Calcul parallèle

Eric Goubault

Commissariat à l'Energie Atomique

Saclay

Plan du cours

- Compléments RMI
  - autres fonctions liées au `rmiregistry`
  - création de `rmiregistry`
  - introduction aux objets activables à distance
- Protocoles distribués tolérants aux pannes
- Quelques perspectives après ce cours...

`rmiregistry`

- Un `rmiregistry` peut être installé sur n'importe quel serveur `http`.
- On peut connaitre la liste des services accessibles sur ce serveur par `String[] list(String name) throws ...` (`name` est l'URL du `rmiregistry`)

rmiregistry

- On a vu `rebind`, il existe aussi:
- `bind(String name, Remote obj) throws ...`
- `unbind(String name) throws ...`

`rmiregistry`

- On peut créer dans le code de démarrage d'un serveur, un `rmiregistry`: `Registry createRegistry(int port) throws ...` (package `java.rmi.registry.Registry`).
- On peut trouver un service de nommage par `LocateRegistry.getRegistry(String host, int port)`.
- Toutes les fonctions de la classe `Naming` s'appliquent (ex. `lookup`, `rebind`) - ce ne sont plus des méthodes statiques.

De façon encore plus générale, on peut directement manipuler les `stubs` par `exportObject`, sans avoir forcément à hériter d'`UniCastRemoteServer` - mais tout est à gérer à la main alors!

Threads et RMI

- De façon interne, des threads peuvent être créés sur les serveurs, pour traiter les appels aux méthodes distribuées, mais:
- il n'est pas garanti que 2 appels à une méthode distante seront exécutés dans un même thread, ou dans un thread différent
- la seule condition qui permette d'être sûr que 2 threads différents seront créés: quand les appels viennent de 2 JVM distinctes

Reveil de serveur

Problèmes dans les méthodes précédentes:

- Même si les services sont peu souvent utilisés, il faut que les serveurs tournent en permanence,
- les serveurs doivent créer et exporter les objets distants: consommation inutile de mémoire et de CPU,

Depuis JAVA 2 on peut activer à distance un objet distant.

Lourd et compliqué, cf. TD (optionnel)

Activation à distance

- Permet d'enregistrer un service RMI sans l'instancier,
- Le service RMI défini par cette méthode est inactif,
- Est réveillé seulement quand un client y fait appel,
- Un processus démon est chargé d'écouter les requêtes et de réveiller les services: `rmid`.

Inscription du service

- A la place du service, une sorte de “proxy” est enregistré auprès du serveur de services RMI (`rmiregistry`),
- Contrairement aux serveurs instances de `UnicastRemoteObject`, cette “fausse” référence ne s’exécute que pendant un cours instant, pour inscrire le service auprès du `rmiregistry`, puis aux moments d’appels au service.

Activation du service

- Quand le client appelle ce service, le `rmiregistry` fait appel à cette fausse référence,
- Celle-ci vérifie si elle a un pointeur sur le vrai service,
- Si non, elle fait appel au démon `rmid` pour créer une instance du service (prend un certain temps, la première fois),
- Puis transmet l'appel au service nouvellement créé.

En pratique

- Le client n'a en rien besoin de savoir comment le service est implémenté (en activation à distance, ou comme serveur permanent).
- La création du service est un peu différente que du serveur résident, mais son code reste similaire.

En pratique - interface serveur activable à distance

L'interface du service ne contient que les méthodes désirées (sans argument, mais devant renvoyer une exception de type `java.rmi.RemoteException` - imposé par `rmic`):

```
public interface InterfaceObjetActivable
  extends java.rmi.Remote
{  public void MethodeA() throws java.rmi.RemoteException;
   public void MethodeB() throws java.rmi.RemoteException;
  ... }
```

Cette interface peut être en fait implémentée par un objet activable à distance ou un service permanent `UnicastRemoteObject`.

Création d'une implémentation du serveur

- Doit être une instance de `java.rmi.activation.Activatable`,
- Doit implémenter un constructeur,
- Doit implémenter les méthodes désirées `MethodeA` etc.

Exemple - Création d'un objet activable à distance

```
public class ImplementationObjetActivable extends
             java.rmi.activation.Activatable
             implements InterfaceObjetActivable
{
  public ImplementationObjetActivable (
    java.rmi.activation.ActivationID activationID,
    java.rmi.MashalledObject data) throws
    java.rmi.RemoteException
    { super(activationID,0); }
```

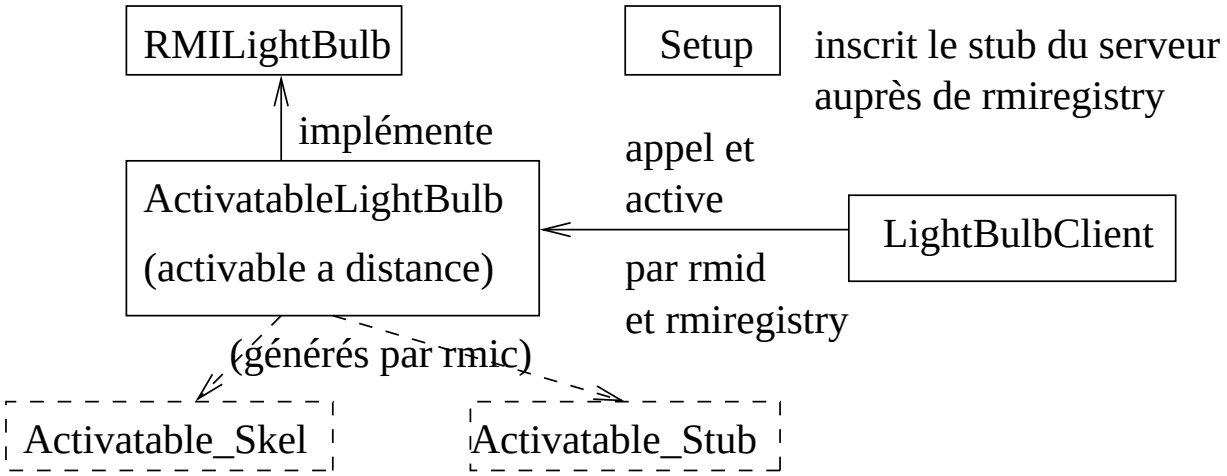
(appelle le constructeur de la classe parent `Activatable(ID,0)` - 0 ou sur un numéro de port spécifique)

Activation de l'objet

- En général inclus dans le `main` de l'implémentation de l'objet (service),
- Code assez compliqué par le fait que doivent être gérés:
  - la politique de sécurité,
  - les propriétés et droits du code exécutable,
  - l'inscription auprès de `rmid` et `rmiregistry`.

Exemple d'une "lampe"

(voir page web du cours)



Complément: Sécurité

- Un programme JAVA peut dans certaines circonstances avoir à s'associer une "politique de sécurité" donnant les droits d'objets provenant de certaines autres machines.
- Exemple: la connection par `socket` à une machine distante passe par la méthode `checkConnect(String host, int port)` du `SecurityManager` (en fait, sous-classe `RMISecurityManager` pour les politiques concernant RMI) courant (définissant la politique de sécurité courante). En cas de non autorisation, messages du type:

```
java.security.AccessControlException: access denied
(java.net.SocketPermission 127.0.0.1:1099 connect,resolve
```

La classe `Policy`

- chaque programme a une politique de sécurité courante, instance de `Policy`, qui est `Policy.getPolicy()`,
- on peut modifier la politique de sécurité courante (comme avant avec le `SecurityManager`) en faisant appel à `Policy.setPolicy`,
- un fichier de permissions (lu au démarrage d'un programme) de la forme:

```
grant codeBase ‘‘file:/home/goubault/-’’ {
  permission java.security.AllPermission;
};
```

donnera tous les droits à tous les programmes dans `/home/goubault/`,

Chargement dynamique de classes

- sur un petit système, on peut gérer les stubs créés etc. (il faut recopier, ou avoir accès par NFS à tous ces fichiers sur toutes les machines considérées)
- on peut acccéder aux serveurs par adresse `http`:

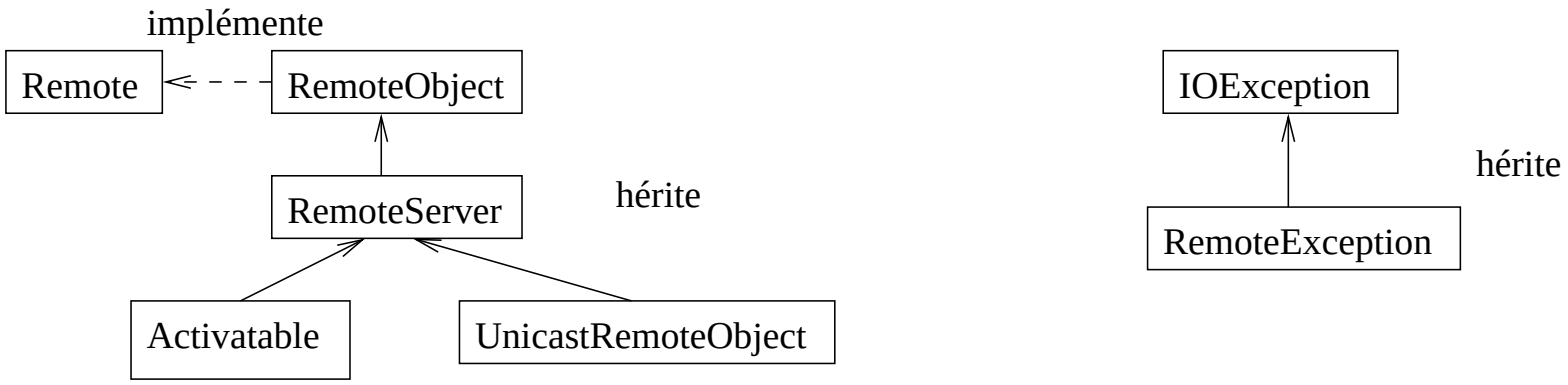
  ```
  java
  -Djava.rmi.server.codebase=http://hostname:port/path
  ```

Pour en savoir plus: packages RMI

- `java.rmi` définit l'interface `RemoteInterface`, et les exceptions,
- `java.rmi.activation` (depuis JAVA2): permet l'activation à distance des objets,
- `java.rmi.dgc`: s'occupe du ramassage de miettes dans un environnement distribué,
- `java.rmi.registry` fournit l'interface permettant de représenter un `rmiregistry`, d'en créer un, ou d'en trouver un,
- `java.rmi.server` fournit les classes et interfaces pour les serveurs RMI.

cf.
`http://java.sun.com/j2se/1.4/docs/guide/rmi/package-use.htm`

Les différentes classes et interfaces



Cycle de développement CORBA

(i) *Ecriture de l'interface de l'objet en IDL* : définition dans le langage de description d'interfaces Corba (ou "Interface Definition Language") des opérations disponibles sur l'objet.

(ii) *Compilation de l'IDL* : génération des modules stub (pour le client) et skeleton (pour le serveur). Ces modules gèrent l'invocation des méthodes distantes.

(iii) *Implémentation de l'objet* : dérivation d'une classe depuis le skeleton généré à l'étape précédente. Les données et méthodes de cette classe doivent correspondre à celles qui sont décrites dans l'IDL. Lors de l'appel des méthodes sur l'objet, le code défini ici sera exécuté par l'application serveur.

Systèmes distribués tolérants aux pannes

Peut-on implémenter certaines “fonctions” sur certaines architectures distribuées, même en présence de pannes?

> Exemple: consensus dans un système asynchrone
>
> NON: FLP’85!

- Il y a une interprétation géométrique du problème “agréable”
- On l’appliquera à des exemples simples, pour comprendre
- Mais cela a permis de résoudre de nombreux problèmes difficiles (on en citera quelques uns)!
- ... c’est un domaine de recherche actif.

Intérêt

- Election d'un leader dans un réseau de machines redondantes...(forme de consensus)
- "Commit" dans une base de données distribuées...(forme de consensus)
- ...

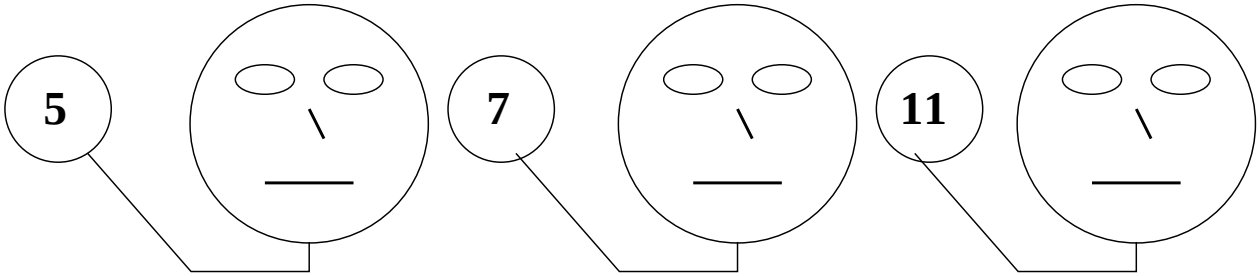
Tâches de décision

Chaque problème sera donné par:

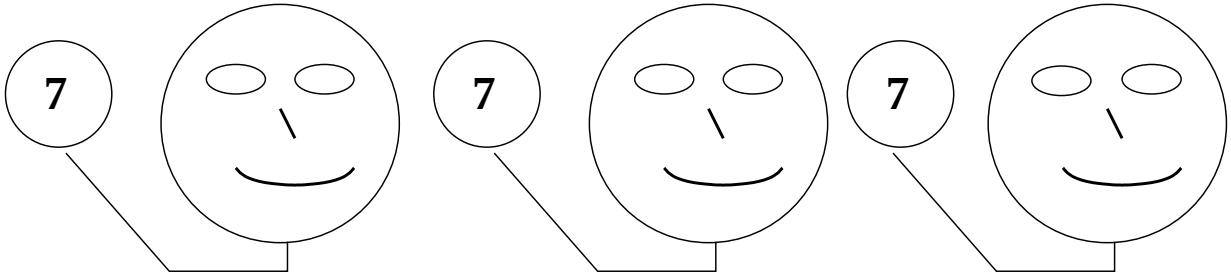
- Pour chaque processeur $P_0, \ldots, P_{n-1}$, un ensemble de valeurs initiales possibles (dans un domaine $\mathcal{K} = \mathbb{N}$ ou $\mathbf{R}$ etc.), i.e. un sous-ensemble $\mathcal{I}$ de $\mathcal{K}^n$: "*input*"
- De façon similaire, on se donne un ensemble de valeurs finales possibles $\mathcal{J}$ dans $\mathcal{K}^n$: "*output*"
- Enfin, on se donne une fonction "*de décision*" $\delta : \mathcal{I} \to \wp(\mathcal{J})$ associant à chaque valeur initiale possible, un ensemble de valeurs finales autorisées.

