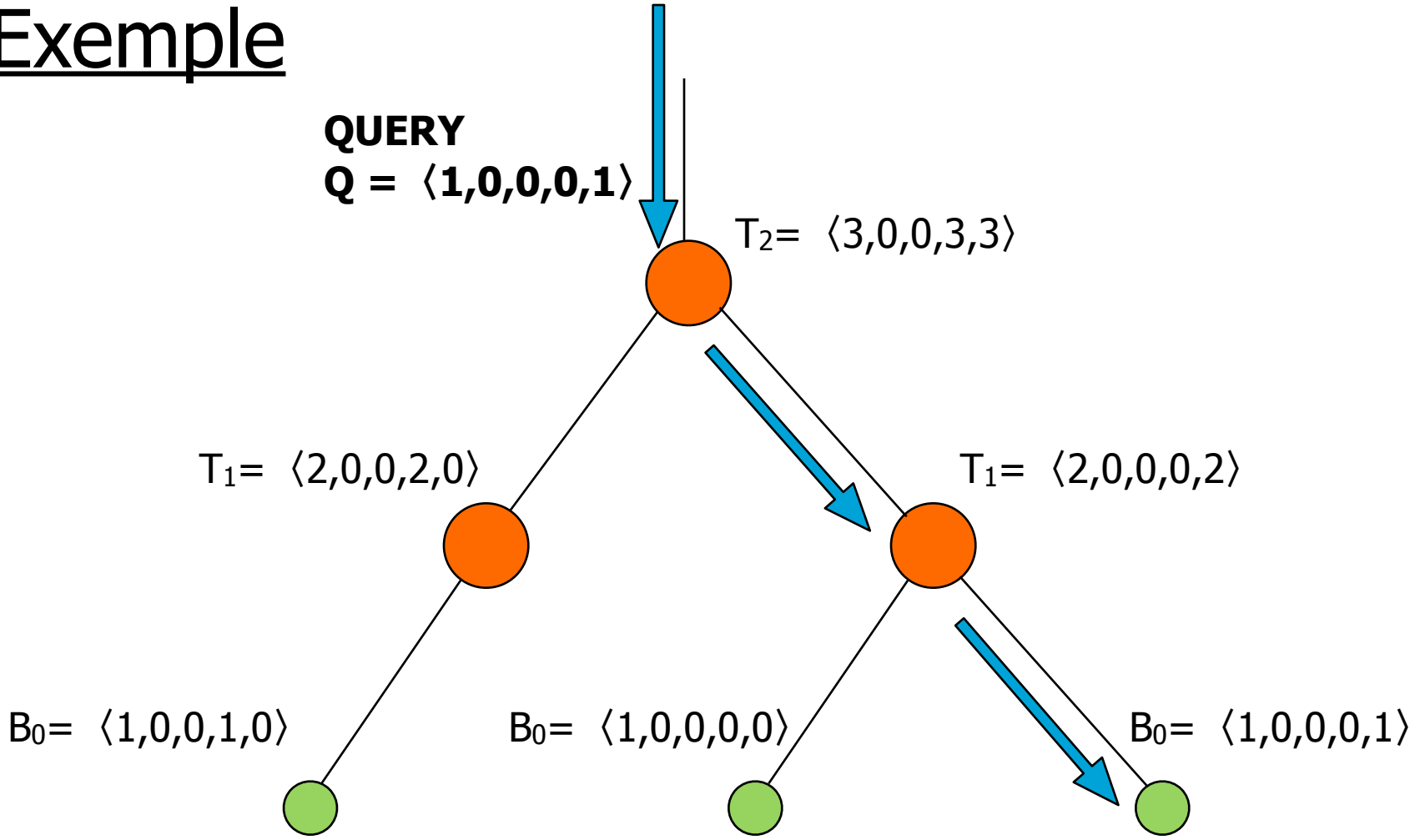
Exemple



Exemple


QUERY
Q = ⟨0,0,0,1,1⟩
T2= ⟨3,0,0,3,3⟩
T1= ⟨2,0,0,2,0⟩
T1= ⟨2,0,0,0,2⟩
B0= ⟨1,0,0,1,0⟩
B0= ⟨1,0,0,0,0⟩
B0= ⟨1,0,0,0,1⟩

Exemple

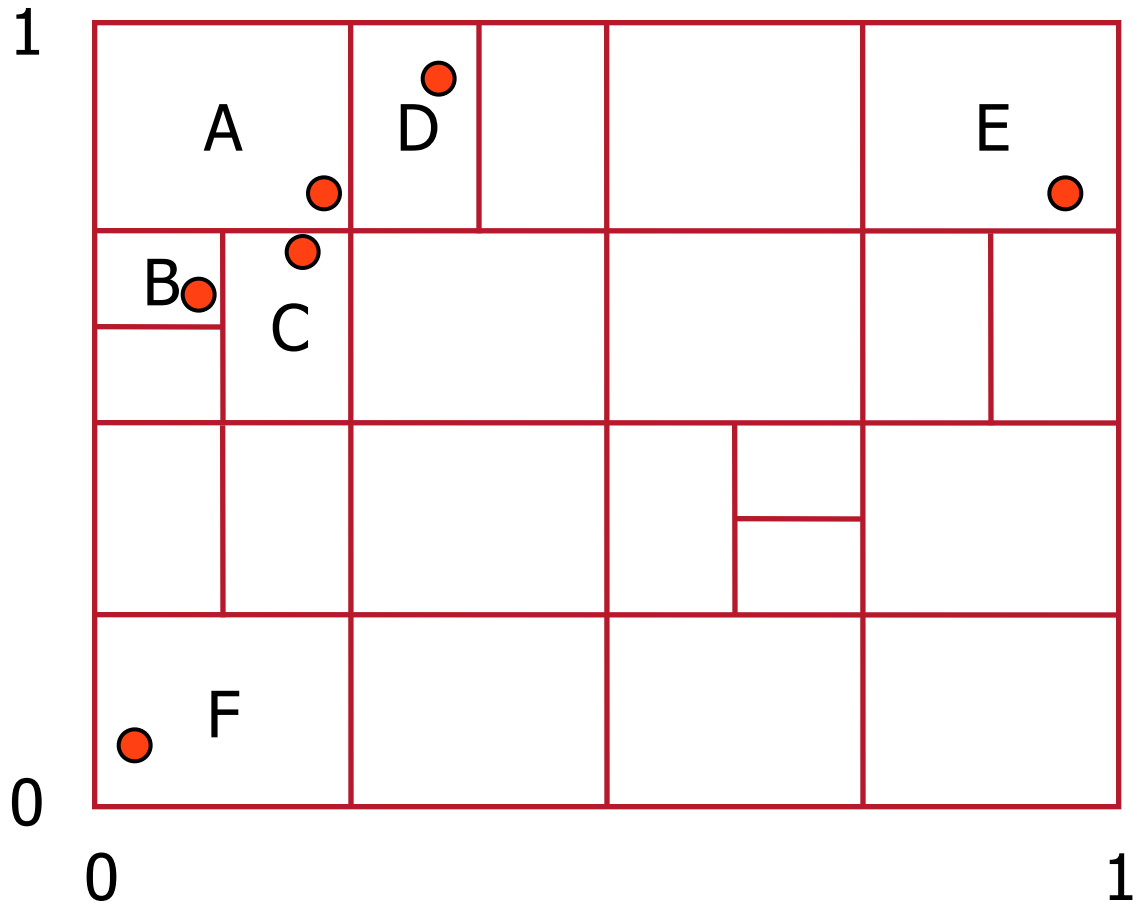

QUERY
Q = ⟨1,0,0,0,1⟩
T2= ⟨3,0,0,3,3⟩
T1= ⟨2,0,0,2,0⟩
T1= ⟨2,0,0,0,2⟩
B0= ⟨1,0,0,1,0⟩
B0= ⟨1,0,0,0,0⟩
B0= ⟨1,0,0,0,1⟩

Content Adressable Network (CAN)

- DHT : h(k) = $\langle x_1,...,x_d \rangle$
- Espace des clés : Hypercube/Tore en dimension d
- Responsabilité d'un pair : d-cube
- Routage par voisinage dans l'espace

Article CAN :
Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp and Scott Shenker. *A scalable content-adressable network.* In Proc. ACM SIGCOMM 2001.
http://www.cs.utexas.edu/users/browne/CS395Tf2002/Papers/Can-p161-ratnasamy.pdf

Content Addressable Network (CAN)



Espace : Tore 2D
h(k)=(x,y)
Pair : 2-cube

Content Addressable Network (CAN)

