

## Systèmes et algorithmes répartis

### Principes et concepts


Philippe Quéinnec, Gérard Padiou  
queinnec@enseeiht.fr  
<http://queinnec.perso.enseeiht.fr/Ens/sar.html>

ENSEEIH  
Département Sciences du Numérique

13 septembre 2023

La présence de 🎵 indique un complément audio : cliquer dessus. 

## Sources


- G. Padiou, « *Précis de répartition* », 2016,  
<http://queinnec.perso.enseeiht.fr/Ens/SAR/precis.pdf>  
Référéncé par [ *Précis 1.5 p.13* ] (section, page)
  - M. Raynal, « *Distributed Algorithms for Message-Passing Systems* », « *Fault-Tolerant Agreement in Synchronous Message-passing Systems* » et « *Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems* », 2010–2012
  - S. Krakowiak, « *Algorithmique et techniques de base des systèmes répartis* », <http://lig-membres.imag.fr/krakowia/Files/Enseignement/M2R-SL/SR/>
  - A.D. Kshemkalyani, M. Singhal, « *Distributed Computing : Principles, Algorithms, and Systems* », 2008  
<http://www.cs.uic.edu/~ajayk/DCS-Book>
- 

## Plan

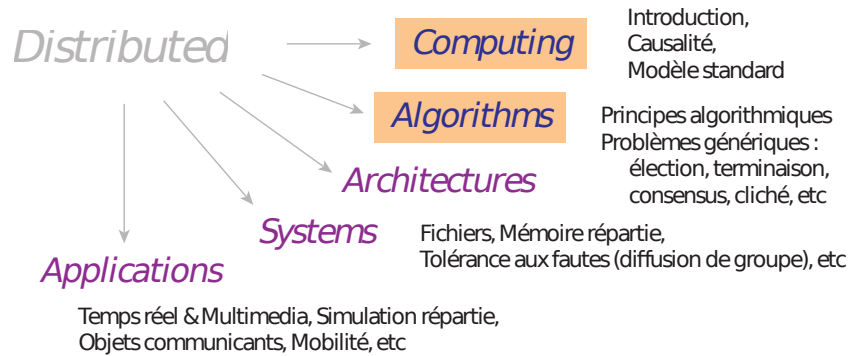
- 1 Préambule
- 2 Définition et problématique
  - Les parfums
  - Exemple
  - Les épines
- 3 Un principe de conception : la transparence

## Préambule : tendance

Répartition  $\equiv$  communication entre objets informatisés

- Actuellement : ordinateurs + téléphones + tablettes toujours connectés
  - L'Internet des objets (*The Internet of things*)
    - 24 milliards d'appareils connectés entre eux en 2020
    - du porte-clefs au réfrigérateur en passant par les plantes
  - L'informatique dans les nuages (*cloud computing*) : l'accès pour tous aux ressources/services informatiques
- 

## Préambule : de quoi allons nous parler ?



## Plan

- 1 Préambule
- 2 Définition et problématique
  - Les parfums
  - Exemple
  - Les épines
- 3 Un principe de conception : la transparence



## Plan du cours



- I. Principes et concepts
- II. Modèle standard et principes algorithmiques
- III. Causalité et datation
- IV. Problèmes génériques
- V. Grande échelle, pair-à-pair
- VI. Consensus, détecteur de défaillances
- VII. Données réparties
- VIII. Construction d'objets concurrents
- IX. Tolérance aux fautes
- X. Simulation répartie



## Modèle centralisé ou réparti



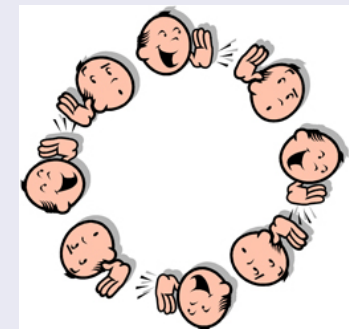
### Modèle centralisé

Les processus se partagent des ressources critiques ou pas



### Modèle réparti

Les processus échangent des données par messages



## Intérêts



### Apports de la répartition

- **Accès aux ressources distantes et partage :**
  - ressources physiques : imprimantes, traceurs...
  - ressources logiques : fichiers
  - données : textuelles, audio, images, vidéo
- **Répartition géographique**
- Puissance de calcul
- Disponibilité
- Flexibilité

[ Précis 1.3 pp.9-10 ]



## Hypothèses



### Connaissance initiale

Chaque site connaît son identité ( $id_i$ ) et l'identité de ses voisins  $voisins_i = \{id_x, id_y, \dots\}$ .  
Le couple  $(id_i, id_j)$  représente le canal entre  $id_i$  et  $id_j$  (et symétriquement).  
Aucun site ne connaît l'identité de tous les sites, ni leur nombre.

### Communication

Un site peut envoyer/recevoir un message uniquement à ses voisins.



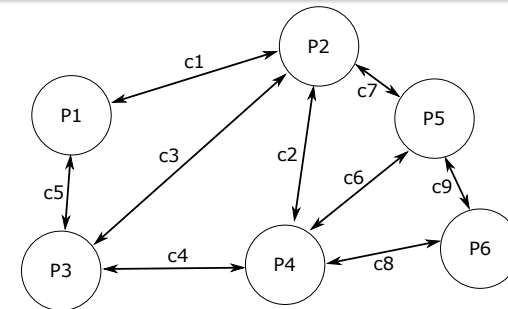
## Exemple : découvrir le graphe de communication



### Le problème

On considère un ensemble de sites connectés deux à deux par des canaux bidirectionnels.

**Comment un site peut-il apprendre la structure du graphe ?**



## Découverte : principe de l'inondation



### Le problème redéfini

Comment chaque site  $id_i$  peut-il connaître l'ensemble des canaux  $(id_j, id_k)$  existants ?

### Principe

- Un site  $i$  qui veut connaître le graphe envoie sa position  $\langle id_i, voisins_i \rangle$  à ses voisins.
- La première fois qu'un site  $i$  reçoit un message, il s'active et envoie sa position  $\langle id_i, voisins_i \rangle$  à ses voisins.
- La première fois qu'un site  $i$  reçoit un message  $\langle id_k, voisins_k \rangle$ , il le transmet à ses voisins et met à jour sa connaissance du graphe (pour  $k$ ). Sinon, il l'ignore.
- Un site conclut quand il a reçu un message de tous les sites dont il a eu connaissance via les voisinages.



## Découverte : principe de l'inondation



### Le problème redéfini

Comment chaque site  $id_i$  peut-il connaître l'ensemble des canaux  $(id_j, id_k)$  existants ?

### Principe

- Un site  $i$  qui veut connaître le graphe envoie sa position  $\langle id_i, voisins_i \rangle$  à ses voisins.
- La première fois qu'un site  $i$  reçoit un message, il s'active et envoie sa position  $\langle id_i, voisins_i \rangle$  à ses voisins.
- La première fois qu'un site  $i$  reçoit un message  $\langle id_k, voisins_k \rangle$ , il le transmet à ses voisins et met à jour sa connaissance du graphe (pour  $k$ ). Sinon, il l'ignore.
- Un site conclut quand il a reçu un message de tous les sites dont il a eu connaissance via les voisinages.



## Questions pas triviales

- Termination : tous les sites finissent-ils par atteindre FIN ?
- Termination : un site peut-il savoir que les autres ont terminé ?
- Correction : à la terminaison de  $i$ ,  $channels\_known_i =$  le graphe ?
- Correction : après terminaison de tous,  $\forall i, j : channels\_known_i = channels\_known_j$  ?
- Coût en messages ?
- Complexité en temps ?
- Résistance à un arrêt de site ?



## Algorithme pour le site $i$



```
on start :
  for each  $id_j \in voisins_i$  do
    send  $\langle id_i, voisins_i \rangle$  to  $id_j$ 
  end for
   $sites\_known_i \leftarrow \{id_i\}$ 
   $channels\_known_i \leftarrow$ 
     $\{ (id_i, id_k) : id_k \in voisins_i \}$ 
```

Variables locales au site  $i$  :  
 $id_i$  : son identité (const)  
 $voisins_i$  : ses voisins (const)  
 $sites\_known_i$  : les sites dont il a reçu un message  
 $channels\_known_i$  : les canaux qu'il connaît

```
on reception  $\langle id, voisins \rangle$  :
  if  $sites\_known_i = \emptyset$  then start(); fi
  if  $id \notin sites\_known_i$  then -- premier message de id
     $sites\_known_i \leftarrow sites\_known_i \cup \{id\}$ 
     $channels\_known_i \leftarrow channels\_known_i \cup \{ (id, id_k) : id_k \in voisins \}$ 
    for each  $id_j \in voisins$  -- propage le message
      send  $\langle id, voisins \rangle$  to  $id_j$ 
    end for
    if  $\forall (id_j, id_k) \in channels\_known_i : \{id_j, id_k\} \subseteq sites\_known_i$  then
       $id_i$  connaît le graphe. FIN
    endif
  endif
endif
```

## Questions pas triviales



- Termination : tous les sites finissent-ils par atteindre FIN ? (oui)
- Termination : un site peut-il savoir que les autres ont terminé ? (non)
- Correction : à la terminaison de  $i$ ,  $channels\_known_i =$  le graphe ? (oui)
- Correction : après terminaison de tous,  $\forall i, j : channels\_known_i = channels\_known_j$  ? (oui)
- Coût en messages ? ( $2 * \text{nombre de sites} * \text{nombre de canaux}$ )
- Complexité en temps ? ( $2 * \text{diamètre}$ )  
(diamètre =  $\max_{(i,j)} \min distance(i, j)$ )
- Résistance à un arrêt de site ? (on perd la terminaison ; ok si le graphe reste connexe et que tout site fait au moins "start")

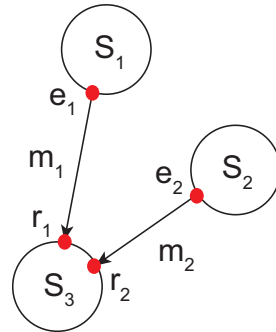


## Modèle d'exécution plus complexe



### Problèmes...

- $m_1$  est-il toujours envoyé avant  $m_2$  dans toute exécution ?
- $m_1$  est-il toujours reçu avant  $m_2$  dans toute exécution ?
- Peut-on déduire ?  
 $date(r_1) < date(r_2)$   
 $\Downarrow ?$   
 $date(e_1) < date(e_2)$



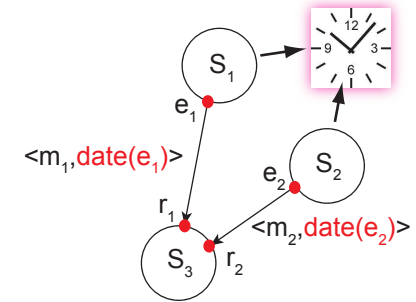
### Dates dans messages

$$date(e_1) < date(e_2)$$

$\Downarrow ?$

$e_1$  avant ?  $e_2$

Pas sûr, car  
l'horloge n'existe pas !!!



Fort **non-déterminisme** : explosion des états possibles

[ Précis 1.2 pp.7-9 ]



## Épine : pas de temps global



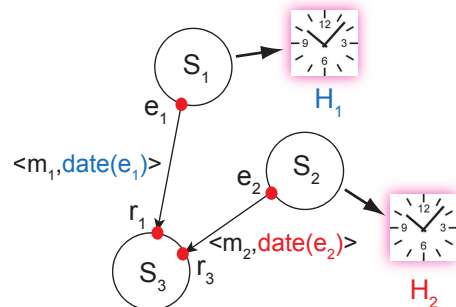
### il existe 2 horloges

$$date(e_1) < date(e_2)$$

$\Downarrow$

$e_1$  avant  $e_2$

Si les horloges  
sont synchronisées !



Pas de référentiel temporel unique



## Épine : le temps

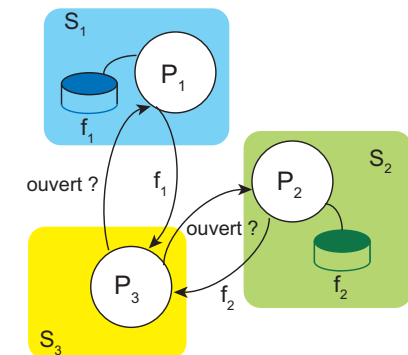


## Épine : pas d'état global immédiat



### Problème

- P3 veut savoir si P1 ou P2 ont ouvert des fichiers ?
- Connaissance instantanée impossible



Un processus ne peut pas connaître instantanément l'état courant de ses partenaires : pas d'état global immédiat.



## Épine : les défaillances

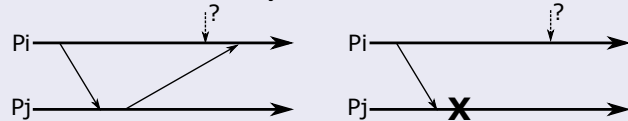


### Défaillance de la communication

Perte de message, modification du contenu, ordre de délivrance  
⇒ solutions réseau ou algorithmiques

### Défaillance de site

- arrêt du site, réponse erronée, transition erronée
- **défaillance partielle** du système
- **non détectable** en asynchrone : lent ou cassé ?



« A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable. » Leslie Lamport, 1987.

## Thèmes de recherche sur la répartition

### Concevoir, modéliser, expérimenter

- Modélisation théorique
- Algorithmique
- Langages
- Systèmes d'exploitation
- Intergiciels (middleware)

[ Précis 1.4 pp.11–12 ]

## Les épines, en résumé



### Impact de la répartition

- **Pas d'horloge globale** : chaque site a son horloge
- **Pas d'état global immédiat** accessible à un site
- **Fiabilité partielle** : possibilité d'arrêt d'une machine, d'un processus quel que part
- Sécurité relative : usagers potentiels nombreux
- Non déterminisme (parallélisme) : système asynchrone

### Conséquence

Modèle de calcul différent du cas centralisé

- Ordre partiel entre les événements d'un calcul
- Calcul d'état global passé

## Plan

- 1 Préambule
- 2 Définition et problématique
  - Les parfums
  - Exemple
  - Les épines
- 3 Un principe de conception : la transparence

## Principe de conception

Une idée clé : **la transparence**

### Principe de conception 1

Un bon système réparti  
est un système  
qui semble centralisé  
(qui s'utilise comme)

### Principe de conception 2

Un bon système réparti  
n'est pas un système centralisé

[ Précis 1.5 pp.13–18 ]

## Transparence d'accès

### Propriété

Accès à une ressource distante  $\equiv$  accès à une ressource locale

### Exemple

- Niveau langage de commande : `sh`  $\neq$  `ssh` (non transparence)
- Niveau service système : `read`, `write` identiques que le fichier opérande soit local ou distant (transparence)
- Niveau langage à objet : appel de méthode local ou à distance **identique** pour l'appelant (transparence)

### Solution : Notion d'interface

Cas des intergiciels à objets : langage IDL et bus logiciel

## Idée : masquer la répartition

### Niveaux de transparence

- Accès
- Localisation
- Partage
- Réplication
- Fautes
- Migration
- Charge
- Échelle

### Mécanismes

- Interface
- Nommage
- Synchronisation
- Groupe
- Atomicité
- Mobilité
- Réflexivité
- Reconfiguration

## Transparence de localisation

### Propriété

La localisation d'une ressource reste cachée.

### Exemple

- Non transparence : commande `scp bach.enseeiht.fr:/foo` .
- Transparence :
  - Niveau service système : `open("nom-fichier", ...)` : nom du fichier indépendant de la localisation du fichier
  - Niveau langage à objet : références aux objets distants sans nécessité de connaître leur localisation

### Solution : Services de nommage gérant des noms globaux

Cas des intergiciels à objets : serveurs de noms

## Transparence du partage

### Propriété

L'usage partagé (et en parallèle) d'une ressource doit rester cohérent ( $\equiv$  sémantique équivalente au cas centralisé).

### Exemple

- Niveau service système : cohérence d'accès à un fichier partagé : assurer les contraintes d'exclusion mutuelle des lecteurs/rédacteurs, mais **coûteux**
- Niveau langage à objets : limiter l'exécution en parallèle des méthodes sur un objet

### Solution : Mécanismes de synchronisation

Problème : mécanismes connus mais souvent coûteux en réparti

nt

## Transparence des fautes

### Propriété

La répartition induit un contexte moins fiable que celui du centralisé : **panne partielle**

### Exemple

- Niveau service système : un service n'est plus accessible (serveur de noms!)
- Niveau langage à objets : un appel à distance de méthode peut échouer

### Solution : Traitement d'exception et atomicité

Atomicité : un traitement s'exécute en entier ou pas du tout

nt

## Transparence de la réplication

### Propriété

La répartition permet la redondance pour plus de fiabilité

### Exemple

- Niveau service système : assurer le maintien de plusieurs copies cohérentes d'un même fichier
- Niveau langage à objets : assurer la réplication transparente d'un objet
- Niveau intergiciel : assurer que plusieurs serveurs répliqués évoluent en cohérence

### Solution : Synchronisme virtuel

Notion de groupe et de protocoles de diffusion atomique

nt

## Transparence de la migration

### Propriété

Permettre la migration de code, de processus, d'agents, d'objets.

### Exemple

- Niveau service système : déplacer un serveur d'une machine chargée à une machine sous-utilisée
- Niveau langage à objets :
  - code mobile : langages de script
  - objets mobiles (ou agents mobiles)

### Solution : la mobilité des traitements et/ou des données

Agents mobiles (contexte d'exécution mobile), code mobile

nt



## Transparence de charge

### Propriété

Masquer (et empêcher) les phénomènes de surcharge, écroulement

### Exemple

La répartition permet naturellement la mise en œuvre de techniques d'équilibrage de charge

- Niveau système : reconfigurer dynamiquement les services sur les machines disponibles selon la charge des serveurs
- Niveau grappe (cluster) : répartir les traitements parallèles de façon équilibrée sur les différents processeurs

### Solutions : réflexivité, machine virtuelle

Réflexivité : possibilité d'auto observation des composants

Machine virtuelle : dissocier environnement d'exécution et support matériel

nt

### En résumé

#### Répartition



Accès et partage de ressources  
via un réseau de communication  
à tout usager qui en a le droit  
et où qu'il soit

### Les épines

- Pas d'horloge globale
- Pas d'état global immédiat
- Fiabilité partielle
- Sécurité relative
- Non déterminisme

nt

## Transparence d'échelle

### Propriété

Permettre l'extension d'un système sans remettre en cause son fonctionnement global

### Exemple

- Niveau système : introduire de nouveaux serveurs sur de nouvelles machines pour s'adapter à une augmentation de l'activité applicative

### Solution : Adaptabilité et autonomie

Adaptabilité et autonomie : mise en œuvre de mécanismes automatique d'adaptation dynamique

nt



## Systèmes et algorithmes répartis

### Modèle standard et principes algorithmiques

Philippe Quéinnec, Gérard Padiou

ENSEEIH  
Département Sciences du Numérique

13 septembre 2023

## Plan

- 1 Le modèle standard
  - Approche événementielle
  - Causalité
  - Abstraction d'un calcul
- 2 Clichés (snapshots)
  - Prise de cliché
  - Utilisation des clichés
- 3 Description des algorithmes
  - Description du comportement des processus
  - Exemple : l'élection

## Modéliser un calcul réparti

### Objectifs

- Description statique et description comportementale
- Abstraction pour faciliter l'analyse
- Validation de propriétés (sûreté et vivacité)

### Les éléments de modélisation

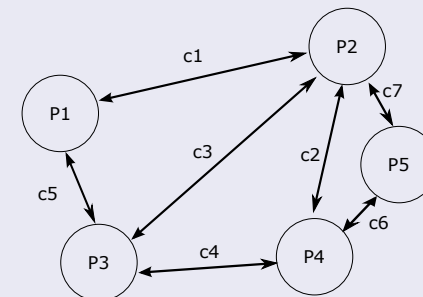
- Les activités, processus, sites, etc  $\Rightarrow$  site logique
- La communication : liens, liaisons, canaux, protocoles (point à point, diffusion)...
- Les connaissances globales de chaque site logique

[ Précis 2.2.1 pp.29–30 ]

## Vision statique : Graphe de processus

### Graphe structurel (statique)

- Sommets  $\equiv$  processus / sites
- Arcs  $\equiv$  liaisons de communication / canaux



## Propriétés

### Propriétés des processus / sites

- Un processus possède une identité unique
- Un processus possède un état rémanent
- Un processus exécute un code séquentiellement
- Un processus n'a qu'une connaissance partielle des autres
- Un processus peut communiquer avec un voisinage
- Défaillance : pause, arrêt définitif, comportement byzantin

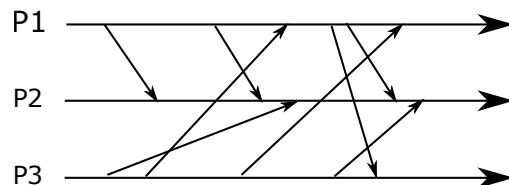
### Propriétés du réseau

- Multiples paramètres : point à point ou diffusion, (a)synchrone, fiable, délais bornés, etc
- Messages : perte, duplication, modification du contenu

## Système asynchrone

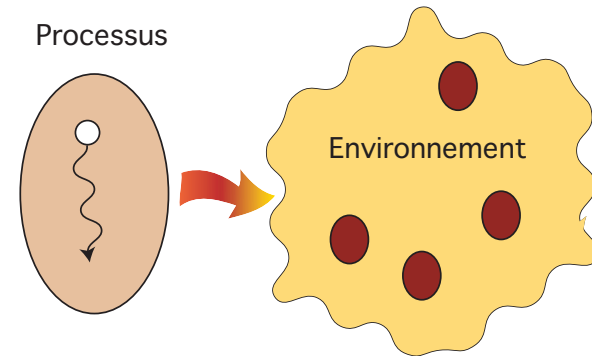
### Modèle asynchrone

- Pas de temps externe commun
- Progression de chaque processus à son rythme
- Délai de transmission arbitraire



Modèle réaliste, faibles hypothèses, plus complexe pour développer et raisonner

## Connaissances d'un processus

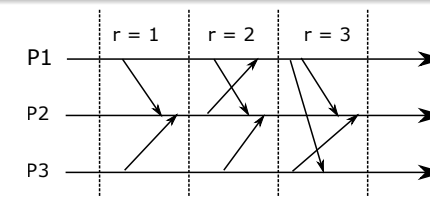


- Nombre de processus ?
- Voisinage de communication ?
- Structure du réseau : maillé, anneau, statique/dynamique, etc

## Système synchrone

### Modèle synchrone

- Borne connue de délai de communication et de pas de calcul
- Pas de calcul (*round*) globaux
- Un message émis dans un pas est reçu au pas suivant / dans le même pas (selon le modèle)
- Cas particulier : rendez-vous = échange synchrone



Modèle peu réaliste, puissant.