Exemple: consensus

Before



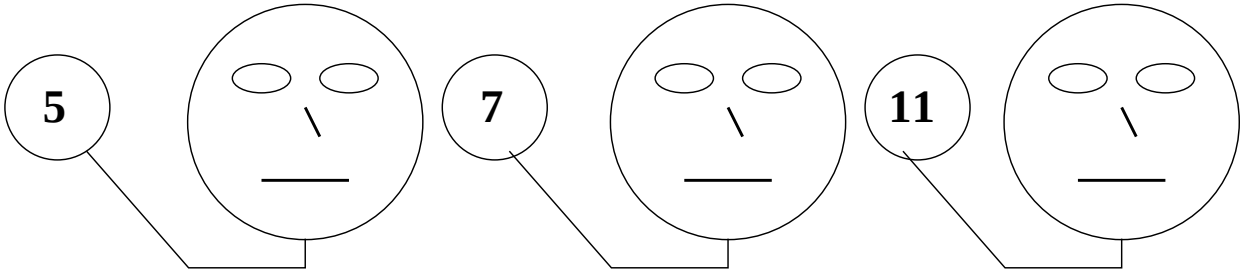
blah blah blah...



After

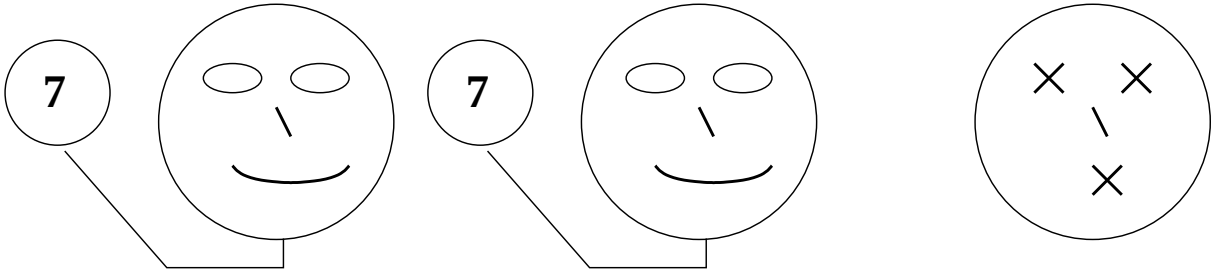
Même si...

Before



blah blah blah...

arghhh...



After

Exemple

- $\mathcal{K} = \mathbb{N}, \mathcal{I} = \mathbb{N}^n$,
- $\mathcal{J} = \{(v, v, \ldots, v) \mid v \in \mathbb{N}\}$,
- $\delta(x_0, x_1, \ldots, x_{n-1}) = \left\{ \begin{array}{c} \{(x_0, x_0, \ldots, x_0), \\ (x_1, x_1, \ldots, x_1), \\ \ldots, \\ (x_{n-1}, x_{n-1}, \ldots, x_{n-1})\} \end{array} \right.$

Idée principale

- Les ensembles d'*entrée* et de *sortie* ont une structure géométrique (ensembles simpliciaux)
- Selon l'architecture, toutes les fonctions de décision ne peuvent pas être programmées
- Il y a des contraintes géométriques sur les fonctions de décision
- Très similaire à la preuve du théorème du point fixe de Brouwer.

Plan

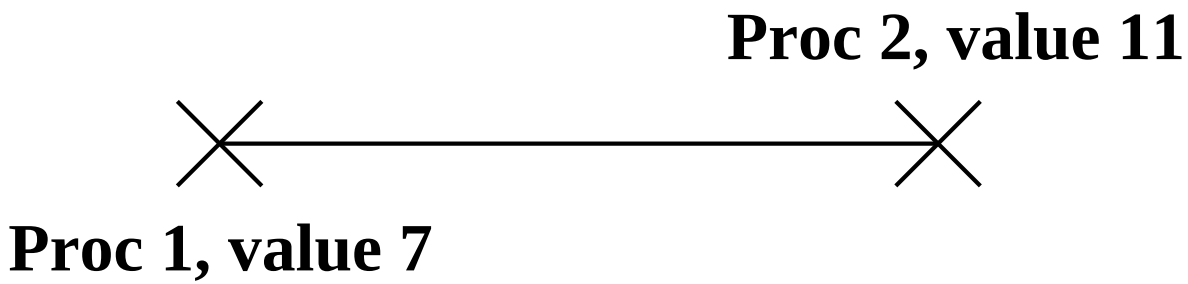
- Les ensembles d'entrée et de sortie sont des ensembles simpliciaux (exemples)
- Un peu de topologie algébrique de base (au tableau)
- La dynamique vue comme ensemble d'ensembles simpliciaux
- Quelques résultats et références

Modèle simplicial des états

Proc 1, value 7

(état local)

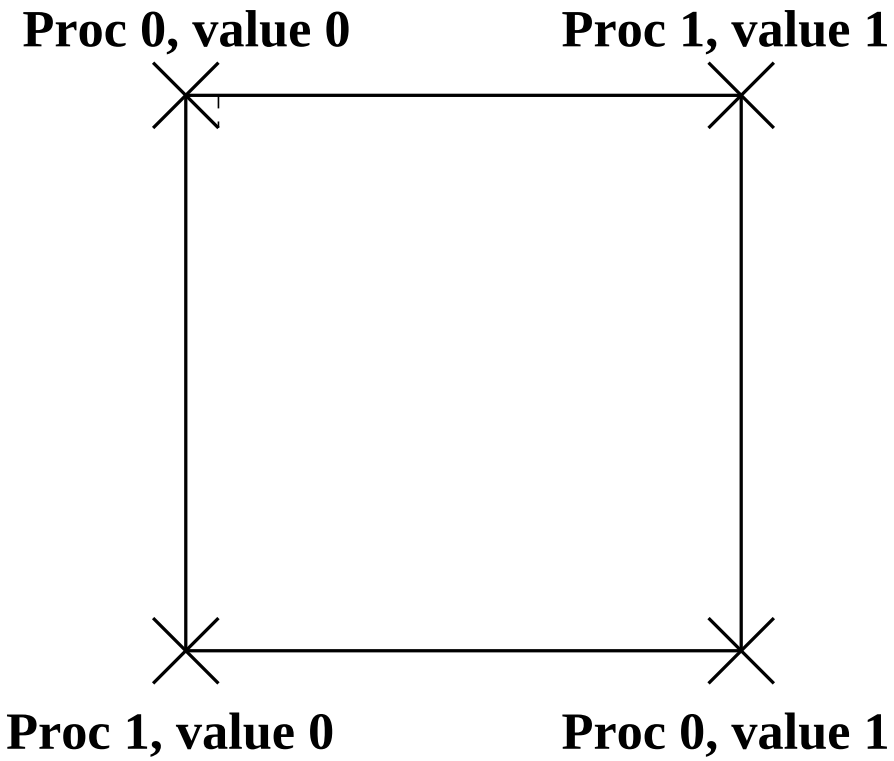
Modèle simplicial des états



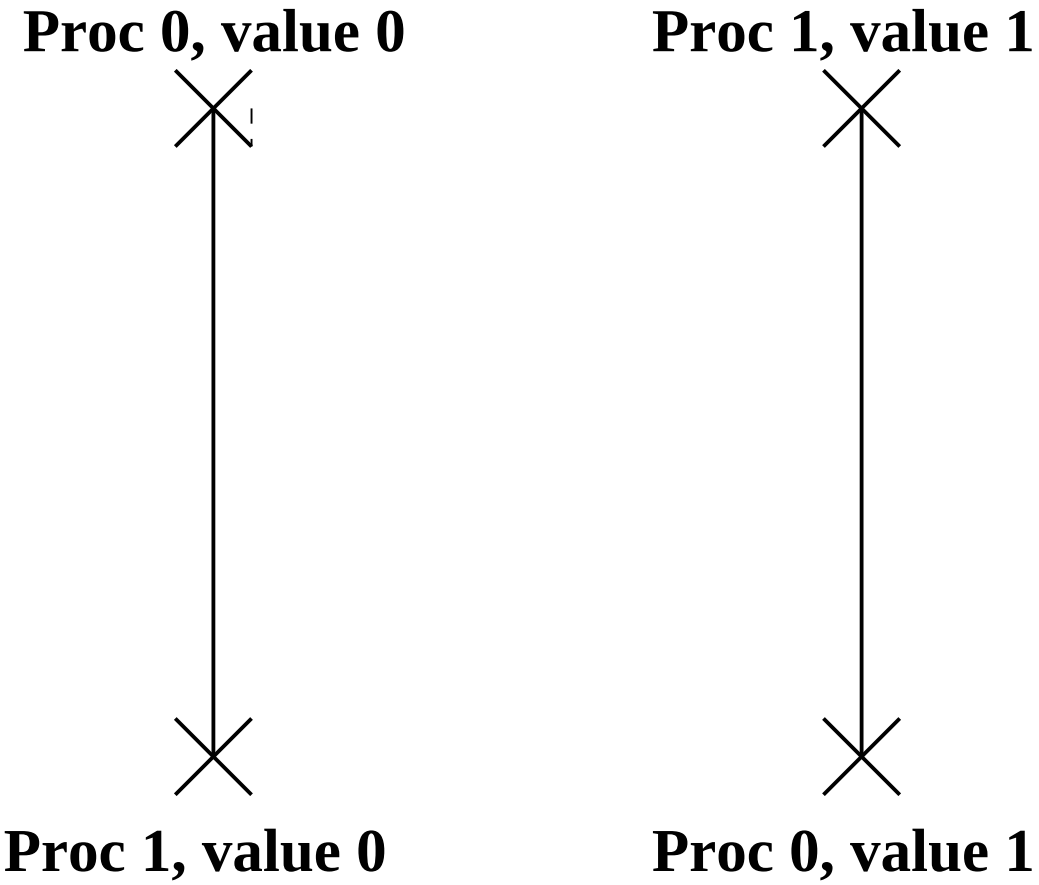
(état composé)

Etats initiaux pour le consensus (binaire)

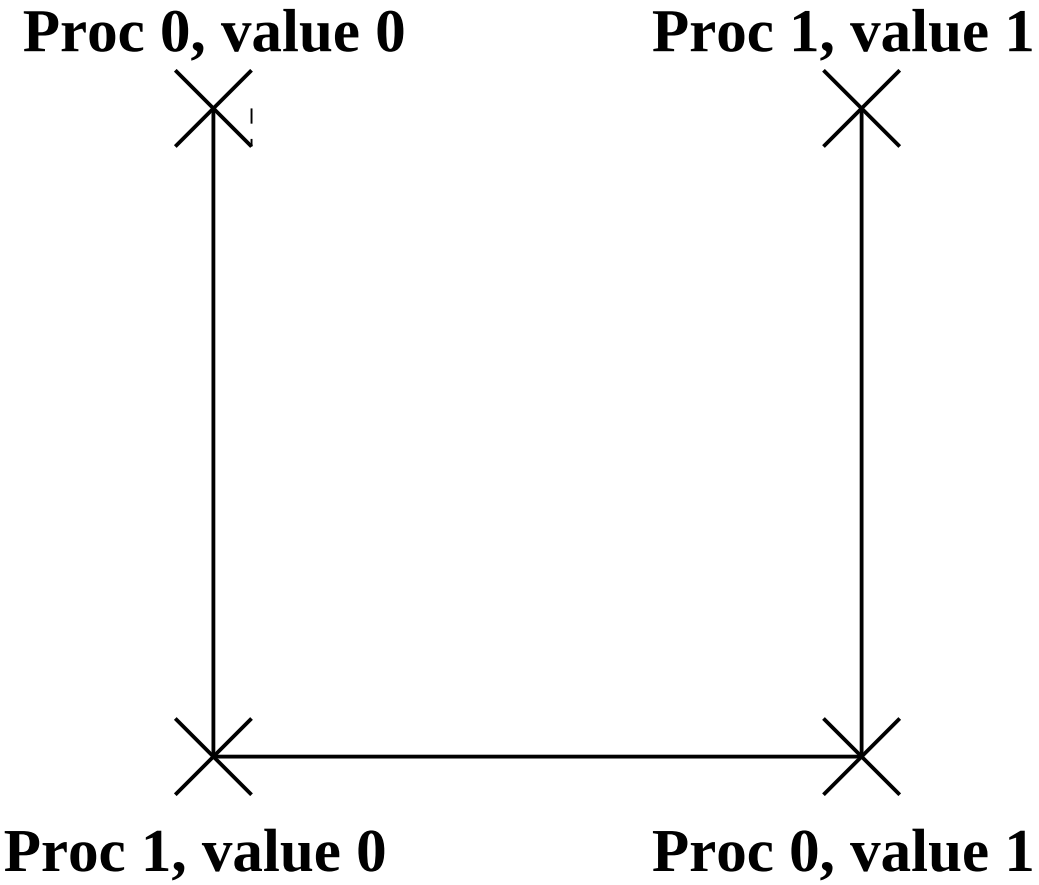
Ici, 2 processus, i.e. dimension 1:



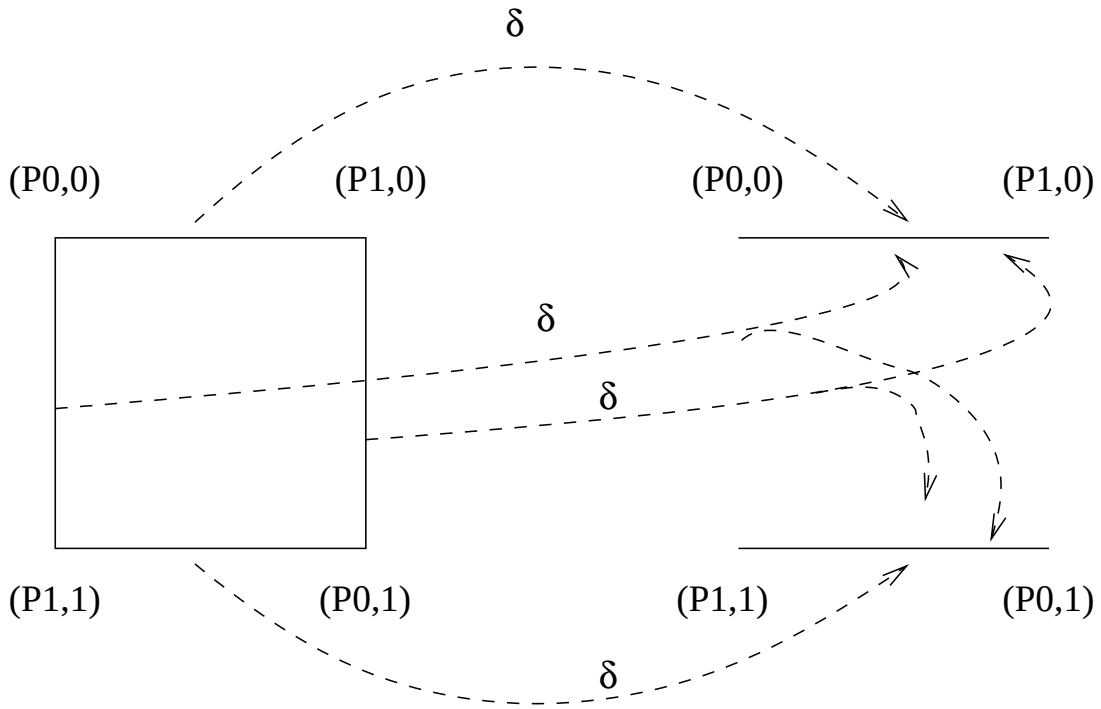
Etats finaux pour le consensus



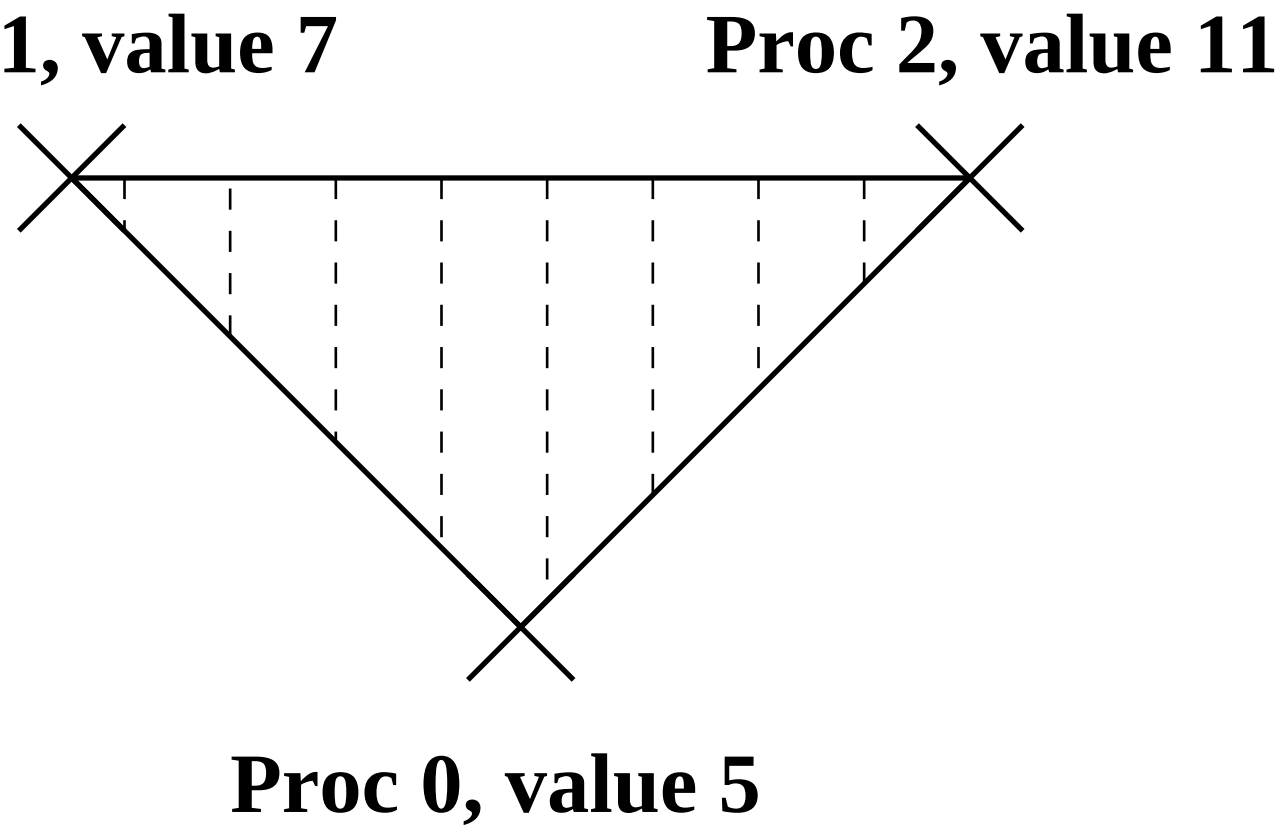
Etats finaux pour le pseudo-consensus



Exemple: spécification du consensus



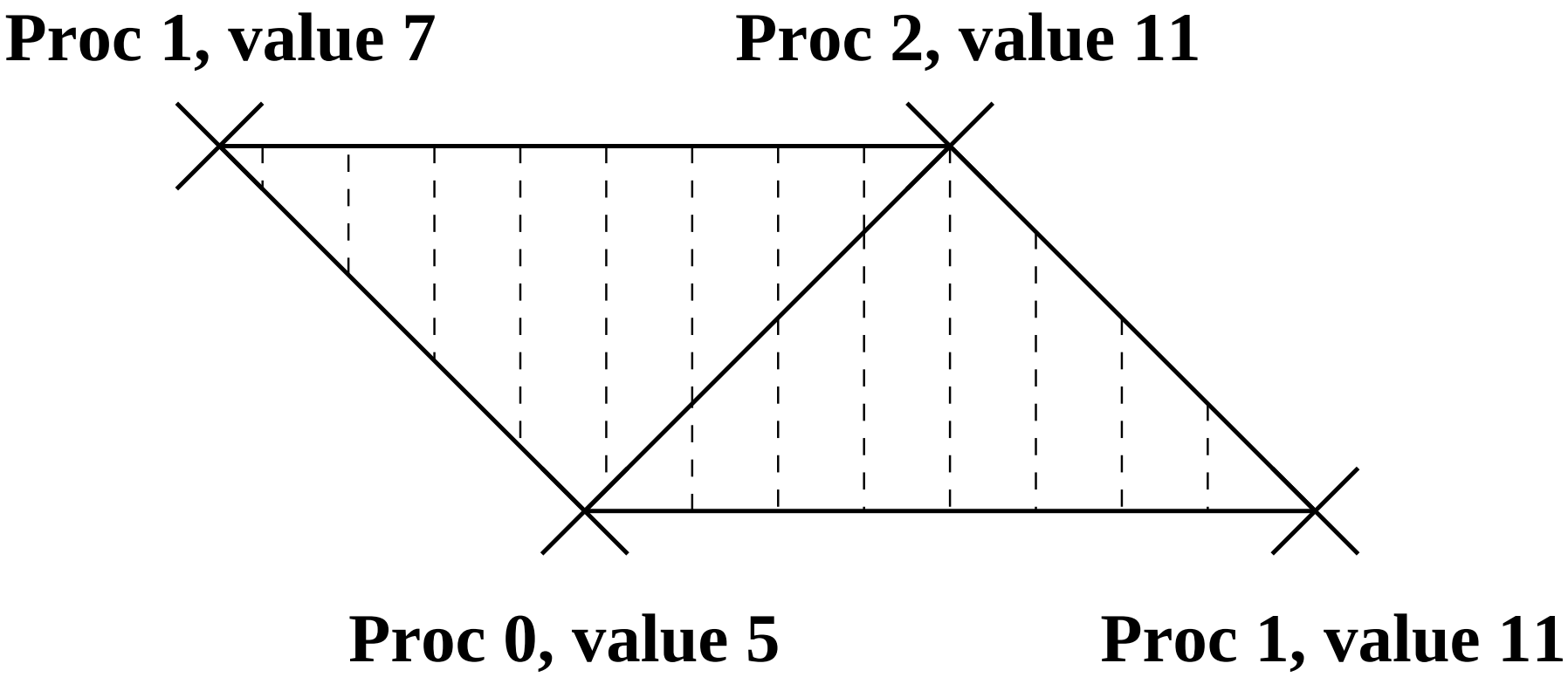
Plus généralement: modèles simpliciaux des états



(Plus généralement [qu'un graphe]: état global)

Exemple

Ensemble simplicial=ensemble d'états globaux (avec certains états locaux, en commun)



Revenons aux *protocoles*

- Programme fini
- Commençant avec des valeurs d'entrée
- Nombre fixé d'étapes
- Termine avec une valeur de décision

Le protocole d'information complète est celui où la valeur locale est l'historique complet des communications.

Protocole générique

```
s = empty;
for (i=0; i<r; i++) {
  broadcast messages;
  s = s + messages received;
}
return delta(s);
```

Exemple

Passage de message synchrone, notion *d'étape*:

- A chaque étape, chaque processeur diffuse sa valeur à tous les autres
- dans n'importe quel ordre
- puis chaque processeur reçoit les valeurs diffusées, et calcule une nouvelle valeur locale

Modèles de pannes

- crash (le processeur se plante et ne communique plus)
- byzantine (le processeur répond mais dit n'importe quoi)
- etc.

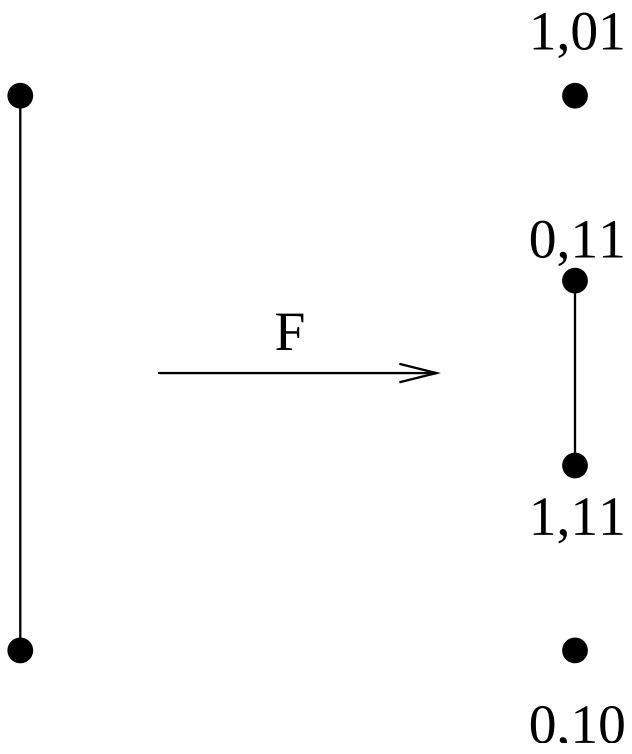
Dans ce qui suit: crash seulement: peut arriver à n'importe quel moment pendant une diffusion, qui peut être faite dans n'importe quel ordre.

Complexe de protocole

Chaque protocole sur une architecture donnée définit:

- un ensemble simplicial (pour toutes les étapes r):
  - noeuds: suite de messages reçus à une étape donnée r
  - simplexes: états composés à l'étape r
- Cela définit un opérateur sur le complexe d'entrée
- Chaque choix de modèle de calcul implique certaines propriétés géométriques du complexe de protocole.