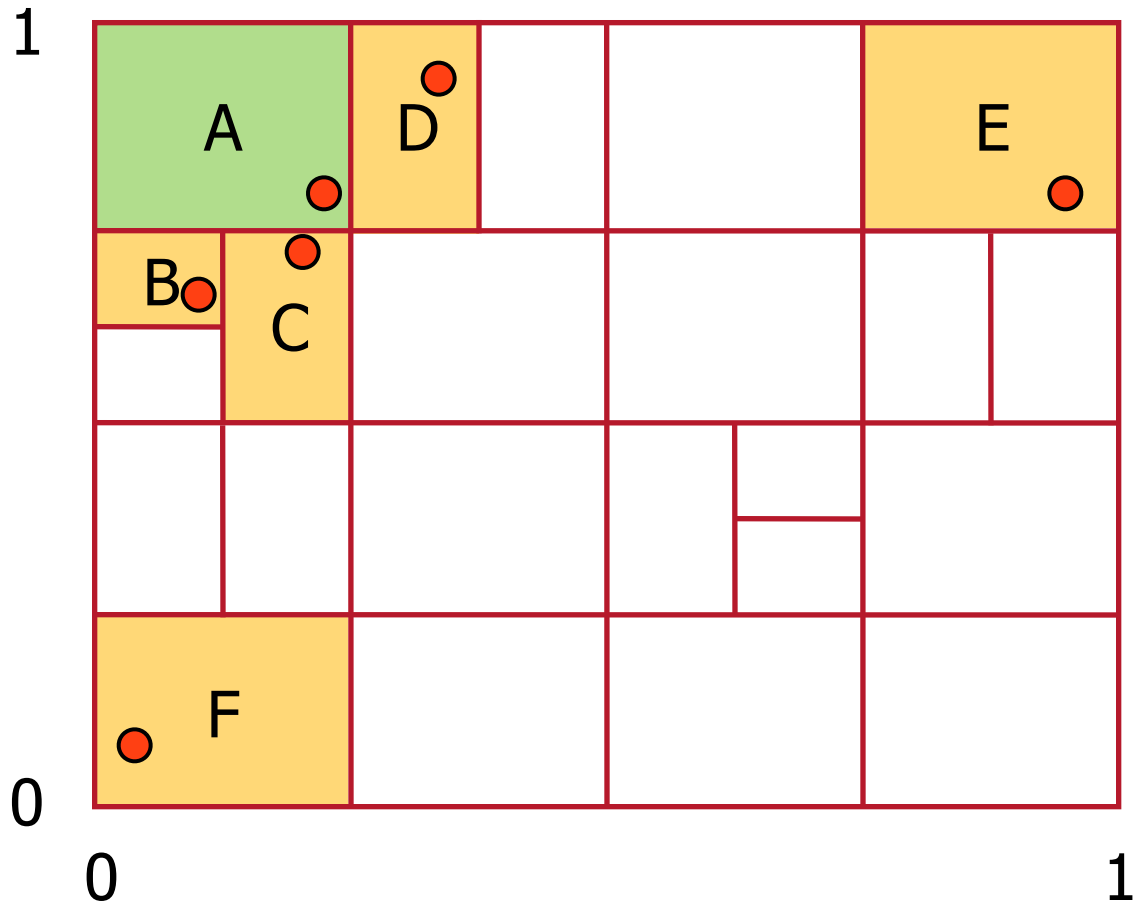
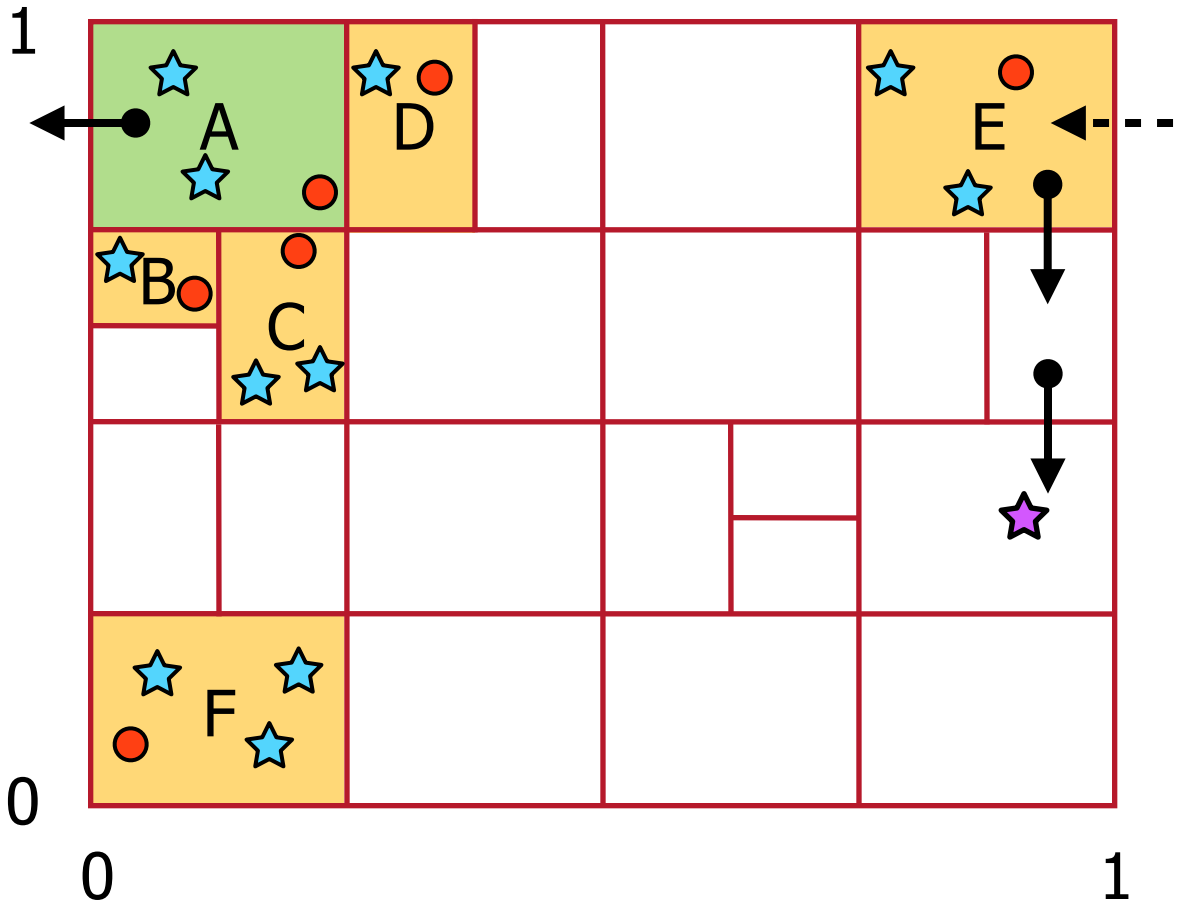


Table de routage
T_A = {B,C,D,E,F}

Content Addressable Network (CAN)



A.Lookup(★)

Analyse de complexité

- Recherche de clé k :

