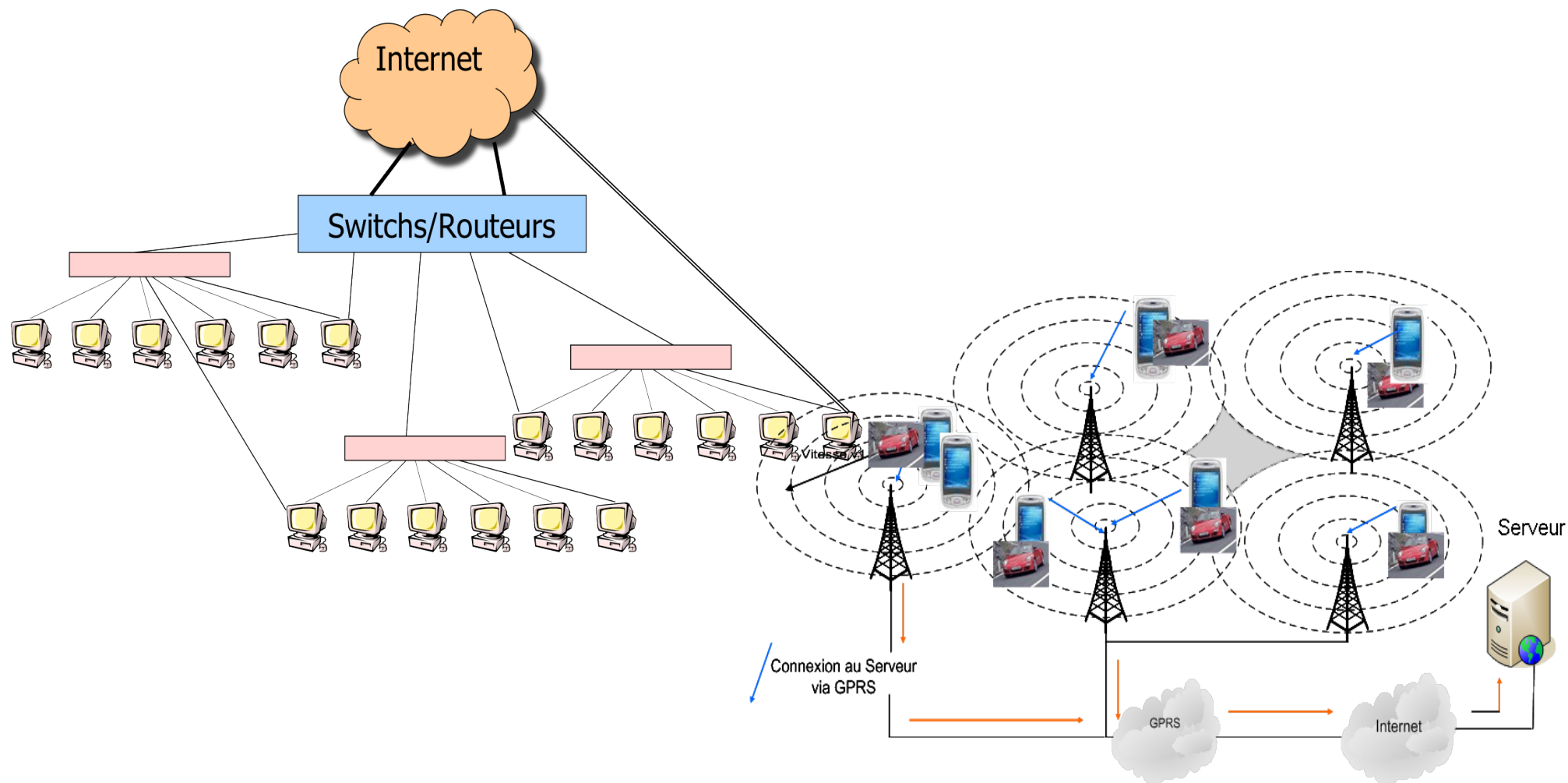


# Distributed Algorithms

**Nabil Abdennadher**

nabil.abdennadher@hesge.ch

# Infrastructure



# Objectifs

- Etude théorique des algorithmes distribués de base
- Complexité : Exprimée en terme de messages échangés

# Plan

- Hypothèses de travail
- Convergecast
- Broadcast
- Constructing Spanning Trees (ST)
- Test\_Connectivity
- Shortest path
- Elections algorithms
- Content Searching Algorithms

- En ce qui concerne le réseau ...

# Hypothèses de travail

- Le réseau est représenté par un graph:  $G = (N, E)$ 
  - N: Les noeuds sont les processeurs (or devices)
  - E: L'ensemble des liens entre les noeuds
- Les liens sont bi-directionnels.
- La communication est basée sur la transmission de messages (***en ce qui concerne les alg. distribués qui seront vus en cours***)
- Un message est représenté par le couple  $(q, Msg)$ 
  - $Msg$  doit être délivré au noeud  $q$
  - $Msg : (u, msg)$  où  $u$  est la tâche exécutée par le noeud  $q$  et  $msg$  est le message à délivrer à la tâche  $u$

# Hypothèses de travail: Routage

- Au niveau de chaque nœud  $r$ , il existe une fonction appelée  $next_r(q)$ 
  - prochain nœud à qui envoyer le message (sachant que nous sommes au niveau du nœud  $r$ ) pour atteindre  $q$
  - $(r, next_r(q))$  appartient à  $E_p$ .
- Cette fonction peut être fixe ou dynamique
- Elle peut être déterministe ( $next_r(q)$  est unique) ou non
- On supposera dans ce cours que la propriété FIFO est vérifiée
  - Au niveau d'un lien
  - Au niveau de l'arrivée des messages au nœud destination

# Hypothèses de travail: Flux de contrôle

- Cette fonction concerne le stockage des données en transit et la gestion des ressources de communication (liens, buffers)
- Les types de flux de contrôle les plus connus
  - Store and forward
  - Circuit switching
  - Wormhole routing



## Flux de contrôle: Store and forward

- $(q, \text{Msg})$  est décomposé en plusieurs packets
- Les packets contiennent les même informations de routage que le message lui même
- Les paquets sont transmis en parallèle
- Pour garantir l'ordre, les packets sont numérotés
- Possibilité de deadlock: taille limitée des buffers

## Flux de contrôle: Circuit switching

- Le chemin est réservé avant le début du transfert
- Possibilité de deadlock: un lien peut faire partie de plusieurs chemins
- Pas efficace pour les petits messages

# Flux de contrôle: flow control (wormhole routing)

- Les packets sont décomposés en flit (flow-control-digit)
- Le flit leader trace le chemin qui sera suivi par les autres flits
- Seul le flit leader contient les informations de routage
- Contrairement à la méthode store-and-forward, la transmission des packets est pipelinée

# Hypothèses de travail: Flux de contrôle

- On supposera dans ce cours que le flux de contrôle utilisé garantit l'absence de deadlock

- En ce qui concerne l'application informatique (l'algorithme) ...

# Une application, un graphe ...

- L'application est représentée par un graphe orienté  $G_T = (N_T, D_T)$ .
- Pour chaque tâche  $t$ :
  - $Int_t$  : les liens entrant dans  $t$ .  $Int_t$  appartient à  $D_T$
  - $Out_t$  : les liens sortant de  $t$ .  $Out_t$  appartient à  $D_T$
- Une tâche  $t$  est réactive (message driven)
  - Elle effectue un traitement en réponse à un message reçu.
  - Dans certains cas :
    - Une ou plusieurs tâches initialisent le traitement de l'application
    - Une tâche peut procéder à une étape d'initialisation

# Template d'un algorithme distribué

- Task\_t
  - Faire un traitement d'initialisation
  - Envoyer un message à un sous ensemble de  $Out_t$  (peut être l'ensemble vide)
  - Répéter
    - A la réception d'un message sur un canal  $c_1$  (de  $int_t$ ) et  $Cond_1 \rightarrow$ 
      - Faire un traitement
      - Envoyer un message à un sous ensemble de  $Out_t$
    - ou ...
    - A la réception d'un message sur un canal  $c_n$  (de  $int_t$ ) et  $Cond_n \rightarrow$ 
      - Faire un traitement
      - Envoyer un message à un sous ensemble de  $Out_t$
  - Jusqu'à (une condition de fin)

# Template d'un algorithme distribué

- Condition d'arrêt
  - La tâche est capable de « détecter » la condition d'arrêt à partir des messages qu'elle reçoit
  - La condition d'arrêt suppose aussi que la tâche ne reçoit plus de messages
- guard → command
  - La commande est exécutée lorsque « guard » est prêt: message reçu et condition booléenne  $Cond_i$  vérifiée
  - Un seul « guard » est exécuté à chaque itération
    - En cas de présence de plusieurs « guard », un est sélectionné au hasard
    - En cas d'absence de « guard », la tâche ne fait rien



## Encore des remarques ...

- Envoi et réception de messages
  - L'envoi est non bloquant
  - La réception est bloquante
  - Ce schéma évite les «deadlocks»
- Les canaux (liens entre les tâches) délivrent les messages dans un ordre FIFO

# Plan

- Hypothèses de travail
- **Collecte (Convergecast)**
- Diffusion (Broadcast)
- Construction d'arbres de recouvrement
- Identification des noeuds d'un réseau
- Calcul du plus court chemin
- Algorithme d'élection
- Algorithmes de recherche

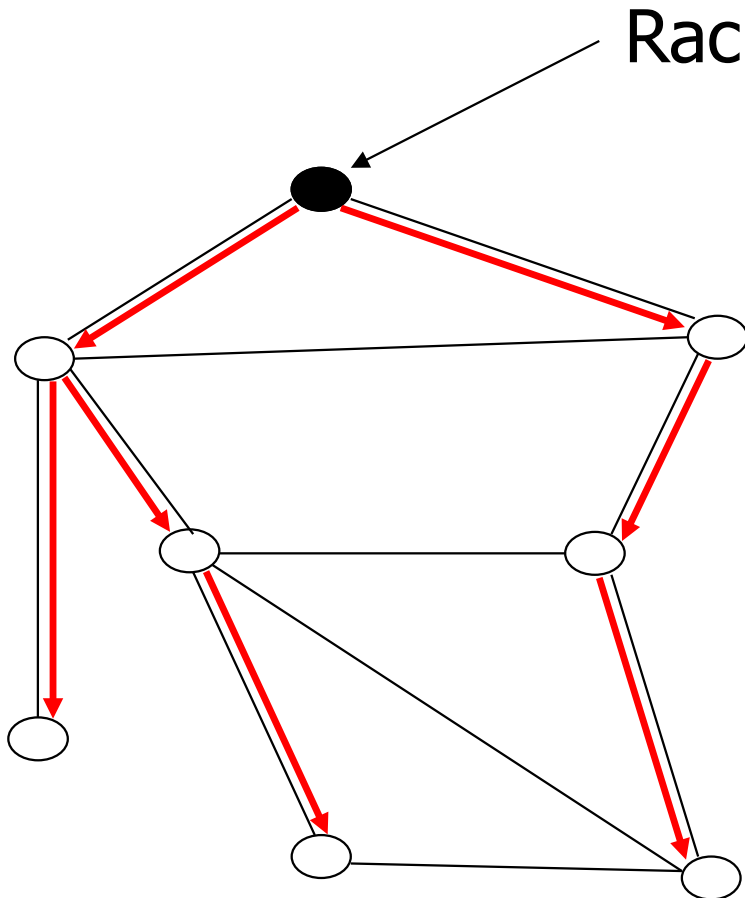
# Qu'est ce qu'un convergecast ?