Complexe de protocole

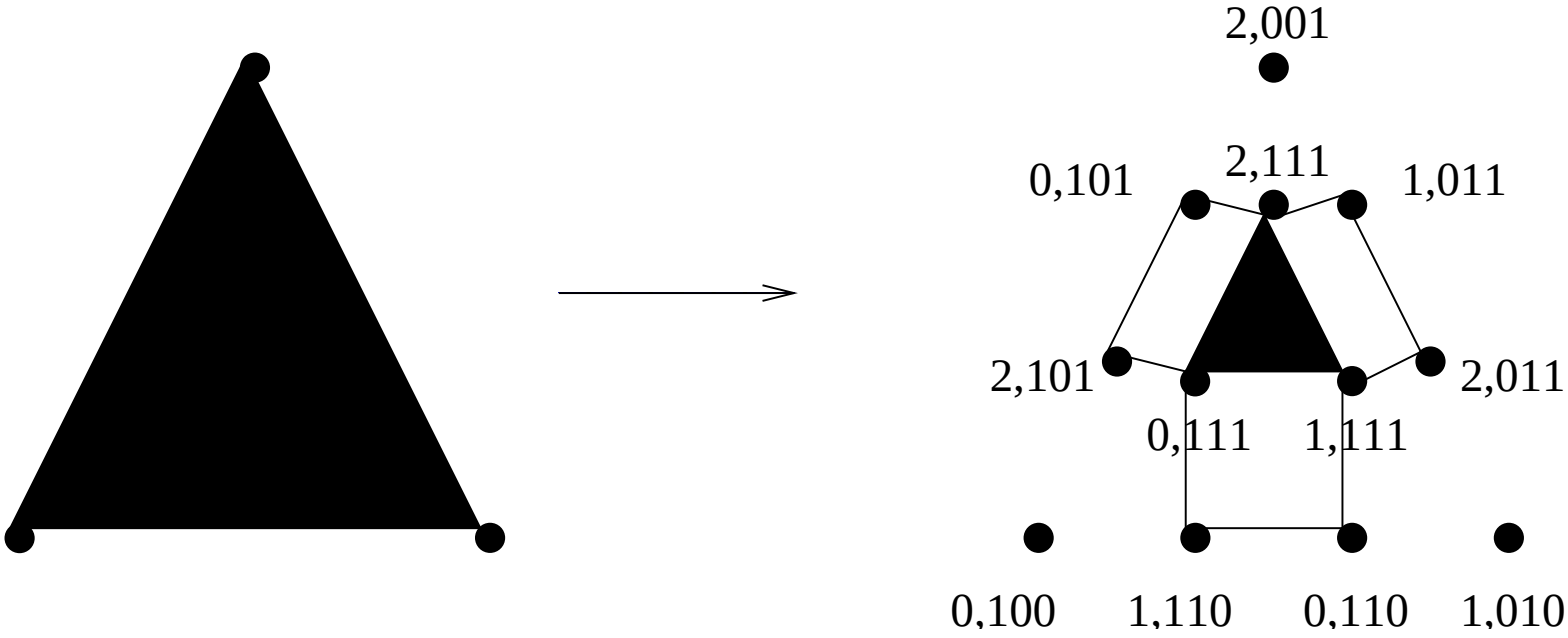


Explication

Dans le modèle synchrone, à l'étape 1:

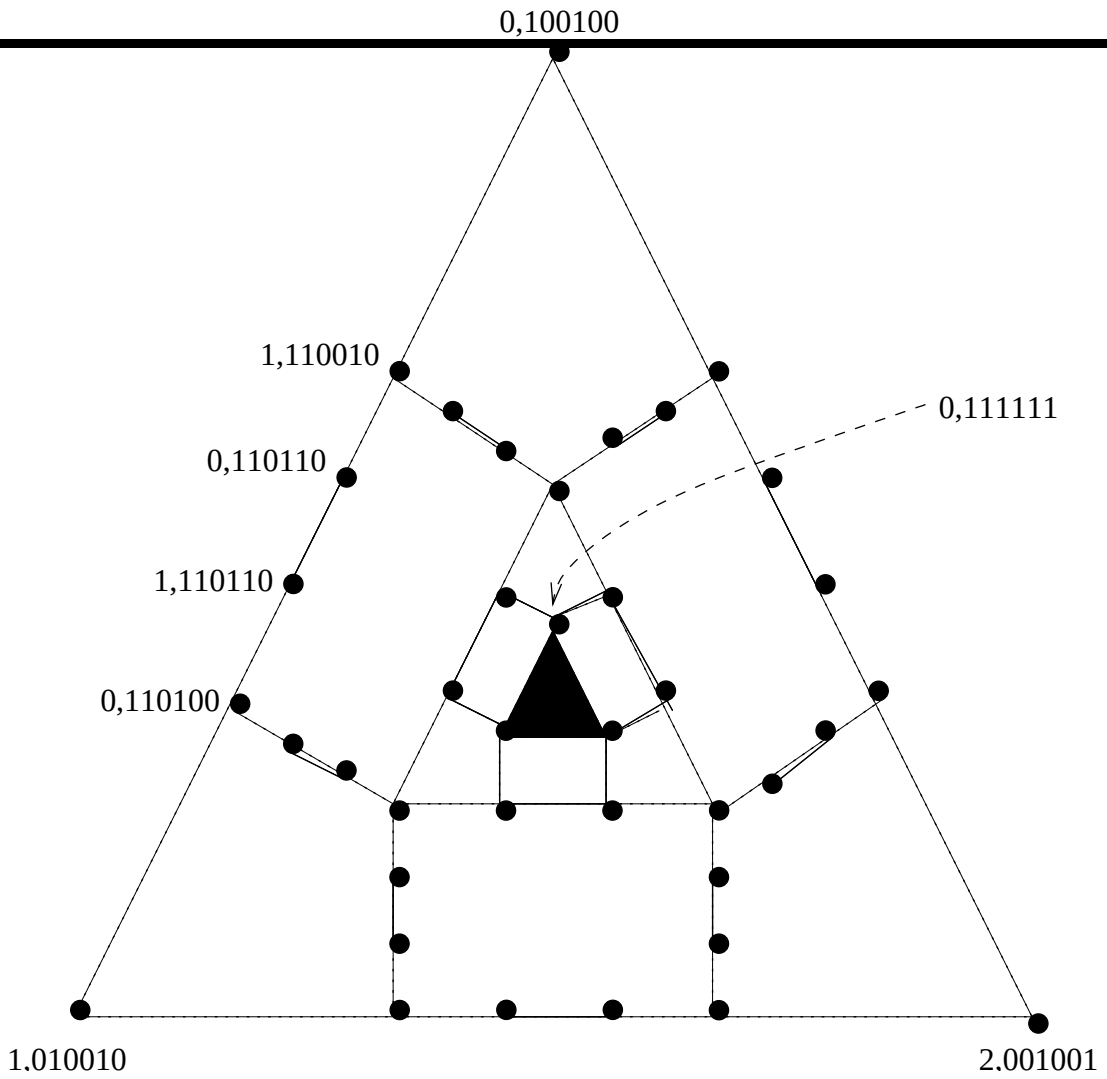
- aucun processus n'a échoué, donc tout le monde a reçu le message de tout le monde (d'où le segment central comme état global)
- un processus a échoué, d'où les deux points comme états possibles

Complexe de protocole synchrone

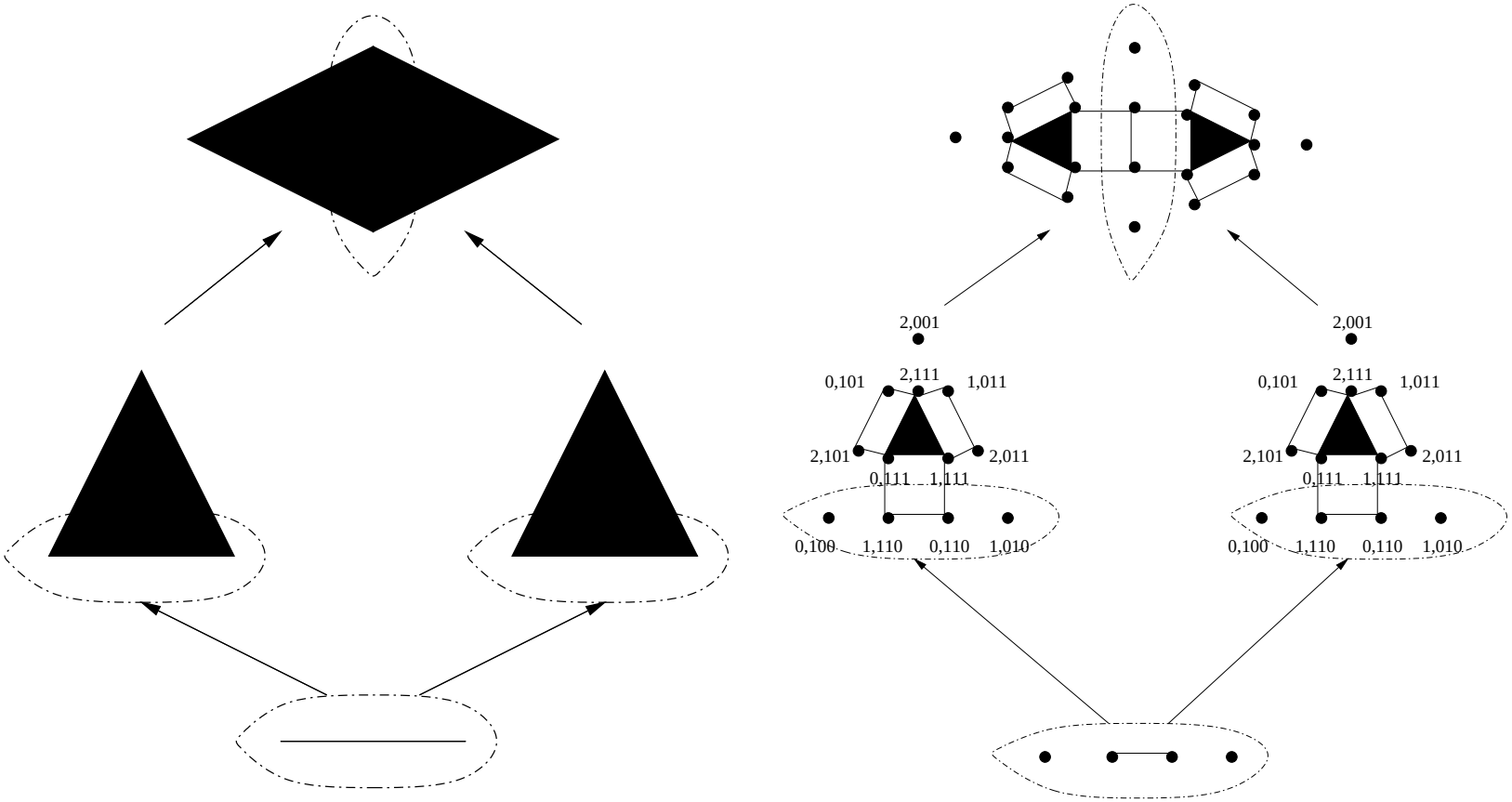


(sans attente - si, jusqu'à une panne, oublier les points isolés!)

Complexe de protocole synchrone - étape 2



Complexe de protocole synchrone



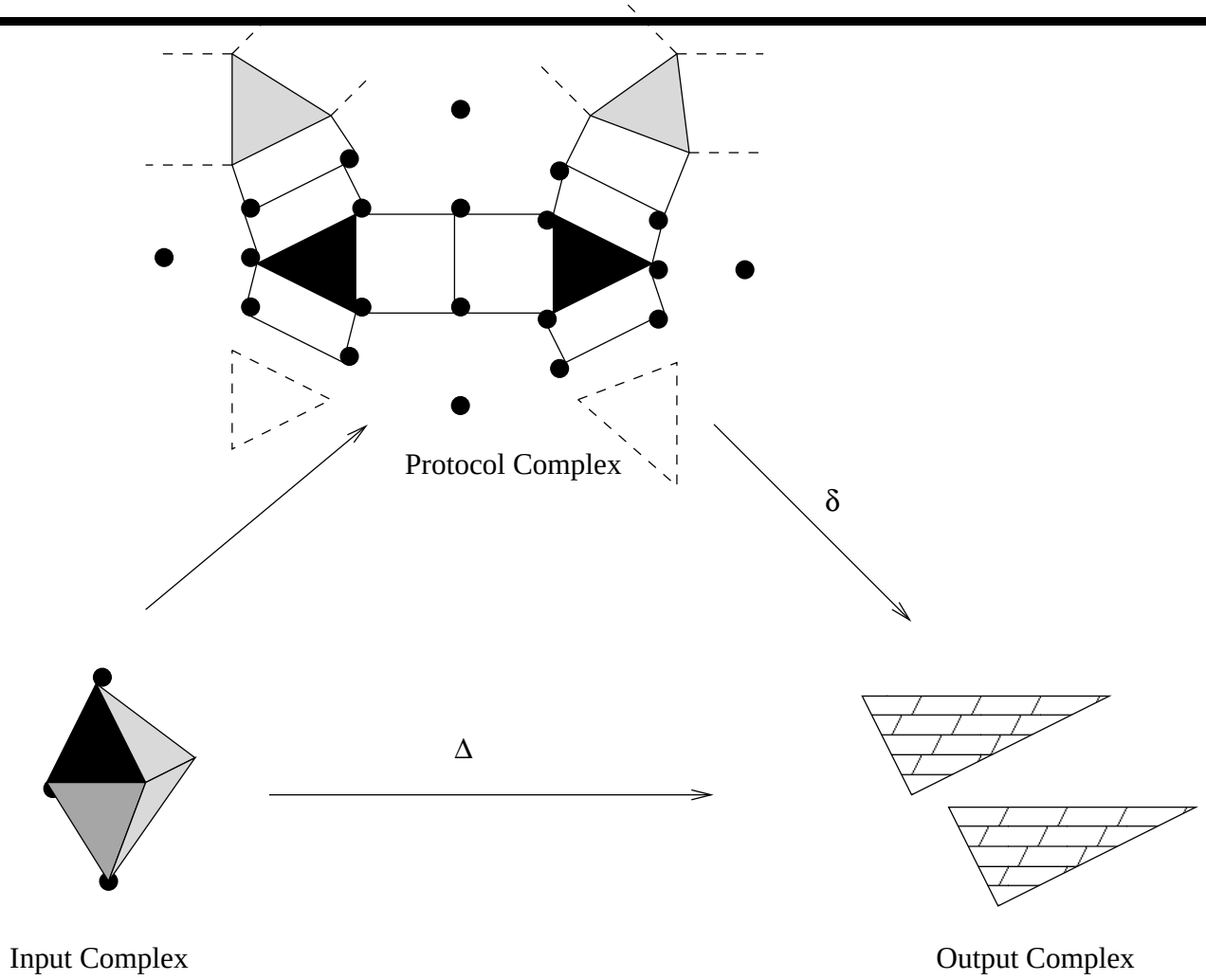
Fonction de décision

Le "delta" dans le protocole générique, est, mathématiquement parlant:

- $\delta : P \to O$ (protocole vers complexe de sortie)
- une fonction simpliciale (c.a.d. une fonction des noeuds vers les noeuds, étendue aux enveloppes convexes)
- respectant la relation de spécification Δ, i.e. pour tout $x \in I$, pour tout $y \in P(I)$, $x\Delta(\delta(y))$

Stratégie de preuve/calcul de complexité: trouver des "obstructions topologiques" au fait que δ puisse être une fonction simpliciale (du complexe de protocole à toutes les étapes/aux étapes jusqu'à k)

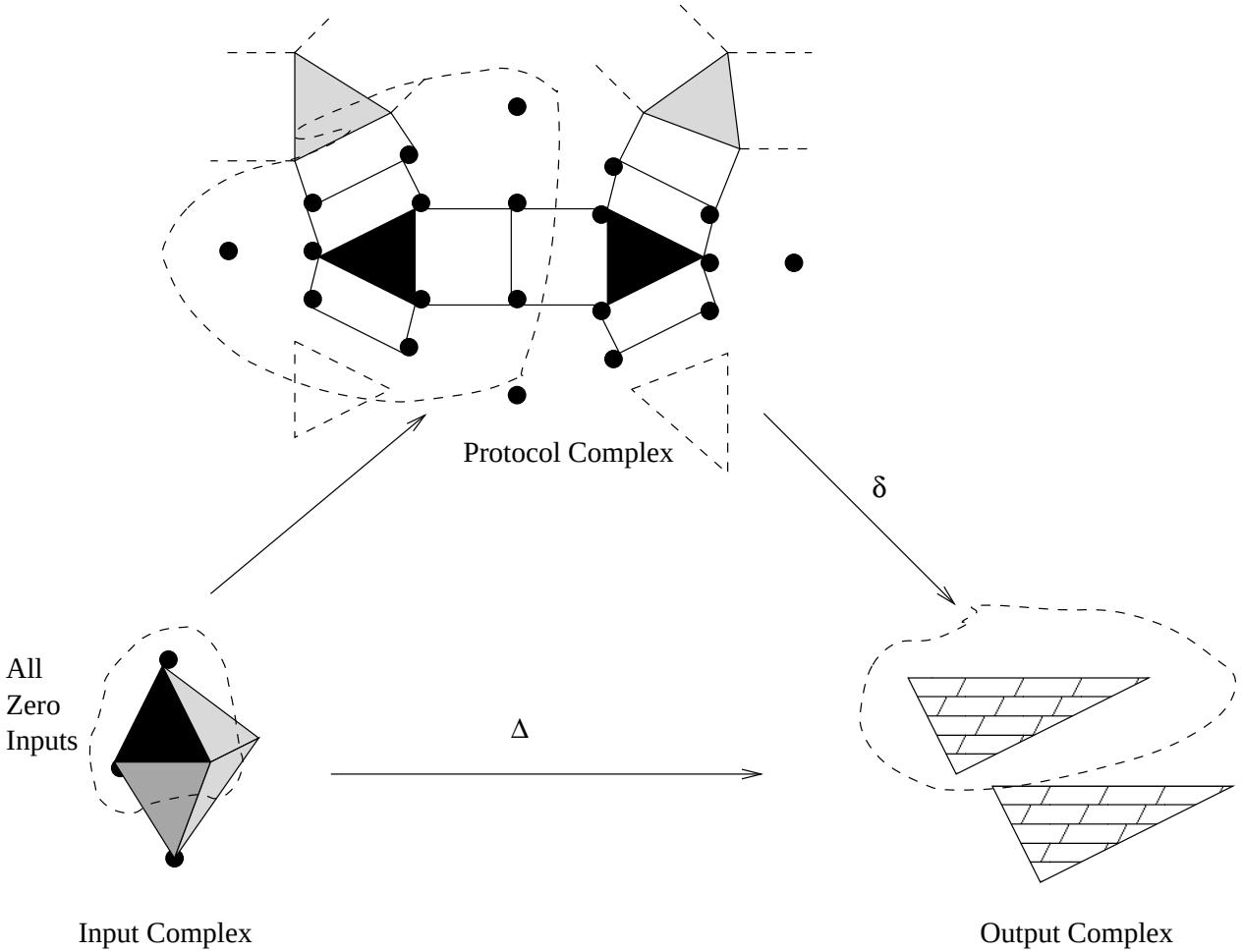
Propriété principale



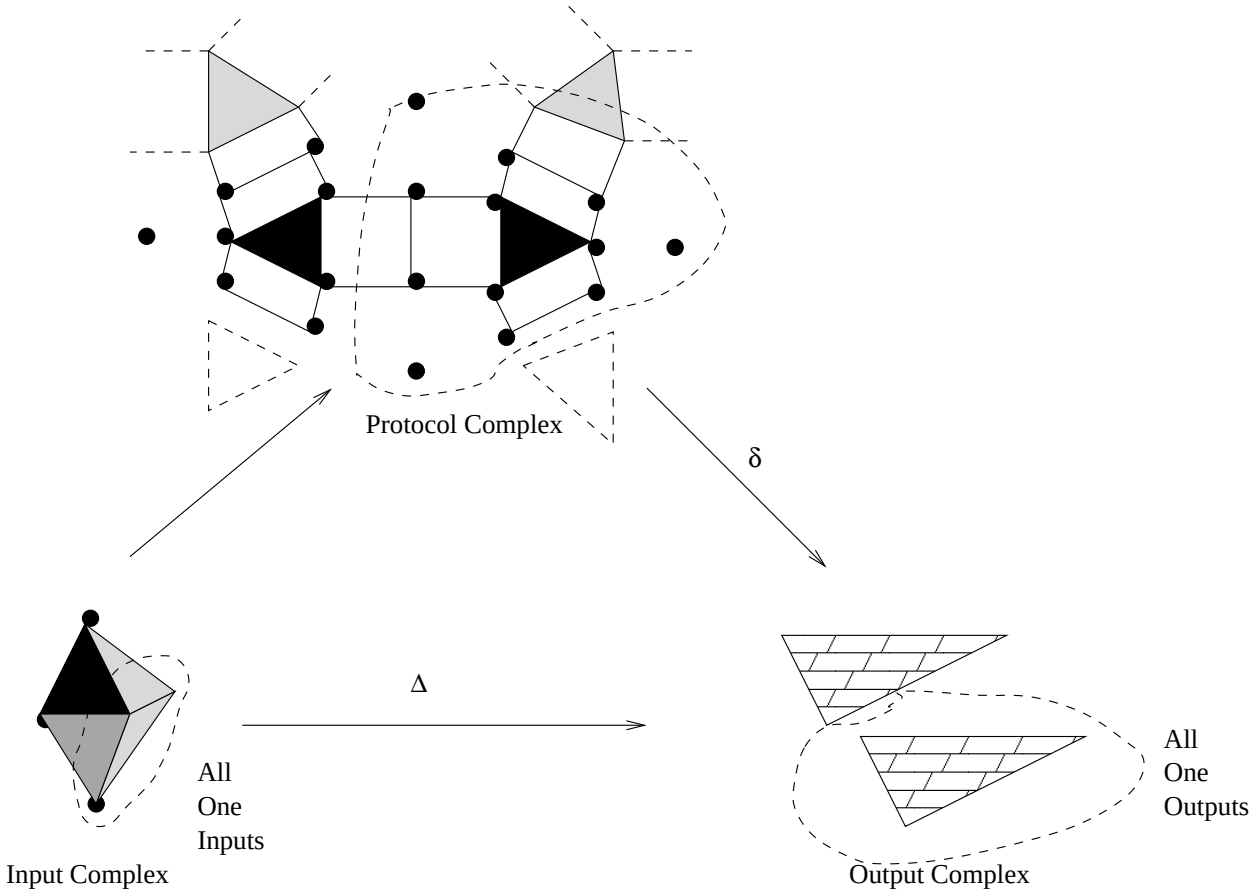
Application simple: encore le consensus...

- Consensus binaire entre 3 processus (modèle passage de message synchrone).
- Le complexe d'entrée est composé de 8 triangles: $(0,0,0)$, $(0,0,1)$, $(0,1,0)$, $(0,1,1)$, $(1,0,0)$, $(1,0,1)$, $(1,1,0)$ et $(1,1,1)$,
- Le complexe d'entrée est homeomorphe (bijection bi-continue) à une sphère (une seule composante connexe); Les quatre premiers triangles déterminent un "hémisphère nord", les quatre derniers, un "hémisphère sud".
- Le complexe de sortie est composé de 2 triangles: $(0,0,0)$ et $(1,1,1)$ (d'où deux composantes connexes),
- On se limite pour le moment à une étape de communication.

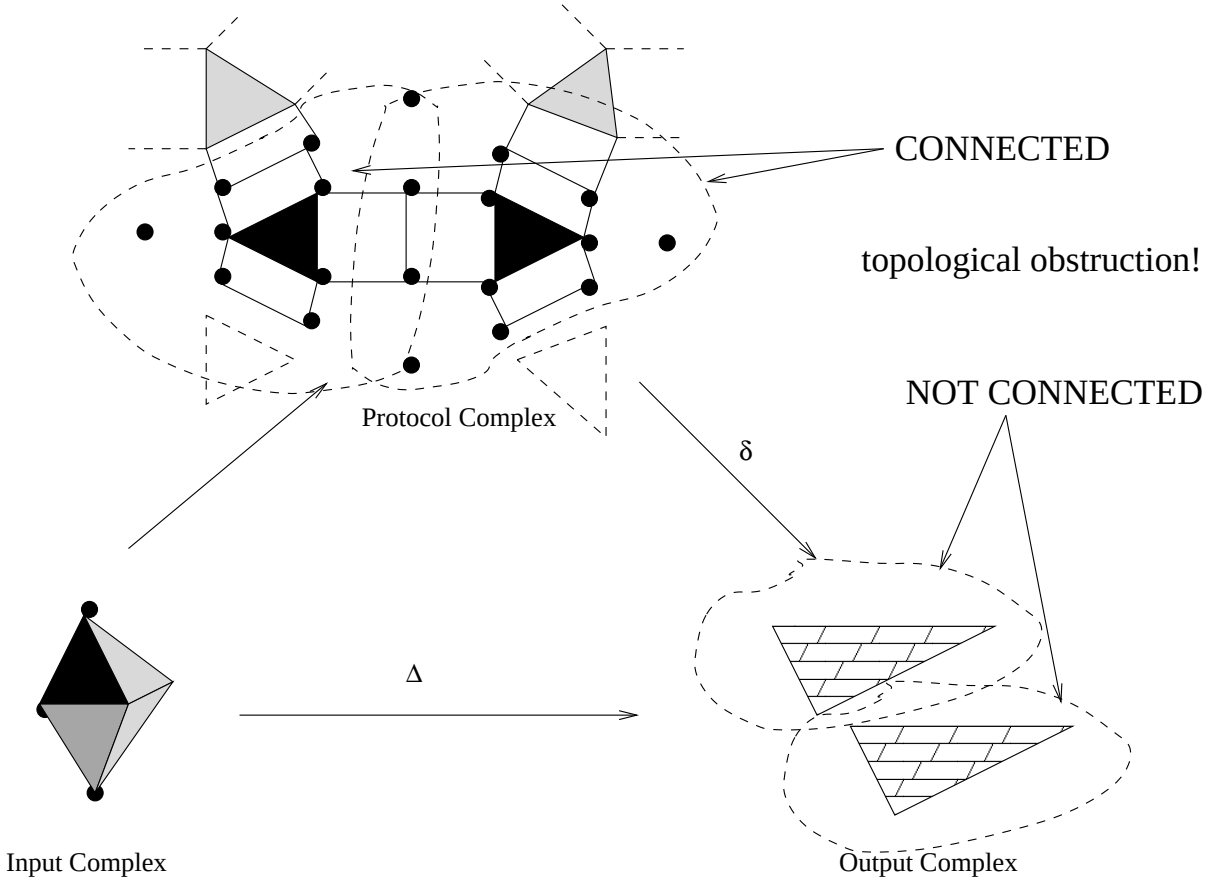
Application simple



Application simple



Application simple - pour au plus $n-2$ pannes seulement!



