# Partage de ressources

## M2

Lélia Blin

lelia.blin@irif.fr

2023

# Partage de ressources

- On aborde ici un des premiers problèmes qui s'est posé lors de l'élaboration des systèmes répartis
- Plusieurs noeuds coopèrent et partagent des ressources communes.

# Partage de ressources

- Ces ressources peuvent être de plusieurs types:
  - matériels:
    - imprimantes
    - scanners
    - ....
  - logiciels :
    - Base de données
    - mémoire commune
    - ....

# Propriétés

- Ces ressources partagées ont des propriétés et doivent être gérées en respectant certaines règles d'accès
- Icion va s'intéresser à une ressource qui aura les propriétés suivantes:
  - Elle est complètement partagée (tous les noeuds peuvent y accéder via le réseau)
  - Au plus $M$ noeuds peuvent y accéder simultanément.

# Propriétés

- Remarque: si à un moment donné, plus de $M$ noeuds y accèdent simultanément
  - nous serons dans une **situation d'erreur**

> - Il faut donc construire un protocole d'accès à cette ressource qui respecte la condition qu'il n'y ait pas d'erreur

# Propriétés

- De manière plus précise, notre protocole devra vérifier les propriétés suivantes pour une ressource à $M$ entrés
  - **Sûreté**: A chaque instant au plus $M$ noeuds utilisent la ressource.
  - **Vivacité**: Chaque noeud demandant un accès à la ressource l'obtiendra au bout d'un temps fini (en l'absence de pannes)
    - On parle d'*absence de famine*.

> Ces deux propriétés sont indispensables pour avoir système:
> - sûr et «équitable»

# Protocole schématique

- Les demandes d’accès sont totalement asynchrones
- Pour encadrer la bonne utilisation d’une ressource, le protocole est exécuté schématiquement de la façon suivante

```
<Acquisition>

<Section critique>

<Libération>
```

# Phase d'acquisition

- La phase d'acquisition consiste pour
  - Le noeud à demander la ressource
    - Nous verrons comment
  - jusqu'à obtenir la ressource

# Section critique

- L'utilisation d'une ressource peut être totalement quelconque
- Nous décrirons pas cette phase qui est propre à chaque ressource
- Nous supposerons malgré tout que la *phase d'utilisation a une durée fini*

# Phase de Libération

- Après l'utilisation de la ressource,
- le noeud qui avait la ressource la **libère** afin que d'autre noeuds puissent l'utiliser à leur tour

# Protocole

- *Les protocoles que nous aurons à décrire sont donc constitués des deux phases*
  - **Acquisition**
  - et **Libération**

*Protocole*

Nous supposerons (sauf précision contraire) que:

- Chaque noeud qui est dans sa phase d’acquisition ne rentre pas dans une nouvelle phases d’acquisition avant d’avoir exécuté sa phase de libération
  - un noeud ne demande qu’une ressource à la fois

> Le système est sans panne

# Partageons une ressource de façon réparti

# Le droit à la parole

- Supposons que vous devez vous occuper d'un groupe d'enfant de maternelle.
- Comment faire pour qu'ils parlent pas tous en même temps?

# Le droit à la parole approche centralisé

- Vous demandez à ceux et celles qui veulent parler de lever la main.
- Et vous choisissez un par un.

> Comment faire de façon réparti?

# Le droit à la parole de façon réparti

- La parole est symbolisé par un "baton"
- Seul celle ou celui qui a le baton peut parlé.

# Billet de concert

- Que garanti un billet pour un organisateur de concert?

# Billet de concert

- Que garanti un billet pour un organisateur de concert?
  - Que la place a été payé.
  - Qu’il y a autant de personnes que de places. (sûreté)

# Protocoles avec jeton

# Ressource à une entrée

- Dans cette partie nous étudions des protocoles pour gérer une ressource avec une seule entrée
  - soit $M = 1$
- Comment représenter le fait d'utiliser cette ressource?

# Jeton

- La ressource est représenté par un jeton
- Le fait qu'il n'y ai qu'un seul jeton qui circule permet de garantir la propriété de sûreté

> Il faut être sûr que notre protocole ne duplique pas le jeton

# Topologies fixes

- Nous allons voir des protocoles de partages de ressource utilisant un jeton pour des topologie fixes
  - **cycle** (anneau)
  - **arbre**

# Dans un cycle

- On va supposer le réseau en forme de cycle unidirectionnel.
- L'idée de l'algorithme est la suivante.
    - au départ le jeton est placé de façon arbitraire sur un noeud
    - Si un noeud n'a pas besoin de la ressource il la fait passer à son voisin
    - Lorsqu'un noeud a besoin de la ressource il attend le passage du jeton

Lélia Blin

# Dans un cycle

- Lorsqu’un noeud demandeur (en phase d’acquisition) reçoit le jeton
  - il le garde
  - et l’utilise pour accéder à la ressource
  - une fois qui la fini d’utiliser la ressource
- il le passe ensuite à son voisin dans le cycle

# Variables du protocole

- Les variables et constante à chaque noeud $v$ sont les suivantes:
  - $Avoir_jeton_v \in \{Vrai, Faux\}$
  - $Demandeur_v \in \{Vrai, Faux\}$
  - $Suivant_v$ désigne le noeud suivant dans le cycle
- Initialement toutes les `Avoir_jeton` et `Demandeur` sont à Faux
- sauf pour un noeud qui a $Avoir_jeton_v = Vrai$