- Tous les noeuds envoient une donnée locale à un même noeud destinataire
- Au départ :
  - Chaque noeud  $n_i$  dispose d'une donnée locale  $D_i$
- A la fin :
  - Toutes les données  $D_i$  sont reçues par un noeud destinataire  $n_r$ .

# Structure de données : Arbre de recouvrement

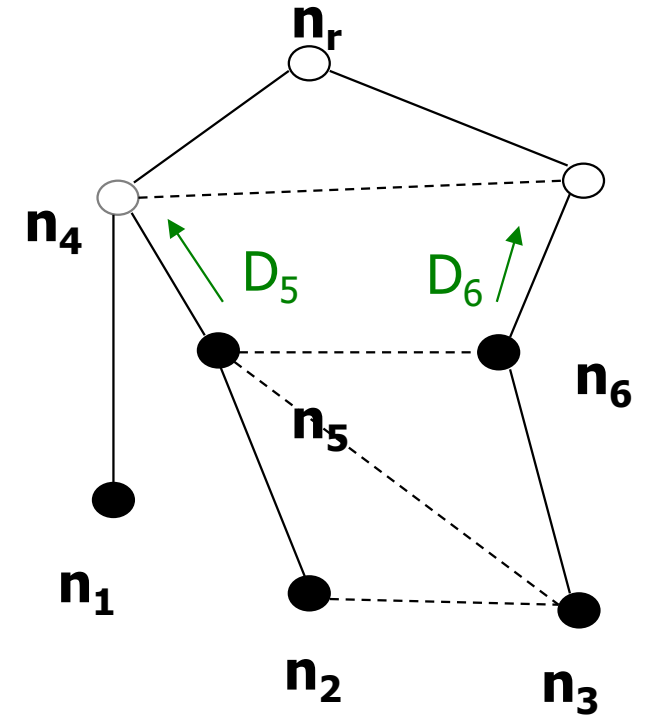
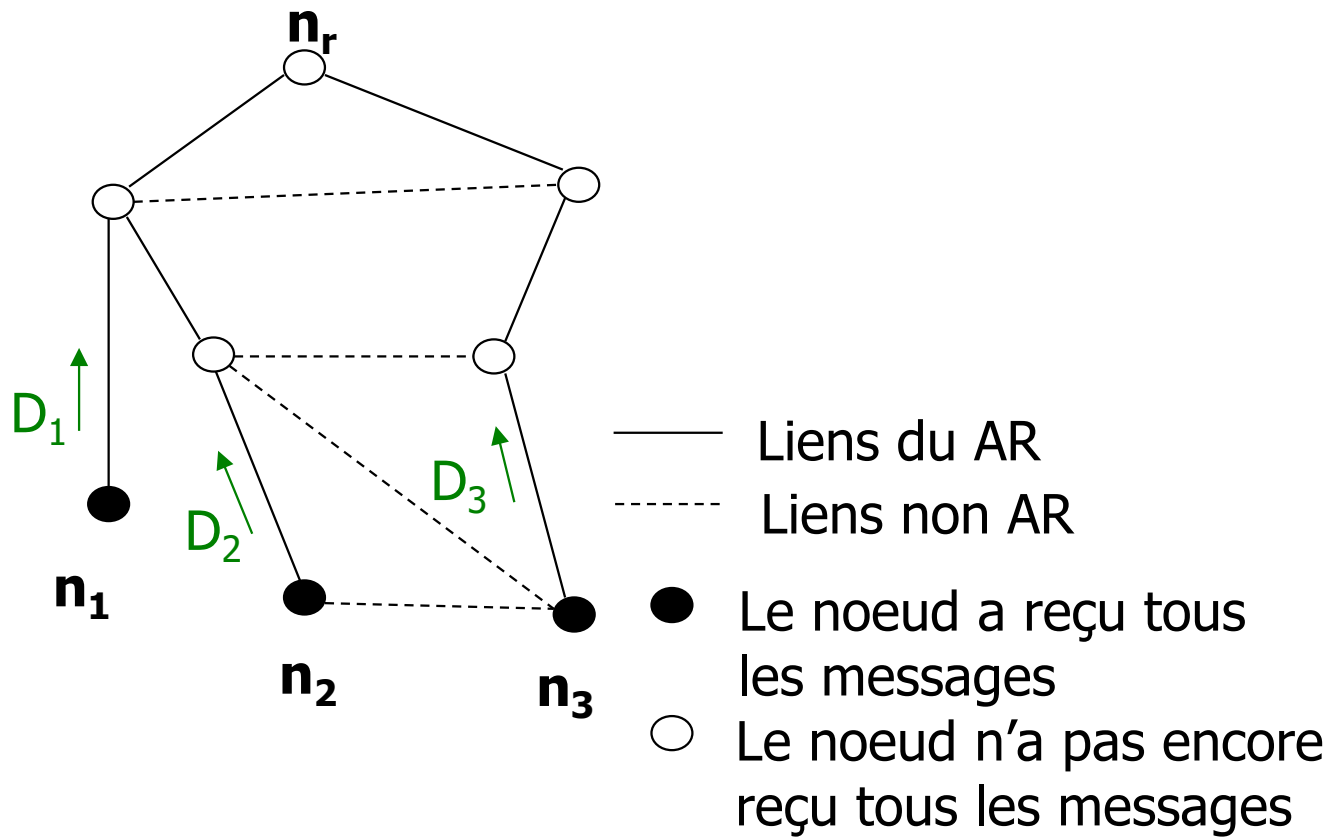
- Le réseau est modélisé par une structure appelée arbre de recouvrement (Spanning Tree)
  - $AR$  est connu
    - Au niveau de chaque nœud : le père + les fils
    - Représentation distribuée
- Pourquoi ?
  - Un arbre est plus facile à parcourir qu'un graphe

# Exemple d'arbre de recouvrement



- Le *AR* est représenté de manière distribuée :
  - Chaque noeud  $n_i$  connaît :
    - son père (NULL pour la racine)
    - ses fils (NULL pour les feuilles)

# Convergecast : Exemple



# Convergecast : Algorithme

- Chaque noeud feuille
  - Envoie la donnée locale  $D_i$  au noeud père
- Chaque noeud non feuille  $n_i$  ( $i \neq r$ )
  - A la réception des messages de tous les noeuds fils
    - Envoie l'ensemble des messages reçus au noeud père
- noeud  $n_r$ 
  - Reçoit les messages de la part de tous les noeuds fils

# Convergecast : Complexité

- $n - 1$  messages
- $n$  opérations



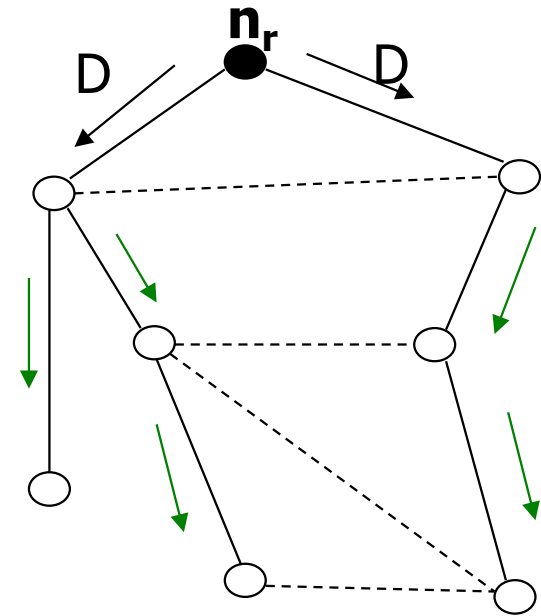
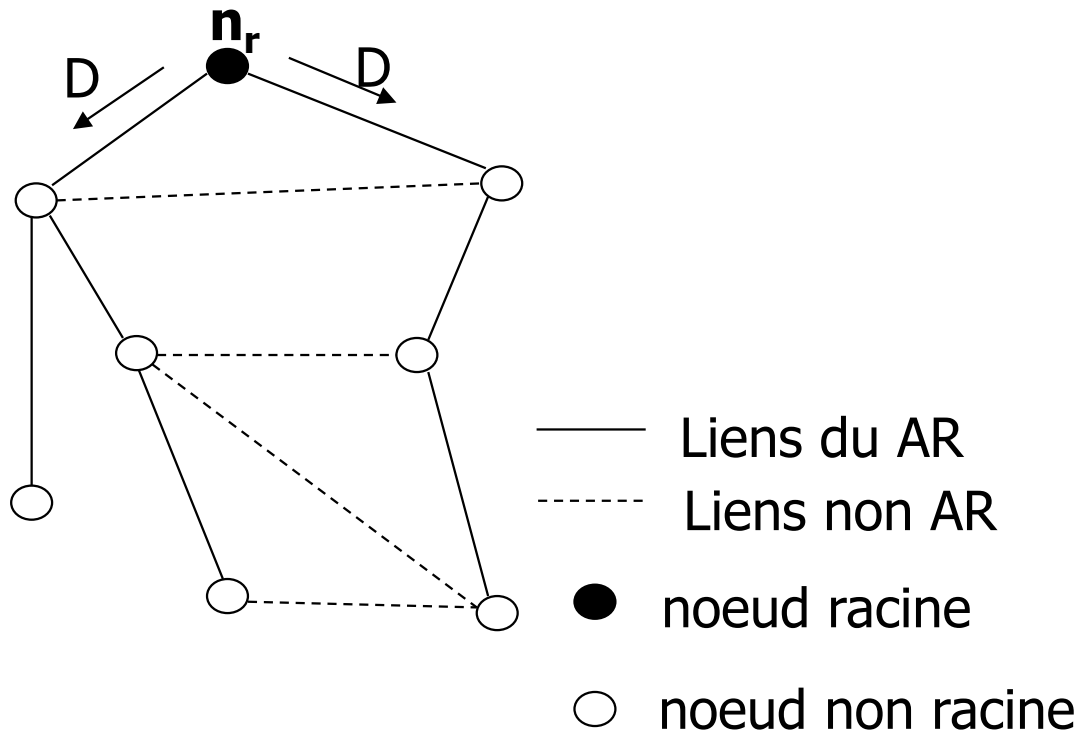
# Plan

- Hypothèses de travail
- Collecte (Convergecast)
- **Diffusion (Broadcast)**
- Construction d'arbres de recouvrement
- Identification des nœuds d'un réseau
- Calcul du plus court chemin
- Algorithme d'élection
- Algorithmes de recherche

# Qu'est ce qu'un braodcast ?

- Un (ou plusieurs) noeuds envoie une donnée  $D$  à tous les noeuds du réseau
- Initialement :
  - $D$  se trouve sur un (ou plusieurs) noeuds : Ensemble  $N_0$
- A la fin :
  - $D$  se trouve sur tous les noeuds du réseau.