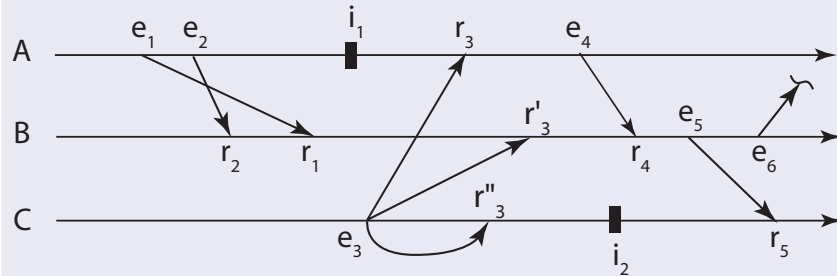
Modèle mixte : sûreté si asynchrone, sûreté + vivacité si synchrone.

## Vision dynamique : Chronogramme



### Représentation événementielle

- Description globale, dans un repère temporel global
- Trois types d'événements : émission, réception, interne
- Modélisation de la communication : diffusion, perte, délais, etc
- **Causalité** entre événements

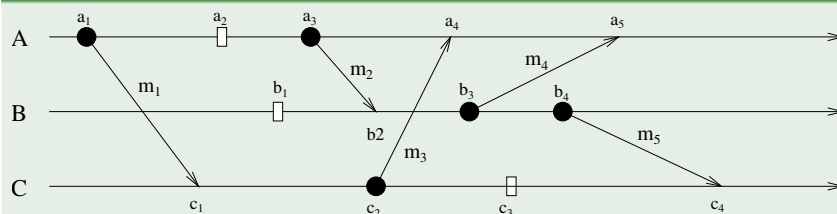


[ Précis 2.2.2, 2.2.3 pp.30-31 ]

## Relation de causalité



### Exemple



$a_1 \prec a_2 \prec a_3 \prec a_4 \prec \dots$   
 $a_1 \prec c_1, c_2 \prec a_4, b_4 \prec c_4$   
 $a_1 \prec c_3$  (car  $a_1 \prec c_1 \prec c_2 \prec c_3$ )  
 $a_2 \prec c_4$   
 $a_3 \parallel c_2$

## Relation de causalité (Lamport 1978)



### Ordre partiel strict entre événements $\prec$

- Les événements d'un processus sont totalement ordonnés :  
e et e' sur le même site, et e précède e', alors  $e \prec e'$ .
- L'émission d'un message précède causalement sa réception :  
Si  $e = \text{émission}(m)$  et  $e' = \text{réception}(m)$ , alors  $e \prec e'$ .
- Transitivité :  $\forall e, e', e'' : e \prec e' \prec e'' \Rightarrow e \prec e''$

- La relation  $\prec$  est un **ordre partiel** :  $e \parallel e' \triangleq e \not\prec e' \wedge e' \not\prec e$
- Indépendant du temps physique mais consistant avec :  
 $e \prec e' \Rightarrow e$  est survenu avant  $e'$  dans le temps absolu

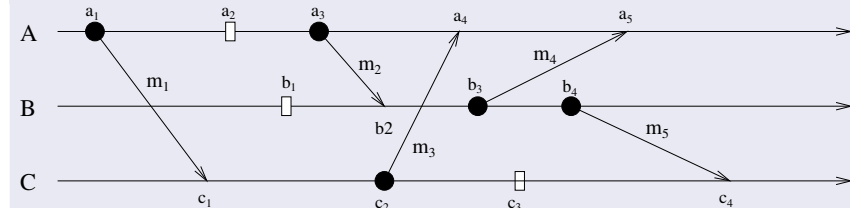
[ Précis 2.2.4 p.31 ]

1. *Time, Clocks and the Ordering of Events in a Distributed System*, Leslie Lamport. Communications of the ACM, July 1978.

## Abstraction d'un calcul réparti



### Exécutions causalement équivalentes



- Ensemble d'événements + relation causale  
→ ensemble d'exécutions réelles équivalentes  
 $a_1; b_1; c_1; a_2; \dots \equiv a_1; a_2; c_1; b_1; \dots$   
 $a_1; c_1; a_2; \dots \not\equiv c_1; a_1; a_2; \dots$  car  $a_1 \prec c_1$
- Le choix des événements **fixe un niveau d'observation**

## Passé / futur causal

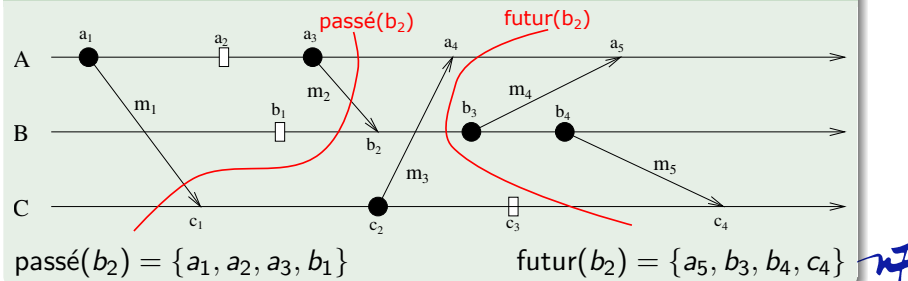
### Partition des événements

$$\text{passé}(e) \triangleq \{f \mid f \prec e\}$$

$$\text{futur}(e) \triangleq \{f \mid e \prec f\}$$

$$\text{concurrence}(e) \triangleq \{f \mid f \not\prec e \wedge e \not\prec f\}$$

### Exemple

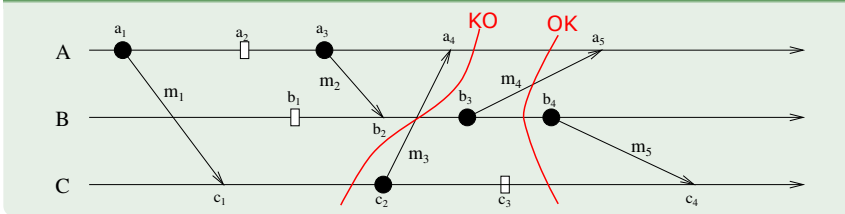


## Passé et coupure cohérente

Une coupure  $C$  est cohérente ssi  $C = \bigcup_{e \in C} (\text{passé}(e) \cup \{e\})$  :

- Pas de « trou » sur un site
- Une réception n'est pas présente sans son émission

### Exemple



*nf*

## Coupure et coupure cohérente



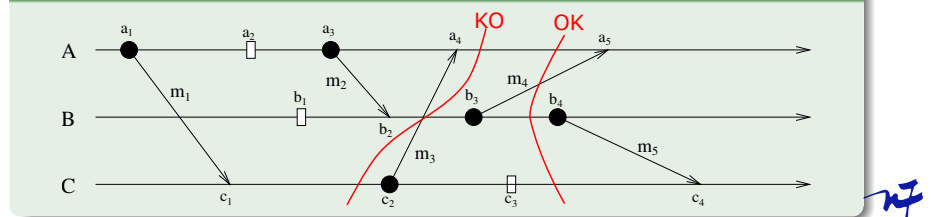
### Coupure

Une **coupure** est un ensemble d'événements qui forment des préfixes complets des histoires locales.

### Coupure cohérente

Une coupure  $C$  est **cohérente** si  $\forall e \in C : \forall e' : e' \prec e \Rightarrow e' \in C$

### Exemple

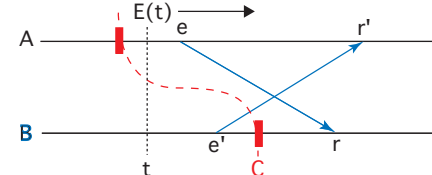


*nf*

## Coupure cohérente et état global



Une coupure cohérente correspond à un état global qui **aurait pu** exister.

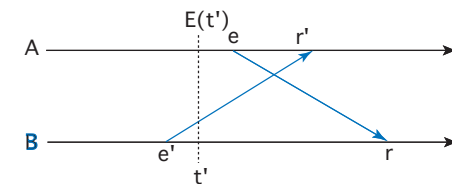


### Réalité

La coupure est cohérente **mais...**

### État effectif

L'état n'a pas existé à un instant global



*nf*

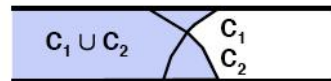
## Treillis des coupures (cohérentes)

### Treillis des coupures

L'ensemble des coupures forme un treillis pour l'inclusion et l'intersection : si  $C_1$  et  $C_2$  sont deux coupures, alors  $C_1 \cup C_2$  et  $C_1 \cap C_2$  sont des coupures.

### Treillis des coupures cohérentes

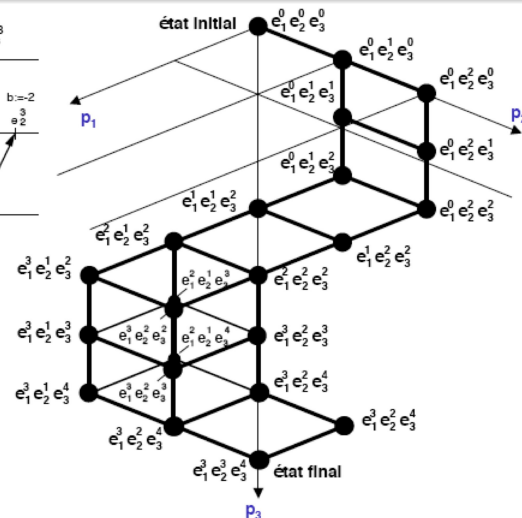
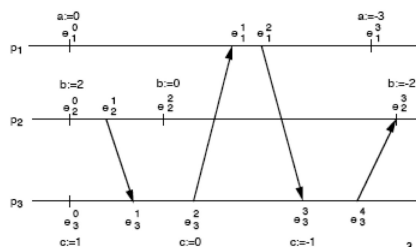
L'ensemble des coupures cohérentes forme un treillis pour l'inclusion et l'intersection : si  $C_1$  et  $C_2$  sont deux coupures cohérentes, alors  $C_1 \cup C_2$  et  $C_1 \cap C_2$  sont des coupures cohérentes.



nt

## Treillis des coupures cohérentes

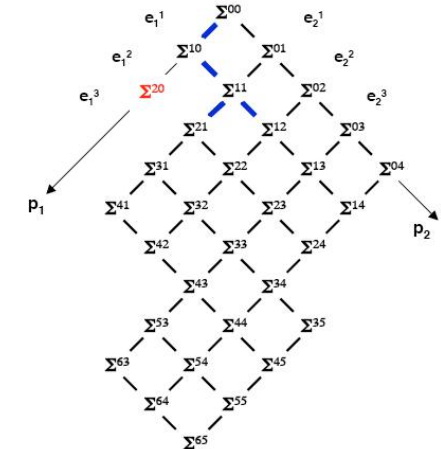
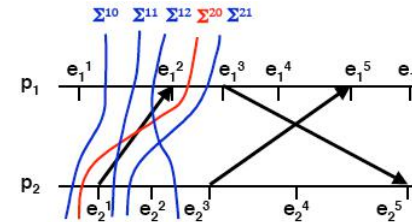
Autre exemple



(dessin : cours S. Krakowiak)

7

## Treillis des coupures cohérentes



- Arc du treillis = occurrence d'un événement possible
- Une exécution = suite d'états globaux cohérents = chemin dans le treillis

Explosion du nombre d'exécutions causalement équivalentes

(dessins : cours S. Krakowiak)

nt

## Plan

- 1 Le modèle standard
  - Approche événementielle
  - Causalité
  - Abstraction d'un calcul
- 2 Clichés (snapshots)
  - Prise de cliché
  - Utilisation des clichés
- 3 Description des algorithmes
  - Description du comportement des processus
  - Exemple : l'élection

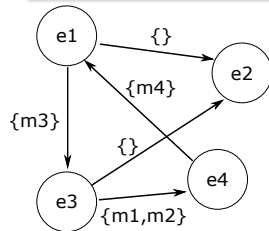
nt

## Prise de cliché (snapshot)



### Définition

Objectif : Capter un état global (passé) des processus et du réseau



- Prise **instantanée** impossible
- Un site collecteur accumule
- Prise **cohérente** de clichés locaux
- Identification des messages en transit

### Cliché global

Clichés locaux + Messages en transit  
 $\{e_1, e_2, e_3, e_4\} + \{m_1, m_2, m_3, m_4\}$

[ Précis 5.1 pp.79–81 ]



## Algorithme de Chandy-Lamport (1985)



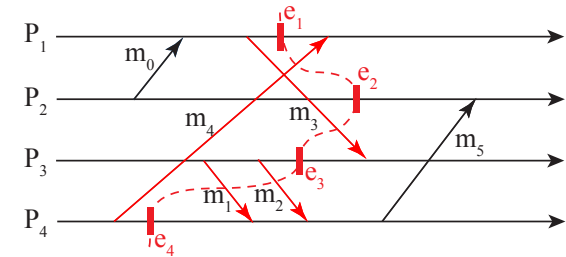
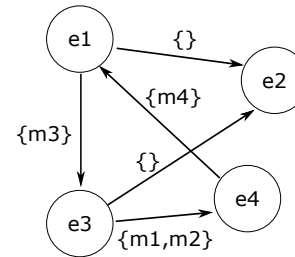
- Un système existant échange des messages ;
- On **superpose** des échanges de messages dédiés pour déclencher des actions locales de sauvegarde de l'état d'un site (= un cliché local + des messages reçus) ;
- Ces états sauvegardés sont **collectés** pour construire un cliché global.

1. *Distributed Snapshots : Determining Global States of Distributed Systems*,  
K. Mani Chandy and Leslie Lamport. ACM Transactions on Computer Systems,  
Feb. 1985



## Prise de cliché (snapshot)

Schéma temporel de la prise de cliché



## Algorithme de Chandy-Lamport

### Idée

- Construire une coupure cohérente au moyen de **marqueurs** visitant les sites.
- Les messages émis par  $S_j$  **avant** le passage du marqueur sur  $S_j$ , et reçus par  $S_i$  **après** le passage du marqueur sur  $S_i$ , sont les messages en transit de  $S_j$  vers  $S_i$ .
- Les messages reçus avant le passage du marqueur sont intégrés à l'état local du site et ne sont plus en transit.
- Les messages émis après le passage du marqueur ne sont pas dans le cliché.



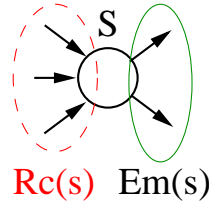


## Algorithme de Chandy-Lamport



### Hypothèses

- Canaux unidirectionnels et **fifo** :  
 $\forall s, Rc(s)$  : canaux en réception  
 $Em(s)$  : canaux en émission
- Réseau fortement connexe ( $\forall s, s' : \exists s \rightarrow^* s'$ )

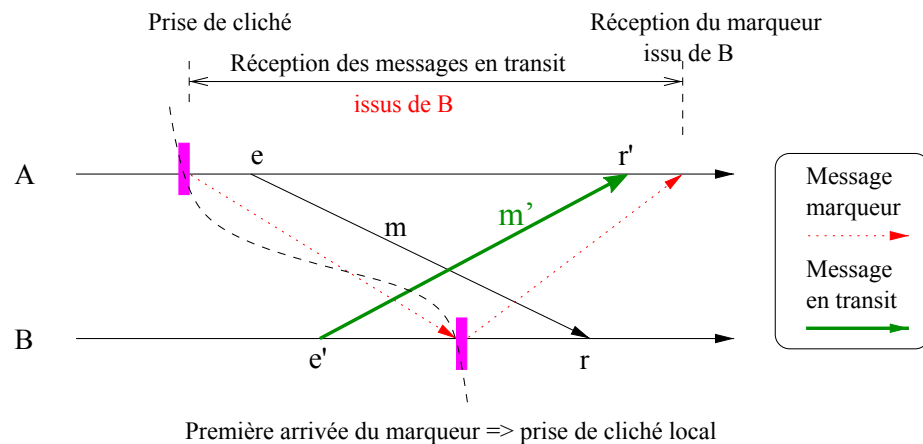


### Principe de l'algorithme

- Utilisation de messages **marqueurs**
- Répartition de l'évaluation : chaque site  $s$  évalue :
  - son cliché local ;
  - les messages considérés en transit sur ses canaux en réception  $Rc(s)$



## Prise de clichés locaux et marqueurs



## Algorithme de Chandy-Lamport



### Comportement d'un site $s$

Sur réception d'un **premier marqueur** :

- 1 Prendre son cliché local  $L_s$  et émettre un **marqueur** sur chaque canal d'émission  $c \in Em(s)$
- 2 Enregistrer dans une liste  $enTransit[c]$  les messages reçus sur chaque canal de réception  $c \in Rc(s)$  jusqu'à la réception d'un **marqueur** sur ce canal
- 3 Lorsqu'un marqueur a été reçu sur **tous** ses canaux de réception, communiquer au collecteur cet état partiel :  
 $\langle L_s, \{enTransit[c] \mid c \in Rc(s)\} \rangle$

Déclenchement de la prise de cliché : envoi d'un message marqueur à un site quelconque.



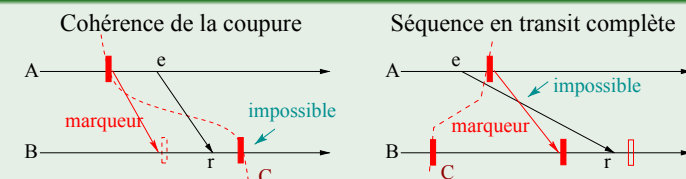
## Vérifier la correction...



### Propriétés

- Sûreté
  - Coupure cohérente
  - Collecte complète des messages en transit
- Vivacité
  - Tout site finit par prendre un cliché local
  - Un marqueur finit par arriver sur chaque canal de réception

### Exemple



## État enregistré = état *possible*

### État enregistré

- $\Sigma_{enreg}$  = cliché enregistré
- $\Sigma_{init}$  = coupure cohérente contenant l'événement déclencheur du cliché
- $\Sigma_{final}$  = coupure cohérente dans lequel le protocole de prise de cliché est terminé

Alors  $\Sigma_{init} \prec \Sigma_{enreg} \prec \Sigma_{final}$

(il existe un chemin de  $\Sigma_{init}$  à  $\Sigma_{final}$  passant par  $\Sigma_{enreg}$  dans le treillis des coupures cohérentes)

Exemple : sur le treillis page 18, si  $\Sigma_{init} = \Sigma^{11}$  et  $\Sigma_{final} = \Sigma^{32}$ ,  $\Sigma_{enreg}$  peut être  $\Sigma^{11}$ ,  $\Sigma^{21}$ ,  $\Sigma^{12}$ ,  $\Sigma^{31}$ ,  $\Sigma^{22}$  ou  $\Sigma^{32}$ , et a pu ne pas être traversé dans la réalité.

nf

## Utilisation du cliché : propriété possible/certaine



### Prédicat possible/certain

Pour un prédicat  $P$  :

- $Pos(P)$  (possibly  $P$ ) : il existe une observation cohérente (= un chemin dans le treillis) qui passe par un état où  $P$  est vrai.
- $Def(P)$  (definitely  $P$ ) : toutes les observations cohérentes (= tous les chemins) passent par un état où  $P$  est vrai.

### Vérification

- $P(\Sigma_{enreg}) \Rightarrow Pos(P)$  mais pas l'inverse...
- $\neg Pos(P) \Rightarrow Def(\neg P)$  mais pas l'inverse...

nf

## Utilisation du cliché : propriété stable



### Prédicat stable

Un prédicat  $P$  sur un état global  $E$  d'un système est stable ssi  $\forall E' : E \prec E' \wedge P(E) \Rightarrow P(E')$

(exemples : le calcul est terminé, il y a eu 10 messages reçus...)