  1D - segment : O(N), avec N #pairs

  2D - carré : $O(2 \cdot \sqrt{N})$

  ...

  nD - hypercube : $O(d \cdot N^{1/d})$

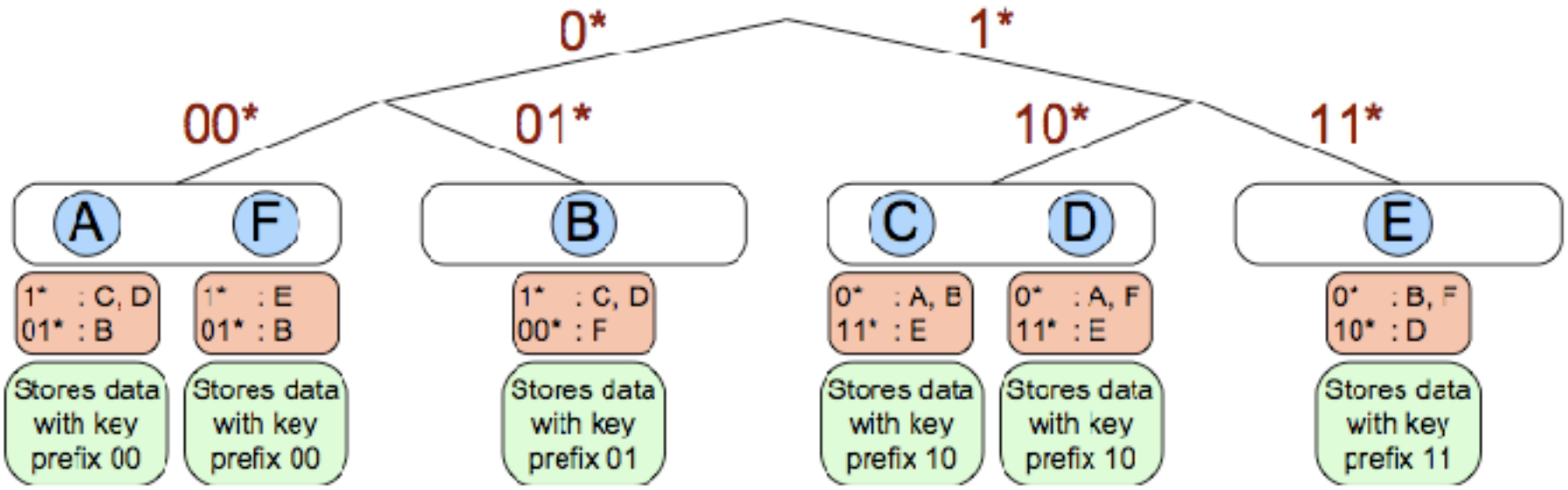
- Moins bien que O(logN), efficace en pratique
- Requêtes par intervalle ?