# Broadcast avec arbre de recouvrement : Exemple



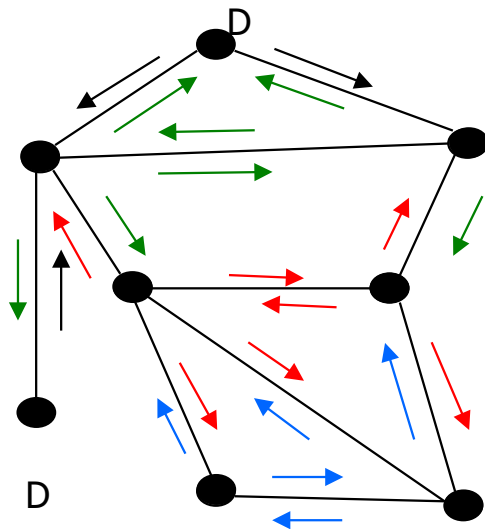
# Broadcast avec arbre de recouvrement : Algorithme

- Nœud racine  $n_r$ 
  - Envoie la donnée  $D$  à tous les fils (arbre  $AR$ )
- Noeuds  $n_i$  ( $i \neq r$ )
  - A la réception de la donnée  $D$  de la part du père
    - Si  $n_i$  n'est pas un nœud feuille
      - Envoyer  $D$  à tous les fils de  $n_i$

# Broadcast avec arbre de recouvrement : Complexité

- Chaque noeud :
  - Reçoit un message de son père (sauf la racine)
  - Envoie un message à ses fils (arbre AR)
- Nombre de messages :  $n - 1$  messages
- Nombre d'opérations :  $n$
- ... mais il faut construire le *AR* auparavant
- Lorsque  $|N_0| > 1$ , on peut utiliser plusieurs *AR* : Spanning Forest.

# Broadcast par vagues (Inondation)



- noeuds possédant D
- noeud ne possédant pas D

- Chaque noeud  $n_i$  gère :
  - Liste des voisins directs :  $v_i$
  - Un booléen ***atteint<sub>i</sub>***
    - initialisé à *faux*
    - mis à *vrai* si :
      - $n_i$  appartient à  $N_0$
      - $n_i$  reçoit  $D$  pour la 1ère fois

# Première vague : générée par $N_0$



# Broadcast par vagues : Algorithme

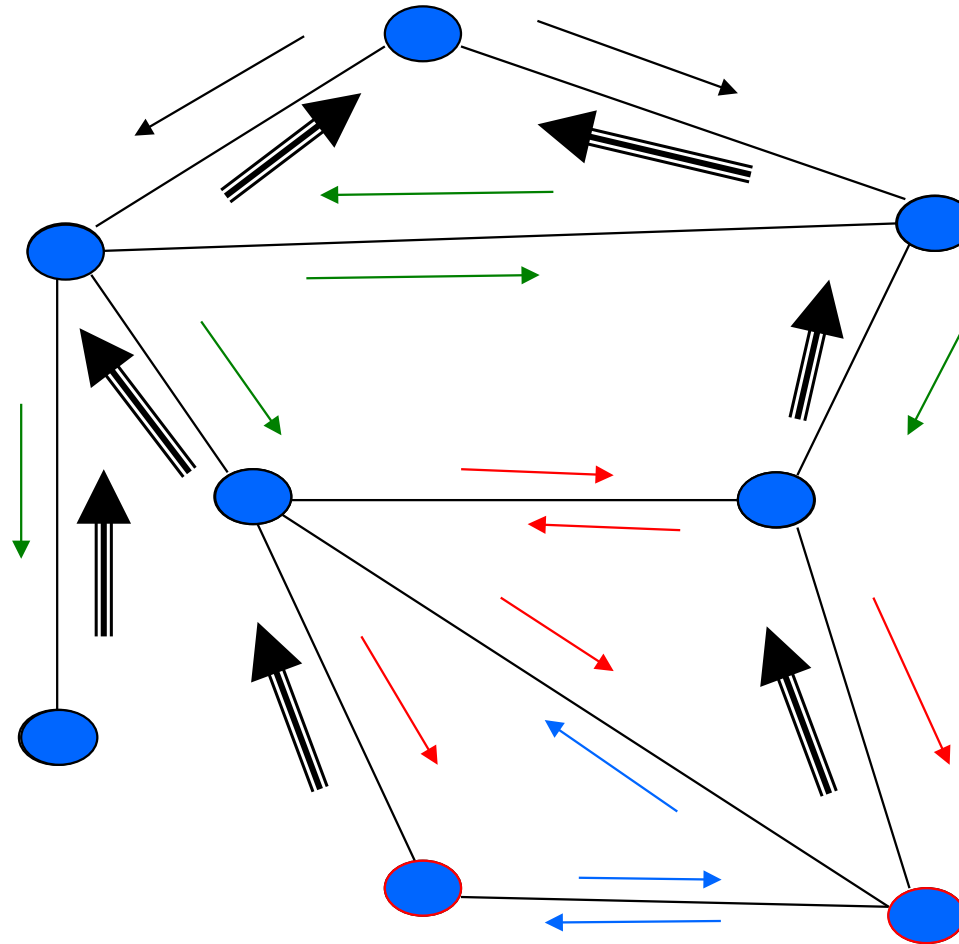
- $atteint_i = \text{faux};$
- Si  $n_i$  appartient à  $N_0$ 
  - $atteint_i = \text{vrai}$
  - Envoyer  $D$  à tous les voisins directs ( $v_i$ )
- A chaque réception de  $D$  (\*  $|v_i|$  réceptions \*)
  - Si  $atteint_i = \text{faux}$  (\* 1ère réception de  $D$  \*)
    - $atteint_i := \text{vrai}$
    - Envoyer  $D$  à tous les voisins directs de  $n_i$



# Broadcast par vagues : Complexité

- La propagation de  $D$  se fait par vagues (waves)
  - $n_i$  envoie une seule fois  $D$  à ses voisins directs
  - $n_i$  reçoit  $D$  autant de fois qu'il a de voisins.
  - Chaque lien du réseau transmet la donnée  $D$  deux fois : Une fois dans chaque direction
  - Complexité
    - $2m$  messages ( $m$  : nombre de liens dans le réseau)
- 33 •  $n$  “opérations” ( $n$  : nombre de noeuds)

# Broadcast par vagues avec accusé de réception: Principe



# Broadcast par vagues avec accusé de réception

- L'ensemble  $N_0$  est un singleton
- Chaque nœud :
  - est adopté par le 1<sup>er</sup> nœud qui lui envoie D
  - Envoie D à tous ses voisins directs, sauf son père
  - Recoit D autant de fois qu'il a de voisins directs
  - A la réception du dernier D, envoie D à son père

# Broadcast par vagues avec accusé de réception

- Chaque noeud  $n_i$  gère :
  - Une liste des voisins directs :  $v_i$
  - Un booléen  $atteint_i$  initialisé à *faux*
  - Un compteur  $count_i$  initialisé à 0
  - $parent_i$  initialisé à NULL

# Broadcast par vagues avec accusé de réception : Algorithme

- Si  $n_i$  appartient à  $N_0$  ( $n_i$  est le noeud racine)
  - $atteint_i := vrai$
  - Envoyer  $D$  à tous les voisins directs

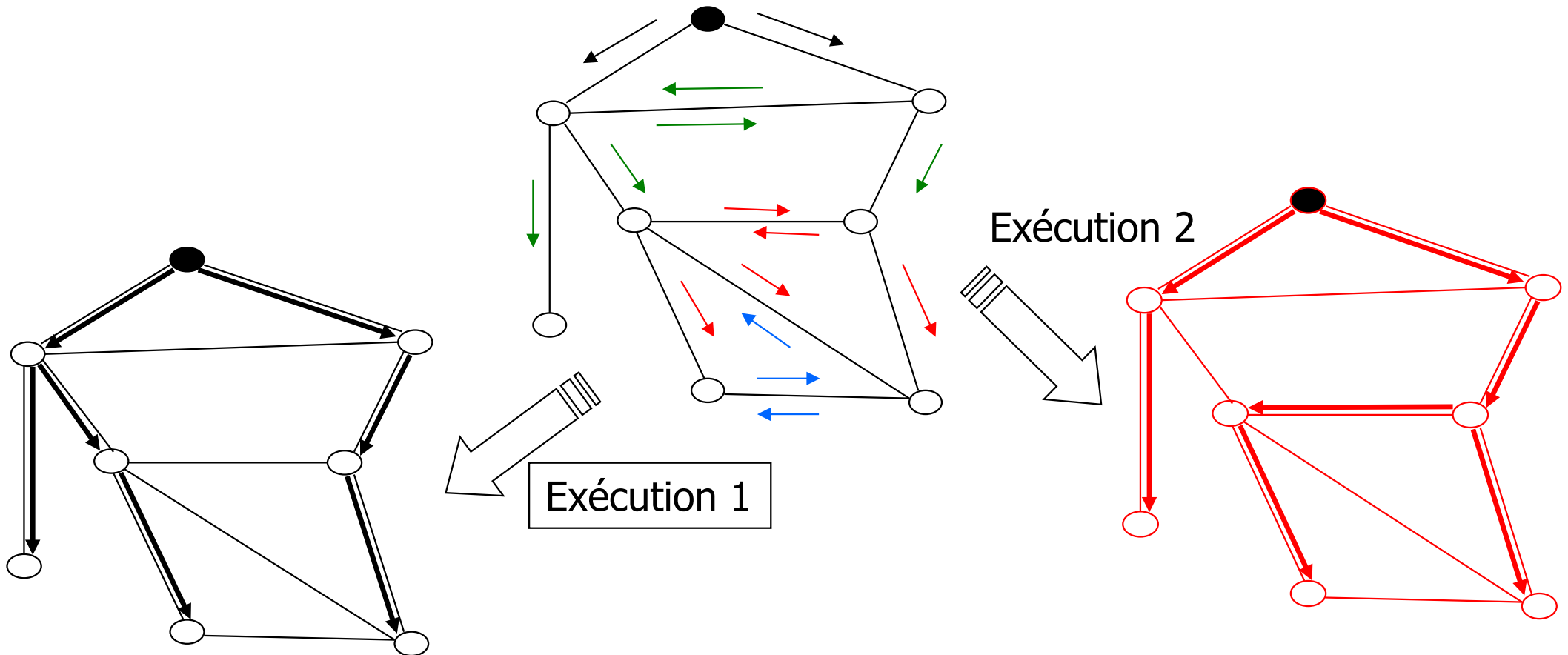
(\* Le reste est exécuté quelque soit le noeud \*)