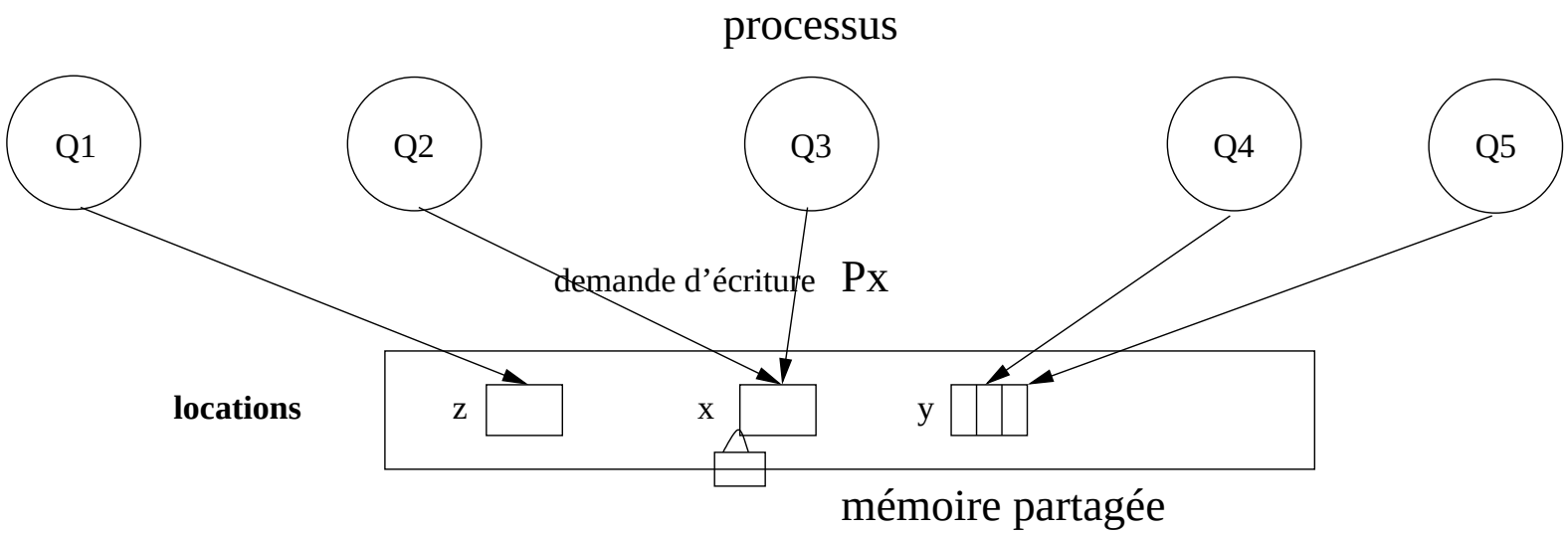
Plus généralement

- Dans tout protocole de ce type à $(n-2)$ étapes, le sous-complexe avec des noeuds tous à zéro, et le "tout 1" sous-complexe sont connectés
- Corollaire: pas de protocole de consensus en $(n-2)$ étapes!

De façon encore plus générale...

- Modèle passage de message synchrone avec r étapes, et au plus k pannes
- $P(S^{n-1})$ est $(n - rk - 2)$-connexe: implique la borne de $(n - 1)$ étape pour le consensus (pour $k = 1$).

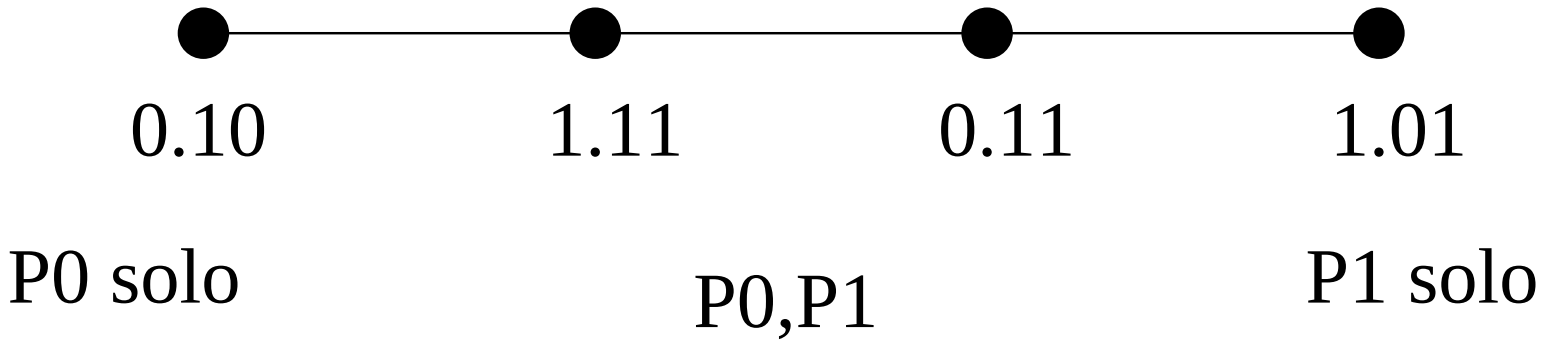
Modèle mémoire partagée



Protocole asynchrones sans attente

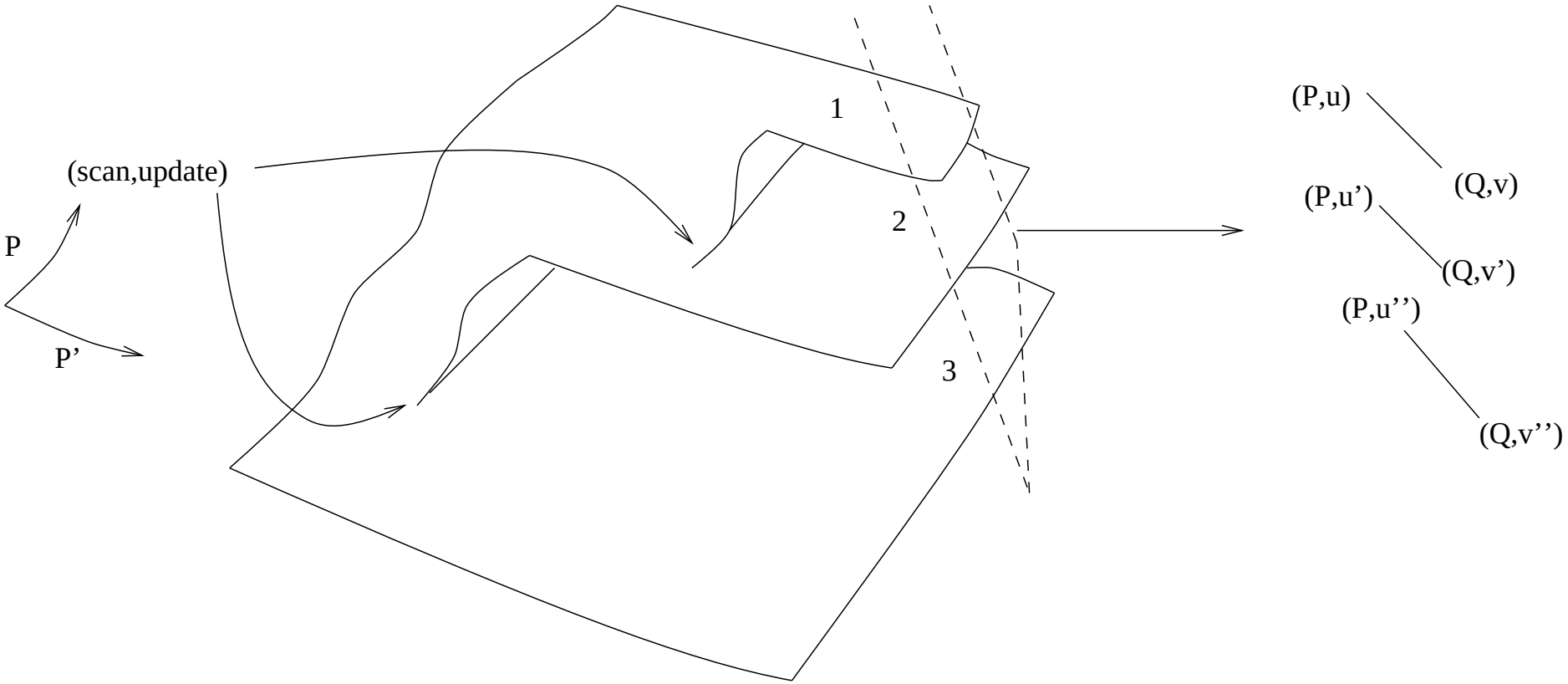
- n processus partagent une mémoire (de taille non bornée) partitionnée: un élément de la partition par processeur
- Chaque processus peut:
  - écrire de façon atomique sur sa partie (update)
  - faire un scan (lecture) atomique de toute la mémoire pour la recopier dans sa mémoire locale
- Equivalent au modèle read/write atomique classique (quand on s'intéresse aux pannes)
- On veut des protocoles *sans-attente* c.a.d. robustes jusqu'à $n-1$ pannes crash.

Complexe de protocole pour une étape (2D)

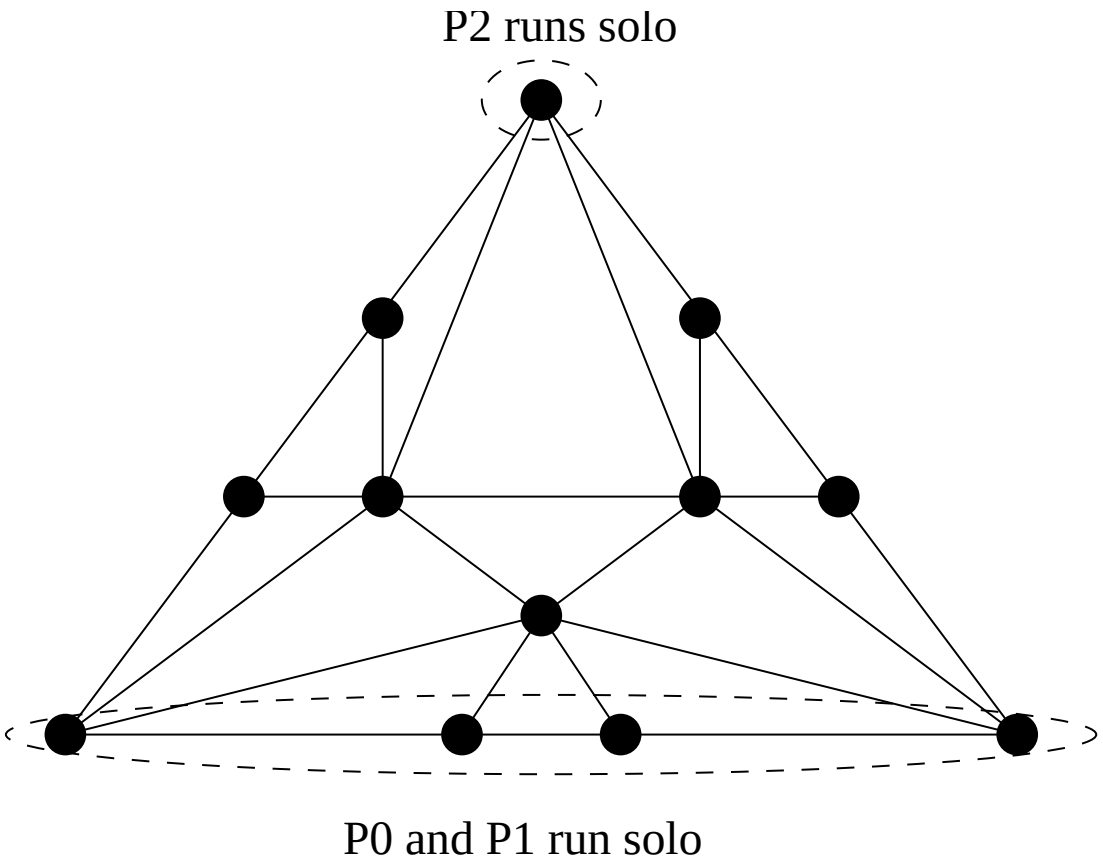


Semantique

Dynamique (et sa “coupe” jusqu’au temps r=complexe de protocole)



Complexe de protocole pour une étape (3D)