P-Grid (Aberer et al., 2001)

- Répartition des valeurs selon un arbre binaire de recherche
- Hash-code qui préserve les préfixes et répartit uniformément les ressources dans l'arbre (pas les @IP)
- Structure auto-adaptative avec réplication
- Recherche par préfixe et par intervalle

P-Grid

- Exemple : Q(F,100) ?



Qu'est-ce qu'un système pair-à-pair ?

- Nombreux sites (en bordure d'internet)
- Ressources réparties
- Sites autonomes (différents propriétaires)
- Sites à la fois clients et serveurs
- Sites avec les mêmes fonctionalités

Pureté P2P

Classes de systèmes P2P

- P2P hybrides Napster
- P2P « purs » Freenet, Gnutella, BitTorrent (trackers?)
- P2P hiérarchiques ou « super-peers »

FastTrack (KaZaA), Gnutella::UltraPeer, eDonkey

- P2P structurés (DHT-like)

Chord, Tapestry, Pastry, P-Grid, CAN, Kademlia (eMule)

- P2P sémantiques Routing Indices, SON

Avantages du P2P

- Mutualisation (peu coûteuse) des ressources inutilisées
- Haute disponibilité et tolérance aux pannes
- Auto-organisation

Critères de comparaison

- Recherche de ressource
  - Topologie : ouverte ou contrôlée
  - Placement des données et index : libre ou dirigé
  - Routage : fonction de choix des successeurs
- Besoins applicatifs
  - Expressivité du langage : égalité, préfixe, intervalle, regexp, ...
  - Exhaustivité du résultat : exact, partiel, top-k...
  - Autonomie des nœuds
- Performance
  - Efficacité : trafic, charge...
  - Grandeur d'échelle : #nœuds, débit Q...
  - Tolérance aux pannes et robustesse