- A chaque réception  $D$  de la part d'un noeud  $n_j$  (\* $|v_i|$  mesg\*)
  - $count_i = count_i + 1$ ;
  - Si  $atteint_i = faux$ 
    - $atteint_i := vrai$
    - $parent_i := n_j$
    - Envoyer  $D$  à tous les voisins directs de  $n_i \neq n_j$
  - Si  $count_i = |v_i|$  et  $parent_i \neq NULL$ 
    - Envoyer  $D$  à  $parent_i$

# Broadcast par vagues avec accusé de réception: Complexité

- Le noeud root  $n_r$  envoie  $|v_r|$  messages
- Chaque noeud  $n_i \neq n_r$ 
  - Envoie  $|v_i| - 1$  messages
  - Envoie un accusé de réception (D) à son père
- Nombre de messages :  $2m$ 
  - $2m - n + 1$
  - En plus de l'accusé de réception :  $n - 1$  (nombre de liens dans l'arbre construit)
- Nombre "d'opérations" :  $n$

# Broadcast par vagues avec ou sans accusé de réception : Non déterminisme



# Les algorithmes Broadcast en bref

- Par vagues (ou inondation)
  - Asynchrone
  - $2m$  messages
  - 😊 • **Avantage** : Les liens utilisés varient d'une exécution à l'autre
- Par vague avec accusé de réception
  - Synchrone
  - $2m$  messages
  - 😊 • **Avantages**
    - Les liens utilisés varient d'une exécution à l'autre
    - Construction de l'arbre de recouvrement
- En utilisant un arbre de recouvrement
  - $n - 1$  messages
  - ☹ • **Inconvénient**
    - Les mêmes liens inter-nœuds sont toujours utilisés
    - Non tolérance aux pannes



# Plan

- Hypothèses de travail
- Collecte (Convergecast)
- Diffusion (Broadcast)
- **Construction d'arbres de recouvrement**
- Identification des noeuds d'un réseau
- Calcul du plus court chemin
- Algorithme d'élection
- Algorithmes de recherche

# Trois algorithmes

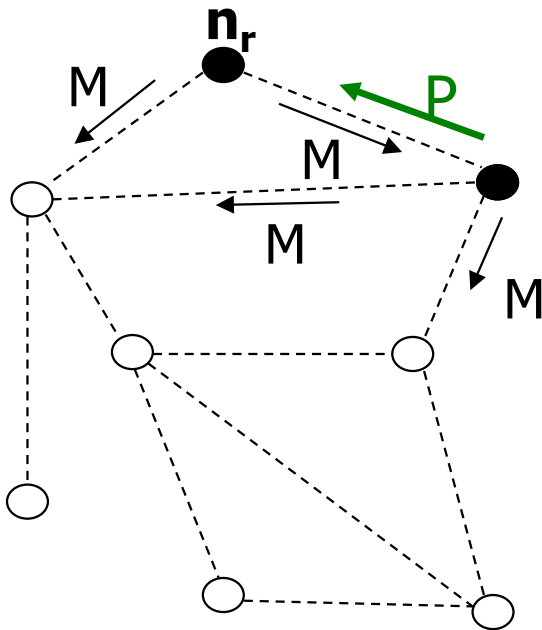
- Construction d'un AR en largeur d'abord (Bredth First Serach : BFS) avec racine identifiée
- Construction d'un AR en profondeur d'abord (Depth First Search : DFS) avec racine identifiée
- Construction d'un AR sans racine identifiée

# Exemple

M : Demande d'adoption

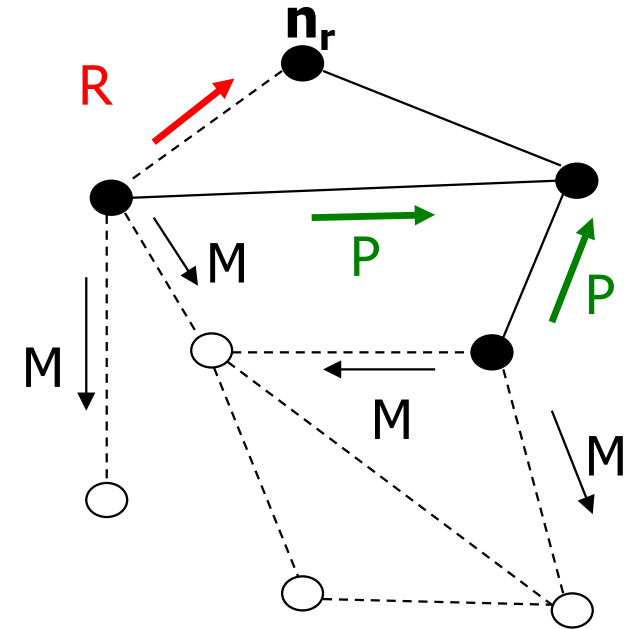
P : Salut papa

R : Non ! Tu n'es pas papa



— Liens du AR  
- - - Liens non AR

● Le noeud a reçu  
le message M  
○ Le noeud n'a pas  
encore reçu M



# Structure de données

- Pour chaque noeud  $n_i$ 
  - **parent** : parent du nœud  $n_i$ . Initialisé à NULL
  - **F** : Ensemble des fils de  $n_i$ . Initialisé à NULL
  - **NF** : Ensemble des Non Fils de  $n_i$ . Initialisé à NULL
  - **v** : Ensemble des voisins de  $n_i$

# Les messages

- Trois types de messages :
  - M : Demande d'adoption d'un fils
  - P : Tu as demandé mon adoption, tu es mon papa
  - R : Tu as demandé mon adoption, merci mais j'ai déjà un papa.

# Algorithme (exécuté par $n_r$ )

- Pour le noeud racine  $n_r$ 
  - Envoi d'un message  $\langle M \rangle$  à tous les voisins
  - Attente de messages  $\langle M \rangle$ ,  $\langle P \rangle$  ou  $\langle R \rangle$
  - A la réception d'un message  $\langle M \rangle$ 
    - Envoyer un message  $\langle R \rangle$
  - A la réception d'un message  $\langle P \rangle$ 
    - Actualiser la liste des noeuds fils :  $F$
  - A la réception d'un message  $\langle R \rangle$ 
    - Actualiser la liste des noeuds non fils :  $NF$

# Algorithme (exécuté par $n_i$ , $i \neq r$ )

- Répéter
  - A la première réception de M de la part de  $n_j$ 
    - $n_i$  envoie un message <P> à  $n_j$
    - $parent = n_j$
    - $n_i$  envoie un message <M> à tous les voisins autres que  $n_j$
  - Pour les autres messages <M> reçus
    - $n_i$  envoie le message <R> à l'émetteur
  - Pour un message <P> reçu
    - Actualiser la liste des processus fils :  $F$
  - Pour un message <R> reçu
    - Actualiser la liste des processus non fils :  $NF$
- <sup>47</sup>Jusqu'à (condition d'arrêt)

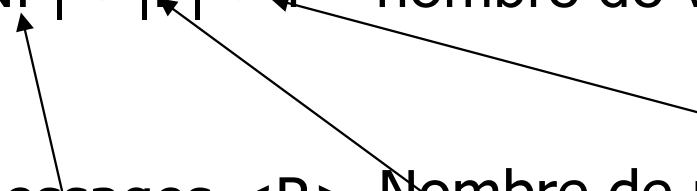


# Test d'arrêt

- Un noeud non racine s'arrête d'attendre des messages lorsque :

- $|NF| + |F| + 1 = \text{nombre de voisins}$

Nombre de messages <R>      Nombre de messages <P>



père

- Le nœud racine s'arrête d'attendre des messages lorsque :

□  $|NF| + |F| = \text{nombre de voisins}$



# Remarques

- L'algorithme BFS est écrit de sorte à privilégier un parcours parallèle des voisins.
  - L'AR généré est en général large avec peu de niveaux
- Est si on désire obtenir un *AR* étroit avec beaucoup de niveaux ?

# Trois algorithmes

- Construction d'un AR en largeur d'abord (Breadth First Search : BFS) avec racine identifiée
- Construction d'un AR en profondeur d'abord (Depth First Search : DFS) avec racine identifiée
- Construction d'un AR sans racine identifiée

# Structure de données

- Pour chaque noeud  $n_i$ 
  - **parent** : parent du nœud  $n_i$ . Initialisé à NULL
  - **F** : Ensemble des fils de  $n_i$ . Initialisé à NULL
  - **NF** : Ensemble des Non Fils de  $n_i$ . Initialisé à NULL
  - **v** : Ensemble des voisins de  $n_i$
  - **NE** : Ensemble des voisins de  $n_i$  non encore explorés. Initialement **NE** = **v**
  - **terminé** : booléen qui arrête l'algorithme au niveau du nœud  $n_i$ . Initialisé à faux.