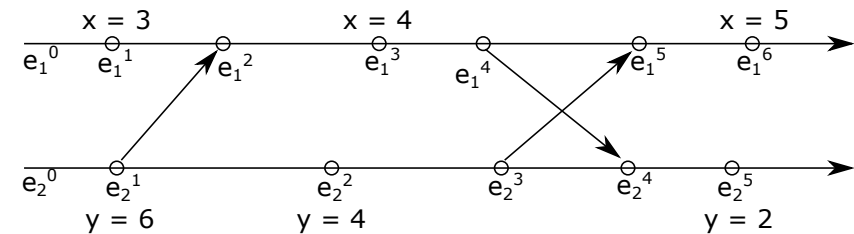
### Vérification de $P$

Si  $P$  est un prédicat stable alors :

- $P(\Sigma_{enreg}) \Rightarrow P(\Sigma_{final})$  (et tout état ultérieur)
- $\neg P(\Sigma_{enreg}) \Rightarrow \neg P(\Sigma_{init})$  (et tout état antérieur)

nf

## Exemple de vérification de propriétés



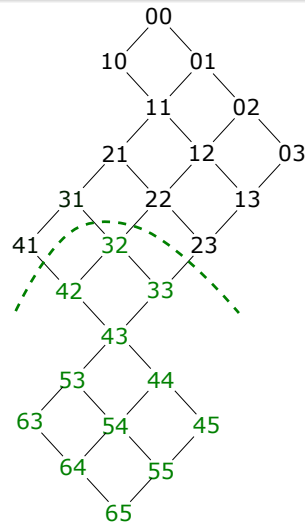
- $x - y \geq 0$ ?  
(en supposant  $x$  croissant,  $y$  décroissant  $\Rightarrow$  propriété stable)
- $Pos(x = y - 2)$ ?
- $Def(x = y)$ ?

(d'après Lorenzo Alvisi)

nf

## Exemple de vérification

Propriété stable

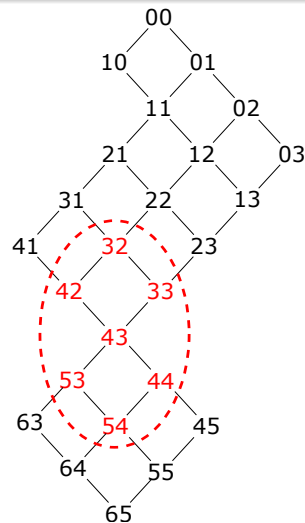


- $x - y \geq 0$  ?  
(sous l'hypothèse  $x \uparrow, y \downarrow$ )
- N'importe quel cliché  $\Sigma_{enreg}$  obtenu après  $\Sigma^{32}$  permet de le vérifier



## Exemple de vérification

Certitude

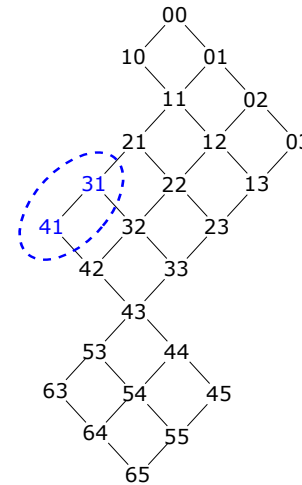


- $Def(x = y)$  ?
- Vrai
- $Pos(x = y)$  pas détecté si on capture un état antérieur à  $\Sigma^{32}$  ou postérieur à  $\Sigma^{54}$
- La capture d'un état (p.e.  $\Sigma^{42}$ ) ne permet pas de conclure



## Exemple de vérification

Possibilité



- $Pos(x = y - 2)$  ?
- $x = y - 2$  est vrai dans les états cohérents  $\Sigma^{31}$  et  $\Sigma^{41}$
- Détecté uniquement si  $\Sigma_{enreg} \in \{\Sigma^{31}, \Sigma^{41}\}$
- $\Sigma_{enreg}$  pas nécessairement survenu dans la réalité



## Utilisation du cliché : propriété possible/certaine

Principe de la vérification



- Un processus moniteur  $M$  collecte tous les états locaux
- $M$  construit le treillis des coupures cohérentes (à partir d'un codage complet de la relation de causalité, cf chapitre suivant)
- Pour évaluer  $Pos(P)$  : parcourir le treillis depuis l'état initial, niveau par niveau, et s'arrêter au premier état où  $P$  est vrai. Aucun état  $\Rightarrow \neg Pos(P)$ .
- Pour évaluer  $Def(P)$  : parcourir le treillis depuis l'état initial, niveau par niveau, en ne développant que les états vérifiant  $\neg P$ . Si plus d'état, alors  $Def(P)$  ; si état final atteint (et  $\neg P$  dans cet état) alors  $\neg Def(P)$ .
- Explosion combinatoire : pour  $N$  sites ayant chacun au plus  $m$  états, possiblement  $m^N$  coupures cohérentes.



## Plan

- 1 Le modèle standard
  - Approche événementielle
  - Causalité
  - Abstraction d'un calcul
- 2 Clichés (snapshots)
  - Prise de cliché
  - Utilisation des clichés
- 3 Description des algorithmes
  - Description du comportement des processus
  - Exemple : l'élection

*nf*

## Description des algorithmes

🎵

```

Process P(id : 0..N-1)
  <variables locales>
  on <condition-logique> :
    <Action>
  on reception message(arg) [ from P(j) ] :
    <Action>
  on <condition-logique> ^ reception message(arg) :
    <Action>
  on start : // initialement
    <Action>
  ...

```

- Action : modification des variables locales et/ou envoi(s) de message, ou terminaison (**terminate**)
- Envoi : **send** Msg(<args>) **to** <destinataire(s)>
- Choix d'un événement à traiter : non déterministe parmi ceux ayant la garde vraie et un message à consommer

*nf*

## Principes algorithmiques

- Algorithmes symétriques
  - code répliqué
  - données initiales propres : identité, voisinage de communication
  - pas de variables partagées, éventuellement connaissances statiques communes (p.e. graphe, nombre de sites)
- Structurer les échanges de messages :
  - maillage (graphe complet)
  - anneau
  - arbre de recouvrement
- Étudier des problèmes génériques :
  - Les services : datation, exclusion mutuelle, consensus, élection...
  - Les observations de propriétés stables : terminaison, interblocage
  - La tolérance aux fautes : réplication, atomicité

[ Précis 2.2.5, 2.2.6 pp.32–34 ]

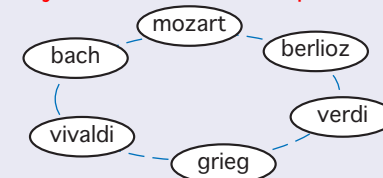
*nf*

## Exemple : l'élection

🎵

### Le problème de l'élection

Objectif : Élire un seul processus



- Un processus a une identité unique qu'il connaît
- Un processus ne connaît pas le nombre global de processus
- Un processus ne connaît pas l'identité des autres
- Communication sur un anneau logique

[ Précis 2.2.9 pp.39–41 ]

1. An improved algorithm for decentralized extrema-finding in circular configurations of processes, Ernest Chang and Rosemary Roberts. Communications of the ACM, May 1979.

*nf*

## Solution correcte ou fausse ?



On suppose que les processus sont totalement ordonnés (ici par leur indice, en pratique, par leur adresse IP par exemple)

```
Process P(id : 0..N-1)
//  $\ominus$  et  $\oplus$  : opérateurs modulo  $N$ 
type Etat = {candidat, élu};
Etat étatCourant ← candidat;
on reception Candidat(proc) from P[id $\ominus$ 1] :
    if (proc < id) send Candidat(proc) to P[id $\oplus$ 1];
    else if (proc = id) étatCourant ← élu;
    else nop; // ignorer le message
on (étatCourant = élu) :
    terminate;
```

Pourquoi cela ne marche-t-il pas ?



## Solution plus complète : tous les processus terminent



```
Process P(id : 0..N-1) {
    type Etat = {candidat, élu, perdant};
    Etat étatCourant ← candidat;
    on start :
        send Candidat(id) to P[id $\oplus$ 1]; // chacun candidate
    on reception Candidat(proc) from P[id $\ominus$ 1]:
        if (proc < id) send Candidat(proc) to P[id $\oplus$ 1];
        else if (proc = id) étatCourant ← élu;
        else nop; // ignorer le message
    on (étatCourant = élu) :
        send Elu(id) to P[id $\oplus$ 1];
    on reception Elu(proc) from P[id $\ominus$ 1]:
        if (proc  $\neq$  id) then
            étatCourant ← perdant;
            send Elu(proc) to P[id $\oplus$ 1];
        endif
    terminate
```

## Solution qui conduit à l'élection du plus petit



```
Process P(id : 0..N-1)
type Etat = {candidat, élu};
Etat étatCourant ← candidat;
on start:
    send Candidat(id) to P[id $\oplus$ 1]; // chacun candidate
on reception Candidat(proc) from P[id $\ominus$ 1]:
    if (proc < id) send Candidat(proc) to P[id $\oplus$ 1];
    else if (proc = id) étatCourant ← élu;
    else nop; // ignorer le message
on (étatCourant = élu) :
    terminate;
```

Pas parfait : un seul processus se termine



## Déclenchement spontané individuel



Pas nécessairement tous candidats au départ (mais tous éligibles)

```
Process P(id : 0..N-1)
type Etat = {candidat, élu, perdant};
Etat étatCourant ← candidat;
on random() :
    send Candidat(id) to P[id $\oplus$ 1];
on reception Candidat(proc) from P[id $\ominus$ 1]:
    if (proc < id) send Candidat(proc) to P[id $\oplus$ 1];
    else if (proc = id) étatCourant ← élu;
    else if (proc > id) send Candidat(id) to P[id $\oplus$ 1];
    :
```



## Conclusion

- Modélisation par des événements locaux
- Relation entre ces événements, en particulier la causalité
- Représentation avec des chronogrammes
- Notion d'état global, de coupure
- Calcul d'un état global



# Systèmes et algorithmes répartis

## Causalité et datation

Philippe Quéinnec, Gérard Padiou

ENSEEIH  
Département Sciences du Numérique

13 septembre 2023

*nf*

## Plan

- 1 **Problème de datation**
  - Temps logique
  - Horloge de Lamport
  - Horloge vectorielle de Fidge-Mattern
- 2 **Les protocoles de communication**
  - Délivrance ordonnée
  - Protocoles ordonnés
  - Protocole causalement ordonné
  - Diffusion causalement ordonnée

*nf*

## plan

- 1 **Problème de datation**
  - Temps logique
  - Horloge de Lamport
  - Horloge vectorielle de Fidge-Mattern
- 2 **Les protocoles de communication**
  - Délivrance ordonnée
  - Protocoles ordonnés
  - Protocole causalement ordonné
  - Diffusion causalement ordonnée

*nf*

## Datation des événements



### Objectif

Associer une date à chaque événement pour :

- **ordonner** les événements → compatible avec la causalité
- **identifier** les événements → dates uniques

### Moyens

- Horloges matérielles
- Horloges logiques

### Difficultés

- Pas d'horloge globale
- Tous les événements ne sont pas causalement liés
- Datation cohérente avec la relation causale :

$$\forall e, e' : e \prec e' \Rightarrow d_e < d_{e'}$$

*nf*

## Horloges matérielles



### Idée

- Utiliser les horloges matérielles de chaque site
- Risque** : Datation incohérente de l'événement de réception d'un message : la date de réception précède la date d'émission
- Possible** si l'horloge du site de réception est en retard (suffisamment) sur celle du site de l'émetteur

### Difficultés

- Cohérence avec la causalité
- Datation définissant un ordre total
- Pas d'unicité des dates

[ Précis 3.3.1 pp.43–44 ]



## Temps logique



### Principes de base

- Associer à chaque site une horloge logique locale
- L'horloge compte les événements au lieu du temps réel
- Surcharger les messages avec leur date d'émission
- Recaler si nécessaire** l'horloge locale d'un site lors de chaque réception de message
- Avantage** : vision plus abstraite d'un calcul réparti

[ Précis 3.3.2 pp.44–47 ]



## Horloges matérielles



### Solutions

- Synchronisation des horloges locales :  
invariant  $\max_{i=1..N}(h_i) - \min_{i=1..N}(h_i) < \epsilon$
- Causalité respectée si  $\epsilon$  est inférieur au temps de transmission d'un message
- Unicité en utilisant couple  $(date, id \text{ du site})$

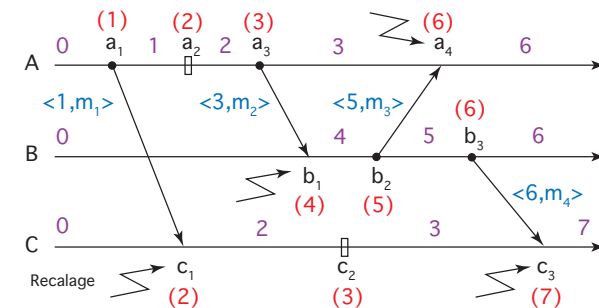
Protocole de synchronisation d'horloges possible mais dans des contextes réseaux assurant une certaine qualité de service ou par l'usage d'un signal externe (horloge atomique, GPS)



## Une tentative...



- Un compteur « horloge » sur chaque site
- Surcharge des messages et recalage de compteur



- 😊  $e \prec e' \Rightarrow d(e) < d(e')$
- 😊😞 Mêmes dates  $\Rightarrow$  pas causalement liés
- 😞  $d(e) < d(e') \not\Rightarrow e \prec e'$  (ex :  $d(c_2) < d(b_3)$  mais  $c_2 \not\prec b_3$ )





## Horloge de Lamport



### Propriétés

- Introduit un ordre total entre événements  
→ Dates distinctes pour tout couple d'événements
- Date = (compteur local, numéro de site)  
ordre total sur les sites ⇒ ordre lexicographique total

```
struct Date { int cpt; // compteur d'événements
              int s;   // numéro de site
}

bool prec (Date d1, Date d2) {
    return (d1.cpt < d2.cpt)
        || ((d1.cpt == d2.cpt) && (d1.s < d2.s));
}
```

1. *Time, Clocks and the Ordering of Events in a Distributed System*, Leslie Lamport. Communications of the ACM, July 1978.

## Horloge vectorielle de Fidge-Mattern (1988)



### Objectif

Représenter exactement la relation de causalité

### Propriétés

- Utilisation de vecteurs de dimension égale au nombre de sites
- Pour un événement  $e$ ,  $HV(e)[i]$  = nombre d'événements du passé de  $e$  sur  $p_i$  (y compris  $e$ )
- Coût plus élevé : surcharge des messages par un vecteur

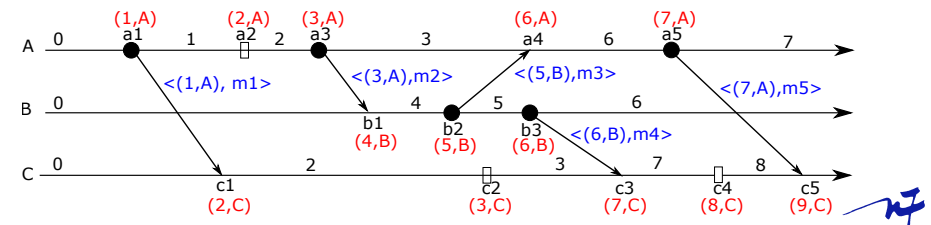
1. *Timestamps in Message-Passing Systems That Preserve the Partial Ordering*, Colin J. Fidge. 11th Australian Computer Science Conference, 1988.  
2. *Virtual Time and Global State in Distributed Systems*, Friedemann Mattern. Int'l Workshop on Parallel and Distributed Algorithms, 1989.

## Actions associées aux événements



Chaque site  $s$  possède une horloge entière  $H_s$ . Chaque événement est daté avec le couple  $(H_s \text{ après l'action, id du site})$ .

| Type d'événement sur un site $s$                                 | Action sur le site $s$   |
|--|--|
| Événement interne sur $s$  | $H_s \leftarrow H_s + 1;$  |
| Émission sur $s$ de $m$  | $H_s \leftarrow H_s + 1;$<br>envoi de $\langle H_s, s \rangle, m \rangle;$ |
| Réception sur $s$ de $\langle \langle dm, s' \rangle, m \rangle$ | $H_s \leftarrow \max(H_s, dm) + 1;$  |

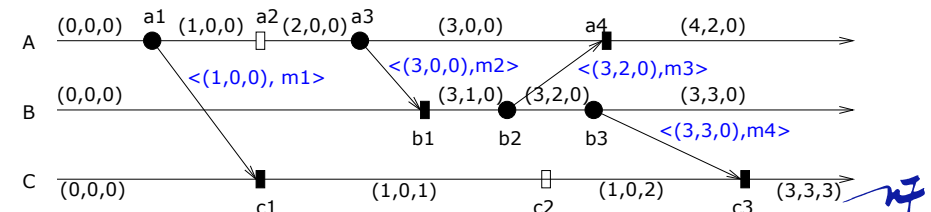


## Actions associées aux événements



Chaque site  $s$  possède une horloge vectorielle  $H_s$ .  
Chaque événement est daté avec le résultat de l'action.

| Type d'événement sur un site $s$             | Action sur le site $s$  |
|--|---|
| Événement interne sur $s$                    | $H_s[s] \leftarrow H_s[s] + 1$  |
| Émission sur $s$ de $m$                      | $H_s[s] \leftarrow H_s[s] + 1$<br>envoi de $\langle H_s, m \rangle$                             |
| Réception sur $s$ de $\langle dm, m \rangle$ | $H_s[s] \leftarrow H_s[s] + 1$<br>$H_s[s'] \leftarrow \max(H_s[s'], dm[s']), \forall s' \neq s$ |



## Horloge vectorielle de Fidge-Mattern



### Expression des relations entre dates

$$\begin{aligned} D \leq D' &\triangleq \forall i : D[i] \leq D'[i] \\ D < D' &\triangleq D \leq D' \wedge \exists k : D[k] < D'[k] \\ D \parallel D' &\triangleq \neg(D < D') \wedge \neg(D' < D) \end{aligned}$$

### Datation isomorphe à l'ordre causal

$$\begin{aligned} e \prec e' &\Leftrightarrow D(e) < D(e') \\ e \parallel e' &\Leftrightarrow D(e) \parallel D(e') \end{aligned}$$

*Handwritten signature*

## Datation et coupure cohérente

### Coupure cohérente

$$C = (c_1, \dots, c_n) \text{ cohérente} \Leftrightarrow HV(C) = \langle HV(c_1)[1], \dots, HV(c_n)[n] \rangle$$

- Soit  $C$  cohérente. Alors  $\forall i, j : HV(c_i)[i] \geq HV(c_j)[i]$ .  
En effet, l'incréméntation de  $HV(c_i)[i]$  ne peut venir que d'un événement local ou résulter d'un message provenant du passé.
- Soit  $C$  non cohérente.  
 $\Rightarrow$  il existe un événement  $e_i$  hors coupe qui est dans le passé causal de  $C$   
 $\Rightarrow$  Sur le site  $i : HV(c_i)[i] < HV(e_i)[i]$  et  $HV(e_i)[i] \leq HV(C)[i]$   
 $\Rightarrow HV(C)[i] > HV(c_i)[i]$ , donc  $HV(C) \neq (HV(c_1)[1], \dots, HV(c_n)[n])$

*Handwritten signature*

## Datation des coupures

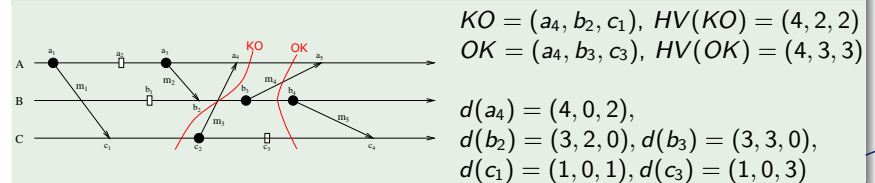


On identifie une coupure  $C$  par  $(c_1, \dots, c_n)$ , ses événements maximaux sur chaque site (les événements internes à la coupure sont implicites)

### Date d'une coupure

- Date d'une coupure  $C = (c_1, \dots, c_n)$  :  
 $HV(C) \triangleq \sup(HV(c_1), \dots, HV(c_n))$
- $C_2 \subsetneq C_1 \Leftrightarrow HV(C_2) < HV(C_1)$

### Exemple (CH2, modèle standard)



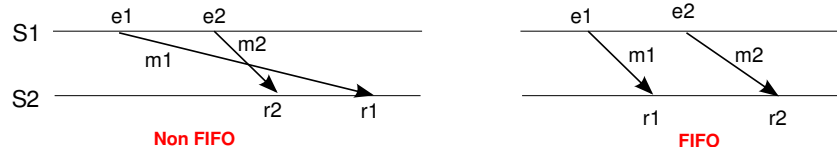
*Handwritten signature*

## Plan

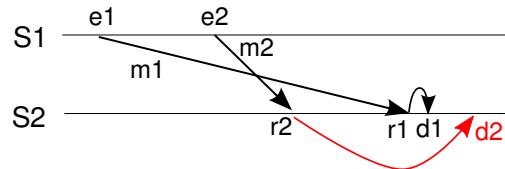
- 1 Problème de datation
  - Temps logique
  - Horloge de Lamport
  - Horloge vectorielle de Fidge-Mattern
- 2 Les protocoles de communication
  - Délivrance ordonnée
  - Protocoles ordonnés
  - Protocole causalement ordonné
  - Diffusion causalement ordonnée

*Handwritten signature*

## Protocole FIFO



$$\forall m, m' : s_1 \xrightarrow{m} s_2 \wedge s_1 \xrightarrow{m'} s_2 \wedge e(m) \prec e(m') \Rightarrow r(m) \prec r(m')$$



délivrance  $\neq$  réception



## Protocole FIFO



### Objectif

Garantir la cohérence des réceptions sur un même site par rapport à leur éventuelle émission depuis un même site

$\Rightarrow$  réordonner les messages reçus sur un site

- Trois événements au lieu de deux par message :
  - l'émission  $e$ ,
  - la réception  $r$ ,
  - la **délivrance**  $d$ .
- Causalité :  $e \prec r \prec d$

- S'exprime par la propriété :

$$\forall s_1, s_2, m, m' : s_1 \xrightarrow{m} s_2 \wedge s_1 \xrightarrow{m'} s_2 \wedge e(m) \prec e(m') \Rightarrow d(m) \prec d(m')$$

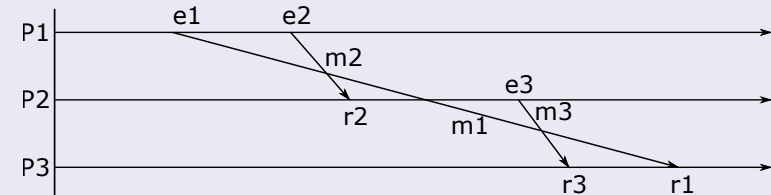


## Protocole causalement ordonné



Objectif : Mettre de l'ordre

### Réceptions incohérentes par rapport aux émissions



$\Rightarrow$  délivrance  $\neq$  réception

1. *Reliable communication in the presence of failures*, Kenneth P. Birman and Thomas A. Joseph. ACM Transactions on Computer Systems, January 1987.