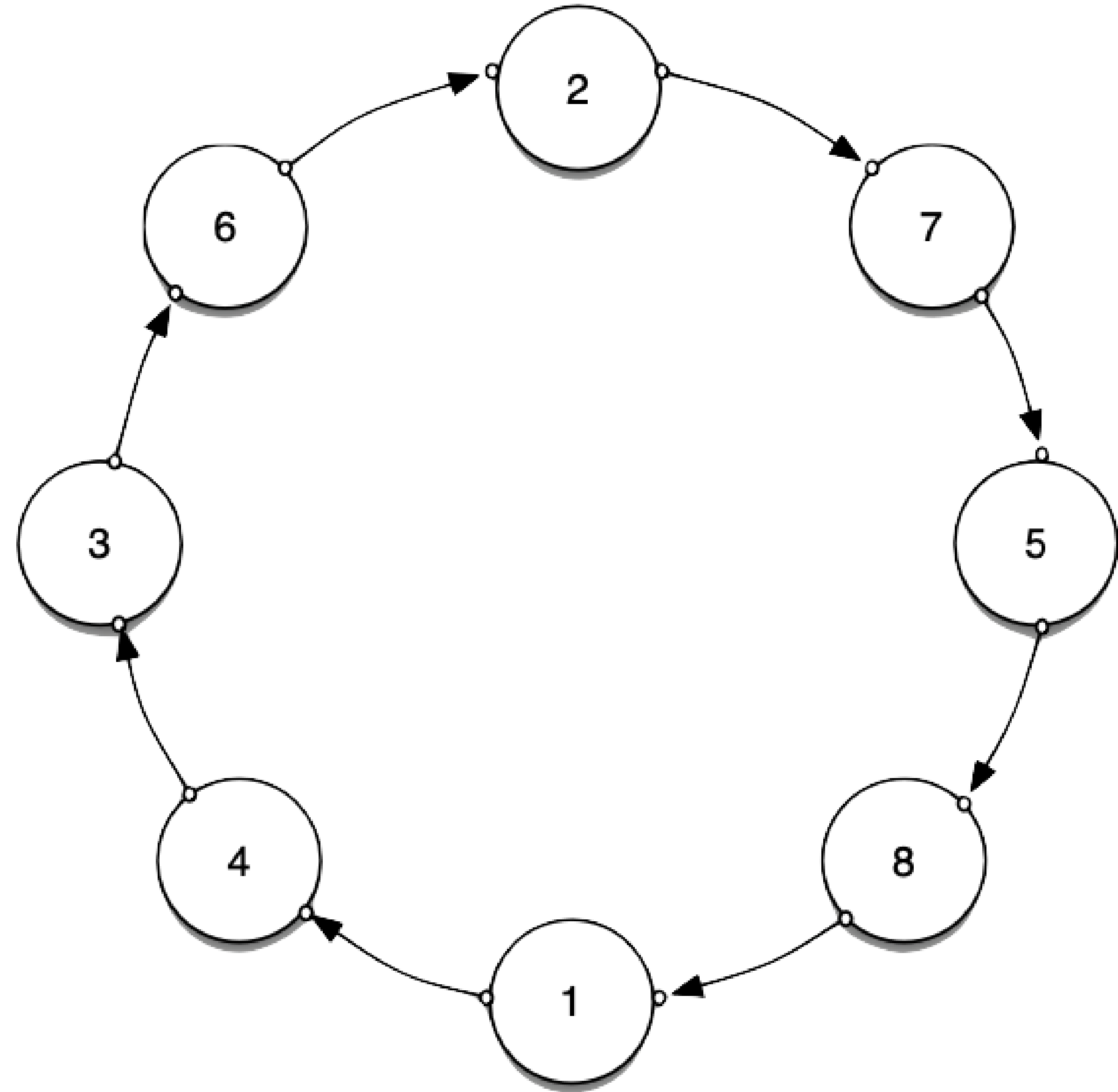
# Protocole

```
Acquisition
 Demandeur_v=Vrai
 Attendre(Avoir_jeton_v)

Liberation
 Demandeur_v=Faux
 Envoyer (<JETON>) à Suivant_v
 Avoir_jeton_v=Faux

Lors de la réception de <JETON>
        Si (non Demandeur_p_) alors
                Envoyer (<JETON>) à Suivant_v
        Sinon Avoir_jeton_v=Vrai
```