# Algorithme (1)

- Pour le noeud racine  $n_r$ 
  - parent =  $n_r$
  - *Choisir un nœud  $n_k$  et l'enlever de NE*
  - *Envoi  $\langle M \rangle$  à  $n_k$*

# Algorithme (2)

- Au niveau d'un noeud  $n_i$ 
  - A la réception de  $\langle M \rangle$  de la part de  $n_j$ 
    - *si*  $parent = NULL$  /\* Premier Message M reçu \*/
      - $parent = n_j$
      - Enlever le noeud  $n_j$  de  $NE$
      - *si*  $NE \neq NULL$ 
        - Choisir un noeud  $n_k$  et l'enlever de  $NE$
        - Envoi  $\langle M \rangle$  à  $n_k$
      - *sinon* Envoi  $\langle P \rangle$  à  $parent$
    - *sinon* /\* Le Message M reçu n'est pas le premier \*/
      - $n_i$  envoie le message  $\langle R \rangle$  à l'émetteur
  - A la réception de  $\langle P \rangle$  ou  $\langle R \rangle$  de la part de  $n_j$ 
    - Si le message reçu est  $\langle R \rangle$  alors ajouter  $n_j$  à  $NF$  ainsi
    - Si le message reçu est  $\langle P \rangle$  alors ajouter  $n_j$  à  $F$  ainsi
    - Si  $NE_i \neq NULL$  /\* Il reste encore des voisins non explorés \*/
      - Choisir un noeud  $n_k$  et l'enlever de  $NE$
      - Envoyer  $\langle M \rangle$  à  $n_k$
    - Sinon /\* Il n'y a plus de voisins à explorer \*/
      - Si  $parent \neq n_i$  Envoi  $\langle P \rangle$  à  $parent$  ainsi; /\*  $n_i$  est le noeud racine d'un sous arbre de l'arbre de recouvrement final. Il envoie alors un message P à son père \*/
      - $terminé = vrai$

# Test d'arrêt

- Au niveau du nœud  $n_i$ , l'algorithme s'arrête lorsque l'arbre de recouvrement, dont la racine est  $n_i$ , est construit.
- Dans ce cas,
  - le booléen *terminé<sub>i</sub>* du nœud  $n_i$  est mis à vrai.
  - $n_i$  ne reçoit plus aucun message puisque ses nœuds voisins ont été tous explorés: ils ont tous construits un sous arbre dont ils sont la racine et ils ont déjà arrêté leur algorithme (leur booléen *terminé* a été mis à vrai)

# Trois algorithmes

- Construction d'un AR en largeur d'abord (Breadth First Search : BFS) avec racine identifiée
- Construction d'un AR en profondeur d'abord (Depth First Search : DFS) avec racine identifiée
- Construction d'un AR sans racine identifiée

# Construction d'un AR, racine non identifiée

- Chaque nœud  $n_i$  est identifié par un identificateur  $id_i$
- Chaque nœud candidat tente de construire un AR de type DFS en envoyant son propre  $id$ .
- Si deux arbres AR essaient de s'approprier un nœud, celui-ci est affecté à l'AR ayant la plus grande racine. L'autre arbre est bloqué et ne finira jamais.
- Chaque nœud gère une variable **leader** : le plus grand  $id$  reçu.



# Construction d'un AR, racine non identifiée

- Lorsqu'un nœud  $n_i$  reçoit un message ***id*** de la part de  $n_j$ 
  - Si ( $id > leader$ )
    - $n_i$  change de AR et envoie  $id$  à ses voisins ( $\neq n_j$ )
  - Si (***id***  $<$  leader)
    - $n_i$  bloque la construction de l'arbre
  - Si ( $id = leader$ ) //  $n_i$  et  $n_j$  appartiennent au même AR ayant la racine *leader*
    - $n_i$  envoie un message  $\langle R \rangle$  à  $n_j$
- Un seul nœud finit : celui qui a le plus grand id
- **Les autres nœuds restent en attente !**
- **Question : comment faire pour les « finir » ?**

# Structure de données

- Pour chaque noeud  $n_i$ 
  - ***id*** : identifiant du nœud  $n_i$
  - ***parent*** : parent du nœud  $n_i$ . Initialisé à NULL
  - ***leader*** : Plus grand *id* reçu par  $n_i$
  - ***F*** : Ensemble des fils de  $n_i$ . Initialisé à NULL
  - ***NF*** : Ensemble des Non Fils de  $n_i$ . Initialisé à NULL
  - ***v*** : Ensemble des voisins de  $n_i$
  - ***NE*** : Ensemble des voisins de  $n_i$  non encore explorés. Initialement ***NE*** = ***v***
  - ***terminé*** : booléen qui arrête l'algorithme au niveau du nœud  $n_i$ . Initialisé à faux.

# Algorithme

- Pour tous les noeuds
  - $parent = NULL;$
  - $leader = 0;$
  - $F = NULL;$
  - $NF = NULL;$
  - $NE = v;$
  - $terminé = faux$
- Chaque nœud  $n_i$  candidat
  - $parent = n_i$
  - $leader = id_i$
  - Choisir un nœud  $n_k$  et l'enlever de  $NE$
  - Envoi  $\langle leader \rangle$  à  $n_k$

# Algorithme ... suite (au niveau de $n_i$ )

- A la réception de  $\langle y \rangle$  de la part de  $n_j$ 
  - Si  $leader < y$  /\* Passage à un autre arbre AR \*/
    - $leader = y$ ;  $parent = n_j$ ;  $NE = v - n_j$ ;  $F = NULL$ ;  $NF = NULL$ ;
    - si  $NE \neq NULL$  /\* Tous les voisins ne sont pas explorés \*/
      - Choisir un nœud  $n_k$ , l'enlever de  $NE$  et envoyer  $\langle leader \rangle$  à  $n_k$
    - Sinon  $n_i$  envoie un message  $\langle P \rangle$  à  $parent$
  - Sinon /\*  $leader \geq y$  \*/
    - si  $leader = y$  alors envoi du message  $\langle R \rangle$  à l'émetteur  $n_j$  /\*  $n_i$  est déjà dans le même arbre que  $n_j$ . Dans la cas où  $leader > y$ , la construction de l'arbre est bloqué \*/
- A la réception de  $\langle P \rangle$  ou  $\langle R \rangle$  de la part de  $n_j$ 
  - Si le message reçu est  $\langle R \rangle$  alors ajouter  $n_j$  à  $NF$
  - Si le message reçu est  $\langle P \rangle$  alors ajouter  $n_j$  à  $F$
  - Si  $NE_i = NULL$ 
    - Si  $parent \neq n_i$  Envoi  $\langle P \rangle$  à  $parent$ ;
    - else  $terminé = vrai$  /\*  $n_i$  finit l'algorithme comme nœud racine de l'arbre de recouvrement final \*/
  - Sinon
    - Choisir un nœud  $n_k$ , l'enlever de  $NE$  et envoyer  $\langle leader \rangle$  à  $n_k$

# Test d'arrêt

- Le nœud racine s'arrête lorsque *terminé* est à vrai
- Les autres nœuds restent en état d'attente.
- **Question : comment les arrêter ?**

# Plan

- Hypothèses de travail
- Collecte (Convergecast)
- Diffusion (Broadcast)
- Construction d'arbres de recouvrement
- **Identification des nœuds d'un réseau**
- Calcul du plus court chemin
- Algorithme d'élection
- Algorithmes de recherche

# Problématique

- Objectif
  - Disposer d'une vue du réseau au niveau de chaque nœud
  - Chaque nœud s'identifie aux autres :
    - Type, Processeur, OS, performance, ressources, etc.
- Pourquoi ?
  - Tolérance aux pannes : panne de certains nœuds du réseau
  - Gérer la qualité de service (Quality of Service)
    - Choix du meilleur nœud pour un service donné

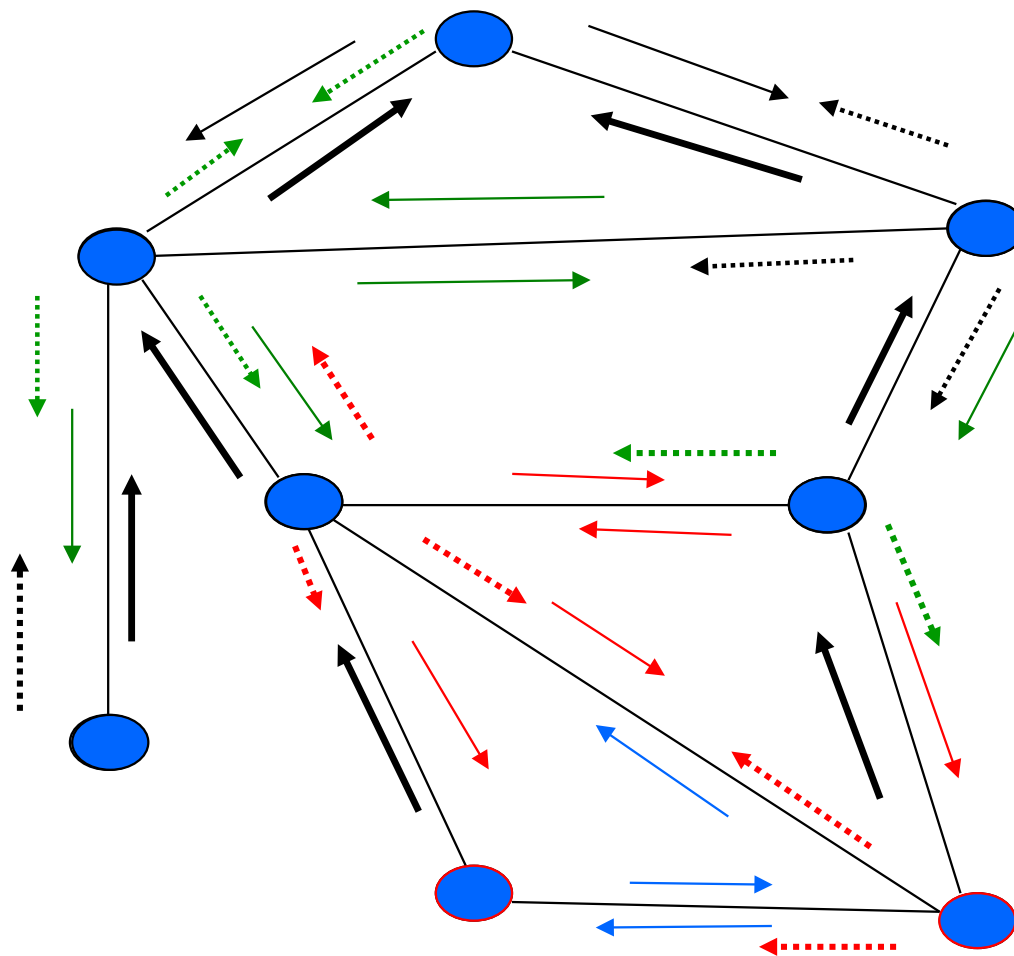
- 1ère solution
  - Exécuter séquentiellement l'algorithme Broadcast avec ACK  $n$  fois
    - Chaque nœud envoie son identification aux autres
- 2ème solution
  - Exécuter en parallèle l'algorithme Broadcast avec ACK: Algorithme *Test\_Connectivity* (**TC**)



# Algorithme *TC* : Principe

- Algorithme *Test\_Connectivity*
  - Exécution en parallèle de l'algorithme Broadcast avec accusé de réception
    - Génération de  $n$  arbres de recouvrement (AR)
  - Chaque nœud envoie son identificateur ***id*** à tous ses voisins
  - Les voisins propagent l'***id*** reçu vers tous les nœuds voisins (sauf le nœud émetteur)

# Algorithme TC



# Algorithme TC : Structure de données

- **$id_i$**  : Identifiant du nœud  $n_i$
- **$v_i$**  : liste des voisins directs
- **$atteint_i(j)$** 
  - Booléen initialisé à faux et mis à vrai lors de la première réception de  $id_j$  par le nœud  $n_i$
- **$count_i(j)$**  : nombre de réception du message  $id_j$  par le nœud  $n_i$
- **$parent_i(j)$**  :
  - Initialisé à NULL.
  - Pointe sur le nœud  $n_k$  de  $v_i$  qui a envoyé, pour la première fois, le message  $id_j$  à  $n_i$
- **$init_i$**  : booléen positionné à vrai si :
  - $n_i$  appartient à  $N_0$  ou
  - lors de la réception du premier message par  $n_i$

