Chapitre 1 — Gestion de pannes

1 Types de pannes

1.1 Panne permanente

Une panne sans résolution.

Un processus est **correct** en termes de panne permanente s'il ne **tombera jamais** en panne permanente.

1.2 Panne récupérable

Une panne avec résolution, moyennant parfois une perte d'informations. Un processus qui en revient en sera conscient.

Un processus est **correct** en termes de panne récupérable quand il **existe un instant** T après lequel il **ne tombera plus en panne**.

1.3 Panne arbitraire

Une panne suite à laquelle tout est possible. Englobe les pannes récupérables ou permanentes.

Un processus est **correct** en termes de panne arbitraire quand il suivra **toujours l'algorithme attendu**.

2 Détecteur de pannes parfait théorique

2.1 Prérequis

- **Complétude** : Un jour, tout processus en panne sera détecté par tout processus correct.
- **Précision** : Si un processus p est détecté par un quelconque processus, alors p est en panne.

2.2 Suppositions

- Pas de perte/duplication/réordonancement.
- Toute panne est permanente
- Il existe une borne supérieure constante T sur la durée de transit de tout message. On parle de système distribué « synchrone ».
- Les durées de traitement sont négligeables.

2.3 Algorithme *heartbeat*

- Périodiquement, le surveillant envoie un ping, et lance un timer de $2T$.
- À la fin du timer :
  - Si le pong a été reçu : un nouveau ping est envoyé, un nouveau timer est lancé
  - Sinon : Une panne est signalée
- Le surveillé répond à ping par pong.

Un système synchrone est irréaliste. Les vrais réseaux ne nous donnent pas de garantie sur la durée de transit d'un message.

3 Détecteur de pannes parfait un jour

3.1 Prérequis

- **Complétude** : Un jour, tout processus en panne sera détecté par tout processus correct.
- **Précision un jour** : Un jour, aucun processus correct ne sera suspecté par un processus correct.

3.2 Suppositions

- Pas de perte/duplication/réordonancement.
- Toute panne est permanente ou récupérable
- Il existe une borne supérieure constante T **inconnue** sur la durée de transit de tout message. On parle de système distribué « partiellement asynchrone ».
- Les durées de traitement sont négligeables.

3.3 Algorithme *heartbeat à timeout dynamique*

- Périodiquement, le surveillant envoie un ping, et lance un timer de Δ.
- À la fin du timer :
  - Si le pong n'a pas été reçu : Δ est incrémenté, une panne est suspectée
  - Envoie un ping et lance un timer de durée Δ
  - Si un pong est reçu d'un suspecté : la suspicion est annulée
- Le surveillé répond à ping par pong.

Le timeout dynamique est nécessaire pour ne pas suspecter constamment un système lent mais correct. La précision un jour est correcte, car grâce au timeout dynamique il y aura un jour un timout Δ pour lequel aucune panne ne sera suspectée.

Chapitre 2 — Timestamps

1 Horloges logiques

1.1 Prérequis

On veut connaitre l'**ordre** des *évènements*.

Un évènement est une action, l'émission ou la réception d'un message.

L'ordre $E_1 \to E_2$ si E_1 arrive avant E_2 sur la même machine ou E_1 est l'émission et E_2 sa réception.

1.2 Propriétés

- **Transitivité** : $E_1 \to E_2$, $E_2 \to E_3 \Rightarrow E_1 \to E_3$
- **Anti-réflexif** : $E_1 \not\to E_1$
- **Anti-symétrique** : $E_1 \to E_2 \not\Rightarrow E_1 \leftarrow E_2$
- **Ordre partiel** : il existe deux évènements E_1 et E_2 tels que ni $E_1 \to E_2$, ni $E_2 \to E_1$.

1.3 Horloge logique de Lamport

On veut une fonction H telle que si $E_1 \to E_2$ alors $H(E_1) < H(E_2)$.

Attention, $H(E_1) < H(E_2) \not\Rightarrow E_1 \to E_2$. On projette un ordre partiel sur un ordre total. H donne une relation à deux évènements qui n'en ont pas.