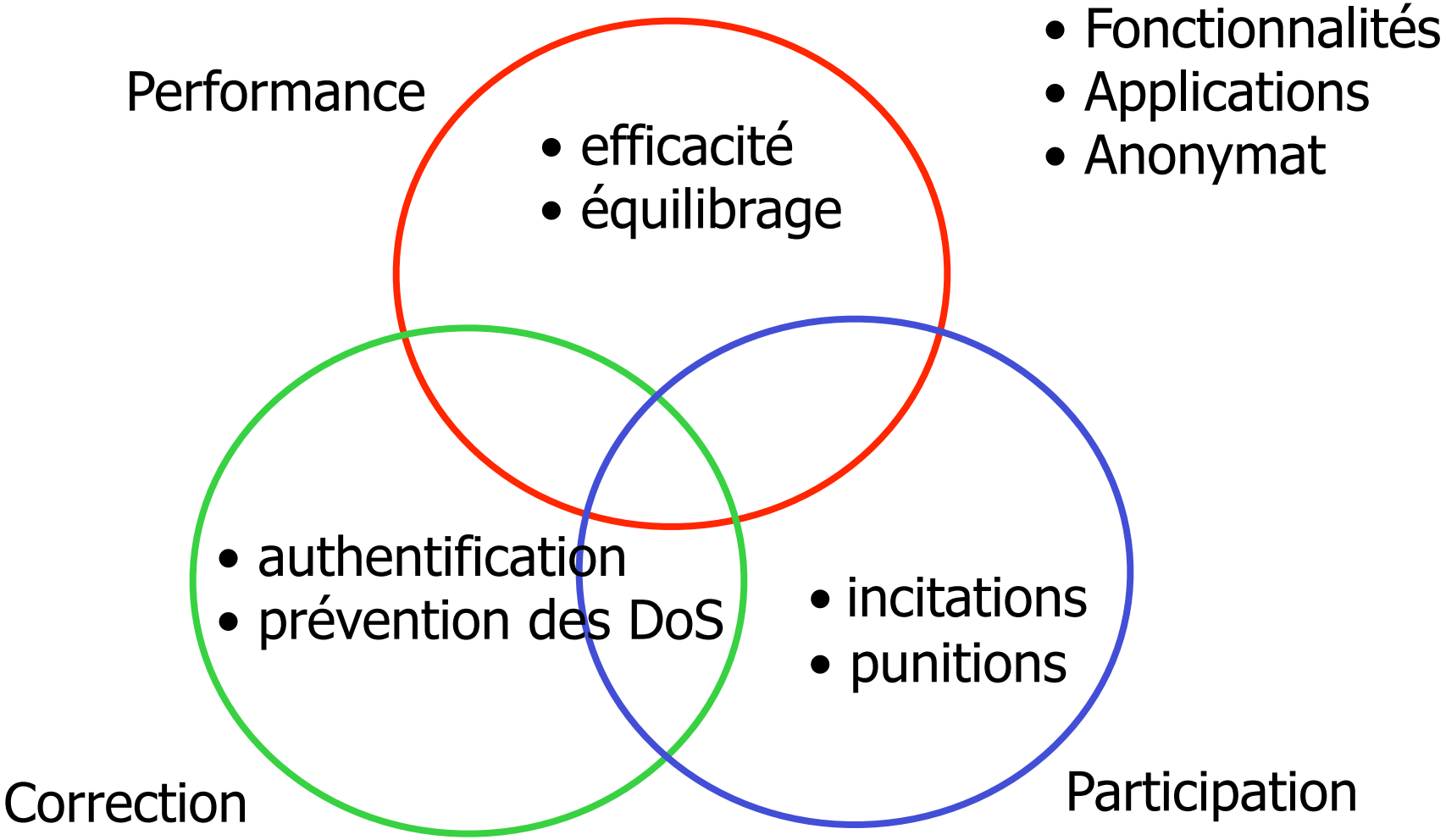
Comparaison

	Gnutella	CAN	Others?
Expressivness	****		
Comprehensivness	**		
Autonomy	****		
Efficiency	*		
Robustness	***		
Topology	pwr law		
Data Placement	arbitrary		
Message Routing	flooding		

Comparaison

	Gnutella	CAN	Others?
Expressivness	****	*	
Comprehensivness	**	****	
Autonomy	****	**	
Efficiency	*	***	
Robustness	***	**	
Topology	pwr law	grid	
Data Placement	arbitrary	hashing	
Message Routing	flooding	directed	

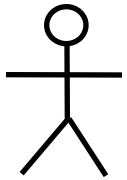
Problèmes ouverts



Problèmes ouverts : “mauvais garçons”

- disponibilité (e.g., traitement des attaques DoS)
- authentification
- anonymat
- contrôle d'accès (e.g., protection des IPs, paiement sécurisé, ...)

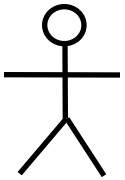
Authentification



?

titre: L'origine des espèces
auteur: Charles Darwin
date: 1859
contenu: Sur une île très loin d'ici...
...

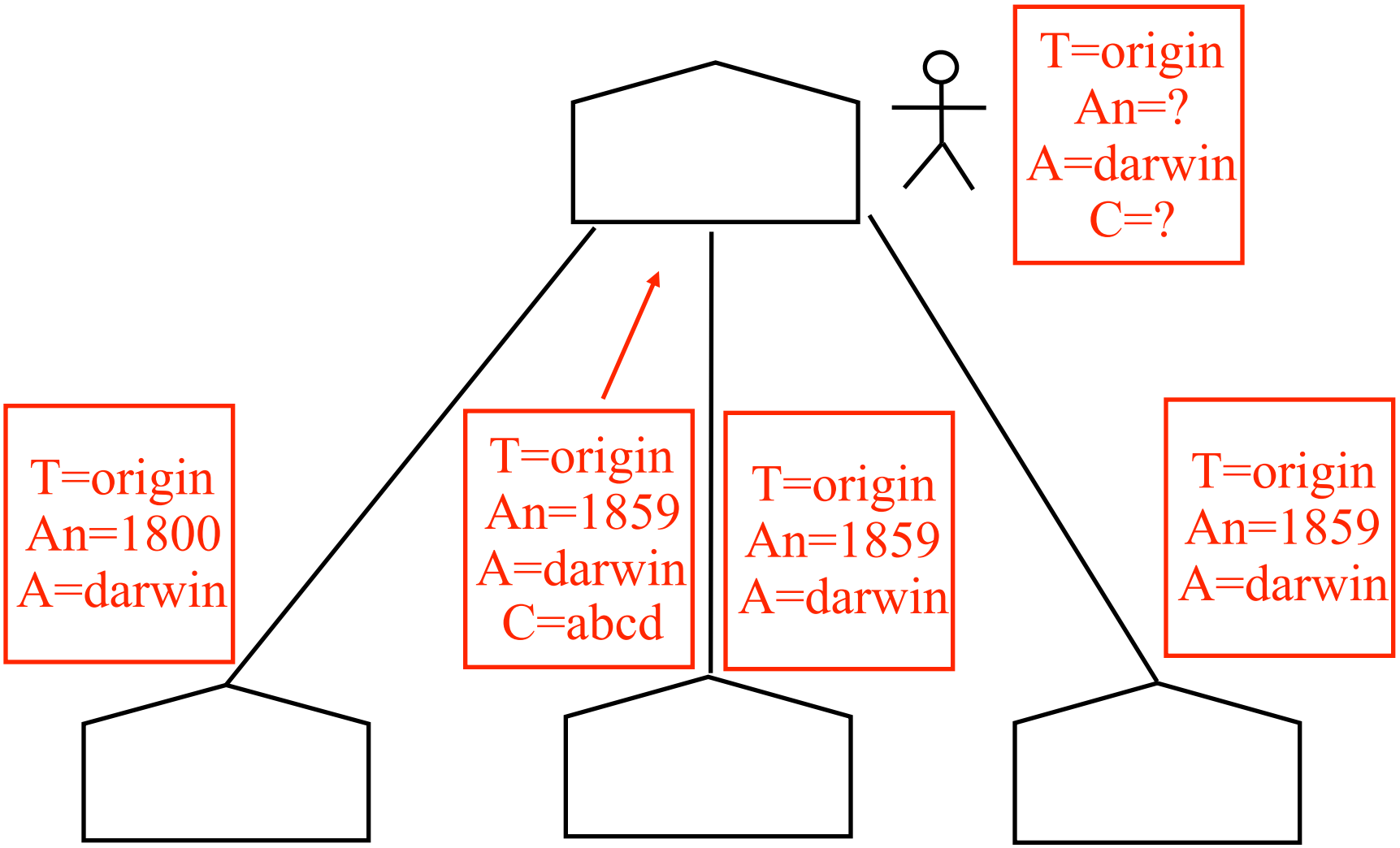
Authentification



?

titre: L'origine des espèces
auteur: Charles Darwin
date: 1859 00
contenu: Sur une île très loin d'ici...
...