# Algorithme *TC*

- $n_i$  appartient à  $N_0$ 
  - $\text{init}_i := \text{vrai};$
  - $\text{atteint}_i(i) := \text{vrai};$
  - Envoyer  $\text{id}_i$  à tous les nœuds de  $v_i$

# Algorithme TC ... Suite

- Répéter
  - A la réception de  $id_k$  de la part de  $n_j$ 
    - si non ( $init_i$ ) //  *$n_j$  envoie ses informations*
      - $init_i := \text{vrai}$ ;  $atteint_i(i) := \text{vrai}$ ;
      - Envoyer  $id_i$  à tous les nœuds de  $v_i$
    - $count_i(k) := count_i(k) + 1$ ;
    - si ( $\text{non} (atteint_i(k))$ )
      - $atteint_i(k) := \text{vrai}$
      - $parent_i(k) := n_j$
      - Envoyer  $id_k$  à tous les voisins directs de  $n_i \neq n_j$
    - si ( $count_i(k) = |v_i|$ ) et ( $parent_i(k) \neq \text{NULL}$ )
      - Envoyer  $id_k$  à  $parent_i(k)$
  - <sup>69</sup>Jusqu'à (condition d'arrêt)

# Algorithme *TC* : Complexité

- $n$  exécutions de l'algorithme broadcast avec ACK
  - $2 n \cdot m$  messages
  - $n^2$  exécutions de l'algorithme broadcast avec ACK

# Plan

- Hypothèses de travail
- Collecte (Convergecast)
- Diffusion (Broadcast)
- Construction d'arbres de recouvrement
- Identification des noeuds d'un réseau
- **Calcul du plus court chemin**
- Algorithme d'élection
- Algorithmes de recherche

# Calcul des plus courts chemins :

## Calcul\_PCC

- Quoi ? : Calcule le plus court chemin entre deux nœuds quelconque du réseau.
- Pourquoi ? : Routage des messages.
- A la fin de l'exécution de l'algorithme, chaque nœud  $n_i$  est informé :
  - de la plus petite distance qui le sépare de chaque nœud du réseau
  - du plus court chemin qui relie  $n_i$  à un autre nœud  $n_j$ .
    - Pour le chemin  $(n_i, n_j)$ :  $n_i$  connaît le voisin  $n_k$  qui appartient au chemin.  $n_k$  connaît à son tour le prochain voisin direct  $n_h$  qui le relie à  $n_j$ , et ainsi de suite.
- Deux algorithmes :
  - Algorithme synchrone :  $S\_Calcul\_PCC$
  - Algorithme asynchrone :  $A\_Calcul\_PCC$



# Algorithme Synchrone :

## *S\_Calcul\_PCC*

***dist<sub>i</sub>(j)*** : la plus petite distance qui sépare le nœud  $n_i$  du nœud  $n_j$

- A l'instant  $t = 0$  : Chaque nœud  $n_i$  envoie son identificateur à tous ses voisins.
- A l'instant  $t = 1$  :
  - Chaque nœud  $n_i$  a reçu les messages de la part de ses voisins, il connaît alors les nœuds  $n_j$  tels que  $dist_i(j) = 0$  ou 1.
  - $n_i$  construit l'ensemble des nœuds  $n_j$  tels que  $dist_i(j) = 1$  et les transmet vers ses voisins.
- A l'instant  $t = 2$  :
  - Chaque nœud  $n_i$  a reçu les messages de la part de ses voisins, il connaît alors les nœuds  $n_j$  tels que  $dist_i(j) = 0, 1$  ou 2.
  - $n_i$  construit l'ensemble des nœuds  $n_j$  tels que  $dist_i(j) = 2$  et les transmet vers ses voisins.

# Algorithme Synchrone :

## *S\_Calcul\_PCC*

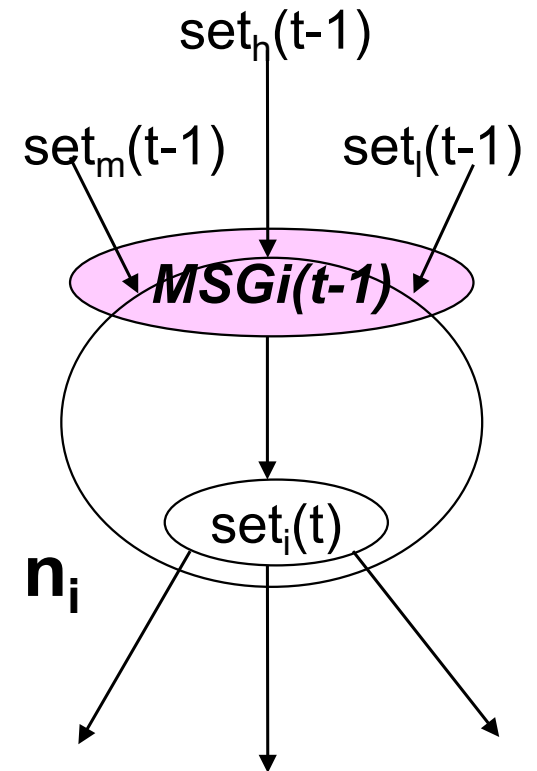
- A l'instant  $t \geq 0$ , chaque nœud  $n_i$  envoie à ses voisins l'ensemble des nœuds  $n_j$  tels que la distance  $dist_i(j) = t$ .
  - Pour  $t = 0$ , cet ensemble est le singleton  $n_i$ .
  - Pour  $t > 0$ , cet ensemble représente les nœuds reçus durant les précédents  $t-1$  instants. La distance entre ces nœuds et  $n_i$  est au plus égale à  $t$ :
    - Ces nœuds ont une distance  $t-1$  des voisins de  $n_i$ .

## Structure de données (*S\_Calcul\_PCC*)

- **$dist_i(j)$**  : la plus petite distance qui sépare le nœud  $n_i$  du nœud  $n_j$ 
  - Initialement,  $dist_i(j) = n$  pour  $i \neq j$  et  $dist_i(i) = 0$ .
- **$first_i(j)$**  : nœud, appartenant à  $v_i$ , qui est le premier nœud du chemin le plus court qui relie les nœuds  $n_i$  et  $n_j$ .
  - Initialement,  $first_i(k) = null$
- **$set_i(t)$**  : L'ensemble des nœuds que  $n_i$  envoie à ses voisins à l'instant  $t$ .
  - Initialement,  $set_i(0) = \{n_i\}$
- **$MSG_i(t)$**  : Ensemble des nœuds reçus par le nœud  $n_i$  à l'instant  $t-1$  de la part de tous les nœuds voisins
  - **$MSG_i(t)$**  est l'union des messages  **$set_i(t)$**  reçus de la part des

# Algorithme : $S\_Calcul\_PCC$

- $MSG_i(0) = \{ \}$  ;
- Pour (tous les nœuds de  $N$ ) faire envoyer  $set_i(0)$  aux voisins  $v_i$  FinPour
- Pour  $t$  allant de 1 à  $n - 1$  faire // *longueur max d'un chemin =  $n - 1$*
- Pour (chaque réception de  $MSG_i(t-1)$ ) faire
  - $set_i(t) = \{ \}$  ;
  - Pour (chaque  $set_j(t-1)$  appartenant à  $MSG_i(t-1)$ ) faire
    - Pour chaque nœud  $n_k$  appartenant à  $set_j(t-1)$  faire
      - Si  $(dist_i(k) > t)$  alors
        - $dist_i(k) = t$  ;
        - $first_i(k) = n_j$  ;
        - $set_i(t) = set_i(t) + \{n_k\}$
    - FinPour
  - FinPour
  - Envoyer  $set_i(t)$  à tous les voisins de  $n_i$  (ensemble  $v_i$ )
  - $t = t + 1$  ;
- FinPour
- FinPour



# Version asynchrone ...

## Structure de données (*A\_Calcul\_PCC*)

- ***dist<sub>i</sub>(j)*** : la plus petite distance qui sépare le nœud  $n_i$  du nœud  $n_j$ 
  - Initialement,  $dist_i(j) = n$  pour  $i \neq j$  et  $dist_i(i) = 0$ .
- ***level<sub>i</sub>(j) = d*** :  $n_i$  a reçu de  $n_j$  les nœuds ayant une distance ***d*** de  $n_j$ .
  - $n_j$  appartient à  $v_i$
  - Initialement,  $level_i(j) = -1$
  - $level_i(j)$  peut être considéré comme l'horloge par rapport au voisin  $n_j$
- ***first<sub>i</sub>(j)*** : nœud, appartenant à  $v_i$  : le premier nœud du chemin le plus court qui relie les nœuds  $n_i$  et  $n_j$ .
  - Initialement,  $first_i(j) = null$

## Structure de données (*A\_Calcul\_PCC*)

- ***state<sub>i</sub>*** : *state<sub>i</sub>* = *d* signifie que *n<sub>i</sub>* a reçu tous les identificateurs des nœuds qui sont à une distance *d*.
  - Initialement, *state<sub>i</sub>* = 0
  - *state<sub>i</sub>* peut être considérée comme l'horloge des «horloges» *level<sub>i</sub>* (*j*) (en ce qui concerne le nœud *n<sub>i</sub>*)
- ***set<sub>i</sub>*** : Nœuds à envoyer par *n<sub>i</sub>* à ses voisins.
  - Initialement, *set<sub>i</sub>* = {*n<sub>i</sub>*}

## Structure de données (*A\_Calcul\_PCC*)

- ***Pour tous les nœuds  $n_i$*** 
  - ***$dist_i(i) = 0$*** ;
  - ***$dist_i(j) = n$  pour  $j \neq i$***
  - ***$level_i(j) = -1$***
  - ***$first_i(j) = \text{null}$  pour  $i \neq j$***
  - ***$state_i = 0$***
  - ***$init_i = \text{faux}$***
  - ***$set_i = \{n_i\}$***
- ***Pour les nœuds  $n_i$  appartenant à  $N_o$*** 
  - ***$init_i = \text{vrai}$***
  - Envoyer  $set_i$  à tous les nœuds voisins ( $v_i$ )