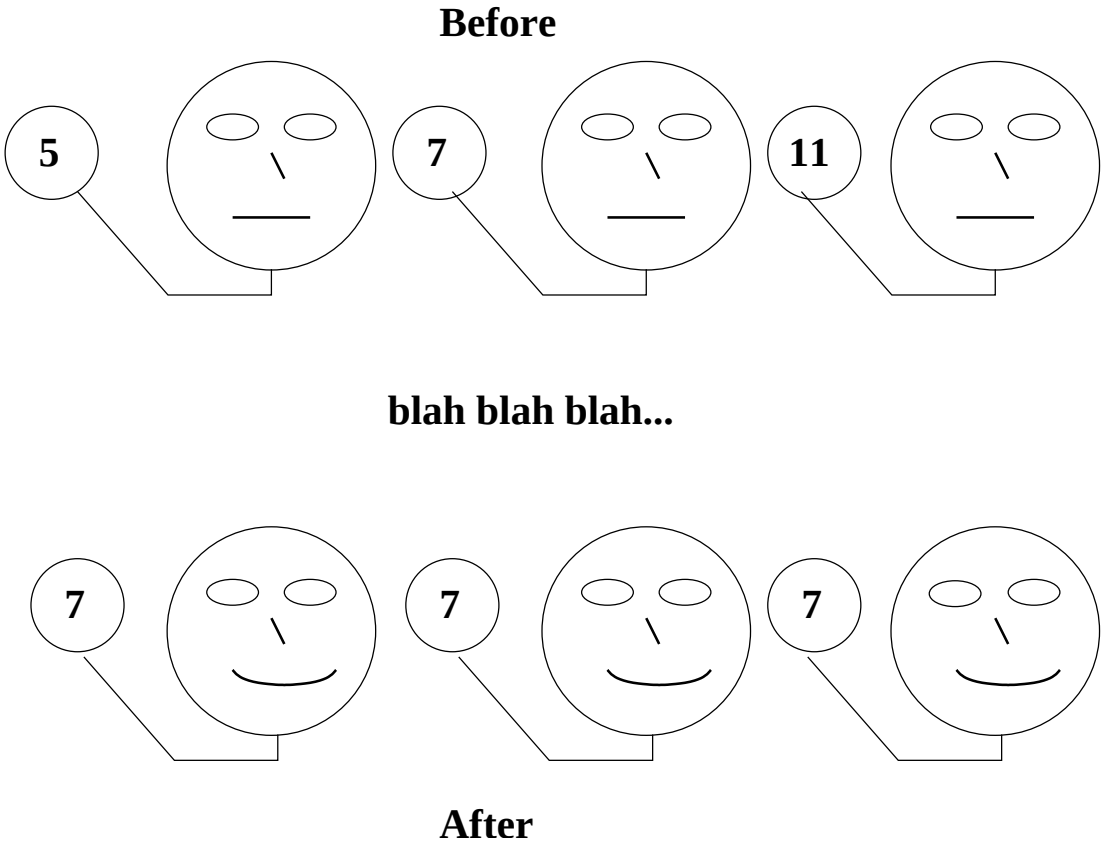


Théorème

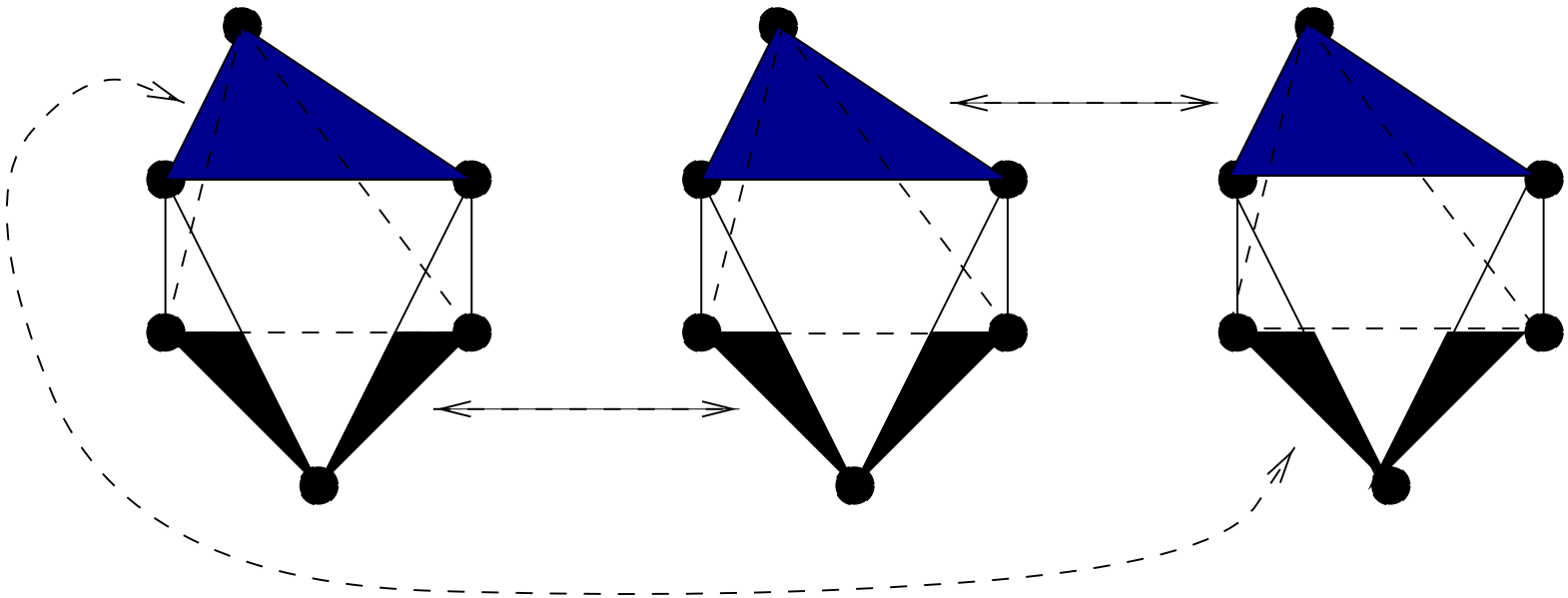
- Les protocoles sans-attente asynchrones sont:
  - $(n-1)$-connexes (pas de trou dans aucune dimension)
  - quel que soit le nombre d'étapes
- Application: "k-set agreement" (accord sur k valeurs)

Accord sur k valeurs

Généralisation du consensus; les processus doivent terminer avec au plus k valeurs différentes (prises parmi les valeurs initiales):



Complexe de sortie ($n = 3$, $k = 2$)



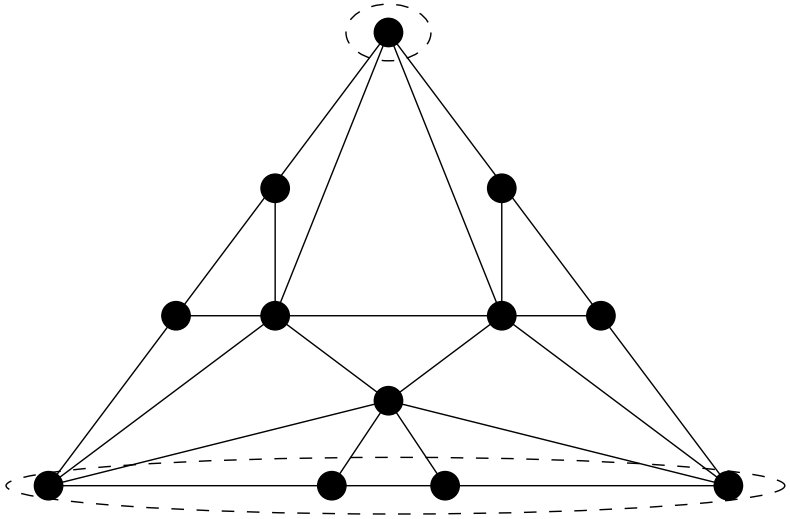
3 sphères collées ensemble moins le simplexe formé des 3 différentes valeurs: pas simplement connexe.

Ebauche de preuve

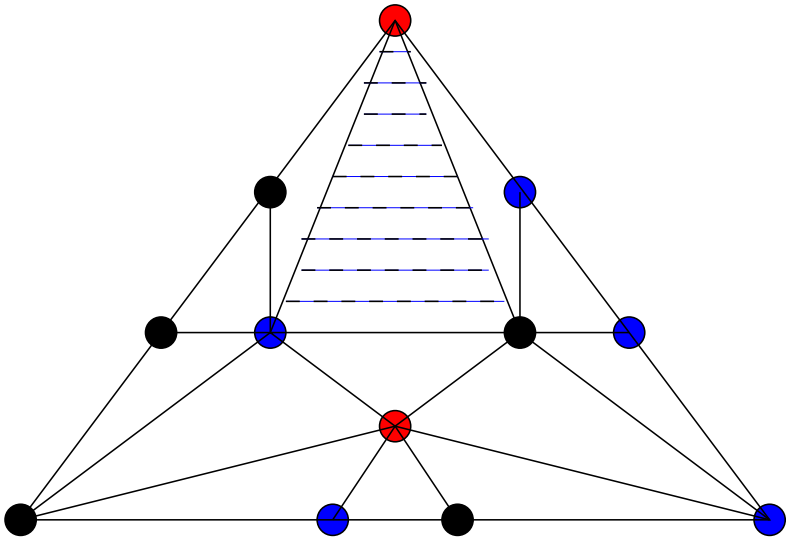
On va utiliser un lemme très ancien de topologie algébrique combinatoire (lemme de Sperner):

- Subdiviser un simplexe
- Donner à chaque “coin” une “couleur” distincte
- Donner à chaque noeud sur le bord une des couleurs données aux coins
- Donner à chaque noeud intérieur n’importe quelle couleur

Lemme de Sperner



P0 and P1 run solo



$\Rightarrow$ Il existe au moins un simplexe qui a toutes les couleurs.

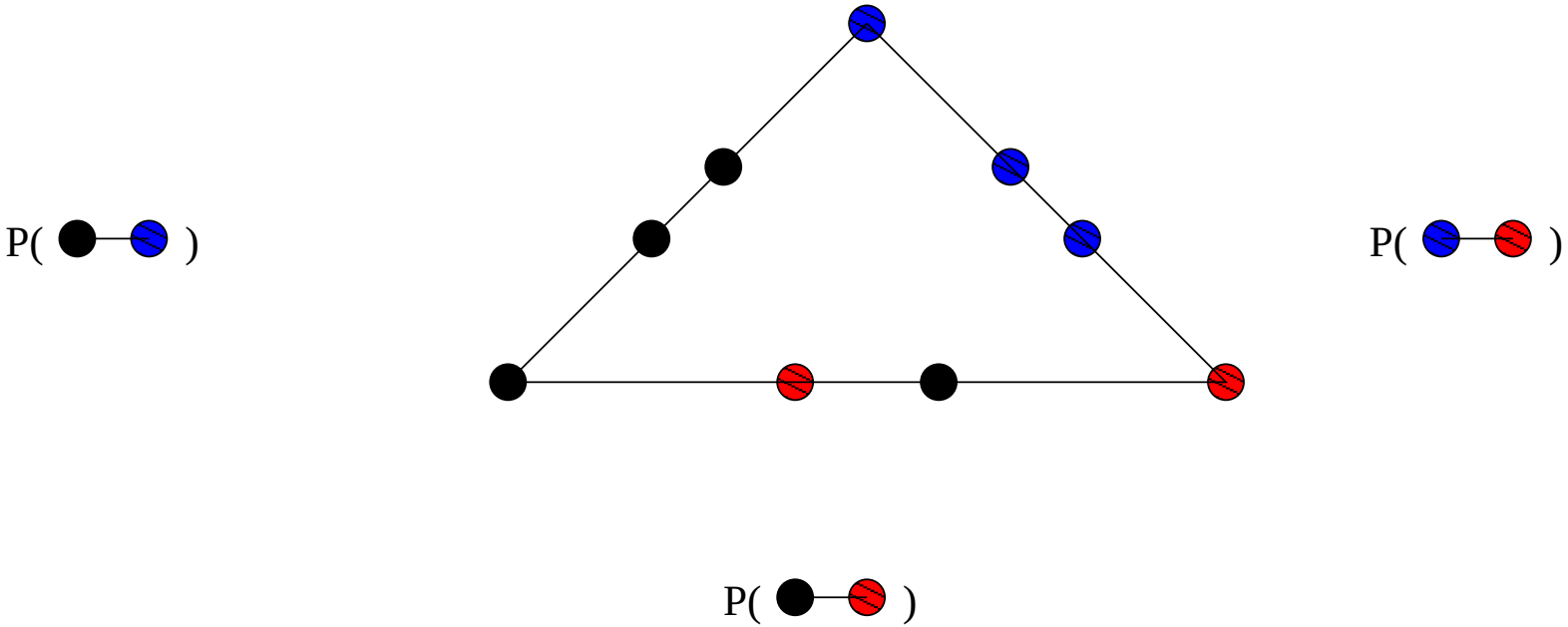
Complexes d'entrée et de sortie

- Chaque processus est coloré avec une entrée différente
- Chaque point est coloré par la valeur de décision

Complexe de protocole

- Pour une exécution avec un seul processus: même point et même couleur (ne peut décider quoi que ce soit d'autre)
- Pour une exécution avec 2 processus:
  - le complexe de protocole est connexe
  - tous les noeuds sont d'une des deux couleurs

Complexe de protocole - pour toutes les exécutions à 2 processus



Complexe de protocole à information complète

- Parce que le complexe est simplement connexe
- On peut “remplir” les chemins combinatoires
- Les noeuds sont colorés avec les couleurs d’entrée

Fin de la preuve

On applique le lemme de Sperner:

- Un des simplexes a les trois couleurs
- Ce simplexe correspondrait à une exécution du protocole qui déciderait des 3 valeurs!

Réciproque?

- En fait, on peut prouver bien plus:
- Une tâche a un protocole asynchrone sans attente si et seulement si il existe une fonction simpliciale μ:
  - du complexe d'entrée subdivisé
  - vers le complexe de sortie
  - qui respecte Δ