## Protocole FIFO



Réalisation : il suffit de numérotter les messages pour **chaque canal** (couple site d'émission, site de réception)

Récepteur pour **un canal** :

```
type Message = <contenu, numéro>;
int prochain = 0;
SortedSet<Message> enAttente; // trié par numéro
while (true) {
    recevoir m;
    enAttente.add(m);
    while (enAttente.first().numéro == prochain) {
        m ← enAttente.removeFirst();
        délivrer m.contenu;
        prochain++;
    }
}
```



## Protocole causalement ordonné



### Objectif

Garantir la cohérence des réceptions sur un même site par rapport à leur causalité éventuelle en émission  
⇒ réordonner les messages reçus sur un site

- Trois événements au lieu de deux par message :

- l'émission  $e$ ,
- la réception  $r$ ,
- la **délivrance**  $d$ .
- Causalité :  $e \prec r \prec d$

- S'exprime par la propriété :

$$\forall s, m, m' : \neg \xrightarrow{m} s \wedge \neg \xrightarrow{m'} s \wedge e(m) \prec e(m') \\ \Rightarrow d(m) \prec d(m')$$

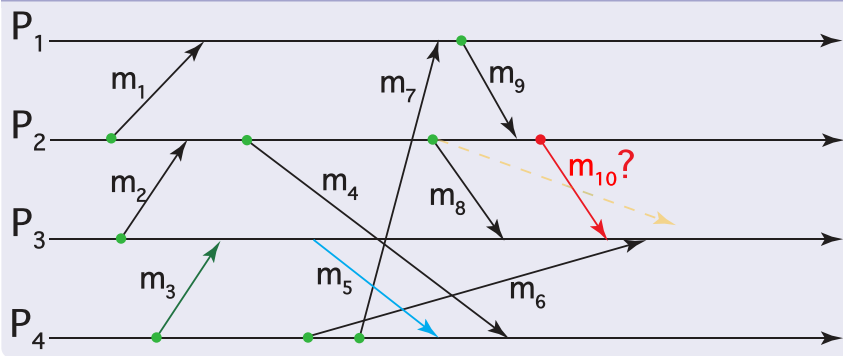
[ Précis 3.2 pp.47–50 ]



## Histoire causale : exemple



### Exemple



$$H_c(m_{10}) = \{m_8, m_4, m_9, m_7, m_1, m_6, m_3, m_2\}$$

$$H_c(m_6) = \{m_3\}$$

$$H_c(m_8) = \{m_4, m_1, m_2\}$$



## Histoire causale



### Histoire causale d'un message $H_c(m)$

L'histoire causale  $H_c(m)$  d'un message  $m$  est l'ensemble des messages qui ont leurs événements d'émission précédant causalement l'émission de  $m$  :

$$H_c(m) = \{m' : e(m') \prec e(m)\}$$

### Critère de délivrance d'un message

Un message  $m$  est délivré sur un site  $s$  ssi tous les messages de son histoire causale **ayant aussi  $s$  comme site de destination** ont été déjà délivrés :

$$\forall s, m' \in H_c(m) : \neg \xrightarrow{m} s \wedge \neg \xrightarrow{m'} s \wedge \Rightarrow d(m') \prec d(m)$$



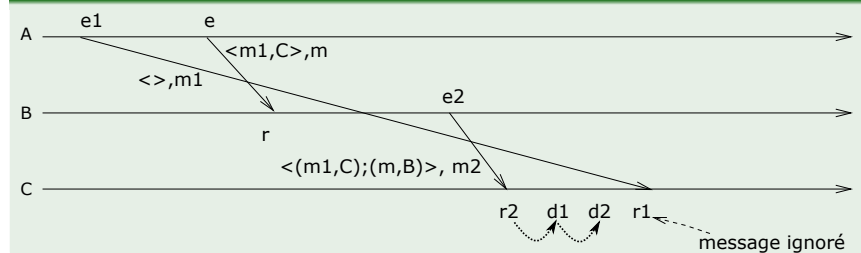
## Approche par surcharge (piggybacking)



### Principe

- Surcharger chaque message avec l'histoire des messages qui le précèdent causalement
- Dans le contexte du courrier électronique : Approche similaire du « réexpédier » avec copie de ce que l'on a reçu

### Exemple



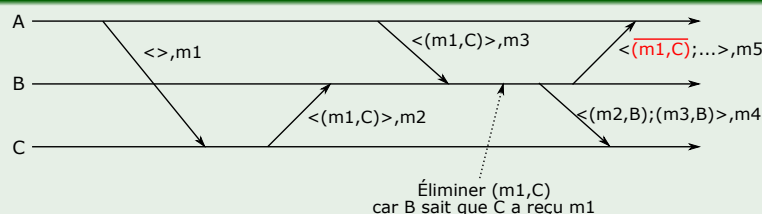
## Approche par surcharge (piggybacking)



### Mise en œuvre

- 😊 Simple et apport d'une certaine tolérance aux pertes de messages par redondance
- 😞 Messages de + en + longs  
⇒ Quand, comment réduire les histoires ?

### Exemple



## Approche par matrice



### Structures de données

Représenter l'histoire causale de chaque message  
Chaque site  $S_s$  gère :

- $MP_s$  : une matrice de précédence causale  
 $MP_s[i, j]$  = nombre de messages émis de  $S_i$  vers  $S_j$ , connu de  $S_s$
- $Dernier_s$  : un vecteur de compteurs des messages reçus de chaque site  
 $Dernier_s[i]$  = nombre de messages reçus du site  $S_i$  sur  $S_s$
- Tout message est surchargé par une copie de la matrice  $MP$  du site émetteur

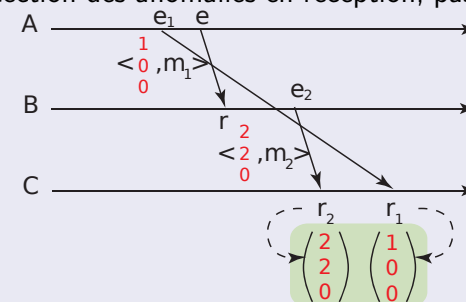
1. *The Causal Ordering Abstraction and a Simple Way to Implement it*, Michel Raynal, André Schiper and Sam Toueg. Information Processing Letters, 1991.

## Datation, premier essai...



### Datation causale (horloge vectorielle)

Permet la détection des anomalies en réception, pas leur prévention



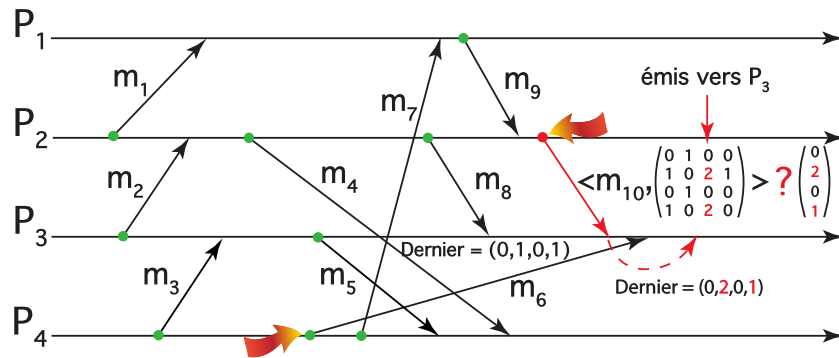
- Lors de  $r_2$ , C sait qu'il y a 2 événements sur A et 2 sur B (dont  $e_2$ ) dans le passé de  $m_2$ , mais le concernent-ils ?
- Lors de  $r_1$ , C découvre qu'un événement de A, causalement antérieur à  $r_2$ , le concerne.

## Actions associées aux événements

| Type d'événement sur un site $s$                           | Action sur le site $s$  |
|--|---|
| Émission sur $s$ de $m$ vers $s'$                          | $MP_s[s, s'] ++$<br>envoi de $\langle MP_s, M \rangle$  |
| Réception sur $s$ de $\langle MP, m \rangle$ issu de $s'$  | $MP_s \leftarrow \max(MP, MP_{s'})$<br>$Dernier_s[s'] ++$   |
| Délivrance sur $s$ de $\langle MP, m \rangle$ issu de $s'$ | $Délivrable(m) \triangleq$<br>$Dernier_s[s'] = MP[s', s]$<br>$\wedge \forall i \neq s : Dernier_s[i] \geq MP[i, s]$ |

$m$  délivrable  $\triangleq$  FIFO entre  $s'$  et  $s$  et il ne précède pas les messages dont l'émission le précède causalement.

## Contrôle des délivrances



## Horloges de plus en plus précises

### Horloges de Lamport

Passé de l'ensemble du système, connu de  $s$ , réduit à la longueur de la plus longue chaîne causale aboutissant à l'événement.

### Horloges vectorielles

Connaissance de premier ordre : passé de  $s'$  que  $s$  connaît, connaissance par  $s$  que  $s'$  a eu un certain nombre d'événements.

### Horloges matricielles

Connaissance de second ordre : connaissance par  $s$  de la connaissance par  $s'$  du passé de  $s''$ .

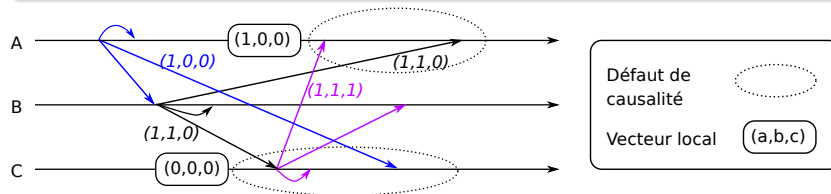
Par exemple, permet de savoir que *tous les sites* connaissent un événement donné.

## Diffusion causalement ordonnée

La **diffusion** ordonnée est plus simple que la communication point-à-point !

### Approche par matrice causale

- Il suffit de gérer un **vecteur** d'émission au lieu d'une matrice
- Toutes les colonnes sont identiques : un processus envoie le même nombre de messages à tous



[ Précis 3.2.2 pp.50–51 ]

1. *Lightweight Causal and Atomic Group Multicast*, Kenneth P. Birman, André Schiper and Pat Stephenson. ACM Trans. on Computer Systems, 1991.

## Conclusion

- Datation logique** des événements
- Protocoles de communication **ordonnés**
- Distinction réception / **délivrance**

# Systèmes et algorithmes répartis

## Problèmes génériques

Philippe Quéinnec, Gérard Padiou

ENSEEIH  
Département Sciences du Numérique

13 septembre 2023

## Plan

- 1 Exclusion mutuelle
  - Le problème
  - Jeton circulant
  - Algorithme de Ricart-Agrawala
  - Algorithme à base d'arbitres
- 2 Détection de la terminaison
  - Le problème
  - Terminaison sur un anneau
  - Algorithme des quatre compteurs
  - Algorithme des crédits
- 3 Détection de l'interblocage
  - Le problème
  - Caractérisation de l'interblocage
  - Algorithme de Chandy, Misra, Haas
- 4 La diffusion fiable

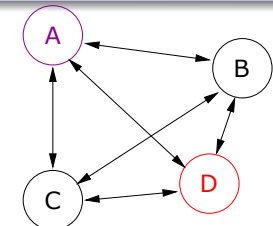
## Plan

- 1 Exclusion mutuelle
  - Le problème
  - Jeton circulant
  - Algorithme de Ricart-Agrawala
  - Algorithme à base d'arbitres
- 2 Détection de la terminaison
  - Le problème
  - Terminaison sur un anneau
  - Algorithme des quatre compteurs
  - Algorithme des crédits
- 3 Détection de l'interblocage
  - Le problème
  - Caractérisation de l'interblocage
  - Algorithme de Chandy, Misra, Haas
- 4 La diffusion fiable

## Spécification du problème

```
process P(int i) :
  ...; Entrer(i); <SC>; Sortir(i); ...
```

### Diagramme d'état



D est en exclusion  
A est candidat

- Sûreté : Un processus **au plus** en exclusion  
 $\forall i, j :: P_i.\text{exclusion} \wedge P_j.\text{exclusion} \Rightarrow i = j$
- Vivacité faible : pas d'interblocage (certains candidats finissent par entrer)
- Vivacité forte : Tout candidat finit par entrer
- Protocole : Tout processus en exclusion finit par sortir

[ Précis 4.1 pp.63–65 ]

## Élection vs exclusion mutuelle

### Problèmes similaires...

Isoler un processus parmi tous : introduire une **asymétrie**

### ... mais bien différents

- Élection d'un **quelconque** des processus mais exclusion mutuelle parmi les **candidats**
- L'élection est définitive mais l'exclusion mutuelle se termine et se transmet  $\Rightarrow$  évolution **dynamique**

## Algorithme à base de jeton circulant

```

Process P(i : 0..N-1) {
  type Etat = {hors,candidat,exclusion}
  Etat EC ← hors;
  bool jeton ← (i = 0); // jeton initialement sur site 0
  on (EC = hors) :           // hors → candidat
    EC ← candidat;
  on (EC = candidat ∧ jeton) : // candidat → exclusion
    EC ← exclusion;
  on (EC = exclusion) :      // exclusion → hors
    EC ← hors;
    send MsgJeton to Pi⊕1
    jeton ← false;
  on reception MsgJeton :    // réception jeton
    jeton ← true;
  on jeton ∧ EC = hors :      // transmission jeton
    send MsgJeton to Pi⊕1
    jeton ← false;
}

```

## Algorithme à base de jeton circulant

### Algorithme basé sur le contrôle d'un objet circulant

- Anneau logique (indépendant de la structure du réseau physique) : chaque site a un successeur
- Jeton circulant :
  - un site non demandeur transmet le jeton à son successeur
  - un site demandeur attend le jeton pour obtenir l'exclusion mutuelle
  - un site qui sort d'exclusion mutuelle transmet le jeton à successeur

### Propriétés

- Sûreté : unicité du jeton
- Vivacité : intégrité de l'anneau (existence et circulation du jeton)

## Algorithmes à base de permission

Un processus candidat doit demander à d'autres processus la **permission** d'entrer en exclusion

- À tout  $P_i$  on associe un ensemble  $D_i$  contenant les processus à contacter
- Correction :  $\forall i \neq j : j \in D_i \vee i \in D_j$
- Deux types de permissions :
  - **individuelles** : un processus donne son autorisation selon son propre état  $\Rightarrow$  n'engage que lui
  - **d'arbitre** : les processus s'échangent des permissions préexistantes en nombre fixé  $\Rightarrow$  engage tous les processus qui dépendent de lui
- **Objectif** : Minimiser les ensembles  $D_i$

1. *An Optimal Algorithm for Mutual Exclusion in Computer Networks*, Glenn Ricart and Ashok K. Agrawala. Communications of the ACM, January 1981.



## Permissions individuelles : Ricart et Agrawala



### Hypothèses

- Chaque processus connaît l'identité des  $N$  processus
- Réseau de communication fiable et maillé (non FIFO)
- Pas de défaillance de processus

### Solution

- Utilisation de permissions individuelles avec :  
 $\forall i : D_i = \text{Tous} - \{i\}$
- Ordonnancement des requêtes par datation

### Messages

- Message de requête : le site  $i$  demande l'autorisation à  $j$
- Message d'autorisation : le site  $j$  donne son autorisation à  $i$
- Pas de message de refus : le refus est temporaire



## Algorithme de Ricart-Agrawala

```

Process P(i : 0..N-1) {
  type Etat = {hors,candidat,exclusion}; Etat EC ← hors;
  Date hloc ← new Date(0,i); // horloge locale
  Date dr; // date de la requête de ce site
  Set<int> Att ← ∅; // sites lui ayant demandé l'autorisation
  Set<int> D; // sites dont i attend l'autorisation
  on (EC = hors) : // hors → candidat
    EC ← candidat;
    D ← 0..N-1 \ {i};
    dr ← hloc.Top();
    for k ∈ D : send Request(i,dr) to Pk;
  on reception Request(p, d) :
    hloc.Recaler(d);
    if (EC ≠ hors ∧ dr < d) then Att ← Att ∪ {p};
    else send Perm(i) to Pp;
  on reception Perm(p) : // EC = candidat nécessairement
    D ← D \ {p};
    if (D = ∅) then EC ← exclusion; // candidat → excl
  on (EC = exclusion) : // exclusion → hors
    for k ∈ Att : send Perm(i) to Pk;
    Att ← ∅; EC ← hors;
}

```

## Algorithme de Ricart et Agrawala

### Principes

- Requêtes d'entrée totalement ordonnées :  
⇒ Utilisation d'horloges de Lamport
- Chaque processus  $P_i$  candidat ou en exclusion connaît la date de sa requête courante  $Date(R_i)$
- Un candidat entre en exclusion s'il a obtenu les permissions de tous les autres  
⇒ il possède alors la requête la plus ancienne
- Revient à vérifier que la requête  $R_i$  d'un processus  $P_i$  est la plus vieille requête des processus candidats ou en exclusion :  
 $\forall k : P_k.Etat \neq \text{hors} \Rightarrow Date(R_i) \leq Date(R_k)$



## Permissions d'arbitres



### Principe

Obtenir la permission de tous les arbitres contactés.

Condition nécessaire :  $\exists \text{ arbitre commun} : \forall i, j : D_i \cap D_j \neq \emptyset$

### Exemple

| $i$ prend pour arbitre : | 1 | 2 | 3 | 4 | 5 |
|--------------------------|---|---|---|---|---|
| 1                        |   | • | • |   |   |
| 2                        |   |   | • | • |   |
| 3                        |   | • |   | • |   |
| 4                        |   | • |   | • |   |
| 5                        |   | • | • |   |   |

Note : 1 et 5 ne sont pas arbitres



## Quorums

### Définition

Un système de quorum est un ensemble d'ensembles, tel que tout couple d'ensembles a une intersection non vide :

$$\mathcal{Q} = \{Q_1, \dots, Q_n\}, \forall i, j : Q_i \cap Q_j \neq \emptyset$$

Un quorum  $Q_i$  est responsable pour l'ensemble total.

Idéalement :

- Effort identique : Tous les quorums ont la même taille :  $\forall i : |Q_i| = K$
- Responsabilité identique : Tous les sites appartiennent au même nombre de quorums :  $\forall i : |\{j : i \in Q_j\}| = D$
- Minimalité :  $K = D =$  le plus petit possible (théorie :  $\lceil \sqrt{n} \rceil$ )

*MF*

## Permissions d'arbitres

ATTENTION : pas de miracle...

### Modèle parallèle processus $\leftrightarrow$ ressources critiques

- Un processus doit collecter des jetons  $\equiv$  ressources critiques
- Problème : **risque d'interblocage**
  - Solution préventive : ordonner les jetons et les demander en respectant cet ordre
  - Solution dynamique :
    - ordonner les requêtes (approche transactionnelle) et appliquer l'algorithme « wound-wait » ou « wait-die »
    - nécessite un ordre total sur les requêtes :  $\Rightarrow$  usage d'horloges de Lamport
    - Risque de livelock

*MF*

## Permissions d'arbitres

Construction facile d'un système de quorum quasi optimal :

- Construire une matrice arbitraire, éventuellement en dupliquant certains sites, p.e. pour 14 sites :

|    |    |    |    |
|----|----|----|----|
| 1  | 2  | 3  | 4  |
| 5  | 6  | 7  | 8  |
| 9  | 10 | 11 | 12 |
| 13 | 14 | 1  | 2  |

- Tout processus utilise les arbitres de sa colonne et de sa ligne  
 $D_6 = \{2, 5, 7, 8, 10, 14\}$
- Tout processus utilise au plus  $2\lceil \sqrt{n} \rceil - 1$  arbitres
- Tout arbitre appartient à au plus  $2\lceil \sqrt{n} \rceil - 1$  ensembles

1. A  $\sqrt{n}$  Algorithm for Mutual Exclusion in Decentralized Systems, Mamoru Maekawa. ACM Transactions on Computer Systems, May 1985.