# Algorithme A\_Calcul\_PCC (exécuté par tous les $n_i$ )

- Tant que ( $state_i < n-1$ ) faire
  - A la réception d'un message  $set_j$  de la part de  $n_j$  (appartenant à  $v_i$ )
    - Si (non  $init_i$ ) alors  $debut\ init_i = vrai$ ; envoyer  $set_i$  à tous les voisins fin fsi
    - $level_i(j) = level_i(j) + 1$ ;
    - Pour (tous les nœuds  $n_k$  appartenant à  $set_j$ ) faire
      - Si ( $dist_i(k) > level_i(j) + 1$ ) alors
        - $dist_i(k) = level_i(j) + 1$  ;
        - $first_i(k) = n_j$ ;
    - finpour
    - Si ( $level_i(j) \geq state_i$ ) pour tous les  $n_j$  de l'ensemble  $v_i$ ) alors
      - $state_i = state_i + 1$ ;
      - $set_i = \{n_k \mid dist_i(k) = state_i\}$
      - Envoyer  $set_i$  à tous les voisins de  $n_i$  (ensemble  $v_i$ )
    - Finsi
  - FinTantque

# Plan

- Hypothèses de travail
- Collecte (Convergecast)
- Diffusion (Broadcast)
- Construction d'arbres de recouvrement
- Identification des noeuds d'un réseau
- Calcul du plus court chemin
- **Algorithme d'élection**
- Algorithmes de recherche

# Algorithmes d'élection

- On s'intéresse aux graphes complets
- 1<sup>ère</sup> solution
  - Tous les nœuds envoient leur identifiants aux autres nœuds
  - A la réception des identifiants, chaque nœud peut identifier le nœud leader
  - Complexité :  $O(n^2)$  pour les messages et  $O(1)$  pour le traitement

# Algorithme *S\_Elect\_Leader*

- 2<sup>ème</sup> solution
  - Les nœuds candidats sont ceux qui désirent être leaders
  - Chaque nœud envoie son identifiant (*id*) à un seul ( $2^0$ ) destinataire, puis à 2 ( $2^1$ ) destinataires, puis à 4 ( $2^2$ ) destinataires, etc.
  - Lors de la  $k^{\text{ème}}$  étape, un nœud envoie son *id* à  $2^{k-1}$  destinataires
  - Pour envoyer à tous les nœuds, il faut  $\log(n)$  étapes
  - Complexité
    - $O(n \log(n))$  pour les messages
    - $O(n)$  pour le traitement

# Algorithme *S\_Elect\_Leader*

- 2<sup>ème</sup> solution ... suite
  - L'envoi de l'*id* est considéré comme une tentative de «capture» du nœud destinataire
  - Pour être élu, il faut réussir à capturer tous les nœuds du réseau
  - Un nœud  $n_i$  réussit à capturer un nœud  $n_j$  si  $id_i$  est plus grand que les  $id$  de tous les nœuds qui tentent de capturer  $n_j$ . De plus,  $id_i$  doit être supérieur à  $id_j$ .
  - A l'étape  $k$ ,  $n_i$  ne tente de capturer des nœuds que s'il réussit à capturer tous les nœuds de l'étape  $k-1$ . Dans le cas contraire, il cesse d'être candidat.

# Algorithme d'élection: *S\_Elect\_Leader*

- Structure de données
  - *candidat<sub>i</sub>*
    - Initialisé à vrai pour les nœuds de  $N_o$ , à faux pour les autres
  - *tried<sub>i</sub>(j)*
    - concerne les nœuds appartenant à  $v_i$ . Il est à **vrai** pour les voisins à qui  $n_i$  a déjà envoyé un message, **faux** sinon
  - *owner<sub>i</sub>*
    - Propriétaire du nœud  $n_i$ , initialisé à NULL

# Algorithme *S\_Elect\_Leader* (exécuté par tous les $n_i$ )

- Initialisation

- $t = 0$ ; // horloge ou pulsation
- $MSG_i = \{\}$ ;
- Pour les nœuds candidats (appartenant à  $N_o$ ), faire
  - $candidat_i = \text{true}$
  - $owner_i = id_i$
  - Choisir un nœud  $n_j$  de  $v_i$
  - Envoyer  $capture(id_i)$  à  $n_j$
- $t = t + 1$ ;

# Algorithme *S\_Elect\_leader* (exécuté par tous les $n_i$ )

- $t$  impair:
  - Recevoir  $MSG_i$  //union des captures reçus
  - Sélectionner  $n_k$  tel que  $id_k \geq id_j$ ,  $n_j$  étant les nœuds qui ont envoyé une demande de capture à  $n_i$ .
  - Si  $owner_i < id_k$  alors
    - Si  $candidat_i$  alors  $candidat_i = \text{faux}$ ;
    - $owner_i = id_k$
    - Envoyer ACK à  $n_k$
  - $t = t + 1$ ;



# Algorithme *S\_Elect\_Leader* (exécuté par tous les $n_i$ )

- $t$  est pair:
    - Recevoir  $MSG_i$  //union des ACK reçus
    - Si  $candidat_i$  alors
      - Si «le nombre de messages envoyés (captures) est supérieur (strictement) à celui des ACK reçus»
      - alors  $candidat_i = \text{faux}$ ;
      - sinon
        - Si  $t < 2 (\log n)$  alors
          - Sélectionner parmi les nœuds  $n_j$  de  $v_i$  un ensemble  $S$  tels que leur  $tried_i(j) = \text{faux}$
          - Mettre  $tried_i(j)$  à vrai pour les éléments de  $S$
          - Envoyer  $capture(id_i)$  aux éléments de  $S$
- 89
- $t = t + 1$ ;

# Version asynchrone ...

# Algorithme asynchrone

## *A\_Elect\_Leader*

- L'algorithme doit s'assurer que deux candidats ne doivent/peuvent jamais capturer un même nœud
  - Un nœud a toujours un seul et unique leader (propriétaire)
- La comparaison ne se fait pas en fonction des identifiants *id* mais en fonction du niveau d'avancement «*level*»
- *level* : nombre de groupes capturés par le nœud: 1, 2, 4, 8, etc.
  - $level_i = E [\log (owns_i + 1)]$  où  $owns_i$  est le nombre de nœuds capturés par  $n_i$

## A\_Elect\_Leader : comment ça marche?

- Pour capturer  $n_j$ ,  $n_i$  envoie un message:
  - *capture (level<sub>i</sub>, id<sub>i</sub>)*
- A la réception de ce message,  $n_j$  teste:
  - $(level_j, propriétaire_j) < (level_i, id_i)$
  - Test lexicographique: si les variables *levels* sont identiques, on compare les variables *propriétaire<sub>j</sub>* et *id<sub>i</sub>*.

## A\_Elect\_Leader : comment ça marche?

- $n_i$  demande la capture de  $n_j$  :
  - $n_i$  envoie à  $n_j$  un message **capture** ( $level_i, id_i$ )
- A la réception de ce message,  $n_j$  effectue le test:
- Si  $(level_j, propriétaire_j) > (level_i, id_i)$  alors
  - $n_j$  envoie un message **nack** à  $n_i$
- Sinon
  - $level_j = level_i$
  - Si  $n_j$  est candidat
    - $n_j$  n'est plus candidat
    - $n_j$  envoie un message **ack** à  $n_i$
    - A la réception du message **ack**,  $n_i$  devient le propriétaire de  $n_j$ .

# A\_Elect\_Leader : comment ça marche?

- Si  $n_j$  n'est pas candidat
  - $p\_propriétaire_j = n_i$  //garder trace de la candidature de  $n_i$
  - $n_j$  envoie un message **check (k)** à  $n_i$ 
    - $n_k$  est le propriétaire actuel de  $n_j$ .
    - Objectif: Régler le contentieux entre  $n_i$  et  $n_k$ .
  - A la réception du message *check*,  $n_i$  (s'il est toujours candidat):
    - Envoie un message **eliminate (level<sub>i</sub>, id<sub>i</sub>)** à  $n_k$ .
  - A la réception du message *eliminate (level<sub>i</sub>, id<sub>i</sub>)*,  $n_k$  teste :
    - Si  $(level_k, id_k) < (level_i, id_i)$ 
      - $n_k$  envoie un message **eliminated** à  $n_i$
    - Sinon
      - $n_k$  envoie un message **nack** à  $n_i$

# *A\_Elect\_Leader* : Algorithme

- Structure de données
  - ***candidat<sub>i</sub>***
    - Initialisé à vrai pour les nœuds de  $N_o$ , à faux pour les autres
  - ***tried<sub>i</sub>(j)***
    - concerne les nœuds appartenant à  $v_j$ . Il est à ***vrai*** pour les voisins à qui  $n_i$  a déjà envoyé un message de capture, ***faux*** sinon
  - ***propriétaire<sub>i</sub>***
    - Propriétaire du nœud  $n_i$ , initialisé à NULL
  - ***level<sub>i</sub>*** :  $\log(\text{owns}_k)$   $n_k$  est le propriétaire de  $n_i$
  - ***nb\_capturés<sub>i</sub>*** : nombre de nœuds capturés
  - ***p\_propriétaire<sub>i</sub>*** : propriétaire potentiel du nœud  $n_i$
  - ***p\_capturé<sub>i</sub>*** : nœud en cours de capture par  $n_i$

# *A\_Elect\_Leader* : Algorithme ... suite

- Si  $n_i$  appartient à  $N_o$ 
  - $candidat_i = \text{vrai}$ ;
  - $owner_i = id_i$
  - $owns_i = 0$ ;
  - $level_i = 0$ ;
  - $p\_owner_i = \text{NULL}$
  - $p\_capturé_i = \text{NULL}$
  - On choisit un nœud  $n_j$  appartenant à  $v_i$  ( $v_i$  contient les autres nœuds du graphe puisque  $G$  est complet)
  - $tried_i(j) = \text{vrai}$
- 96 • Envoyer *capture* ( $level_i, id_i$ ) à  $n_j$



# A\_Elect\_Leader : Algorithme

## ... suite

- A la réception de ***capture (level<sub>j</sub>, id<sub>j</sub>)***
  - Si  $p\_owner = id_j$ 
    - $owner_i = id_j$
    - Envoyer `ack ()` à  $n_j$
  - Si  $(level_i, owner_i) < (level_j, id_j)$  alors
    - $level_i = level_j$  ;
    - Si  $candidat_i$  alors
      - $candidat_i = \text{faux}$
      - $owner_i = id_j$
      - Envoyer `ack` à  $n_j$
    - Sinon
      - $p\_owner_i = id_j$
      - Envoyer `check ()` à  $n_k$  (propriétaire actuel de  $n_i$  ( $owner_i = id_k$ ))
  - Sinon envoyer `ack ()` à  $n$

# *A\_Elect\_Leader* : Algorithme ... suite

- A la réception de **nack**
  - Si  $candidat_i$  alors  $candidat_i = \text{faux}$ ;
- A la réception de **check (j)**
  - Si  $candidat_i$  alors envoyer *eliminate* ( $level_i$ ,  $id_i$ ) à  $n_j$ ;