Principe de la preuve

$\Rightarrow$

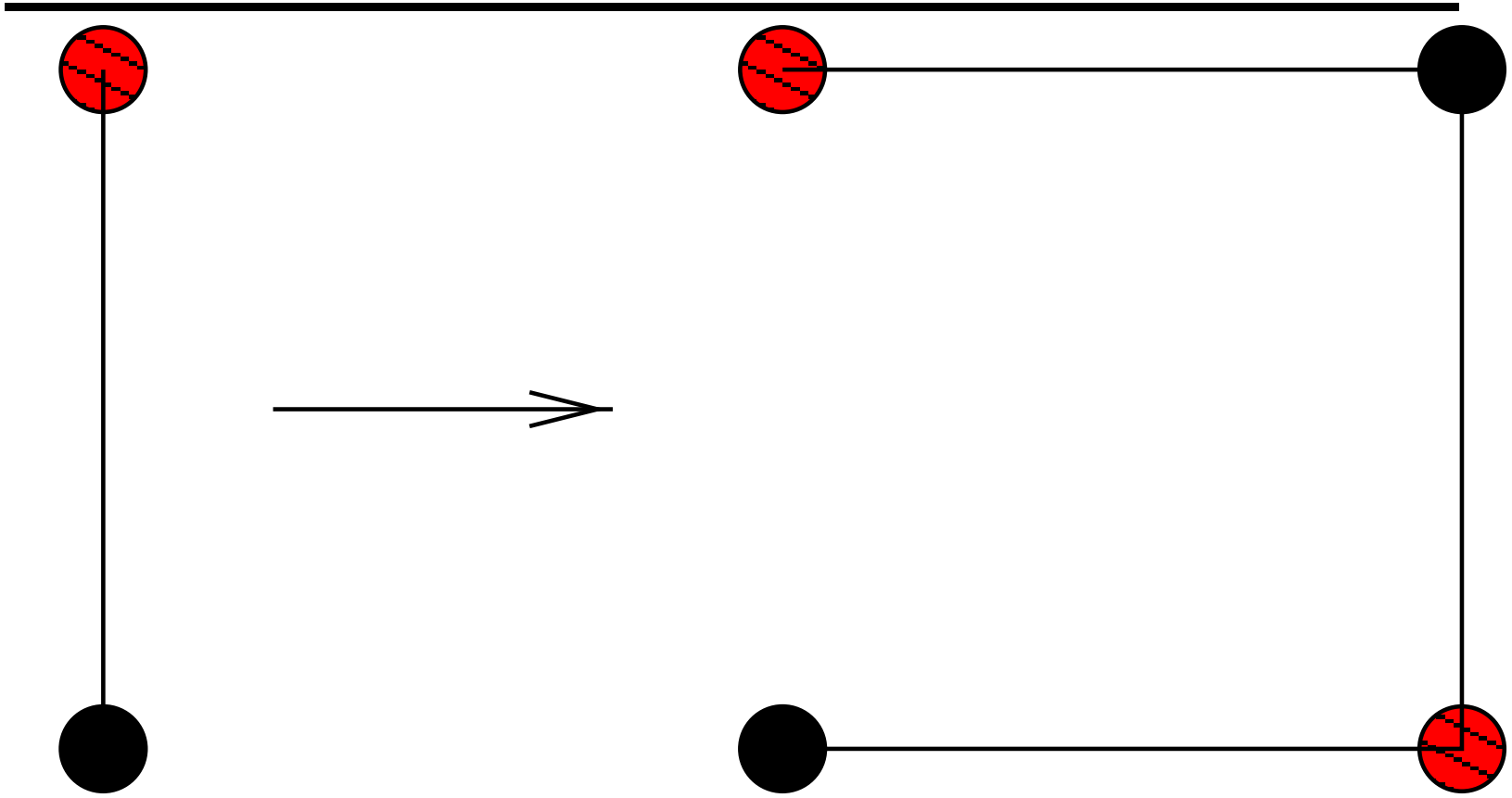
- Le complexe de protocole est $(n-1)$-connexe (en utilisant des calculs homologiques, et en particulier la suite exacte longue de Mayer-Vietoris)
- On exploite la connectivité:
  - on inclut le complexe d'entrée suffisamment subdivisé dans le complexe de protocole
  - on mappe le complexe de protocole vers le complexe de sortie
  - exactement comme la preuve pour l'accord sur k valeurs

Principe de la preuve

$\Leftarrow$

- On peut réduire toute tâche à "l'accord sur un simplexe" [en utilisant l'algorithme du "participating set" de Borowsky et Gafni 1993]
- On démarre aux coins du simplexe subdivisé
- On doit terminer sur les noeuds d'un seul simplexe dans la subdivision

Exemple



Subdivision d'un segment en 3.

Protocole

$$
\begin{array}{lll}
P & = & update; \\
 & & scan; \\
 & & case\ (u, v)\ of \\
 & & \quad (x, y') :\ u = x'; update; [] \\
 & & \quad default :\ update
\end{array}
\qquad
\begin{array}{lll}
P' & = & update; \\
 & & scan; \\
 & & case\ (u, v)\ of \\
 & & \quad (x, y') :\ v = y; update; [] \\
 & & \quad default :\ update
\end{array}
$$

Preuve

On a les trois traces essentielles suivantes:

(i) $scan$ de P avant $update$ de P': P ne connait pas y donc il choisit d'écrire x. $Prog$ finit avec $((P, x), (P', y))$.

(ii) Cas symétrique: $Prog$ finit avec $((P, x'), (P', y'))$.

(iii) L'opération $scan$ de P est après l'$update$ de P' et le $scan$ de P' est après l'$update$ de P. $Prog$ finit avec $((P, x'), (P', y))$.

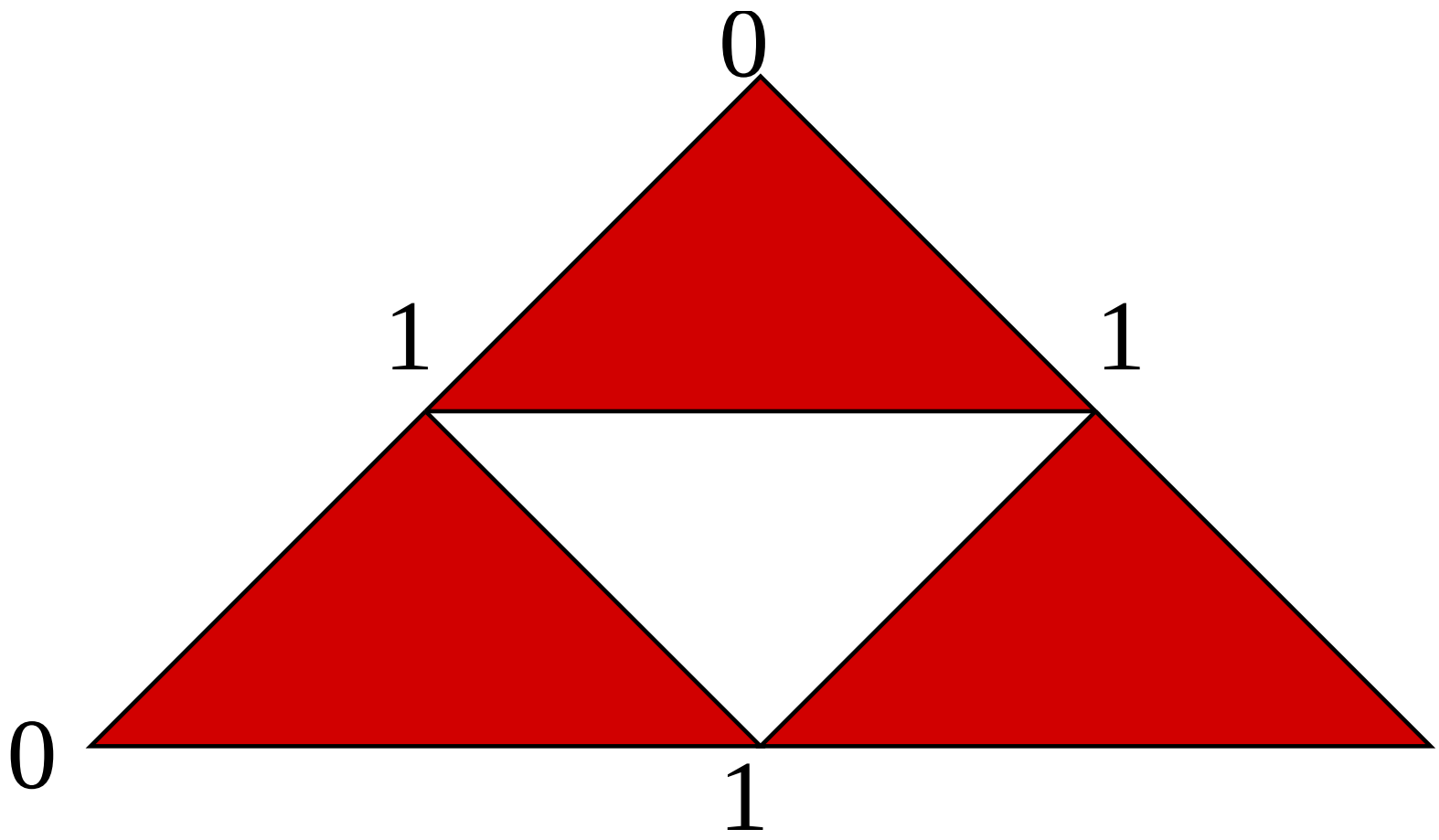
Autres primitives de communication

Les vrais systèmes multiprocesseurs modernes offrent d'autres primitives de synchronisation:

- test&set
- fetch&add
- compare&swap
- files...

Cela donne d'autres complexes de protocoles et d'autres résultats.

Exemple: complexe de protocole test&set



Test&Set

- Les protocoles Test&Set sans attente sont:
  - $(n-3)$-connexes
  - plus puissants que read/write (consensus entre 2 processus faisable)
  - mais toujours pas de consensus entre 3 processus
- Un résultat similaire tient pour d'autres primitives

Références et principaux résultats

- Démarre par un article de Fisher-Lynch-Patterson (“FLP”) en 1985: il existe une simple tâche qui ne peut pas être résolue dans un système distribué fonctionnant par passage de messages, avec au plus un crash potentiel.
- A créé un secteur de recherche très actif, voir par exemple le livre “Distributed Algorithms” (N. Lynch, 1996)

Références et principaux résultats

- suite par Biran-Moran-Zaks dans PoDC'88: caractérisation des tâches qui peuvent être résolues dans un système par passage de messages simple, en présence d'une panne
- l'argument est basé sur la "similarity chain", qui est une version unidimensionnelle de ce que l'on vient de voir.
- S'est révélé être difficile à généraliser à des modèles avec plus de pannes

Références et principaux résultats

A PoDC'1993, indépendamment:

- Borowsky-Gafni, Saks-Zaharoglou et Herlihy-Shavit ont trouvé une borne inférieure pour l'accord sur k valeurs (problème posé par Chaudhuri en 1990)

  [au moins $\lfloor \frac{f}{k} \rfloor + 1$ étapes dans un modèle synchrone]

- Saks-Zaharoglou et Herlihy-Shavit ont exploité des propriétés géométriques pour trouver cette borne

Références et principaux résultats

- Renommage: Attiya-BarNoy-Dolev-Peleg JACM 1990,
- La tâche de $(n+1, K)$-renommage démarre avec $n+1$ processus auxquels on a donné un nom initial unique dans $0, \ldots, N$ et on veut qu'ils choisissent un nom de sortie unique dans $0, \ldots, K$ avec $n \leq K < N$ (indépendamment d'un "process id" - i.e. il s'agit d'un "renommage anonyme"!).
- Ont montré que dans le modèle passage de messages, il y a une solution sans attente pour $K \geq 2n+1$, et aucune pour $K \leq n+2$
- En utilisant des invariants topologiques: il a été montré qu'il n'y a pas non plus de renommage possible quand $K \leq 2n$
- Herlihy et Shavit STOC'93: on a le même résultat pour le modèle asynchrone sans attente (en utilisant l'homologie explicitement).

Références et principaux résultat

Plus tard...

- Caractérisation complète des tâches asynchrones avec read/write atomiques dans “The topological structure of asynchronous computability”, M. Herlihy et N. Shavit, J. of the ACM, jan. 2000
- Utilisation de span algébriques dans “Algebraic Spans”, M. Herlihy et S. Rajsbaum comme méthode unificatrice pour le renommage, l’accord sur k valeurs etc.
- Utilisation de pseudo-sphères...

Références et principaux résultats

- Nombre de consensus (M. Herlihy puis E. Ruppert SIAM J. Comput. vol 30, No 4, 2000 par exemple). Remarque: (M. Herlihy): un objet qui résout le problème du consensus pour n processus peut simuler (sans attente et en utilisant aussi des read/write atomiques) tout objet pour n ou moins de processus.
- Exemple: read/write atomiques ont un nombre de consensus égal à un test&set, queues, stacks, fetch and add ont un numéro de consensus de 2 etc.
- Exemple: il n'y a aucun protocole de $(n+1, 2j)$-renommage sans attente si les processus partagent une mémoire avec read/write atomique et des objets de nombre de consensus $(n+1, j)$.

Quelques perspectives

- Modèles sémantiques du parallélisme
- Autres langages ou paradigmes:
  - Linda, Facile, CHOCS, Erlang, CML, OCCAM, PVM/MPI etc.
  - algèbres de processus: CCS, CSP, π-calcul, Join-calcul etc.
- Analyse statique par interprétation abstraite
- Algorithmes distribués... (voir N. Lynch "Distributed Algorithms")
- Systèmes distribués tolérants aux pannes - Systèmes auto-stabilisants
- Projets GRID

Modèles sémantiques du parallélisme

[MPRI]

- Systèmes de transitions
- Structures d'événements
- Réseaux de Petri
- etc.

Motivations

- Associer à un programme (ici parallèle) un modèle mathématique
- Manipulable, et dont on puisse tirer (par algorithmes) des propriétés d'intérêt
- Exemple: points morts, états atteignables, chemins d'exécutions, valeurs que peuvent prendre les variables etc.

Autres langages

[MPRI]

- PVM/MPI: voir par exemple:
  http://www-unix.mcs.anl.gov/mpi/
- Algèbres de processus: exemple CCS, voir prochain cours
- Join calcul: voir par exemple: http://join.inria.fr/
- Paradigme Linda: “tuple space” et récupération/accès par pattern matching (voir aussi JavaSpaces)
- cf. également www.cs.yale.edu/Linda/linda.html

Analyse statique

[MPRI]

- Vous avez vu un premier exemple avec la logique de Hoare en IF
- Voir maintenant http://www.di.ens.fr/~cousot
- Par exemple, à partir d'une sémantique par systèmes de transitions, "abstraire" les valeurs que peuvent prendre les variables par des intervalles d'entiers (abstraction ici de la logique de Hoare)
- Sujet très actif aussi bien dans la recherche qu'industriellement (cf. ENS/X, CEA/LSL, Polyspace Technologies par exemple)

Systèmes auto-stabilisants

- Cas particulier de protocoles “tolérants aux pannes”
- Communauté importante, longtemps après le premier article (1974) sur le sujet, de E. W. Dijkstra
- Chaque processus doit arriver en un temps fini à “réparer” une mauvaise valeur fournie à un moment donné par un processus
- Voir par exemple:
  http://www.cs.uiowa.edu/ftp/selfstab/bibliography/

Projets GRID

- Un sujet à la mode!
- Systèmes distribués “coopératifs” à l’échelle de la planète... (cluster de stations sur internet par exemple)
- Au menu: algorithmique hétérogène, tolérance aux pannes etc.
- Voir par exemple http://www.gridcomputing.com/