# Une exemple



# Avantages et Inconvénients de ce protocole

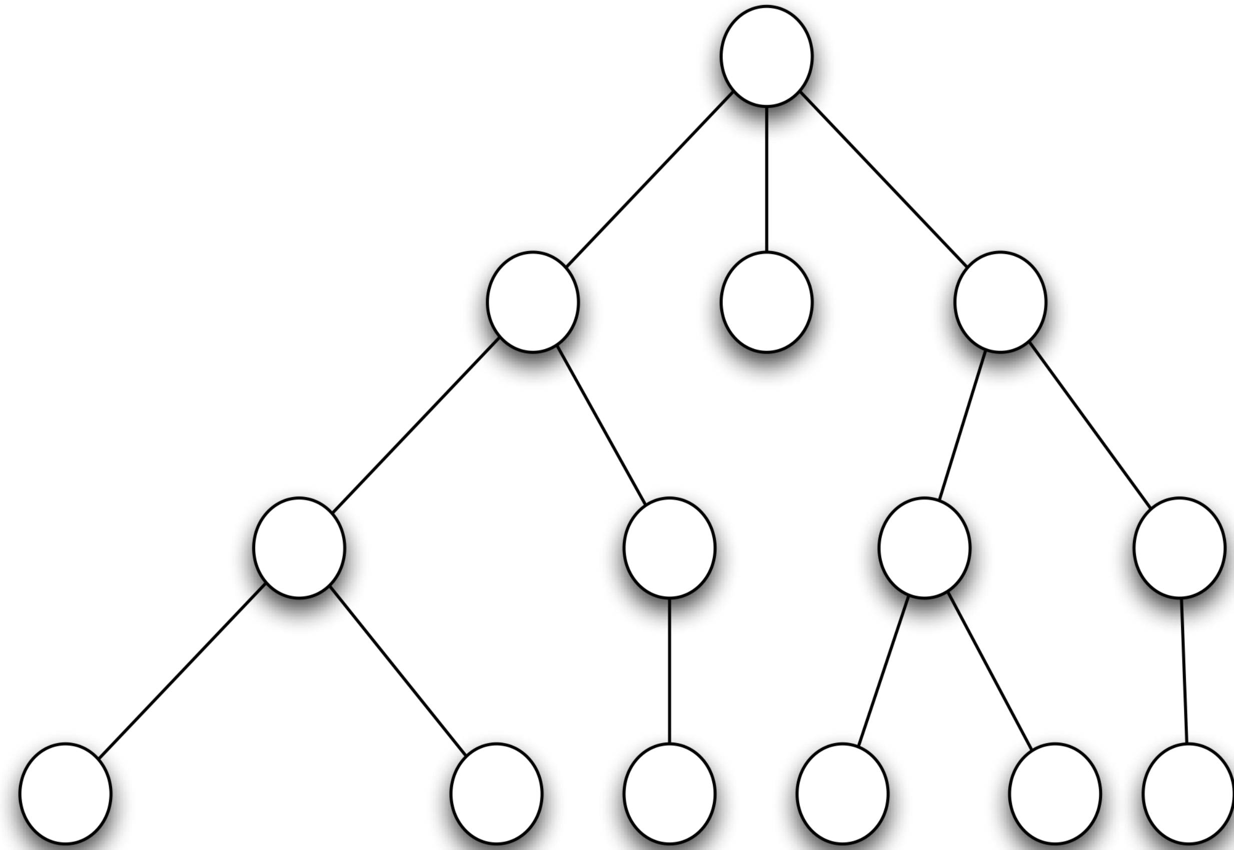
# Avantages de ce protocole

- Le premier et sa simplicité
- Il peut être étendu automatiquement
  - ajoute de noeud au système
  - suppression de noeud dans le système
- Aucune connaissance globale
- Taille du message $O(1)$ bits

# Inconvénients du protocole

- Longue attente, même sans autre demande
- Le jeton circule constamment
  - parfois pour rien
- Il n'y a pas de priorité temporelle
  - les noeuds ne sont pas forcément servi dans l'ordre des demandes

# Protocole dans un arbre



# Variables du protocole

- Les variables et constante à chaque noeud $v$ sont les suivantes:
  - $Avoir_jeton_v \in \{Vrai, Faux\}$
  - $En_SC_v \in \{Vrai, Faux\}$
  - $Racine_v$ identifiant de noeud qui indique à $v$ vers quel voisin dans l'arbre $v$ doit demander la ressource
  - $Request_v$ est une file d'identifiants de noeud, initialement vide
- Initialement les variables sont à Faux
- Sauf pour un noeud qui a $Avoir_jeton_v = Vraie$

# Protocole



```
Acquisition
        si(non Avoir_jeton) alors
                si (FileVide(Request)) alors
                        Envoyer <DEMANDE> à Racine
                ajouter(Request,id)
                Attendre(Avoir_jeton)
        sinon En_SC:=Vrai

Libération
        En_SC=Faux
        si (non file_vide(Request))
                Racine:=defiler(Request)
                Envoyer <JETON> à Racine
                Avoir_jeton:=Faux
                si (non file_vide(Request))
                Envoyer <Demande> à Racine
```

```
Lors de la réception de <DEMANDE> envoyer par u
        si(Avoir_jeton) alors
                si (En_SC) alors
                        ajouter(Request,u)
                sinon
                        si (Request={}) alors Racine:=u
                        sinon
                                Racine:=defiler(Request)
                                ajouter(Request,u)
                        Envoyer <JETON> à Racine
                        Avoir_jeton:=Faux
        sinon
                si (file_vide(Request)) alors
                        Envoyer <DEMANDE> à Racine
                        ajouter(Request,u)
```

```
Lors de la réception de <JETON> envoyé par u
        Racine=Defiler(Request)
        si (Racine=id) alors
                Avoir_jeton=Vrai
        sinon
                Envoyer <JETON> à Racine
                si (FileVide(Request)) alors
                        Envoyer <DEMANDE> à Racine
```