Chargement de plusieurs fichiers



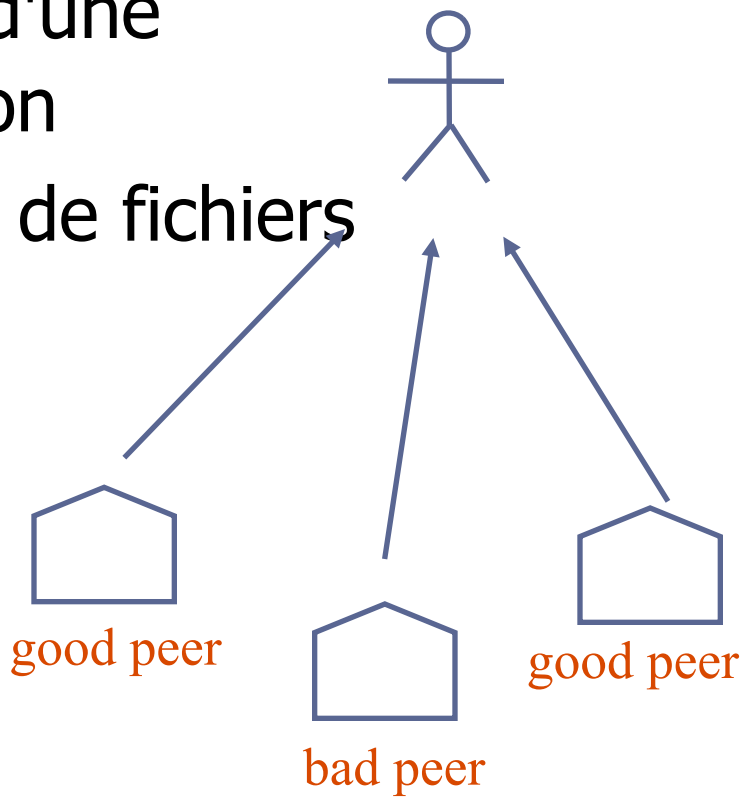
Solutions

- Fonction d'authentification A(doc): V/F
  - sur des sites de confiance, sur tous les sites ?
  - signature expert → sig(doc) → utilisateur
- Vote
  - la vérité est détenue par la majorité
- Estampille
  - e.g., la version (disponible) la plus ancienne est authentique

Autre défi : l'efficacité

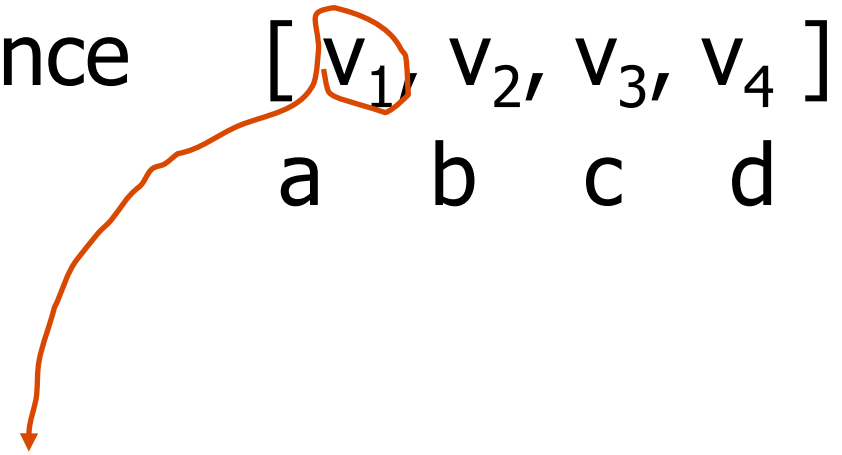
- Exemple: partage de musique
  - tout le monde dispose d'une fonction d'authentification
  - mais le téléchargement de fichiers est coûteux

- Solution:
  tracer le comportement des pairs



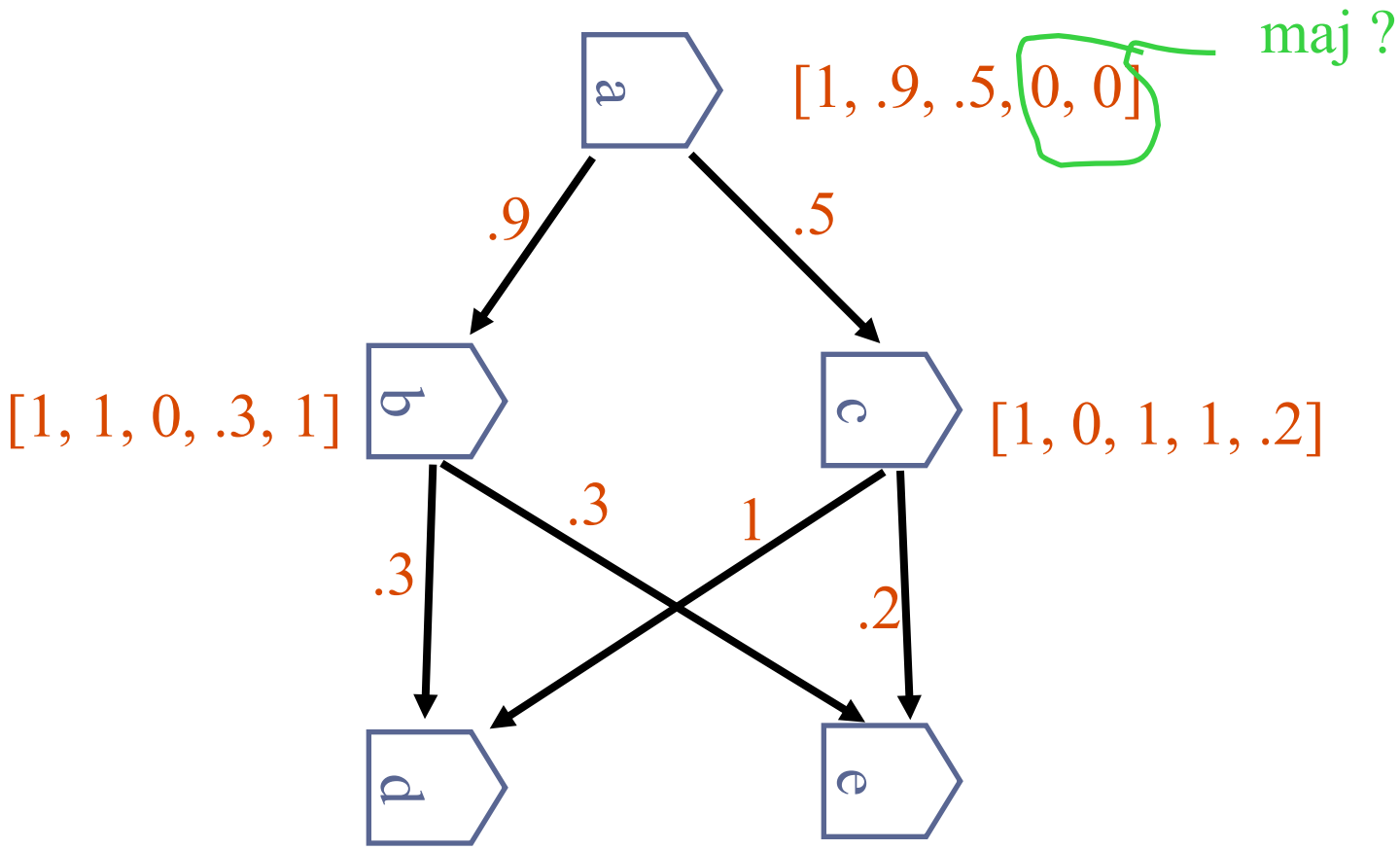
Comment faire ?