*MF*

## Plan

- 1 Exclusion mutuelle
  - Le problème
  - Jeton circulant
  - Algorithme de Ricart-Agrawala
  - Algorithme à base d'arbitres
- 2 Détection de la terminaison
  - Le problème
  - Terminaison sur un anneau
  - Algorithme des quatre compteurs
  - Algorithme des crédits
- 3 Détection de l'interblocage
  - Le problème
  - Caractérisation de l'interblocage
  - Algorithme de Chandy, Misra, Haas
- 4 La diffusion fiable

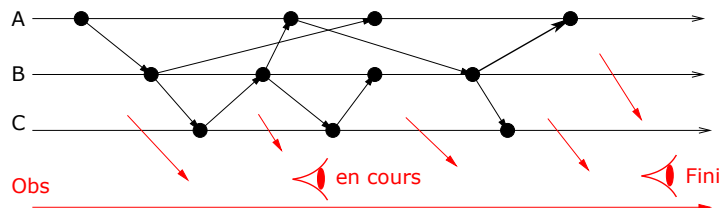
*MF*

## Terminaison : détecter une propriété stable



### Spécification

- Propriété **stable** à détecter :  
Tous les processus sont passifs **ET** pas de message en transit.
- Sûreté : Pas de **fausse détection** :  
 $Term \Rightarrow (\forall i :: P_i.passif \wedge EnTransit = \emptyset)$
- Vivacité : La terminaison **finit par** être détectée :  
 $(\forall i :: P_i.passif \wedge EnTransit = \emptyset) \leadsto Term$



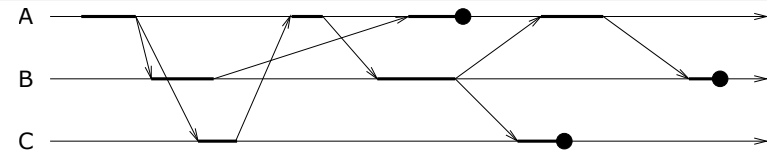
[ Précis 4.2 pp.66–69 ]

## Calcul diffusant



### Définition d'un calcul diffusant

- Un processus initial émet un ou plusieurs messages
- Puis, tous les processus adoptent le même comportement :  
**loop** { /\* un pas de calcul \*/  
recevoir( $m$ );  
traiter  $m$ ;  
envoyer 0 à  $N - 1$  messages;  
}



⇒ Les phases actives peuvent être vues comme atomiques

## Terminaison sur un anneau (Misra, 1983)



### Principe

Les sites sont organisés en anneau (communication FIFO depuis le précédent / vers le suivant mais à destinataire arbitraire).  
Parcourir l'anneau et vérifier que tous les sites sont passifs.

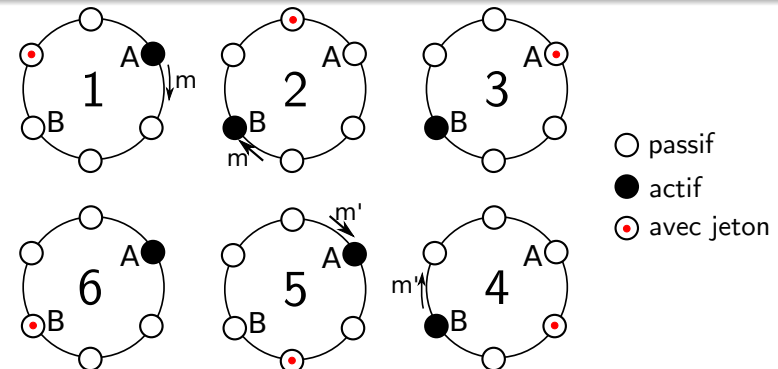
### Difficulté

Un message émis avant le passage du visiteur sur le site émetteur peut être reçu après le passage du visiteur sur le site récepteur et réactiver un site trouvé passif

⇒ faire **deux** tours en vérifiant qu'aucun site n'a changé d'état entre temps

1. *Detecting Termination of Distributed Computation Using Markers*, Jayadev Misra. 2nd ACM Symposium on Principles of Distributed Computing, 1983.

## Terminaison – Misra



- tous trouvés passifs, mais calcul pas terminé!
- sites A et B trouvés passifs, mais ils ont été actifs entre temps

⇒ faire **deux** tours en vérifiant qu'aucun site n'a changé d'état entre temps

```

Process P(i : 0..N-1) {
  variables : couleur ∈ {blanc,noir}
             état ∈ {actif,passif}
             jeton ∈ {true,false}
:
  on reception message_applicatif : // action
    couleur ← noir;
    état ← actif;
  on reception jeton(val) : // réception jeton
    jeton ← true; nb ← val;
    if (nb = N ∧ couleur = blanc) then TERMINAISON DÉTECTÉE
  on jeton ∧ état = passif : // envoi jeton
    if (couleur = blanc) then send jeton(nb + 1) to  $P_{i \oplus 1}$ 
    else send jeton(1) to  $P_{i \oplus 1}$ 
    couleur ← blanc;
    jeton ← false;
  else : // attente
    état ← passif;

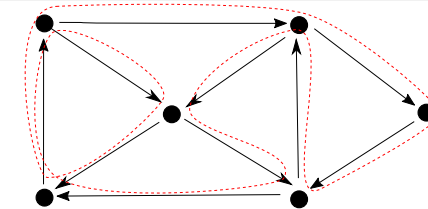
```

## Terminaison avec un anneau logique



## Graphe de communication arbitraire

- Circuit logique contenant **tous les arcs** (éventuellement plusieurs fois)
- Communication **FIFO** : le jeton ne peut pas dépasser un message antérieur sur le même arc
- Terminaison comme précédemment, avec  $N$  = longueur du circuit

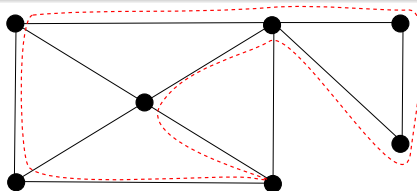


## Terminaison avec un anneau logique



## Graphe de communication arbitraire – avec la causalité

- Circuit logique contenant tous les sites (mais pas nécessairement tous les arcs)
- Communication **causale** : le jeton en transit depuis un site  $s$  ne peut pas arriver sur  $s'$  **avant** les messages émis avant sa visite sur  $s$  et envoyés directement de  $s$  vers  $s'$
- Terminaison comme précédemment, avec  $N$  = longueur du circuit

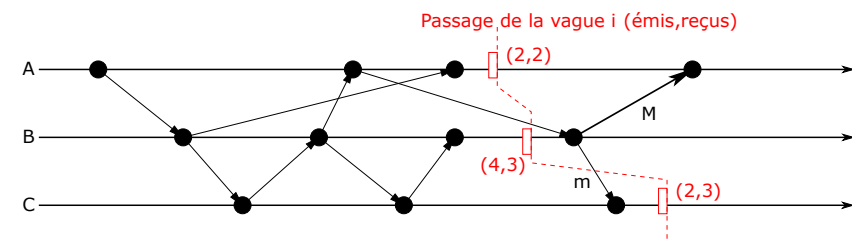


## Algorithme des quatre compteurs



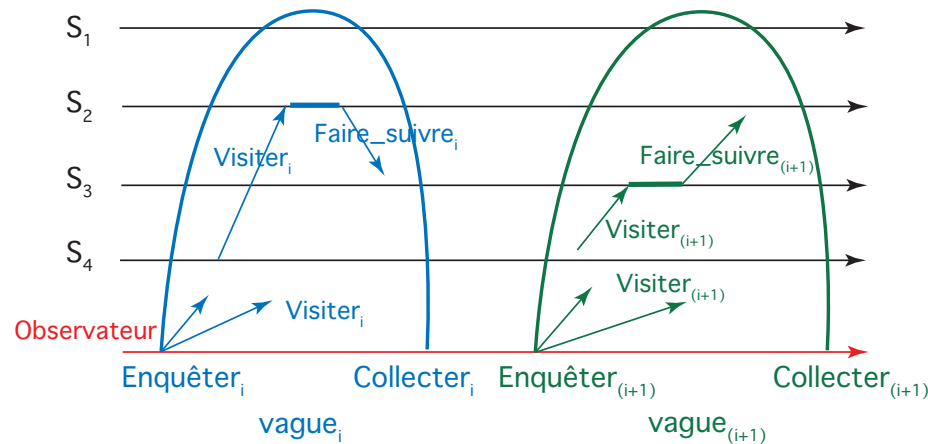
## Principe

- Terminaison  $\triangleq \text{Emis}(t) = \text{Reçus}(t)$  **mais** impossible à évaluer
- Approche : Compteurs **locaux** des messages émis et reçus
- Mécanisme de **vague** pour collecter les valeurs des compteurs
- La vague  $i$  collecte  $R_i \triangleq \sum r_i$  et  $E_i \triangleq \sum e_i$



1. *Algorithms for Distributed Termination Detection*, Friedemann Mattern. Distributed Computing, 1987.

## Vague : itération répartie

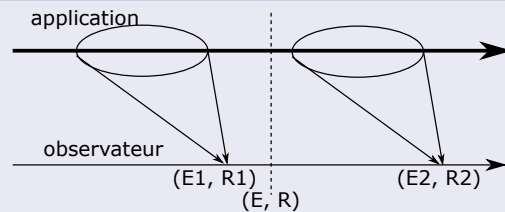


## Algorithme des quatre compteurs – preuve

**Vivacité** : si le calcul est terminé, alors la terminaison est détectée

Calcul terminé  $\Rightarrow$  il existe une date à partir de laquelle les compteurs de messages émis/reçus par site ne changent plus  $\Rightarrow$  deux vagues successives trouveront  $E_1 = R_1 = E_2 = R_2$ .

**Sûreté** : si la terminaison est annoncée, alors le calcul est terminé

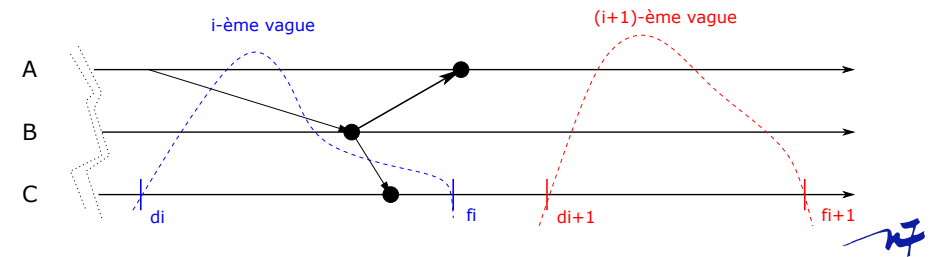


- |                           |                      |   |
|---------------------------|----------------------|---|
| (1) $E \geq R$            | calcul réparti       | (E, R) valeurs réelles<br>(et inconnues) des<br>compteurs entre deux<br>vagues successives. |
| (2) $E_1 \leq E \leq E_2$ | compteurs croissants |   |
| (3) $R_1 \leq R \leq R_2$ | idem                 |   |
| (4) $E_2 = R_1$           | détection annoncée   |   |
| (5) $E \leq R$            | d'après 2,3,4        |   |
| $E = R$                   | d'après 1,5          |   |

## Algorithme des quatre compteurs

### Détection de la terminaison

- Nécessite **deux** vagues successives
- Terminaison si :  $R_i = E_{i+1}$  (car  $\Rightarrow \exists t < d_{i+1} : E(t) = R(t)$ )
- Détection avec un retard d'au plus la durée de la dernière vague

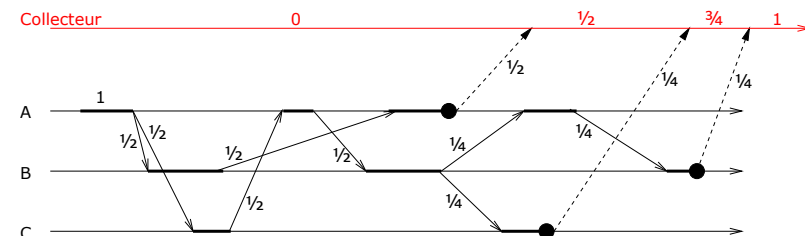


## Algorithme des crédits (Mattern)

Un peu oublié mais pourtant simple et performant. . .

### Principe

- Le processus initial possède un crédit de 1
- Le crédit courant est **partagé** entre les messages émis
- Un processus **rend** son crédit s'il n'envoie pas de message
- Terminaison lorsque la **somme** collectée égale 1



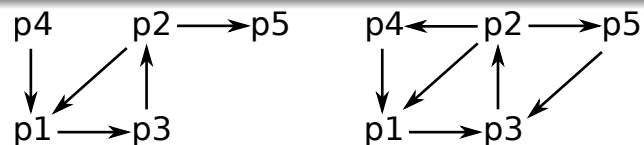
## Plan

- 1 Exclusion mutuelle
  - Le problème
  - Jeton circulant
  - Algorithme de Ricart-Agrawala
  - Algorithme à base d'arbitres
- 2 Détection de la terminaison
  - Le problème
  - Terminaison sur un anneau
  - Algorithme des quatre compteurs
  - Algorithme des crédits
- 3 Détection de l'interblocage
  - Le problème
  - Caractérisation de l'interblocage
  - Algorithme de Chandy, Misra, Haas
- 4 La diffusion fiable

## Graphe d'attente

### Graphe d'attente

Graphe dont les nœuds sont les processus, et un arc  $p_i \rightarrow p_j$  si  $p_i$  est bloqué / en attente de  $p_j$



### Modèles de communication

- Modèle ET : un processus est bloqué tant qu'il n'a pas reçu un message depuis **tous** ceux qu'il attend.
- Modèle OU : un processus est bloqué tant qu'il n'a pas reçu un message depuis **l'un** de ceux qu'il attend.

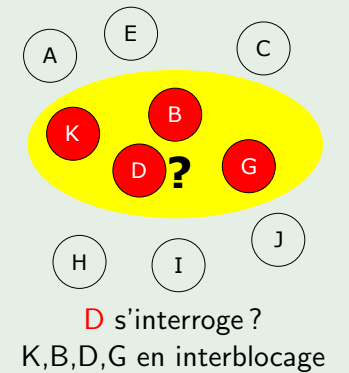
## Spécification du problème

### Détection d'une propriété stable

- Interblocage dû aux communications : attente de la réception d'un message
- Un processus est-il définitivement bloqué ?
- Sûreté : pas de fausse détection
- Vivacité : un processus bloqué finit par le savoir

Note : définition identique en centralisé, résolutions différentes (absence d'état global)

### Exemple



[ Précis 4.3 pp.72–75 ]

## Caractérisation de l'état d'interblocage

### Modèle ET

Existence d'un cycle dans le graphe d'attente

### Modèle OU

Existence d'une composante fortement connexe terminale dans le graphe d'attente

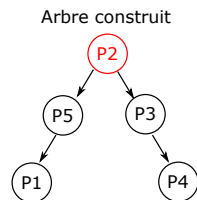
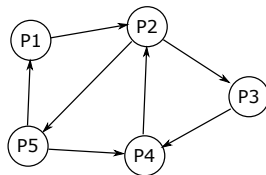
### Composante fortement connexe terminale (CFCT)

Un sous-graphe  $G'$  d'un graphe  $G = \{S, A\}$  est une CFCT (*knot*) ssi il existe un chemin entre tout couple de sommets de  $G'$  **et** si tout sommet de  $G'$  a ses successeurs dans  $G'$  :

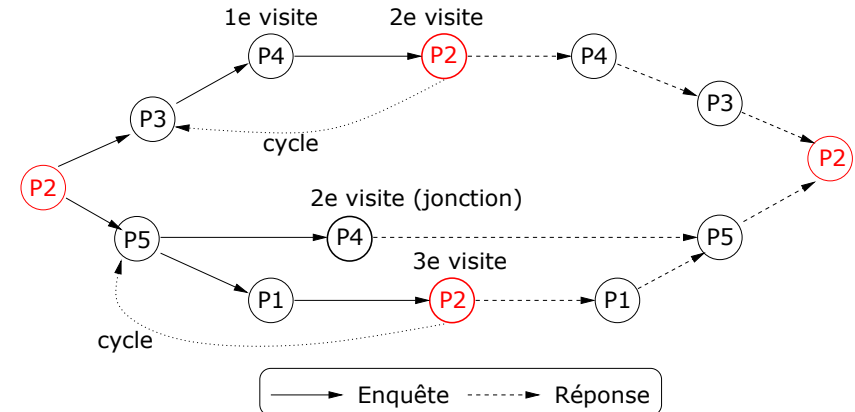
$$\forall s, s' \in G' : \exists s \xrightarrow{*} s' \wedge \forall s \in G' : \text{succ}(s) \neq \emptyset \wedge \text{succ}(s) \subset G'$$

## Algorithme : calcul diffusant et arbre de contrôle

- Phase 1 : construction d'un arbre de recouvrement des sites bloqués (message d'enquête)
- Phase 2 : un site répond lorsqu'il est bloqué et que tous ses successeurs ont répondu, ou qu'il a déjà été visité (dans ce cas, cycle ou jonction avec une enquête en cours)
- Si le site initiateur obtient une réponse de tous ses successeurs, il y a interblocage



1. *Distributed Deadlock Detection*, K. Mani Chandy, Jayadev Misra and Laura Haas. ACM Transactions on Computer Systems, May 1983.



### Propriétés

- Terminaison de la construction de l'arbre  $\Rightarrow P_2$  est interbloqué
- Pas de terminaison : Il faudra recommencer...

## Plan

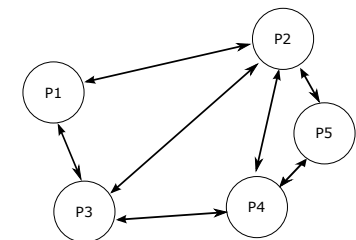
1. Exclusion mutuelle
  - Le problème
  - Jeton circulant
  - Algorithme de Ricart-Agrawala
  - Algorithme à base d'arbitres
2. Détection de la terminaison
  - Le problème
  - Terminaison sur un anneau
  - Algorithme des quatre compteurs
  - Algorithme des crédits
3. Détection de l'interblocage
  - Le problème
  - Caractérisation de l'interblocage
  - Algorithme de Chandy, Misra, Haas
4. La diffusion fiable

## La diffusion fiable

Envoyer un message à un ensemble de destinataires, tels que tous les processus corrects le délivrent, ou aucun.

### Hypothèses

- Réseau point-à-point fiable (tout message finit par arriver, intact, s'il existe un lien entre l'émetteur et le destinataire)
- Réseau connexe, pas nécessairement complet
- Défaillance d'arrêt** : un processus peut s'arrêter définitivement, à tout moment





## Réalisation par inondation

### Diffuser(m), sur p

```
-- p = émetteur, m = message
∀ s ∈ voisins(p) ∪ {p} faire
    envoyer(<p,m>) à s
fin pour
```

### Réception(<p,m>), sur q

```
si q n'a pas déjà délivré m alors
    si p ≠ q alors                -- propagation
        ∀ s ∈ voisins(q) faire
            envoyer(<p,m>) à s
        fin pour
    fin si
    délivrer(m)
fin si
```

## Conclusion

Quelques problèmes standards :

- Prise de cliché (chapitre II)
- Élection (chapitre II)
- Exclusion mutuelle
- Interblocage
- Terminaison d'un calcul réparti
- Diffusion fiable

## Propriétés

- Diffusion fiable *uniforme* : tous les processus (corrects ou ultérieurement défaillants) délivrent le message, ou aucun.
- Tout processus qui délivre un message l'a au préalable envoyé à ses voisins. Pour qu'un processus ne reçoive pas un message, il faudrait donc qu'aucun processus ne le lui ait envoyé, et donc aucun n'a pu le délivrer.
- Nombre de messages nécessaires = nombre de liens de communication (\*2)
- Le protocole tolère des arrêts de processus, tant que le graphe reste connexe.
- Si le graphe cesse d'être connexe  $\Rightarrow$  partitions. La propriété de fiabilité devient « tous les destinataires d'une même partition le délivrent, ou aucun ».



# Systèmes et algorithmes répartis

## Systèmes à grande échelle, pair à pair

Philippe Quéinnec

ENSEEIH  
Département Sciences du Numérique

13 septembre 2023



## Plan

- 1 Passage à grande échelle
- 2 Diffusion à grande échelle
  - Algorithmes structurés
  - Algorithmes probabilistes
- 3 Systèmes pair à pair
  - Principales difficultés
  - Classification
  - Systèmes non structurés
  - Systèmes structurés



## plan

- 1 Passage à grande échelle
- 2 Diffusion à grande échelle
  - Algorithmes structurés
  - Algorithmes probabilistes
- 3 Systèmes pair à pair
  - Principales difficultés
  - Classification
  - Systèmes non structurés
  - Systèmes structurés



## Passage à grande échelle

### Grande échelle

- Grand nombre de sites, d'objets. . .
- Grand nombre d'interactions
- Grande taille (géographique)

Grand = ? (ça dépend !)