# Remarques

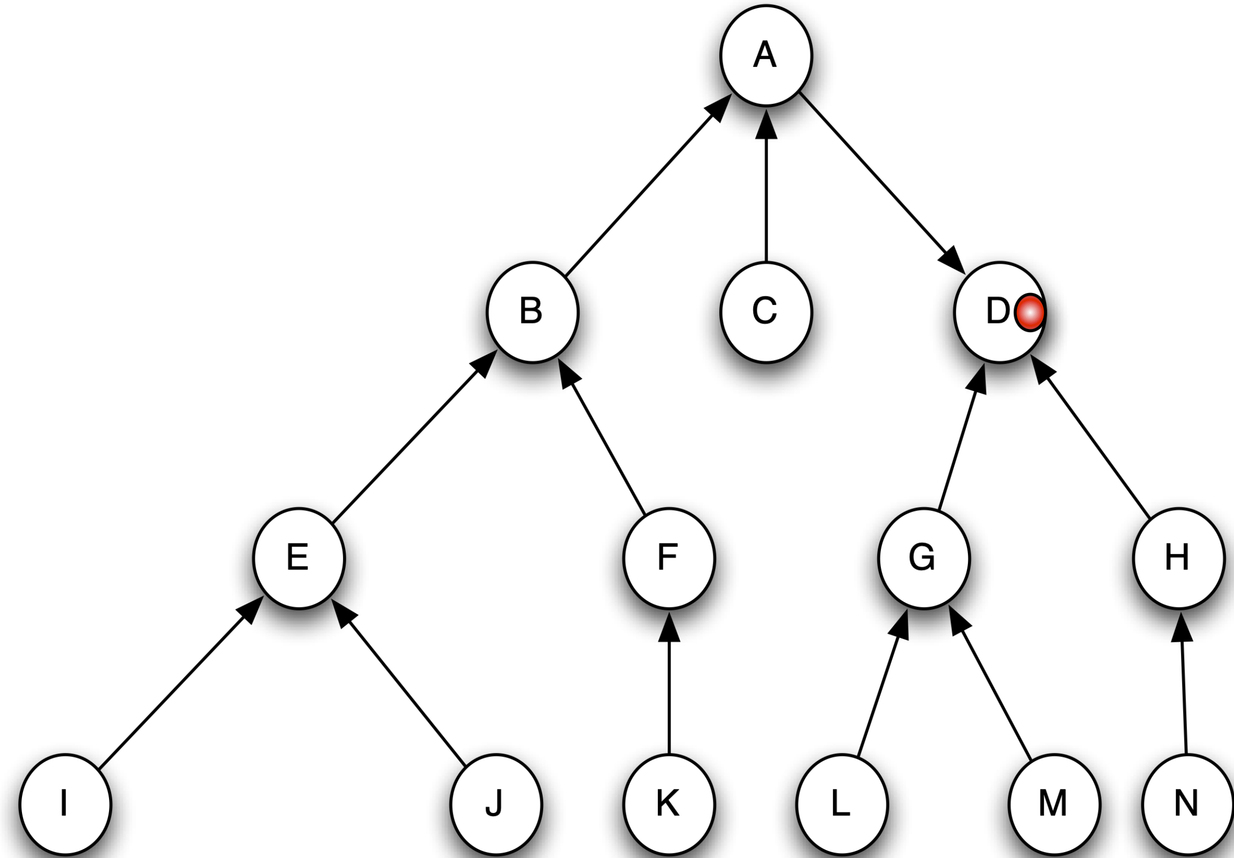
- La file Request sert à ordonner les demandes de chaque sous-arbre
- Notons:
  - Si un noeud reçoit un message `<DEMANDE>`
    - Si sa file est vide
      - le noeud le transmet dans son sous-arbre dans lequel il y a le jeton
    - Si sa file n’est pas vide
      - Le noeud bloque la demande à son niveau
      - Le noeud débloquera la demande que lorsque les autres demande qui sont avant dans la file auront été servies.

# Remarques

- Lors de la phase de libération
  - le noeud possédant le jeton le transmet
    - au premier qui lui a demandé
    - autrement dit le premier de sa file
  - Si après l'envoie du jeton sa file reste non vide:
    - il transmet une nouvelle demande pour que le suivant dans la file puisse être servi.
- La file d'attente permet d'assurer la vivacité du mécanisme

# Exemple

Le noeud E fait une demande de jeton, suivit par une demande de K et avant que E soit servi.



# Avantage

- Le jeton circule que si il y a des demandes
- L'ordre des demandes est plus respecté que dans l'algorithme sur le cycle
- Taille des messages: O(1) bits
- Nombre de messages par demande  est proportionnel au diamètre de l'arbre en moyenne $O(log_2 n)$

# Protocoles à base de permissions

# Une autre façon de partager

# Estampillage

- Pour ces protocoles nous avons besoin d'estampiller les message avec des marques d'horloge
- Rappel: aucune horloge globale n'est disponible
- On va donc utiliser un mécanisme  à base d'estampillage  basé sur les *horloges logiques* locale
- Ce mécanisme peut être vu comme un sous-protocole

# Estampillage protocole

variable locale: `horloge`

```
Initialisation:
        horloge:=0
Lors de la réception d'un message <M,h>
        horloge=max{horloge,h}+1
        Délivrer(M)
Lors de l'envoie du message M
        horloge=horloge+1
        Envoyer(< M,horloge>)
```

# Remarques

- Les messages sont donc transportés avec l’heure logique à laquelle ils ont été émis
- Cependant on peut se retrouver avec deux message ayant la même estampille $\longrightarrow$ ordre partiel
- On a besoin d'un ordre total $\longrightarrow$ le moyen usuel dans ce cas et de prendre en compte l’identifiant de l’envoyeur

# Remarques

- Ainsi à la place de considérer uniquement l'horloge logique
- on prendre en compte le **couple** ($h$,$i$)
- où $i$ est l'identifiant du noeud émetteur

# Priorité

- L'ordre sur le couple $(h_i, i)$ et $(h_j, j)$ ou $i$ et $j$ sont les identifiant des émetteurs des messages est le suivant
- $(h_i, i) < (h_j, j) \Leftrightarrow (h_i < h_j$ ou $(h_i = h_j$ et $i < j))$

# Priorité

- Ce qui se traduit par $i$ est plus prioritaire que $j$ car
  - soit la demande de $i$ a été faite avant celle de $j$ en prenant en compte les horloges $h_i$ et $h_j$
  - Soit les horloges sont les mêmes mais l'identité de $i$ est plus petite que celle de $j$
- Cet ordre est donc total **toutes les paires sont comparables**

# Algorithme de Ricart et Agrawala

# Principe de l'algorithme

- Dans cet algorithme
- chaque noeud désirant la ressource
- va demander le permission à tous les autres

# Conflits

- on départage les conflits en utilisant les étiquettes
- chaque étiquette donnant l'heure à laquelle la demande a été faite
- Les demandes les plus anciennes **sont les plus prioritaires**

# Variables locales

- Chaque noeud maintient les variables locales suivantes:
  - `Heure_demande` l'heure à laquelle le noeud a fait sa dernière demande
  - `Rep_attendues` est un entier qui compte le nombre de réponses attendu par le noeud suite à sa demande
  - `Demandeur` booléen qui est à vrai si demande la ressource faux sinon (initialisé à faux)
  - `differe[j]` tableau de $n$ cases de booléens, `differe[j]=vrai` ssi le noeud a différé sa réponses à la demande de $j$ ; initialisé à faux