- Vecteur de confiance



- Valeur scalaire entre 0 et 1 ?
- Paire de valeurs
  [dl totaux, bons dl] ?

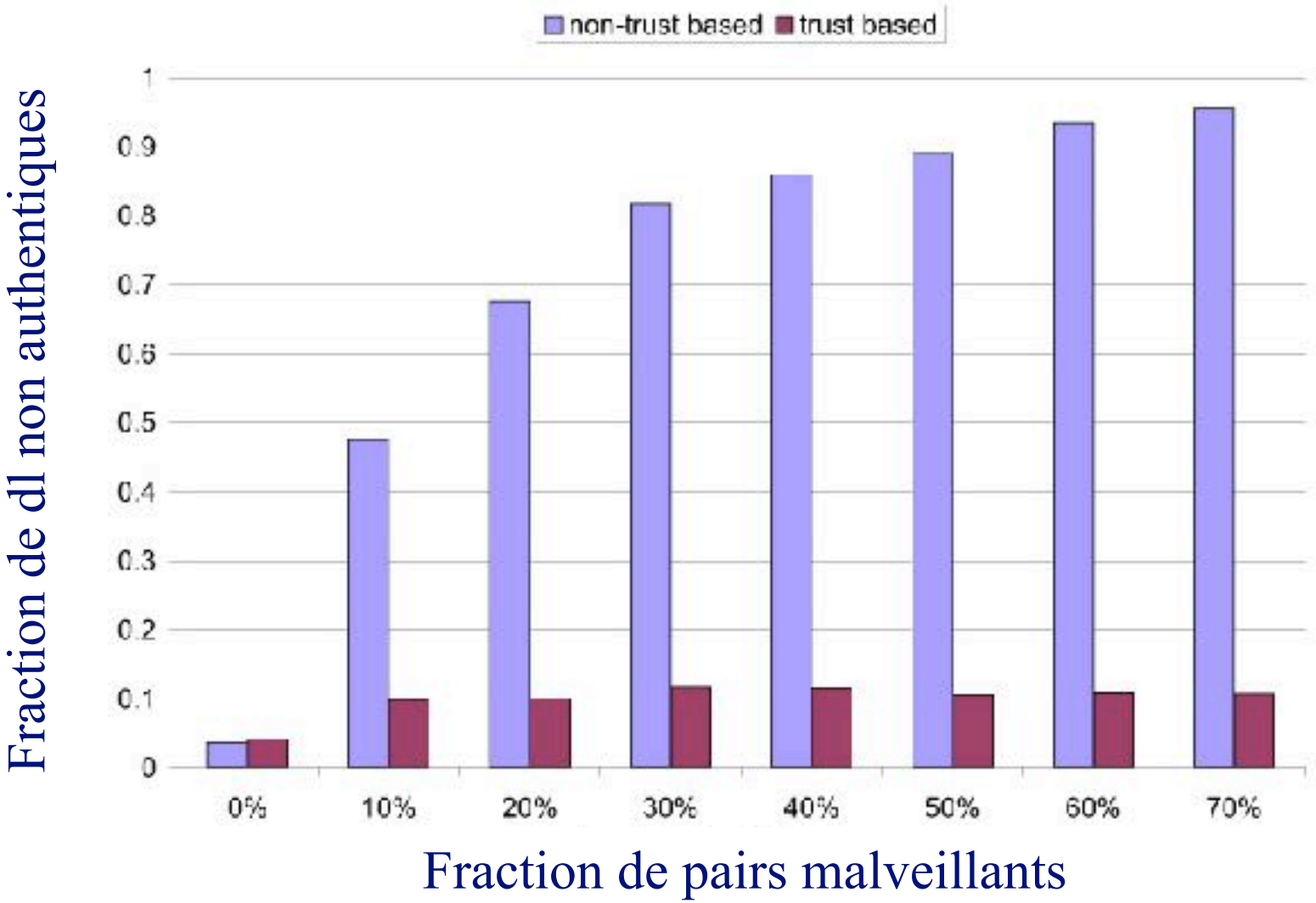
Opérations de confiance



Problèmes

- Calcul de confiance dans les systèmes dynamiques
- Surcharge des pairs de confiance
- Contenu intéressant sur les mauvais pairs
- Réputation des mauvais pairs
- Communautés de mauvais pairs
- ...