### Scalability

La capacité de croissance (*scalability*) est la propriété pour un système de conserver ses qualités (performance, robustesse. . .) lorsque sa taille change d'échelle

taille 4 → 128, 32 → 1024, 1000 → 100 000



## Champs d'application

- Découverte, observation et accès à des ressources nombreuses
- Collecte de données (surveillance d'installations, capteurs)
- Détection de pannes
- Base de données à grande échelle (SIG – système d'information géographique)
- Absence d'infrastructure « officielle » : rôle symétrique des sites (tous client et serveur)

## Outil : les réseaux de recouvrement

### Réseau de recouvrement ou *overlay*

Réseau logique, virtuel, au-dessus d'un réseau physique existant

- Couche applicative :
  - Réimplantation du routage
  - Ajout de fonctionnalité : nommage, stockage
- Intérêt :
  - Indépendance par rapport au(x) réseau(x) physique(s) sous-jacent(s)
  - Souplesse et évolutivité (niveau applicatif)

## Fiasco pour le passage à grande échelle

- Algorithmes centralisés, point de contrôle unique  
⇒ véritables algorithmes répartis
- Algorithmes linéaires ( $O(n)$ ) en le nombre de sites, ou pires  
⇒  $O(\log n)$
- Hypothèse sur la structure statique du système  
⇒ ajout et retrait de sites, reconfiguration du réseau, partitionnement
- Considérer que la défaillance de site est un événement exceptionnel  
⇒ il existe des sites défaillants en permanence
- S'adresser à l'ensemble des sites (diffusion générale)  
⇒ propagation (par inondation, arborescente, probabiliste)

## Plan

- 1 Passage à grande échelle
- 2 Diffusion à grande échelle
  - Algorithmes structurés
  - Algorithmes probabilistes
- 3 Systèmes pair à pair
  - Principales difficultés
  - Classification
  - Systèmes non structurés
  - Systèmes structurés

## Diffusion à grande échelle

### Besoins

- Nombre de sites inconnu
- Nombre et identité des sites variables
- Grand nombre de sites

### Limites des approches classiques

- Ensemble bien identifié de sites (notion de groupe)
- Propriétés fortes (fiabilité, ordre, atomicité) néfastes au passage à l'échelle

*nf*

## Groupes et diffusion

### Notion de groupe

Cf chapitre « Tolérance aux fautes »

### Limites

- Vision synchrone des arrivées et départs
- Propriétés fortes (fiabilité, ordre), coûteuses et non indispensables
- Taille d'un groupe limitée

*nf*

## Diffusion par inondation

Cf chapitre IV « problèmes génériques »

### Diffuser(m), sur p

```
-- p = émetteur, m = message
∀ s ∈ voisins(p) ∪ {p} faire
    envoyer(<p,m>) à s
fin pour
```

### Réception(<p,m>), sur q

```
si q n'a pas déjà délivré m alors
    si p ≠ q alors                -- propagation
        ∀ s ∈ voisins(q) faire
            envoyer(<p,m>) à s
        fin pour
    fin si
    délivrer(m)
fin si
```

*nf*

## Arbre de recouvrement

- Construire un arbre issu du site de diffusion et contenant tous les sites
- Approximation : graphe orienté acyclique avec détection de messages en doublon
- Difficulté : construire l'arbre...
- ...mais c'est simple quand on a une table de routage hiérarchique, cf transparent 38

*nf*

## Algorithmes épidémiques

### Diffusion à grande échelle

- Grand nombre de sites ( $> 100$ ), voire très grand ( $> 10000$ )
- Nombre inconnu et variable de sites
- Rôle symétrique de tous les sites
- Présence de sites en panne
- Topologie d'interconnexion inconnue (a priori non directement maillée)

### Approche

- Algorithmes probabilistes
- Propagation aléatoire

1. *Epidemic Algorithms for Replicated Database Maintenance*, Alan Demers et al. 6th Symposium on Principles of Distributed Computing. Aug. 1987.

## Algorithme épidémique : Rumeur

### Rumeur (Gossip)

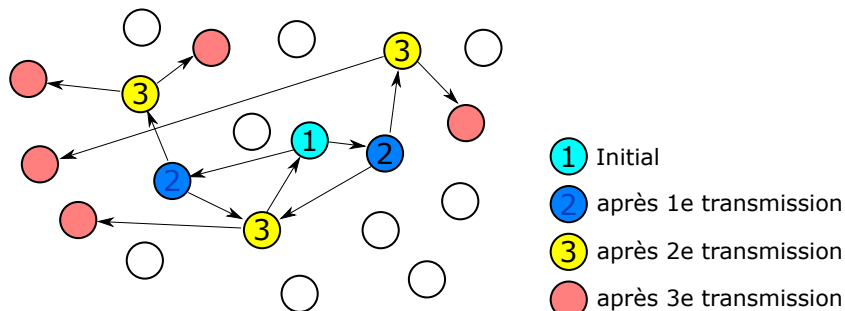
Contamination d'autres sites choisis aléatoirement avec une information supposée nouvelle.

États d'un site :

- Infectable : ne possède pas l'information
- Contagieux : apte à contaminer des sites infectables
- Immunisé : a cessé d'être contagieux
  - Immunisation en aveugle ou avec rétroaction (échec d'une tentative de contamination)
  - Compteur (de tentatives ou d'échecs)
  - Probabilité d'abandon après chaque tentative / échec

## Rumeur : exemple

20 sites, nombre de sites contactés à chaque tour = 2, immunisation en aveugle à 1 tour

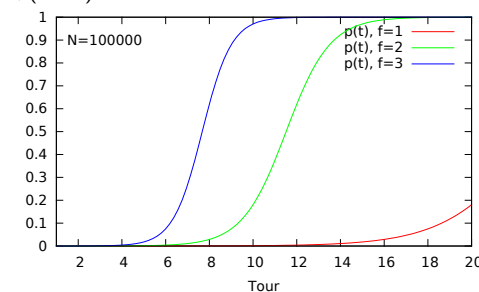


## Rumeur : performance

Modèle simple : *infect forever* = pas d'immunisation

- $N$  = nombre de sites
- $f$  = nombre de sites contactés à chaque tour par chaque site
- $I(t)$  = nombre de sites infectés (contagieux ou immunisés) après le  $t$ -ième tour
- $p(t) = I(t)/N$  = proportion de sites infectés au  $t$ -ième tour

$$\text{Alors } p(t) = \frac{1}{1 + (N-1) * e^{-f * t}}$$



## Rumeur : performance

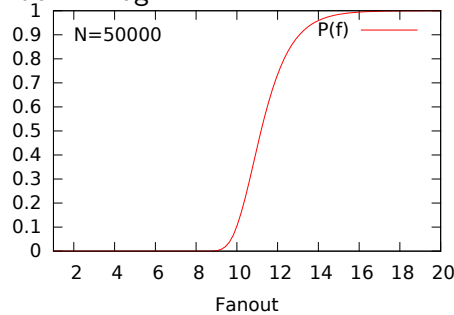


Modèle simple : *infect and die* = immunisation après un seul tour

- $N$  = nombre de sites
- $f$  = nombre de sites contactés par chaque site (*fanout*)
- $P$  = probabilité que tous les sites finissent par être infectés (à l' $\infty$ )

Alors  $P \approx e^{-e^{\log N - f}}$

Inversion autour de  $f = \log N$



*nf*

## Rumeur : coût

- Diffusion **probabiliste**
- Existence de sites non informés (poches d'ignorance si le choix des sites est plutôt local)
- Coût individuel très faible (très peu de sites à contacter en comparaison du grand nombre de sites présents)
- Coût global élevé : grand nombre de messages (échecs de contamination de plus en plus probables quand la diffusion progresse)

*nf*

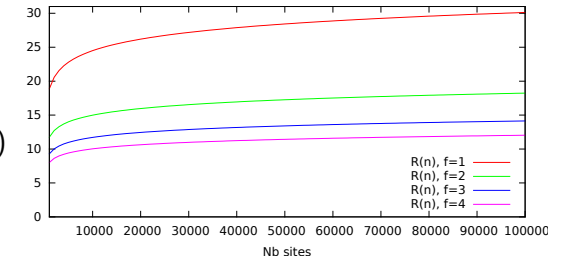
## Rumeur : performance



Nombre de tours pour tout contaminer :

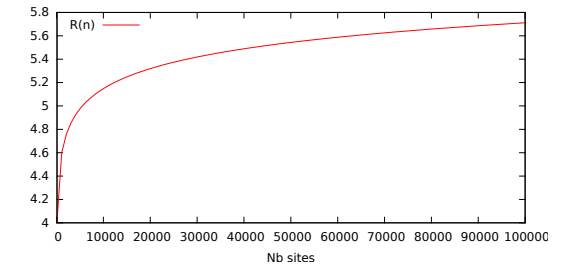
- Modèle sans immunisation (*infect forever*) :

$$R = \log_{f+1} n + \frac{1}{f} \log n + O(1)$$



- Modèle immunisation immédiate (*infect and die*) :  
(fin de l'épidémie avec tous contaminés)

$$R = \frac{\log n}{\log \log n} + O(1)$$



## Algorithme épidémique : Anti-entropie



### Anti-entropie

Périodiquement, chaque site contacte aléatoirement un autre site. Les deux sites échangent alors des informations, et leurs objets sont mis en cohérence.

L'algorithme **converge** vers l'égalité des copies : cohérence à terme.  
Nombre de tours pour tout contaminer =  $O(\log N)$

*nf*

## Graphes petit monde



Les algorithmes précédents n'ont pas de notion de voisinage : choix arbitraire d'un site quelconque (graphe complet).

### Graphe petit monde (*small-world graphs*)

Graphe connexe vérifiant :

- Grand nombre de nœuds ( $N > 10000$ )
- Faible connectivité des nœuds (de l'ordre de 5 à 10)
- Distance entre deux nœuds quelconques  $\approx \log N$
- Remarquablement adapté aux algorithmes épidémiques, résistant aux pannes (de sites et de liens)
- Apparaît spontanément (ex : réseau routier : maillage local, rocade, autoroutes)
- ...ou pas (ex : réseau aérien avec quelques gros hubs à forte connectivité)



## Problème à résoudre

- Stocker de l'information
- Trouver de l'information

⇒

- Utiliser l'ensemble des participants comme serveurs de stockage distribué
- Utiliser une partie des participants comme répertoire de nommage

Chaque nœud est à la fois client et serveur



## Plan

- 1 Passage à grande échelle
- 2 Diffusion à grande échelle
  - Algorithmes structurés
  - Algorithmes probabilistes
- 3 Systèmes pair à pair
  - Principales difficultés
  - Classification
  - Systèmes non structurés
  - Systèmes structurés



## Domaine d'application

### Partage

- informations/fichiers
- ressources de calcul (grid computing)
- ressources de stockage (réplication)
- bande passante (CDN *Content delivery network*)
- interactions (jeux massivement multijoueur)



## Réseaux de recouvrement

### Réseau de recouvrement ou *overlay*

Réseau logique, virtuel, au-dessus d'un réseau physique existant

- Couche applicative
  - Réimplantation du routage
  - Ajout de fonctionnalité : nommage, stockage
- Pairs : réseau formé par les participants, tous (à peu près) égaux

*nt*

## Difficultés (2)

### Équité

- Équilibrer la charge : égalitairement ? proportionnellement ?
- Utilisateurs égoïstes : contrôles et incitations à l'équité

### Défaillances

- Maintenance de l'overlay à tout prix
- Défaillances de pairs, de liens de communication
- Partition temporaire du réseau, réinsertion ?

*nt*

## Difficultés (1)

### Maintenance du réseau de recouvrement

- Démarrage
  - Création du réseau ? (premier site : cas particulier)
  - Insertion/retrait d'un pair dans le réseau
- Maintenance continue
  - Faute, retrait involontaire, expulsion
- Terminaison : arrêt du réseau ?

### Passage à l'échelle

- Éviter un (ou quelques) serveurs centralisés
- Distribuer la charge sur les pairs
- Borner la charge sur chaque pair (CPU, bande passante, stockage)

*nt*

## Difficultés (3)

### Adaptabilité

- Ajout/retrait de sites par vagues (heures ouvrables)
- Ajout/retrait d'informations parfois massif (plusieurs milliers d'un coup)

### Performance

- Efficacité : localisation, accès
- Latence du réseau (localité d'accès)
- Parallélisation

*nt*

## Classification : contrôle

- Contrôle centralisé : un serveur central met en correspondance les pairs
  - + simple
  - fragile
  - faible capacité de croissance
- Contrôle totalement décentralisé : tous les nœuds jouent un rôle symétrique (client et serveur)
  - + pas de point central, confidentialité
  - + disponibilité élevée
  - complexe, gestion hasardeuse
  - performance indéterminée
- Contrôle partiellement décentralisé : un ensemble dynamique de nœuds jouent un rôle privilégié



## Classification

|                            | Réseau virtuel             |                      |               |
|----------------------------|----------------------------|----------------------|---------------|
|                            | non structuré              | faiblement structuré | structuré     |
| contrôle centralisé        | Napster                    |                      |               |
| partiellement décentralisé | eMule, FastTrack, Gnutella |                      |               |
| totalement décentralisé    | BitTorrent                 | Freenet              | Chord, Pastry |

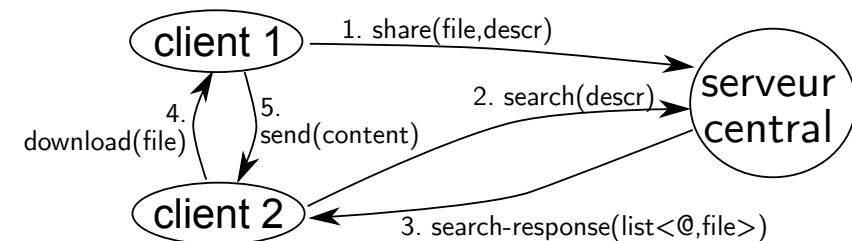


## Classification : réseau virtuel

- Non structuré : le placement des données n'est pas lié à la topologie du réseau
  - + bonne adaptabilité avec un ensemble de nœuds très dynamique
  - recherche inefficace en absence de contrôle centralisé
- Structuré : les données sont placées en des points prédéterminés  $\Rightarrow$  recherche déterministe
  - + recherche rapide, en temps borné
  - ensemble de nœuds dynamique ?
- Faiblement structuré : partiellement déterministe



## Contrôle centralisé – Napster simplifié



- Un serveur central : l'annuaire
- Des clients pairs : stockage





## Contrôle décentralisé – Gnutella

Chaque nœud est client, serveur et routeur

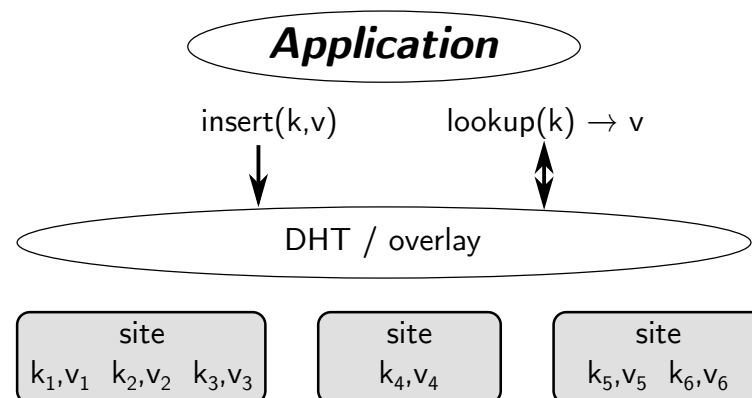
Messages

- ping : pour découvrir des correspondants  
bootstrap : *gnutella caches* notoires puis par rebond
- pong : en réponse (+ informations : nb/taille des fichiers possédés)
- query : recherche (mots clefs)
- query-hit : en réponse (@ IP, id fichiers)

1. *On the Long-term Evolution of the Two-Tier Gnutella Overlay*, Amir Rasti, Daniel Stutzbach and Reza Rejaie. 25th IEEE Int'l Conf. on Computer Communications. April 2006.

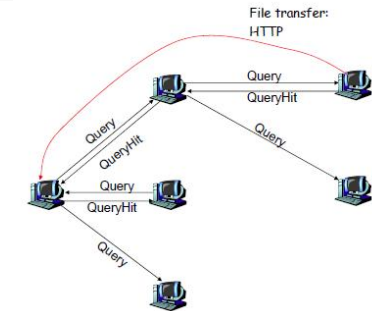
## Table de hachage répartie (DHT)

Table de hachage répartie = *distributed hash table*



## Gnutella : recherche

- Recherche par inondation : requête query transmise de voisin en voisin
- Id unique : éviter les retransmissions en boucle
- nombre de retransmissions (hops) limité



Améliorations :

- Envoi aléatoire, effectué en parallèle
- Distinction entre nœuds feuilles (connectés à 2 ou 3 ultranœuds) et ultranœuds (puissants, fortement interconnectés)  $\Rightarrow$  nombre réduit de hops

(source : Original uploader was ACNS at en.wikipedia – Commons CC BY-SA 3.0)

## Table de hachage répartie (DHT)

L'infrastructure P2P établit le lien entre clef et site :

- Chaque site possède une ID : p.e. hachage de son adresse IP
- Chaque objet possède une clef et une valeur
- La clef d'un objet est p.e. le hachage de sa valeur, ou de son nom, ou de sa description...
- **Chaque site est responsable d'une partie de l'espace de hachage**, p.e. des clefs qui sont proches de son ID

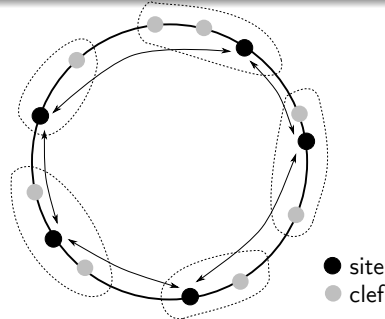
$\Rightarrow$

- Pas de connaissance globale centralisée
- Pas de point unique de défaillance
- Passage à l'échelle
- Répartition de la responsabilité

## Exemple de DHT : Pastry



- Les sites sont organisés en anneau virtuel (rangé par ID)
- Chaque site connaît le suivant et le précédent
- L'espace des clefs est partagé : le site le plus proche est responsable d'une clef



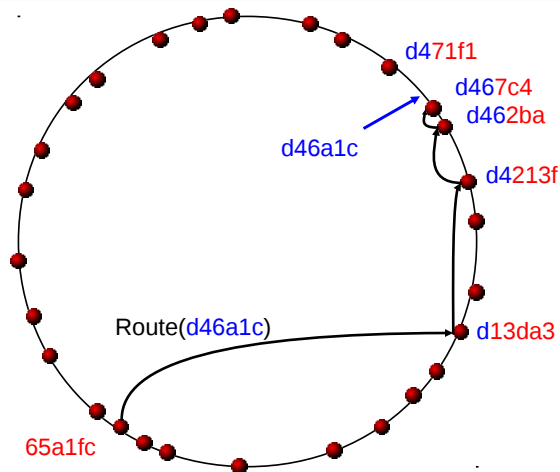
### Routeage : trouver le nœud responsable d'une clef

route(key, msg) : acheminer le message (spécifique à l'application) au site en charge de la clef

1. *Pastry : Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems*, Antony Rowstron and Peter Druschel. Int'l Conf. on Distributed Systems Platforms. Nov. 2001.



## Pastry : Routage



- max  $\log_{16} N$  hops
- table de routage (par site) :  $O(\log N)$



## Pastry : routage



Routage en suivant les liens suivants/précédents inefficace  $\Rightarrow$  table de routage

- $N$  sites ( $N = 16$  millions)
- Chaque site possède une ID :  $P$  digits sur  $B$  valeurs ( $B = 16$ ,  $P = \lceil \log_B N \rceil$ ). ex : 65a1fc
- Chaque site a une table de routage à  $P$  lignes,  $B$  colonnes
- La case  $(i, j)$  est l'adresse d'un site ayant les  $i$  premiers digits identiques au site, et  $j$  en  $i + 1$  digit.

ex : sur le site 65a1fc, ligne 3, col 4 = site 65a4xx

| L0  | 0    | 1    | 2    | 3    | 4    | 5    | 6    | 7    | 8    | 9    | a    | b    | c    | d    | e    | f    |
|-----|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| L1  | 60   | 61   | 62   | 63   | 64   | 65   | 66   | 67   | 68   | 69   | 6a   | 6b   | 6c   | 6d   | 6e   | 6f   |
| L2  | 650  | 651  | 652  | 653  | 654  | 655  | 656  | 657  | 658  | 659  | 65a  | 65b  | 65c  | 65d  | 65e  | 65f  |
| L3  | 65a0 | 65a1 | 65a2 | 65a3 | 65a4 | 65a5 | 65a6 | 65a7 | 65a8 | 65a9 | 65aa | 65ab | 65ac | 65ad | 65ae | 65af |
| ... |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |

- Trouver une clef = chercher le premier digit distinct et transmettre à ce site, qui poursuit le routage



## Pastry : voisinage (*Leaf sets*)

Chaque site maintient les  $L/2$  plus proches sites en deçà et en delà (où  $L$  est un paramètre, valant généralement 2 ou 4)

- efficacité du routage
- résistance du routage
- détection de fautes (ping périodique)



## Pastry : algorithme de routage

```

if (destination ∈ voisinage) then
    transmettre le message au membre concerné
else
    let l  $\triangleq$  longueur du préfix commun entre ce site
                                et la destination
    let d  $\triangleq$  valeur du l-ième digit
    if routage[l,d] est défini et répond then
        transmettre le message à routage[l,d]
    else
        transmettre à un site qui
            - a au moins un préfixe commun de taille l
            - est numériquement plus proche que ce site
    
```

*nt*

## Pastry : départ (défaillance)

Les membres d'un voisinage s'échangent périodiquement des messages de vie.

Absence de réponse sur un message de vie ou un message de routage  $\Rightarrow$  considéré défaillant, enlevé du voisinage et de la table de routage si présent.