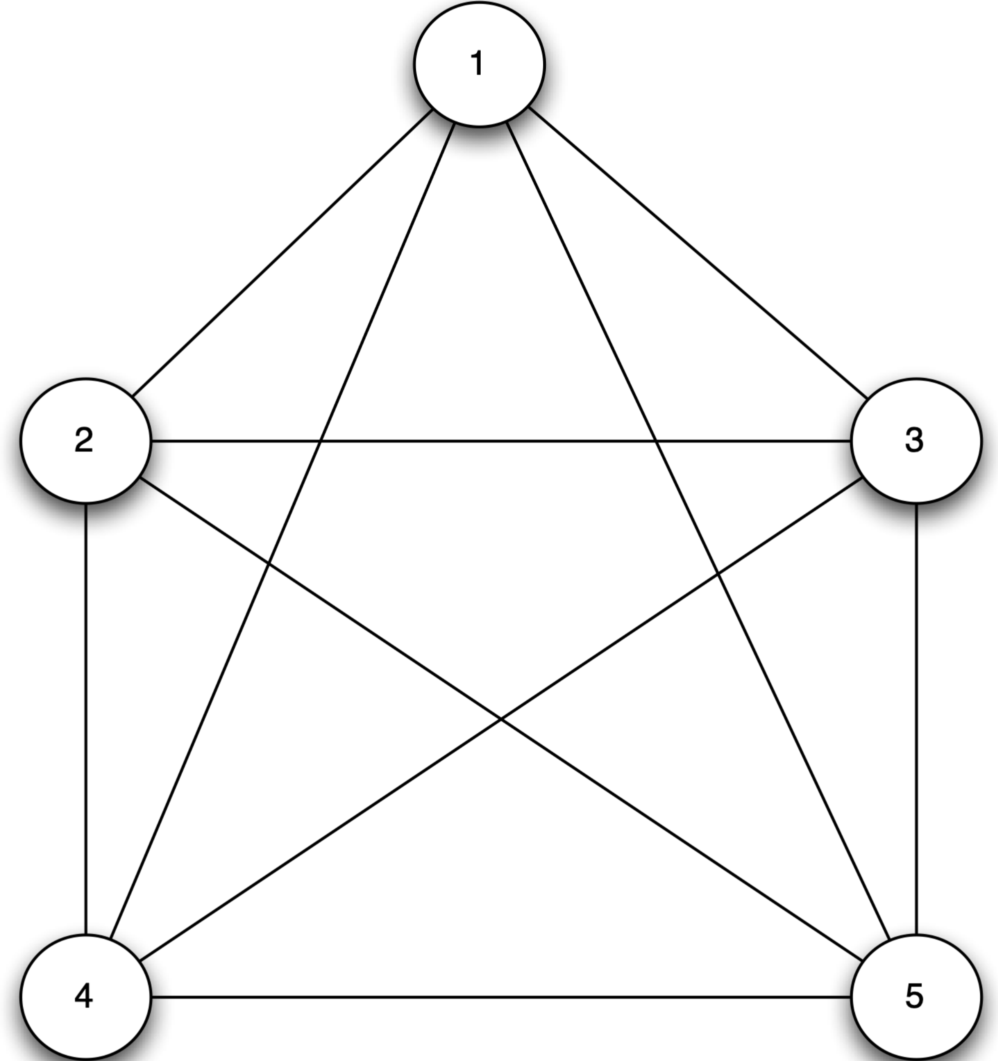
# Algorithme

noeud $i$

```
Procédure acquisition
        Heure_demande:=horloge
        Demandeur:=vrai_
        Rep_attendues:=n-1
        pour tout (x∈V-{i}) faire
                Envoyer(<DEMANDE,(Heure_demande,i)>) à x
        Attendre(Rep_attendues=0)
                en_SC
Procédure libération
        Demandeur:=faux
        pour tout (x ∈V-{i}) faire
                si (differe[x]) alors
                        Envoyer(<REPONSE >) à x
                        differe[x]=faux
Lors de la réception de <DEMANDE,h> envoyé par j
        Si (non Demandeur ou (Heure_demandei,i)>(h,j)) alors
                Envoyer(<REPONSE >) à j
        Sinon
                differe[j]:=vrai
Lors de la réception de <REPONSE> envoyé par j
        Rep_attendues:=Rep_attendues-1
```

# Exemple

2 et 3 sont demandeur avec une heure de demande égale à 1
4 est demandeur avec une heure de demande égale à 2



# Remarques

- La complexité est de $2(n-1)$ messages par demande
- Inconvénient
  - une demande est faite à tous les autres noeuds
  - ce qui entraîne un grand nombre de messages

# L'algorithme de Maekawa et variantes

# Améliorations

- Pour améliorer le nombre de message au lieu de demander un si grand nombre de permissions  on peut imaginer que chaque noeud $i$ a un sous ensemble $S_i$ à qui il va demander la permission d'utiliser le ressource
- Comment construire $S_i$?
- Quelle propriété doit avoir l'ensemble $S_i$?

# Quorum

- Un tel ensemble $S_i$ sera appelé quorum
- Pour être sûr que la **propriété** de **sûreté** est vérifiée il faut que les quorums vérifient la propriété suivante
  - Règle d'intersection: si $i \neq j$ alors $S_i \cap S_j \neq \emptyset$

# Règle d'intersection

- C'est propriété est indispensable
  - pour s'assurer que lorsque deux noeuds $i$ et $j$ demandent les permissions pour entrer en section critique les noeuds qui sont à la fois dans $S_i$ et $S_j$ ne peuvent pas accorder la permission aux deux
  - Ainsi **en cas de demandes simultanées au moins un des noeuds n'aura pas une permission**
- La présence de l'**estampillage** reste obligatoire pour départager les demandes multiples

# Diminuer la charge

- On ne veut pas que la charge de travail relatif au demande soit supporter que par un seul noeud
- On veut équilibrer la charge de travail que doit faire chaque noeud pour la communauté