Echantillon de résultats



Participation & Incitation

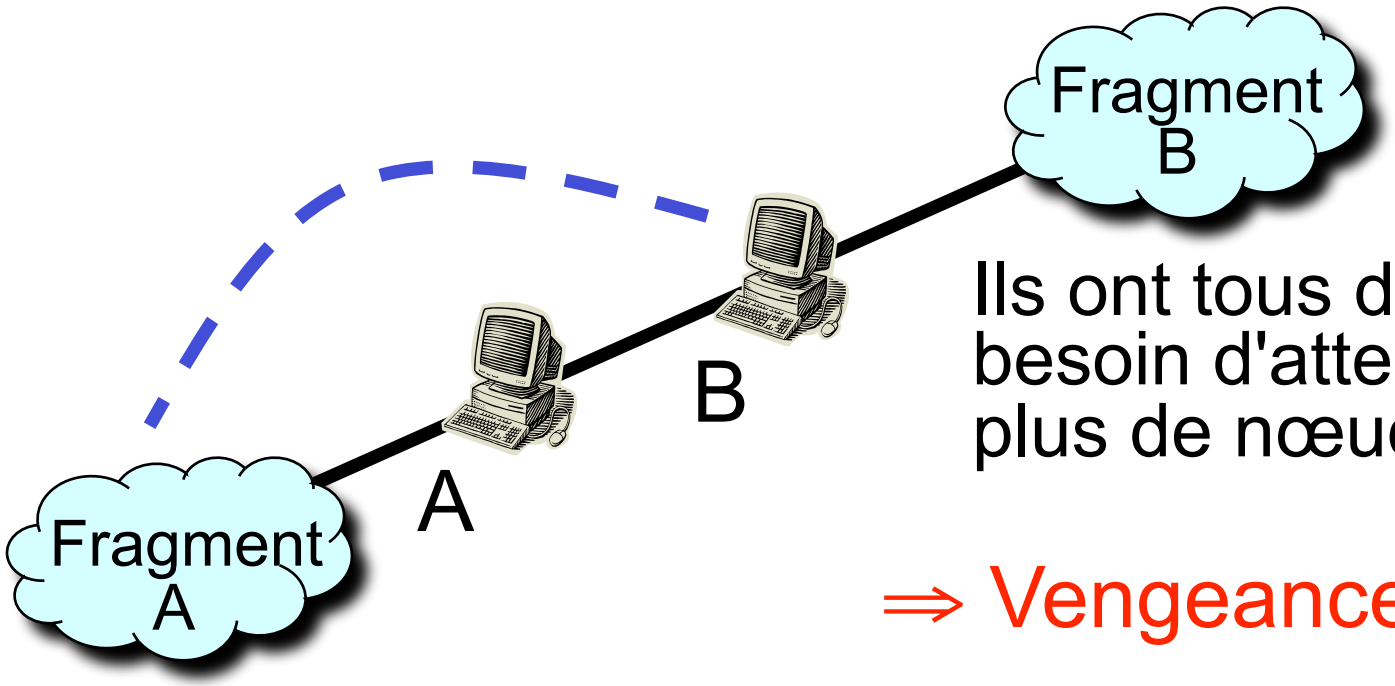
- Les pairs autonomes requièrent un mécanisme d'incitation (une carotte) pour travailler ensemble
  - Transférer les messages
  - Réaliser les calculs
  - Partager/stocker les fichiers
  - Fournir les services
  - Etc.

Les formes d'incitation

- Trois principaux types d'incitation (à ce jour) :
  - Donnant-donnant (Tit for tat)
  - Réputation
  - Argent/Monnaie

Donnant-donnant

- "Je fais pour toi ce que tu fais pour moi"
- Exemple

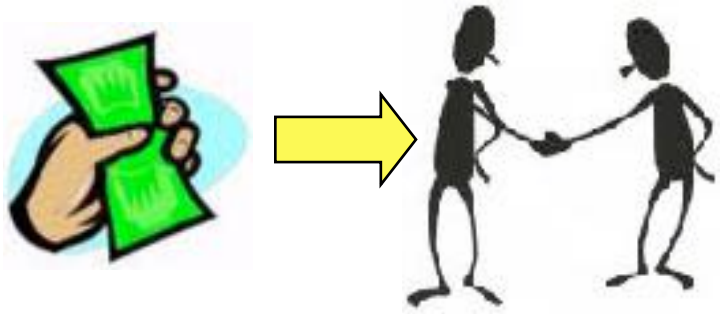


Ils ont tous deux besoin d'atteindre plus de nœuds

⇒ Vengeance !

Réputation et Monnaie

- "Si tu fais quelque chose pour moi, je te donnerai de la réputation/monnaie"



Pros and Cons

- Donnant-donnant
  - Pros : infrastructure et surcoût minimaux, moins sujet à tricherie
  - Cons : relations symétriques requises
- Monnaie
  - Pros : tout le monde veut de l'argent ! Pour certaines applications, c'est même requis
  - Cons : infrastructure très lourde, thésaurisation
- Réputation
  - Pros : convient dans beaucoup de situations
  - Cons : surcoût, problèmes inhérents d'incitation

Réputation et Monnaie

- Pour ces techniques, il y a deux questions importantes :
  - Si on dispose d'argent ou de score de réputation, comment les utiliser pour motiver les pairs à participer ?
  - Comment implémenter l'argent ou le score de réputation à la mode P2P ?

En résumé

- Recherche (localisation/consultation)
- Maj (insertion/défaillance)
- Topologies et étude comparative
  - DHT : Chord, CAN, P-Grid
  - P2P « pur » : Gnutella
  - Super-pairs : KaZaA, Gnutella/UltraPeer
- Traitement des « mauvais garçons »
- Traitement des parasites