- Réparation du voisinage : augmenter son voisinage en interrogeant le site le plus loin de son voisinage actuel
- Réparation du routage : obtenir la table des sites sur la même rangée que le site supprimé, puis en remontant

*nt*

## Pastry : jonction



- 1 Le site d'id  $i$  veut s'insérer
- 2 Il envoie un message *join* à n'importe quel site
- 3 Ce message est routé comme précédemment au site  $j$ , actuellement responsable de  $i$ . Le site  $i$  sera inséré entre  $j$  et  $k$  (précédent ou suivant de  $j$  selon la valeur de  $i$ )
- 4 Le site  $j$  transmet à  $i$  son voisinage pour que  $i$  construise son voisinage initial
- 5 La table de routage de  $i$  est établie à partir des tables de  $j$  et  $k$
- 6 Le site  $i$  interroge tous les sites de son voisinage initial :
  - $i$  établit son voisinage définitif en gardant les  $L/2$  plus proches dans chaque sens
  - Les sites du voisinage apprennent  $i$  et mettent à jour leurs propres table de routage et voisinage
- 7 Le site  $i$  devient actif

Difficulté : deux insertions simultanées entre deux même nœuds !

*nt*

## Pastry : bilan

### Points positifs

- Passe à l'échelle, résistant à  $L/2$  fautes simultanées (et en pratique bien plus), non centralisé (auto-organisation)
- Routage efficace :  $O(\log N)$  hops en situation normale,  $O(N)$  en cas pire (improbable : tables de routage détruites)
- Information par site modeste :  $O(\log N)$  pour le routage
- Partage équitable de la responsabilité et du stockage

### Améliorations

- Routage : prendre en compte la distance (en temps) pour établir la table de routage
- Réplication : un même objet clef/valeur est répliqué sur plusieurs nœuds voisins
- Sécurité : hachage non inversible avec peu de collision (SHA)

*nt*

## Conclusion

- Systèmes non structurés : performant si contrôle assez centralisé  $\Rightarrow$  capacité de croissance ? anonymat ?
- Systèmes structurés : dynamique ?
- Contrôle décentralisé  $\Rightarrow$  qualité de service ?
  - sécurité, confiance ?
  - disponibilité non garantie (mais plutôt bonne)
  - site parasite ?
- Aucun standard, même architecturalement



# Systèmes et algorithmes répartis

## Consensus, détecteur de défaillances

Philippe Quéinnec

ENSEEIH  
Département Sciences du Numérique

13 septembre 2023



## plan

- 1 Le consensus
  - Définition
  - Modèles, défaillances
  - Universalité, impossibilité
- 2 Système synchrone
  - Sans défaillance
  - Défaillance d'arrêt
  - Défaillance byzantine
- 3 Système asynchrone
  - Sans défaillance
  - Défaillance d'arrêt
  - Détecteur de défaillances



## Plan

- 1 Le consensus
  - Définition
  - Modèles, défaillances
  - Universalité, impossibilité
- 2 Système synchrone
  - Sans défaillance
  - Défaillance d'arrêt
  - Défaillance byzantine
- 3 Système asynchrone
  - Sans défaillance
  - Défaillance d'arrêt
  - Détecteur de défaillances



## Le consensus



### Définition

Soit un ensemble de processus  $p_1, \dots, p_n$  reliés par des canaux de communication.

Initialement : chaque processus  $p_i$  propose une valeur  $v_i$ .

À la terminaison de l'algorithme : chaque  $p_i$  décide d'une valeur  $d_i$ .

- **Accord** : la valeur décidée est **la même** pour tous les processus **corrects**
- **Intégrité** : tout processus décide **au plus une fois** (sa décision est définitive)
- **Validité** : la valeur décidée est **l'une des valeurs proposées**
- **Terminaison** : tout processus correct décide au bout d'**un temps fini**

1. *The Byzantine Generals Problem*, Leslie Lamport, Robert Shostak and Marshall Pease. ACM Trans. on Programming Languages and Systems. 1982.



## Remarques

### Correct / défaillant

Un processus est dit **correct** s'il n'est et ne sera jamais défaillant.  
Un processus incorrect peut fonctionner normalement avant de défaillir.

### Valeur proposée / décidée

- Les valeurs proposées ne sont pas nécessairement toutes distinctes.
- Le consensus binaire (uniquement 0/1 comme valeurs possibles) est équivalent au consensus à valeur quelconque.
- La valeur décidée n'est pas nécessairement une valeur majoritaire, ni celle d'un processus correct.

nf

## Variantes – 2



### Consensus uniforme

- Accord uniforme** : la valeur décidée est **la même** pour tous les processus (corrects ou ultérieurement défaillants) qui décident

Correct = n'aura jamais de défaillance. Un processus incorrect peut décider **puis** devenir défaillant.

### k-consensus

- k Accord** : **au plus k** valeurs distinctes sont décidées pour l'ensemble des processus **corrects**

Consensus basique :  $k = 1$

### Consensus approximatif

- $\epsilon$ -Accord** : les valeurs décidées par les processus **corrects** doivent être à distance maximale  $\epsilon$  l'une de l'autre.

nf

## Variantes – simultanéité

### Démarrage

Quand un processus démarre-t-il l'algorithme de consensus ?

- Démarrage simultané (à une heure donnée)
- Démarrage initié par l'un des processus** : diffusion d'un message d'initialisation de l'algorithme.  
(Attention aux propriétés de cette diffusion : fiable, ordonnée)
- Sur réception d'un 1<sup>er</sup> message de l'algorithme : ça complique !

Les trois formes sont équivalentes.

### Consensus simultané

- Terminaison simultanée** : tous les processus corrects décident *en même temps* (= au même tour, modèle synchrone).

nf

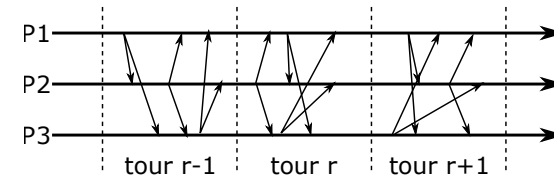
## Modèle temporel



### Synchrone

borne supérieure connue sur le temps de transmission et sur l'avancement des processus.

Usuellement, algorithmes fonctionnant par tours, synchronisés sur tous les processus.



### Asynchrone

Pas de borne connue : avancement arbitrairement lent des processus et du réseau.

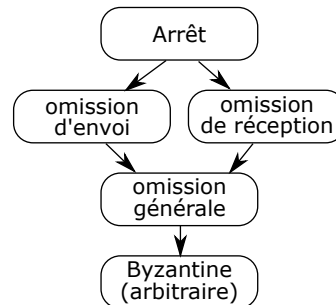
Modèle moins contraint, plus réaliste (mais plus difficile)

nf

## Défaillances d'un processus



- **Arrêt** (*crash failure* ou panne franche) : le processus fonctionne correctement jusqu'à un point où il cesse définitivement d'agir.
- **Omission**
  - omission en émission : le processus omet certaines émissions qu'il aurait dû faire, ou cesse définitivement.
  - omission en réception : le processus ignore certains messages en réception, ou cesse définitivement.
- **Arbitraire** (*byzantine failure*) : le processus ment (par omission ou par contenu arbitraire des messages envoyés)



1. *Fault-Tolerant Broadcasts and Related Problems*, Vassos Hadzilacos and Sam Toueg. In *Distributed Systems*. 1993.



## Communications

### Défaillance

- **réseau fiable** : tout message finit par arriver
- **perte** : certains messages n'arrivent jamais
- **ordre** : respect de l'ordre d'émission ou d'un autre ordre
- **arbitraire** : duplication, modification du contenu...

### Hypothèse de réseau fiable

Les défaillances réseau en asynchrone peuvent être modélisées par des défaillances de site  $\Rightarrow$  on suppose le réseau fiable.



## Utilité du consensus



Le consensus est un **outil générique pour la tolérance aux fautes** :

- Un système informatique est une machine à état :  
(nouvel état, sorties)  $\leftarrow$  fonction(état courant, entrée)
- Assurer la disponibilité = réplication en  $n$  copies
- Transparence = équivalence avec une seule copie
- Consensus pour ordonner identiquement les entrées  
+ si non déterministe, consensus pour décider identiquement des réponses sur les  $n$  copies

1. *The Implementation of Reliable Distributed Multiprocess Systems*, Leslie Lamport. Computer Networks. 1978.



## Universalité



### Spécification séquentielle

Un objet possède une spécification séquentielle si ses comportements corrects sont exprimables par des séquences (= des traces) de ses opérations.

### Universalité du consensus

Le consensus suffit pour implanter en réparti n'importe quel objet possédant une spécification séquentielle.

- En communication par message : consensus + diffusion générale (anonyme)

1. *Wait-Free Synchronization*, Maurice Herlihy. ACM Trans. on Programming Languages and Systems. 1991.



## Problèmes réalisables avec le consensus



- Élection d'un leader = accord de tous sur un processus
- Diffusion fiable avec terminaison = tous les processus corrects délivrent un même message (éventuellement vide si l'émetteur s'est arrêté)
- Diffusion uniforme = tout les processus (corrects ou pas) délivrent ou aucun
- Construction de groupes
- Commit (validation) de transaction distribuée
- Calcul d'une fonction globale portant sur l'ensemble des sites

## Impossibilité du consensus en synchrone avec perte de message



### Résultat d'impossibilité (Gray, 1978)

Le consensus est impossible à réaliser dans un système synchrone où les messages peuvent être arbitrairement perdus.

Intuition de la preuve : le dernier message avant décision peut être perdu sans changer la décision  $\Rightarrow$  il était inutile. On le supprime, et on recommence le raisonnement.

(Piège : la perte de message peut être modélisée par une défaillance d'omission de *n'importe quel* processus. Le consensus est faisable en synchrone avec omission s'il y a  $< n/2$  sites défaillants.)

1. *Notes on Data Base Operating Systems*, Jim Gray. Operating Systems, An Advanced Course, 1978.

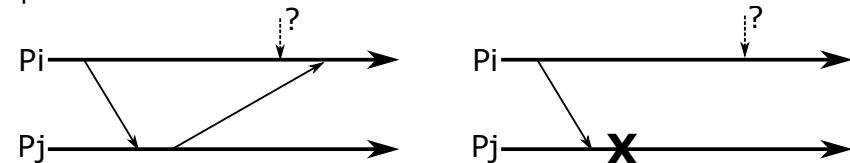
## Impossibilité du consensus en asynchrone avec arrêt



### Résultat d'impossibilité (FLP85)

Le consensus est impossible à réaliser dans un système asynchrone où un seul processus peut subir une défaillance d'arrêt.

Intuitivement, il est impossible de distinguer un processus lent d'un processus arrêté.



1. *Impossibility of Distributed Consensus with One Faulty Process*, Michael Fischer, Nancy Lynch and Michael Paterson. Journal of the ACM. April 1985.

## Résultats d'impossibilité de réalisation du consensus

### Processus communiquant par mémoire partagée

| Défaillance / modèle | Arrêt de processus |
|----------------------|--------------------|
| synchrone            | $\exists$ solution |
| asynchrone           | <b>impossible</b>  |

### Processus communiquant par messages

| Défaillance / modèle | Arrêt de processus | Omission           | Byzantine          | Perte de message  |
|----------------------|--------------------|--------------------|--------------------|-------------------|
| synchrone            | $\exists$ solution | $\exists$ solution | $\exists$ solution | <b>impossible</b> |
| asynchrone           | <b>impossible</b>  | <b>impossible</b>  | <b>impossible</b>  | <b>impossible</b> |





## Contourner FLP



### Affaiblir le problème

- Terminaison probabiliste
- $k$ -consensus
- Consensus approximatif ( $\epsilon$ -consensus)
- *Best effort*

### Renforcer le système

- Système partiellement synchrone
- Système synchrone *suffisamment longtemps*
- **Détecteur de défaillances**



## Plan

- 1 Le consensus
  - Définition
  - Modèles, défaillances
  - Universalité, impossibilité
- 2 **Système synchrone**
  - Sans défaillance
  - Défaillance d'arrêt
  - Défaillance byzantine
- 3 **Système asynchrone**
  - Sans défaillance
  - Défaillance d'arrêt
  - Détecteur de défaillances



## Réalisabilité du consensus

| défaillance | synchrone   | asynchrone |
|-------------|---|------------|
| non         | faisable  | faisable   |
| arrêt       | faisable<br>$f$ défaillances $< n$ processus<br>$\Omega(f + 1)$ tours             | impossible |
| omission    | faisable<br>$f$ défaillances $< n/2$ processus                                    | impossible |
| byzantine   | faisable<br>$f \leq \lfloor (n - 1)/3 \rfloor$ processus<br>$\Omega(f + 1)$ tours | impossible |

- $k$ -consensus : faisable en asynchrone/arrêt avec  $f < k < n$
- Consensus approximatif : faisable en asynchrone/arrêt avec  $5f + 1 \leq n$



## Réalisation en absence de défaillance



### Principe

Tous les processus diffusent leur valeur, chacun garde la plus petite reçue.

Synchrone  $\Rightarrow$  borne supérieure de communication  $\Delta \Rightarrow$  décision en **un tour** et  **$n$  diffusions**.

Processus  $P_i(v_i)$ ,  $0 \leq i < n$

local  $V_i$

on round 0 :

$V_i \leftarrow \{v_i\}$

envoyer( $v_i$ ) à tous les autres

on réception( $v$ ) :

$V_i \leftarrow V_i \cup \{v\}$

on round 1 :

décider  $\min(V_i)$  (toute fonction déterministe)



## Réalisation en défaillance d'arrêt



Tolérance à  $f$  défaillances ( $f < n$ ).

### Principe

Au  $i$ -ième tour, le processus  $i$  diffuse sa valeur. Après  $f + 1$  tours, on est sûr qu'**au moins un** processus correct a diffusé une valeur reçue par au moins  $n - f$  processus corrects.

```
Processus  $P_i(v_i)$ ,  $0 \leq i < n$ 
local x
on start :
   $x \leftarrow v_i$ 
on réception(v) :
   $x \leftarrow v$ 
on round  $i$ ,  $0 \leq i \leq f$ : // au  $i$ -ième tour pour  $P_i$ 
  envoyer( $x$ ) à tous les autres
on round  $(f + 1)$ :
  décider x
```

*nt*

## Algorithme équitable à $f + 1$ tours

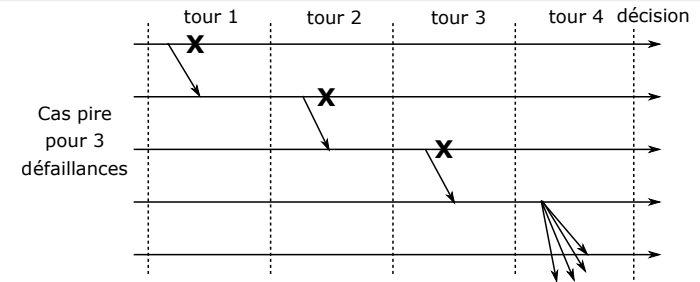
### Principe

À chaque tour, chaque processus diffuse la plus petite valeur qu'il connaît (uniquement si elle a changé).

```
Processus  $P_i(v_i)$ ,  $0 \leq i < n$ 
local x, prevx, received
on start :
   $x \leftarrow v_i$ ,  $prevx \leftarrow \perp$ 
on réception(v) :
   $received \leftarrow received \cup \{v\}$ 
on round  $k$ ,  $0 \leq k \leq f$ : // à chaque tour
   $prevx \leftarrow x$ 
   $x \leftarrow \min(received \cup \{x\})$ 
   $received \leftarrow \emptyset$ 
  si  $x \neq prevx$  alors diffuser( $x$ )
on round  $(f + 1)$ :
  décider x
```

*nt*

## Réalisation en défaillance d'arrêt



- $f + 1$  tours,  $f + 1$  diffusions
- décision simultanée
- non équitable : les valeurs des processus  $f + 1, \dots, n$  ne sont pas considérées  $\Rightarrow$  ajouter un tour préliminaire :  
diffuser( $v_i$ )  
 $x \leftarrow$  l'une des valeurs reçues dans ce tour

*nt*

## Nombre optimal de tours

### Borne inférieure

Il n'existe pas d'algorithme synchrone basé sur des tours qui résolve le consensus avec  $f$  défaillances d'arrêt en moins de  $f + 1$  tours.

Le pire cas est qu'un seul processus défaille à chaque tour.

### Existence

La borne " $f + 1$  tours" est atteignable.

1. A Lower Bound for the Time to Assure Interactive Consistency, Michael Fischer and Nancy Lynch. Information Processing Letters. 1982.

*nt*

## Le problème des généraux byzantins

### Les généraux byzantins

Des divisions de l'armée Byzantine assiègent une cité et doivent décider d'attaquer ou pas. Chaque division est commandée par un général qui communique avec les autres par des messagers fiables.

Chaque général doit éventuellement décider d'un plan d'action (**terminaison**); le plan doit être le même pour tous (**accord**); un général ne doit pas changer de décision une fois prise (**intégrité**); la décision retenue doit avoir été proposée et en particulier s'ils sont unanimes, ceci doit être la décision finale (**validité**).

Certains généraux sont des traîtres qui veulent empêcher les généraux loyaux de conclure. Pour cela, ils peuvent envoyer des messages contradictoires aux autres généraux ou mentir sur ce qu'ils ont reçu des autres généraux. Les traîtres peuvent même se coaliser pour conspirer de manière coordonnée.

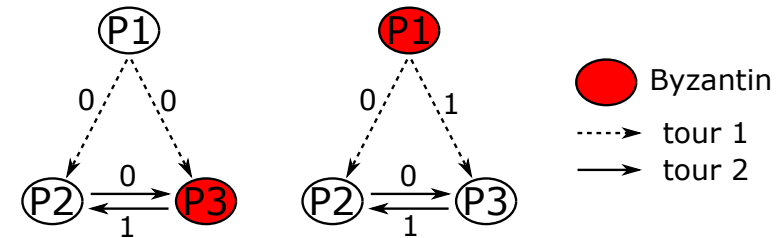
1. *The Byzantine Generals Problem*, Leslie Lamport, Robert Shostak and Marshall Pease. ACM Trans. on Programming Languages and Systems. 1982.

## Algorithme avec défaillances byzantines

Le consensus en synchrone avec défaillance byzantine est impossible pour  $n \leq 3f$ .

Le consensus en synchrone avec défaillance byzantine est possible si  $f \leq \lfloor \frac{n-1}{3} \rfloor$ .

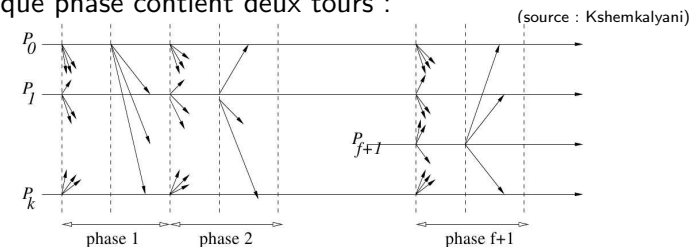
## Impossibilité du consensus à 3 avec une défaillance byzantine



- Défaillance byzantine = le processus peut mentir
- $P_2$  ne peut pas distinguer entre les deux scénarios
- Des échanges supplémentaires ne changent rien
- $P_2$  ne peut pas nécessairement décider identiquement à l'autre processus correct

## Algorithme avec "roi de phase" (Phase King)

- $f + 1$  phases, chaque phase a un unique roi, fixé a priori
- Chaque phase contient deux tours :



- Tour 1 : chaque site diffuse son estimation à tous. Chaque site reçoit les valeurs proposées puis détermine si une valeur est proposée par une majorité qualifiée ( $> n/2 + f$ ), ou une majorité simple ( $> n/2$ )
- Tour 2 : le roi fixe son estimation à sa valeur majoritaire (sa propre valeur si pas de majorité) et la diffuse. Chaque site fixe sa nouvelle estimation à la valeur reçue en tour 1 avec majorité qualifiée, ou sinon à la valeur reçue du roi.

## Algorithme avec "roi de phase" – justification

$f + 1$  phases,  $(f + 1)(n + 1)(n - 1)$  messages,

$f < \lceil n/4 \rceil$  défaillances

- Parmi les  $f + 1$  phases, au moins une où le roi est correct
- Dans cette phase, les processus corrects obtiennent la même estimation que le roi : soit  $p_i$  et  $p_j$  corrects :
  - ou  $p_i$  et  $p_j$  ont chacun une majorité qualifiée ( $> n/2 + f$ ) et le roi a alors aussi cette même estimation
  - ou  $p_i$  a une majorité qualifiée ( $> n/2 + f$ ) et  $p_j$  utilise la valeur du roi ( $> n/2$ )
  - ou  $p_i$  et  $p_j$  utilisent l'estimation du roi
- Si tous les processus corrects ont la même estimation au début d'une phase, ils garderont cette même valeur à la fin (même si le roi est byzantin).

*nf*

## Consensus en asynchrone

### Résultat d'impossibilité (FLP85)

Le consensus est impossible à réaliser dans un système asynchrone où un seul processus peut subir une défaillance d'arrêt.