# Formellement

- On veut que chaque noeud $i$ fasse partie de $D$ quorums $S$ et que pour tout $j$: $|S_j| = k$
- Il faut donc minimiser $k$ et $D$
- Rm: Dans ce cas la complexité en nombre de message est en $O(k)$

# Quorum

- On prend un quorum contant $k$ membres  qui font partie d'au plus $D-1$ autre ensemble
- Ainsi le maximum de quorum qui peut être construits est de $n=k(D-1)+1$

# Quorums

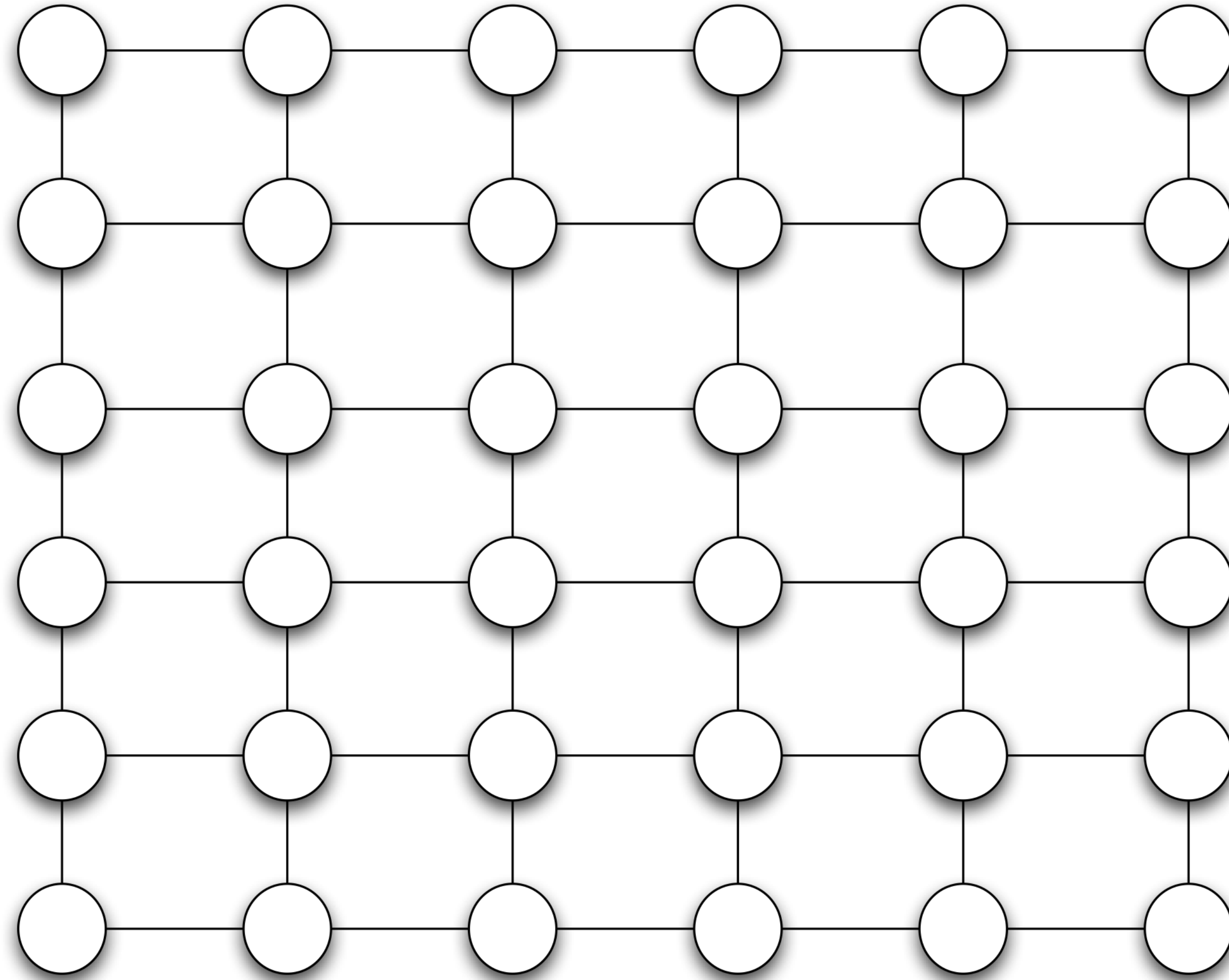
- De plus $Dn/k$ est le nombre maximum de quorums
  - car chaque noeud peut être dans au plus $D$ quorums
  - contenant chacun au plus $k$ noeuds
- Si il y autant de quorum que de noeuds on a
  - $n = Dn/k$ c'est à dire $D = k$
- En combinant ses égalités on obtient
  - $n = k(k-1)+1$ et $k = O(\sqrt{n})$