# A\_Elect\_Leader : Algorithme ... suite

- A la réception de ***eliminate (level<sub>j</sub>, id<sub>j</sub>)***
  - Si non *candidat<sub>i</sub>* alors envoyer *eliminated* à  $n_j$
  - sinon
    - Si  $(level_i, owner_i) < (level_j, id_j)$  alors
      - *candidat<sub>i</sub>* = faux;
      - Envoyer *eliminated* à  $n_j$
    - Sinon envoyer *nack* () à  $n_j$
- A la réception ***eliminated***
  - Si *candidat<sub>i</sub>* alors
    - Soit  $n_j$  le nœud de  $v_i$  tel que  $p\_owned_i = id_j$ .
    - Send capture  $(level_i, id_i)$  à  $n_j$

# *A\_Elect\_Leader* : Algorithme ... suite

- A la réception de **ack ()**
  - $nb\_capturés_i = nb\_capturés_i + 1$ ;
  - $level_i = E [\log(owns_i)]$ ;
  - $S = \{n_j \text{ appartiennent à } v_i \text{ tels que } tried_i(j) = \text{faux}\}$
  - Si  $S \neq \{\}$  alors
    - On choisit un nœud de  $S$  :  $n_j$
    - $tried_i(j) = \text{vrai}$
    - Envoyer *capture* ( $level_i, id_i$ ) à  $n_j$

# Plan

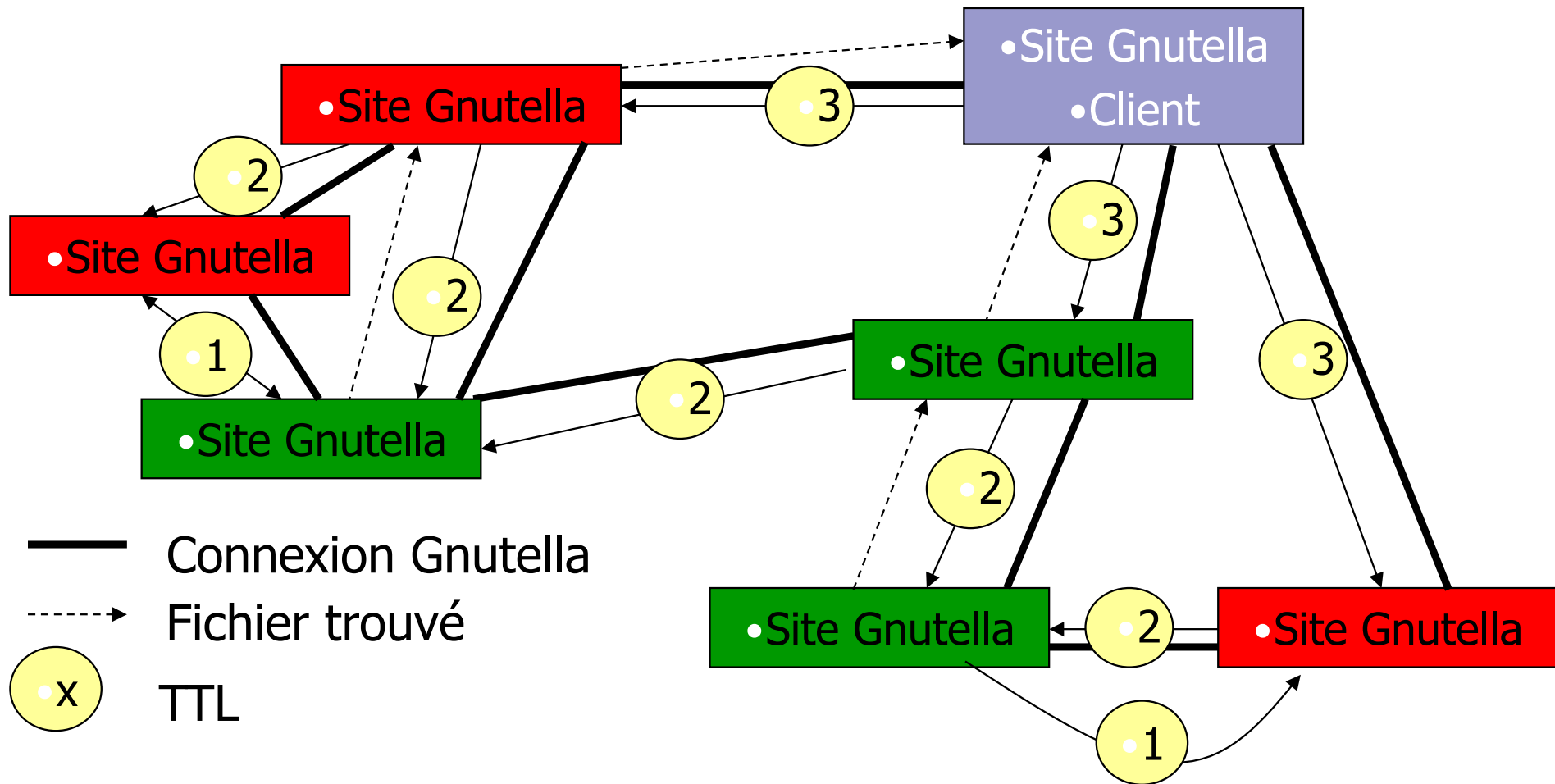
- Hypothèses de travail
- Collecte (Convergecast)
- Diffusion (Broadcast)
- Construction d'arbres de recouvrement
- Identification des noeuds d'un réseau
- Calcul du plus court chemin
- Algorithme d'élection
- Algorithmes de recherche

# Gnutella

- Chaque site dispose :
  - d'un entrepôt contenant les fichiers partagés
  - d'une liste de voisins (au sens Gnutella)
  - d'un Algorithme de recherche distribué
    - Basé sur un algorithme de diffusion
    - Avec une portée limitée (Time To Live : TTL)

# Principe de recherche

- Une requête de recherche :
  - Fichier recherché,
  - Durée de vie (Time To Live : TTL)
- Une requête est envoyée vers les nœuds voisins
- A la réception d'une requête :
  - Recherche dans l'entrepôt local
  - Si le fichier existe, un *relpy* est envoyé à l'émetteur
  - Décrémentation de TTL
  - Si TTL non nul, la requête est envoyée aux voisins directs.





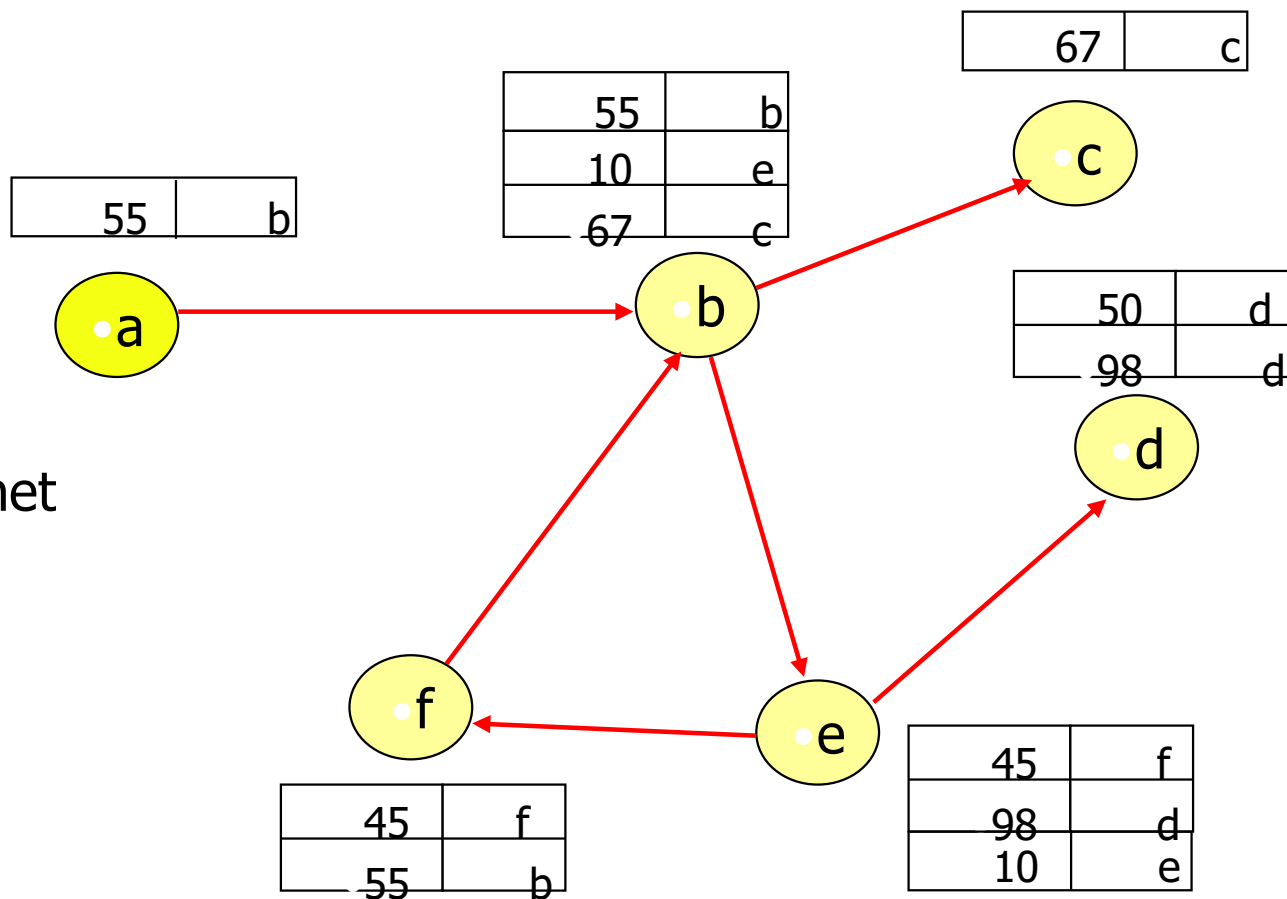
- Application : partage de données
- Chaque site dispose de :
  - son propre entrepôt
  - Un algorithme de recherche
- Un fichier est représenté par :
  - Une clé
  - Son contenu (pas toujours)
  - L'adresse IP de la machine qui l'héberge (propriétaire)

Clé	Contenu (donnée)	Adresse IP (propriétaire)
xYWERaa	Pointeur sur le fichier	129.187.133.141
saWQEa	Pointeur sur le fichier	192.199.111.101

# Réseau Freenet



Connexion Freenet



# Principe de la recherche

- A la réception d'une requête :
  - Recherche du fichier (clé) dans l'entrepôt local
  - Envoie au voisin qui héberge le fichier ayant la clé la plus proche
  - Un site ne traite jamais une requête plus qu'une fois.
- Lorsqu'un site ne peut plus transmettre une requête, il la renvoie au site émetteur ...
- Le résultat est :
  - transmis au client en suivant le même circuit
  - stocké au niveau des sites intermédiaires

# Algorithme de recherche

## Recherche d'un document : clé = 50