→

- Refuser (ou ignorer) les défaillances : *best effort*
- Affaiblir le problème :  $k$ -consensus
- Affaiblir le problème : terminaison probabiliste
- Renforcer le système : détecteur de défaillance

*nf*

## Plan

- 1 Le consensus
  - Définition
  - Modèles, défaillances
  - Universalité, impossibilité
- 2 Système synchrone
  - Sans défaillance
  - Défaillance d'arrêt
  - Défaillance byzantine
- 3 Système asynchrone
  - Sans défaillance
  - Défaillance d'arrêt
  - Détecteur de défaillances

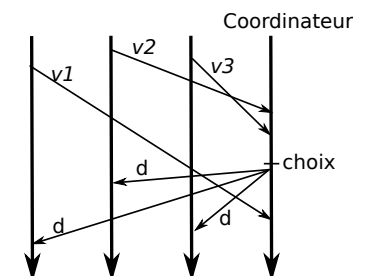
*nf*

## Consensus avec coordinateur

*nf*

Hypothèse : système asynchrone fiable, pas de défaillance

- Chaque processus envoie sa valeur à un coordinateur désigné à l'avance
- Au bout d'un certain temps (après avoir reçu au moins une valeur), le coordinateur choisit une valeur  $d$
- Le coordinateur envoie  $d$  à **tous** les processus (diffusion fiable)
- Chaque processus décide  $d$
- $\Rightarrow$  tous les processus décident identiquement en temps fini **non borné**

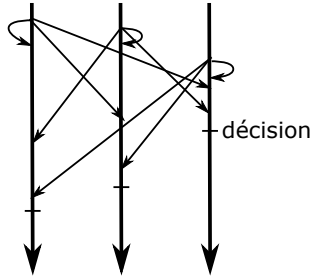


*nf*

## Consensus symétrique



- Chaque processus diffuse sa valeur à tous
- Quand un processus a reçu **toutes** les valeurs, il applique un algorithme déterministe (p.e. min) pour choisir la valeur de décision
- Tous les processus utilisent le même algorithme
- $\Rightarrow$  tous les processus décident identiquement en temps fini **non borné**



*nf*

## Algorithme à terminaison probabiliste



Algorithme à terminaison probabiliste, assurant la sûreté (accord, intégrité, validité) si  $f < \frac{n}{2}$  ( $f$  = nb de fautes,  $n$  = nb de sites).

- Tours asynchrones
- A chaque tour, le processus :
  - Rapporte sa valeur courante en la diffusant à tous
  - Attend  $n - f$  rapports
  - Si une valeur obtient une majorité absolue ( $> \frac{n}{2}$ ) alors propose cette valeur en la diffusant à tous sinon propose "?"
  - Attend  $n - f$  propositions
  - Si une valeur est reçue au moins  $f + 1$  alors décide cette valeur et termine
  - Si une proposition non "?" a été reçue alors prendre cette valeur sinon prendre **au hasard** 0 ou 1

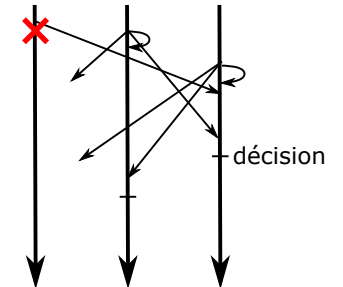
1. Another advantage of free choice : Completely asynchronous agreement protocols, Michael Ben-Or, Principles of Distributed Computing, 1983

*nf*

## k-consensus avec arrêt de processus



- Chaque processus envoie sa valeur à tous les autres
- Quand un processus a reçu (au moins)  $n - (k - 1)$  valeurs, il décide le min.



Si  $f < k < n$  alors :

- Terminaison en temps fini (non borné) pour les processus corrects
- Au plus  $f$  valeurs non reçues par tous  $\Rightarrow$  au plus  $f + 1$  décisions distinctes

*nf*

## Algorithme de Ben-Or

```

Processus  $P_i(v_i)$ ,  $0 \leq i < n$ 
 $x_i \leftarrow v_i$  // estimation courante de  $p_i$ , ( $v_i \in \{0,1\}$ )
 $k_i \leftarrow 0$  // k est le n° de tour
boucle
     $k_i \leftarrow k_i + 1$ 
    envoyer Report( $k_i, x_i$ ) à tous
    attendre  $n - f$  messages Report( $k_i, *$ ) // "*"  $\in \{0,1\}$ 
    si reçu plus de  $n/2$  Report( $k_i, v$ ) avec le même v
        envoyer Proposal( $k_i, v$ ) à tous
    sinon
        envoyer Proposal( $k_i, ?$ ) à tous
    attendre  $n - f$  messages Proposal( $k_i, *$ ) // "*"  $\in \{0,1,?\}$ 
    si reçu au moins  $f + 1$  Proposal( $k_i, v$ ) avec le même v alors
        décider v et terminer
    si reçu au moins un Proposal( $k_i, v$ ) alors  $x_i \leftarrow v$ 
    sinon  $x_i \leftarrow 0$  ou 1 aléatoirement
finboucle
    
```

*nf*

## Algorithme de Ben-Or : preuve

Deux processus  $p_i$  et  $p_j$  ne peuvent pas proposer deux valeurs distinctes dans le même tour.

Si  $p_i$  propose 0, il a reçu  $> \frac{n}{2}$  rapports égaux à 0. Donc  $p_j$  n'a pas pu recevoir  $> \frac{n}{2}$  rapports égaux à 1 (et inversement pour 1/0).

Si un processus  $p_i$  décide  $v$  à un tour  $k$ , tous les processus  $p_j$  démarreront le tour  $k + 1$  avec  $x_j = v$ .

Si  $p_i$  décide  $v$ , c'est qu'il a reçu  $f + 1$  propositions pour  $v$ . Au même tour,  $p_j$  a reçu  $n - f$  propositions, donc au moins une était  $v$  (intersection non vide des propositions reçues). D'après le résultat précédent, il n'a pas pu recevoir d'autre proposition non-?  
→ il fixe son  $x_j$  à  $v$ .

*nf*

## Algorithme de Ben-Or : sûreté

### Accord

Tous les processus (qui décident) décident de la même valeur.

Découle directement des lemmes précédents.

### Validité

La valeur décidée est l'une des valeurs proposées.

Supposons que  $p$  décide  $v$  qui n'était pas proposé initialement. Donc tous les processus avaient  $1 - v$  initialement. Donc tous avaient la même valeur  $1 - v$ , et tous, y compris  $p$ , décident de cette valeur (lemme précédent). Contradiction.

### Intégrité

Un processus ne décide qu'une seule fois.

Le processus termine après avoir décidé.

*nf*

## Algorithme de Ben-Or : preuve (2)

Si tous les processus (non crashés) ont la même valeur au début d'un tour, tous décident cette valeur dans ce tour.

Si tous les processus non crashés ont la même valeur  $v$ , chaque processus rapporte  $v$  aux autres. Comme  $n - f > n/2$ , chacun propose  $v$ . Comme  $n - f \geq f + 1$ , chacun décide  $v$ .

*nf*

## Algorithme de Ben-Or : vivacité

### Terminaison

Tout processus correct finit par décider avec probabilité 1.

- ① Nous disons qu'une valeur  $v$  est  $k$ -fixée si tous les processus non crashés possède  $v$  au tour  $k$  (et donc vont décider  $v$  au plus au tour  $k + 1$ )
- ② À un tour quelconque, une valeur  $v$  a une probabilité  $\geq (\frac{1}{2})^n$  d'être fixée (un processus ne fixe pas nécessairement  $x_i$  par tirage aléatoire, d'où le  $\geq$ ).
- ③ Au tour  $k$ ,  $Prob[\text{aucune valeur n'est } k\text{-fixée}] < 1 - (\frac{1}{2})^n$
- ④  $Prob[\text{aucune valeur n'est fixée pour les } k \text{ premiers rounds}] < (1 - (\frac{1}{2})^n)^k$
- ⑤  $Prob[\text{une valeur est } k\text{-fixée pendant les } k \text{ premiers rounds}] \geq 1 - (1 - (\frac{1}{2})^n)^k$
- ⑥ Converge vers 1 quand  $k$  croît.

*nf*

## Décteur de défaillances : Motivation



### Résultat d'impossibilité (FLP85)

Le consensus est impossible à réaliser dans un système asynchrone où un seul processus peut subir une défaillance d'arrêt.

- Le consensus en asynchrone est impossible car on ne sait pas distinguer un processus défaillant (arrêté) d'un processus lent.
- On va **supposer** l'existence d'un détecteur de défaillances, qui indique si un processus est fautif ou non.
- FLP  $\Rightarrow$  un détecteur parfait est impossible.
- Il existe des détecteurs **imparfaits** (i.e. qui peuvent se tromper) qui suffisent pour réaliser le consensus !
- FLP  $\Rightarrow$  ces détecteurs imparfaits sont impossibles mais on peut en construire des **approximations réalistes**.



## Complétude



### Complétude (*completeness*)

- **Complétude forte** : tout processus défaillant finit par être suspecté par **tout** processus correct
- **Complétude faible** : tout processus défaillant finit par être suspecté par **un** processus correct

Complétude faible et complétude forte sont équivalentes :

- En complétude faible, tout processus défaillant finit par être détecté par au moins un processus
- Périodiquement, chaque processus diffuse sa liste de processus suspectés
- Alors tous les processus finiront par obtenir l'information de suspicion = complétude forte
- (hypothèse : la diffusion est fiable, ce qui est réalisable en asynchrone avec défaillance d'arrêt)



## Décteur de défaillances



### Définition

- Un détecteur de défaillances est un service réparti composé de détecteurs locaux à chaque processus (site).
- Un détecteur fournit à son processus local une liste des processus qu'il **suspecte** d'être défaillants.
- Les détecteurs locaux coopèrent (ou pas) pour établir cette liste.
- Propriétés :
  - Complétude : peut-on ne pas suspecter un processus défaillant ?
  - Exactitude : peut-on suspecter un processus correct ?
- Équivalent à un **oracle** (éventuellement imparfait)

1. *Unreliable Failure Detectors for Reliable Distributed Systems*, Tushar Chandra and Sam Toueg. Journal of the ACM. March 1996.



## Exactitude permanente



Peut-on suspecter à tort un processus ?

### Exactitude permanente (*accuracy*)

- **Exactitude forte** : **aucun** processus n'est suspecté avant qu'il ne devienne effectivement défaillant.
- **Exactitude faible** : il existe **un** processus correct qui n'est jamais suspecté par aucun autre processus.

Exactitude forte : en particulier, un processus correct ne peut jamais être suspecté par aucun autre processus.





## Exactitude inévitable



Exactitude après une période initiale de chaos

### Exactitude inévitable

- Exactitude finalement forte : au bout d'un certain temps, aucun processus n'est suspecté avant qu'il ne devienne défaillant.
- Exactitude finalement faible : au bout d'un certain temps, il existe un processus correct qui n'est plus jamais suspecté par aucun autre processus

Exactitude finalement forte : équivalent à : au bout d'un certain temps, aucun processus correct n'est suspecté par aucun autre processus correct

(Finalement = inévitablement = *eventually*)



## Consensus avec détecteur parfait $P$

### Données

- $f$  = nombre de défaillances d'arrêt à tolérer
- pour chaque processus  $p_i$ , un vecteur  $V_i$  contenant les valeurs proposées par les autres processus et connues de  $p_i$
- $V_i[j]$  = valeur proposée par  $p_j$  telle que connue par  $p_i$
- Initialement  $V_i[i] = v_i$  (valeur proposée par  $p_i$ ) et  $V_i[j] = \perp (j \neq i)$

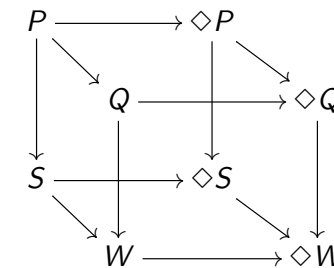


## Classes de détecteurs de défaillances



|                   | Exactitude |        |                  |                   |
|-------------------|------------|--------|------------------|-------------------|
|                   | forte      | faible | finalement forte | finalement faible |
| Complétude forte  | P          | S      | $\diamond P$     | $\diamond S$      |
| Complétude faible | Q          | W      | $\diamond Q$     | $\diamond W$      |

P = perfect, S = strong, W = weak



## Consensus avec détecteur parfait : principe



- $f + 1$  tours asynchrones : à chaque tour :
  - chaque processus envoie aux autres son vecteur  $V_i$
  - chaque processus met à jour son vecteur  $V_i$  avec les nouvelles valeurs apprises (celles telles que  $V_i[j] = \perp$ )
  - quand un processus a reçu un message de tous les processus non suspectés par son détecteur  $P$ , il passe au tour suivant.
- Après  $f + 1$  tours, le processus décide la première valeur non  $\perp$  de son vecteur.

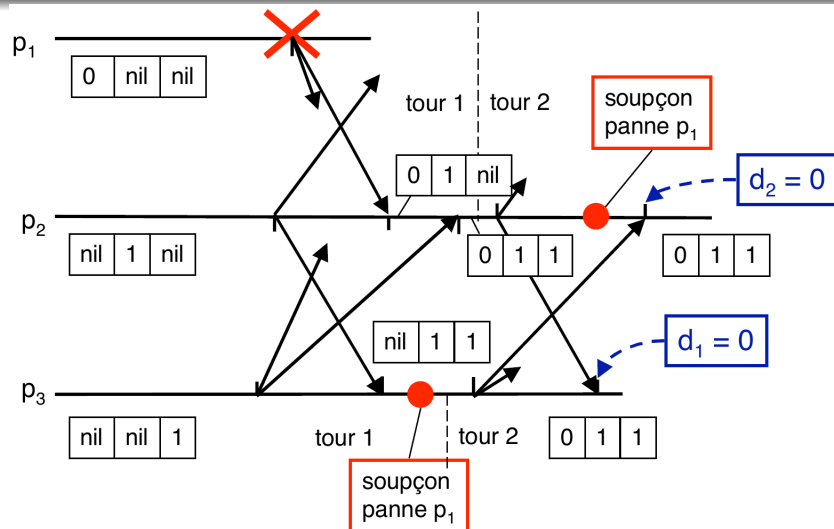
Remarques :

- il suffit de transmettre les nouvelles valeurs apprises au tour précédent  $\Rightarrow$  un processus n'envoie une valeur qu'une seule fois.
- $V_i[j]$  est stable : mis à jour au plus une fois et plus jamais modifié si non  $\perp$ .





## Consensus avec détecteur parfait : exemple



## Consensus avec détecteur parfait : sûreté – intégrité, validité

- **Intégrité** : tout processus décide au plus une fois.  
Unique point de décision après  $f + 1$  tours.
- **Validité** : la valeur décidée est l'une des valeurs proposées.  
Les vecteurs  $V_i$  sont constitués de valeurs connues par les autres sites, et au départ il n'y a que les  $V_i[i]$  qui contiennent les valeurs proposées (pas d'invention de valeurs).

## Consensus avec détecteur parfait : sûreté – accord

La valeur décidée est la même pour tous les processus corrects.

Supposons que  $p_i$  et  $p_j$  décident deux valeurs différentes  $\Rightarrow$  au tour  $f + 1$ ,  $V_i$  et  $V_j$  sont différents. Soit  $x$  une valeur de  $V_i$  non présente dans  $V_j$ .

- $x$  a été transmis à  $p_i$  par un processus correct  $p_k$ , mais  $p_j$  n'a pas attendu ce message car il suspectait  $p_k$ .  
Impossible : **exactitude du détecteur parfait**.
- $x$  a été transmis à  $p_i$  au **dernier** tour par un processus  $p_k$  qui s'est arrêté avant de transmettre  $x$  à  $p_j$  (si pas au dernier tour,  $p_i$  aurait transmis la valeur à  $p_j$ ).  
 $p_k$  connaissait cette valeur depuis un tour  $t$ . Nécessairement,  $t =$  le tour précédent ( $f$ ), car sinon  $p_j$  aurait attendu et reçu un message de  $p_k$  avec la valeur  $x$  ( $p_k$  correct aux tours  $< f + 1$ ).  
Par récurrence, puisque  $p_j$  ne connaît pas  $x$ , le processus qui devait le lui envoyer s'est chaque fois arrêté au tour précédent.  
 $\Rightarrow$   **$f + 1$  défaillances** : impossible

## Consensus avec détecteur parfait : vivacité – terminaison

**Terminaison** : tout processus correct décide au bout d'un temps fini

Supposons qu'un processus  $p_i$  ne décide pas. Alors il est bloqué dans un tour en attente d'un message provenant d'un processus  $p_j$  :

- $p_j$  est correct. Alors son message finira par arriver (en temps fini non borné) ;
- $p_j$  est défaillant. Alors il finira par être suspecté (**complétude du détecteur parfait**) et  $p_i$  ne l'attendra plus.

Consensus avec détecteur  $S$ 

On sait qu'un processus correct n'est jamais suspecté. Faire  $n$  tours pour être sûr de le voir.

- $n - 1$  tours identiques à l'algorithme précédent.  
Au  $n - 1$ -ième tour, chaque processus a un vecteur, mais tous les vecteurs des processus corrects ne sont pas nécessairement identiques car des processus corrects ont pu être suspectés.
- tour supplémentaire : chaque processus diffuse son vecteur et attend un vecteur de chacun des autres processus qu'il ne suspecte pas. Les  $\perp$  reçus effacent de son vecteur les valeurs non  $\perp$ .  
À la fin, tous les processus corrects ont le même vecteur : des valeurs proposées par des processus corrects peuvent ne pas y être (processus suspectés à tort) mais au moins la valeur du processus correct jamais suspecté est présente.
- décision comme précédemment.



## Synthèse

| détecteur                | nombre de défaillances | nombre de tours |
|--------------------------|------------------------|-----------------|
| $P$                      | $n - 1$                | $f + 1$         |
| $S$                      | $n - 1$                | $n$             |
| $\diamond S, \diamond W$ | $\frac{n-1}{2}$        | fini non borné  |

## Plus faible détecteur de défaillances

Le détecteur de défaillances  $\diamond W$  (complétude faible, exactitude finalement faible) est le plus faible détecteur de défaillances permettant de résoudre le consensus en asynchrone avec défaillance d'arrêt.

1. *The Weakest Failure Detector for Solving Consensus*. Tushar Chandra, Vassos Hadzilacos and Sam Toueg. Journal of the ACM. July 1996.

Consensus avec détecteur  $\diamond S$ 

Pour tolérer  $f$  défaillances, il faut  $2f + 1$  processus.

Principe :

- Un coordinateur tournant. Chaque processus sert de coordinateur à tour de rôle.
- Exactitude finalement faible : il existe un moment où un processus correct sera à la fois non suspecté et coordinateur.
- Un coordinateur (non suspecté) qui réunit une majorité de processus corrects (possible car  $n \geq 2f + 1$ ) peut décider d'une valeur.
- Il diffuse alors cette valeur qui est retenue par tous les processus corrects.



## Implantation des détecteurs de défaillances



Les détecteurs de défaillances  $P, S, \diamond S, \diamond W$  ne sont pas réalisables en asynchrone avec défaillances d'arrêt (FLP), mais ils sont réalisables en supposant assez de synchronisme.

On se donne  $\delta$  = délai maximal de transmission d'un message.

## Détecteur actif (ping)

Périodiquement,  $p$  envoie un message à  $q$  et attend un acquittement. En absence de réponse après  $2\delta$ ,  $p$  suspecte  $q$ .

## Détecteur passif (heartbeat)

Périodiquement (période  $T$ ),  $q$  diffuse un message "je suis vivant". Si à  $T + \delta$  après le message précédent, le processus  $p$  n'a rien reçu, il suspecte  $q$ .

Nécessite des horloges synchronisées, réalisables sous l'hypothèse synchrone d'un délai maximal de communication.



## Implantation des détecteurs de défaillances

Estimation de  $\delta$  :

- Trop petite : fausses détections
- Trop grande : temps trop long avant détection

Si le système est effectivement asynchrone, il n'existe pas de  $\delta$  qui évite les fausses détections, et on ne peut pas garantir l'exactitude même faible (tous supposés défaillants par au moins un autre), sauf à tuer les processus suspectés. . .

*nt*

## Conclusion sur le consensus

### Universalité

Consensus  $\rightarrow$  tout objet (ayant une spécification séquentielle)

### Communication par messages, synchrone

- Consensus réalisable même avec défaillances complexes (mais alors très coûteux)
- Modèle peu réaliste

### Communication par messages, asynchrone

- Consensus impossible même avec défaillance simple
- Contournements :
  - détecteurs de défaillances
  - hypothèse de synchronisme suffisamment longtemps (Paxos)

*nt*

## Détecteur de défaillances – bilan

- Apport théorique : identifier le minimum nécessaire pour réaliser le consensus ; cadre unique de comparaison
- Apport pratique : isoler la résolution d'un problème réparti et la gestion de la détection des défaillances

*nt*

### Systèmes et algorithmes répartis

#### Conclusion

Philippe Quéinnec

ENSEEIH  
Département Sciences du Numérique

13 septembre 2023

- 1 Introduction : problématique, transparence
- 2 Modèle événementiel, causalité, état global
- 3 Temps logique, délivrance ordonnée
- 4 Quelques algorithmes génériques : élection, exclusion mutuelle, terminaison
- 5 Grande échelle, pair à pair
- 6 Problème universel : le consensus
- 7 Tolérance aux fautes, groupe
- 8 Gestion des données : réplication, systèmes de fichier répartis
- 9 Simulation répartie

## Que faut-il retenir ?

### Originalité du domaine

- Absence d'état global
- Absence de temps global
- Panne partielle

### Utilité

- Travail collaboratif
- Travail en mode déconnecté
- Puissance de calcul
- Disponibilité (pannes)
- Adaptabilité

## Domaine difficile mais quelques réponses

### Modèle

- Modèle à événements locaux
- Causalité
- État et temps logique

### Outils spécifiques

- Protocoles ordonnés (fifo, causaux. . .)
- Groupe et diffusion
- Littérature : problèmes connus (impossibles ?), par exemple élection, exclusion mutuelle, terminaison, consensus, réplication. . .