# Construction

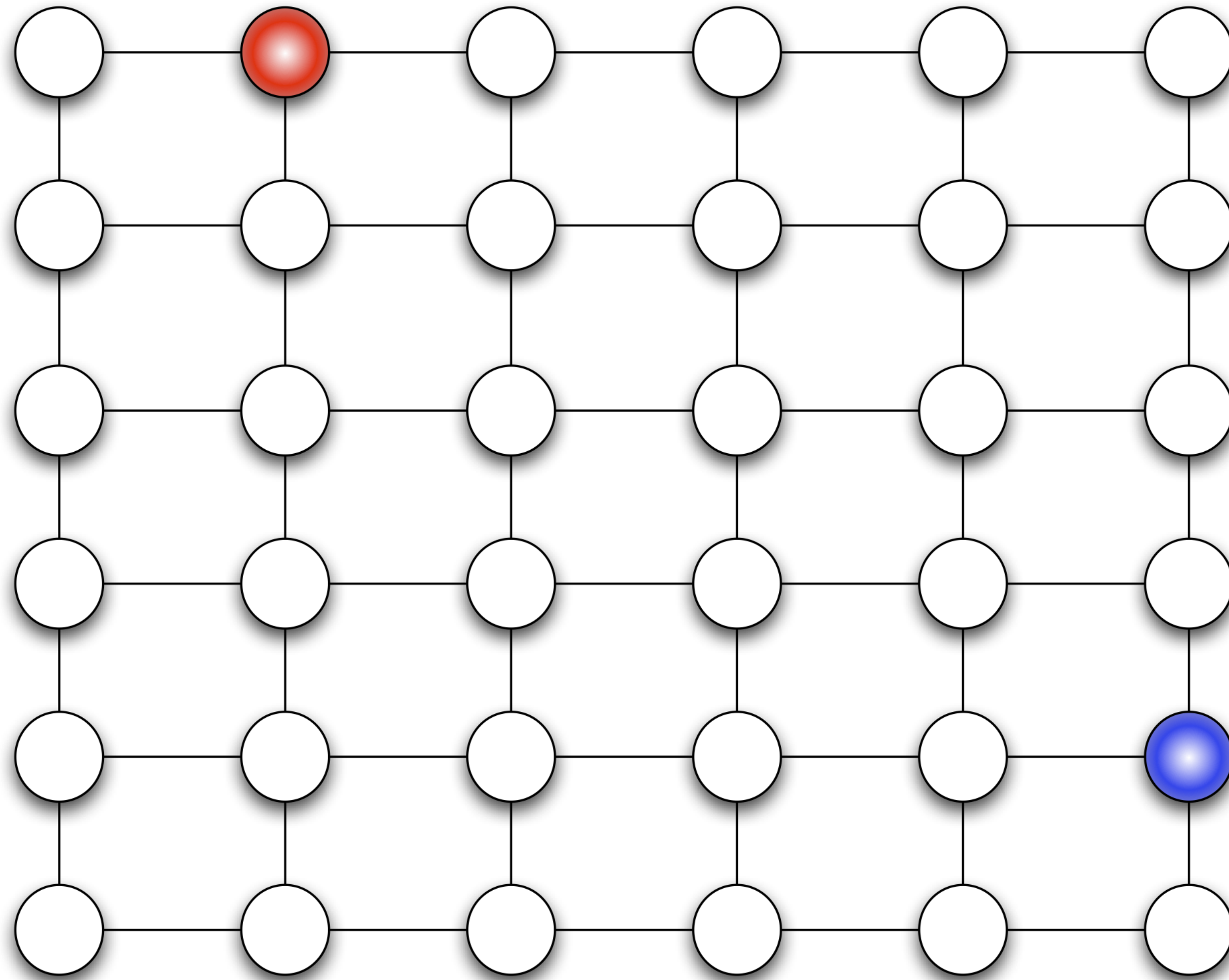
- Construire effectivement un quorums avec ses propriétés est difficile
- Par contre si $n = p^2$ il est facile de construire $n$ quorums  avec $\sqrt{n} = p$ noeuds dans chacun d'eux

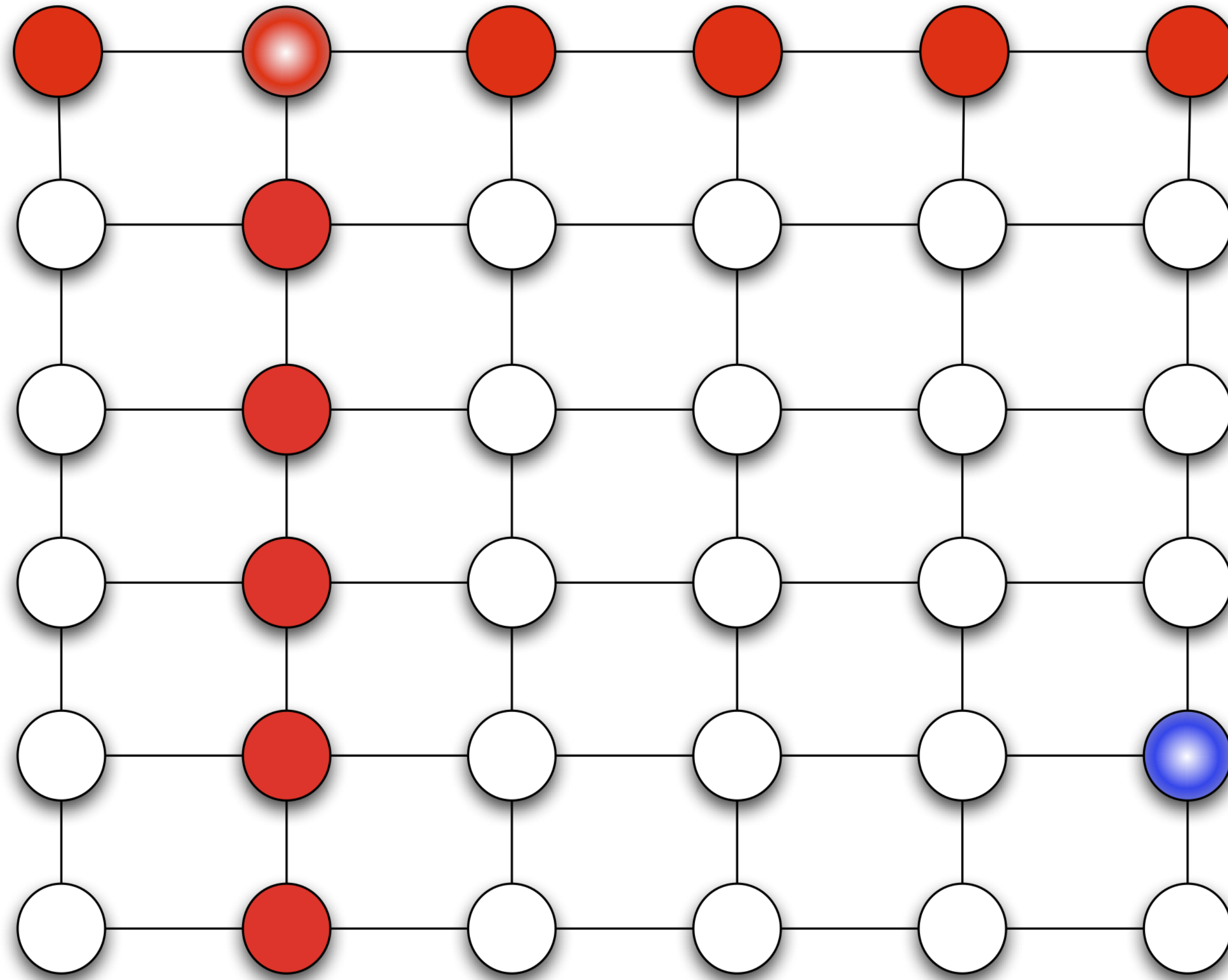
# Grille

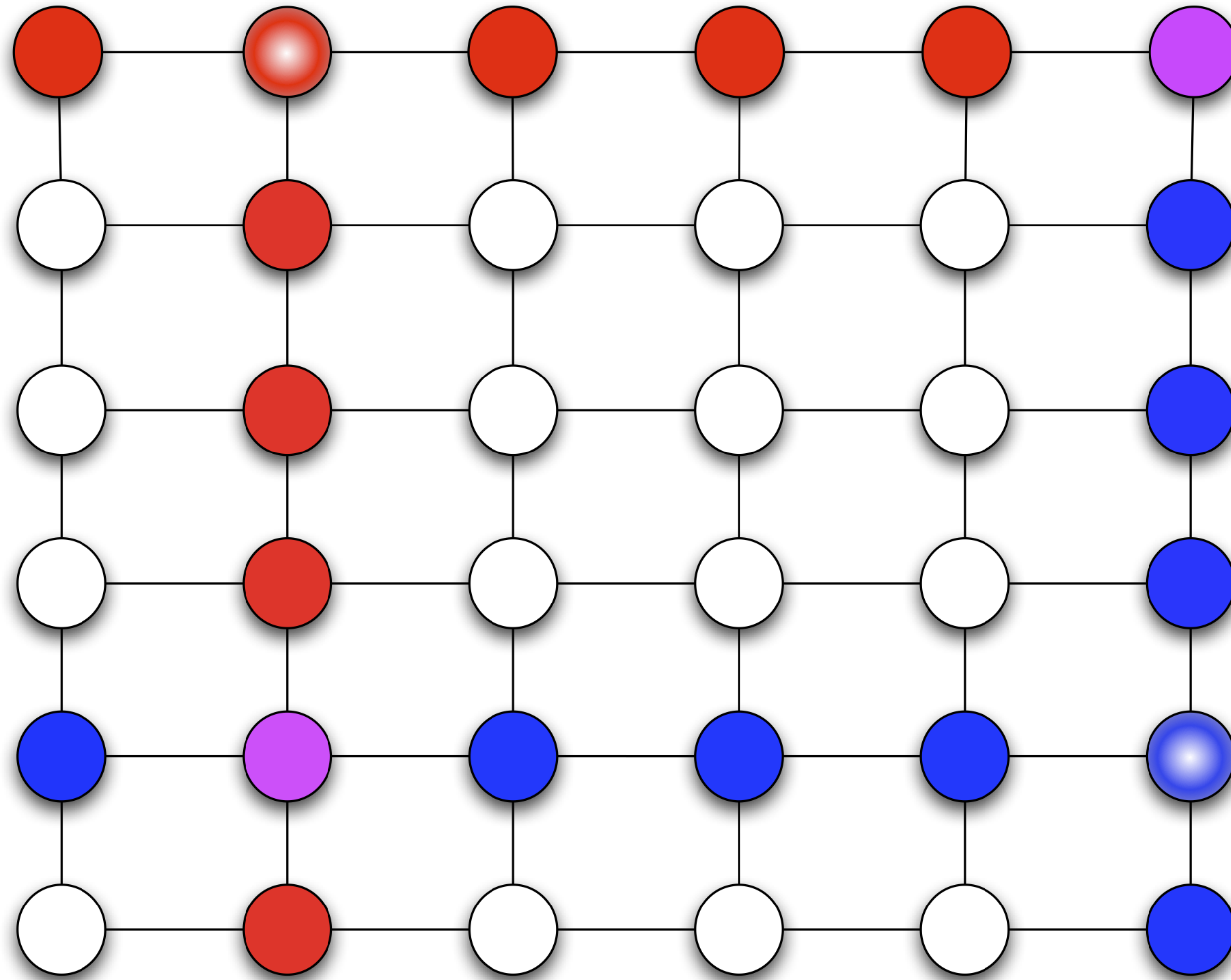
Avec cette construction toute les propriétés sont vérifiées



Lélia Blin







# Algorithme pour grille pxp

- Avec se système chaque noeud $i$ connaît son quorum $S_i$
- Chaque fois que $i$ demande la ressource
  - il demande la permission
  - estampillée par son horloge
  - à tous les noeuds de $S_i$

# Algorithme pour grille pxp

- Lorsqu'un noeud $i$ reçoit une demande de la part de $j$
- on peut être dans trois cas
  - Premier cas:
    - $i$ ne demande pas la ressource il envoie sa permission à $j$

# Algorithme pour grille pxp

- Deuxième cas:
  - $i$ a demandé la ressource
  - si sa demande a une estampille plus ancienne que celle de $j$
    - il lui envoie un message pour dire non
  - sinon il lui donne la permission

# Algorithme pour grille pxp

- Troisième cas:
  - $i$ a déjà donné sa permission à un noeud $u$ dont l'heure de demande avait une estampille plus récente que celle de $j$.
  - Dans ce cas $i$ va essayer de récupérer la permission qu'il a donné à $u$ pour la donner à $j$.
  - Si $u$ a reçu un message de refus de permission il redonne la permission à i qui va la donner à $j$.