Pour contourner le cas où $H(E_1) = H(E_2)$, on ajoute qu'entre deux machines i et j telles que $i < j$, alors si E_1 arrive sur la machine i et E_2 sur la machine j : $E_1 \to E_2$.

1.3.1 Algorithme

- Chaque site maintient un numéro.
- À chaque événement *local*, le timestamp est incrémenté.
- Le timestamp est envoyé avec chaque nouveau message.
- À la réception, le timestamp le plus grand entre local et reçu est utilisé et *incrémenté*.

1.3.2 Conditions nécessaires

Pour tout deux évnements différents E_1 et E_2 dans le même processus et $H(x)$ étant le timestamp pour un évènement x, il est nécessaire que $C(E_1) \neq C(E_2)$.

Il faut donc au moins que :

- L'horloge logique soit configurée pour qu'il y ait au moins un incrément entre E_1 et E_2
- Dans un environnement multi-process, il peut être nécessaire d'y attacher un *pid* pour différentier les évènements E_1 et E_2 qui peuvent simultanément arriver sur deux processus.

2 Mutex réparti

2.1 Propriétés

Correctness : Jamais plus d'un processus sera en SC à un instant T.

Progrès : Toute demande d'entrée en SC sera autorisée un jour.

2.2 Version pile

On request :
- Push demandeur dans la file avec son timestamp
- Réordonne la file par heure de Lamport
- Envoie *ack* avec nouveau timestamp

On release :
- Pop tête de file
- Entrer en SC si je suis la nouvelle tête, et que j'ai reçu tous les *acks*.

La tête de la file est celle actuellement en SC, si elle a reçu un ack de toutes les autres machines.

Le timestamp est nécessaire pour éviter que deux processus différents puissent se retrouver en tête de file entre deux machines, si deux messages envoyés par les deux machines sont reçus en même temps. Cela ne garantit pas l'ordre strict immédiat.

Les *acks* sont nécessaires pour assurer l'ordre strict et partagé entre les deux machines.

2.3 Version tableau

File remplacée par un tableau de taille N avec initialement $\{\text{REL}, 1\}$, puis le dernier message reçu avec le timestamp **sauf si ACK remplacerait REQ**.

Un processus entre en SC si l'heure logique est strictement la plus petite.

Le timestamp est incrémenté en entrée et sortie de SC, et à toute réception de la part d'un autre processus..

On request :
- Stocke le message dans tableau
- Envoie *ACK* avec nouveau timestamp (amélioration possible : ne pas envoyer le *ACK* si le tableau contient *REQ* pour self)

On release : Stocke message dans tableau.

On ack : Stocke message dans tableau, sauf si un *REQ* y est déjà.

Après chaque message : Entre en SC si
- j'ai la plus petite heure logique
- c'est un *REQ*

3 Mutex par jeton

3.1 Algorithme de Ricart et Agrawala

App veut entrer en SC : J'envoie un *REQ* à tout le monde

Réception d'un *REQ* :
- Si je ne suis pas en SC
  - Si je ne veux pas entrer en SC : je réponds avec *OK*
  - Si je veux entrer en SC : je réponds avec *OK* **si je n'ai pas la priorité**, sinon j'attends d'avoir fini.
- Si je suis en SC
  - J'attends d'avoir fini, puis je réponds avec OK

App sort de SC : Je réponds par OK à tous les REQ en attente.

Réception d'un OK : Je le comptabilise et si j'ai le *OK* de tout le monde alors je peux entrer en SC.

3.1.1 Algorithme de Carvalho et Roucairol

App veut entrer en SC : J'envoie un *REQ* à ceux qui ont le jeton

Réception d'un *REQ* :
- Si je suis en SC
  - J'attends d'avoir fini, puis je passe le jeton
- Si je ne suis pas en SC
  - Si je veux entrer en SC :
    - Si j'ai la priorité : j'attends d'avoir fini puis je passe le jeton
    - Sinon : je passe le jeton et renvoie mon *REQ*
  - Sinon : je passe le jeton

App sort de SC : Je passe mon jeton à tous les demandeurs.

Réception d'un OK : Je note avoir reçu le jeton.

Si j'ai tous les jetons : Je peux entrer en SC **quand je veux**

Chapitre 3 — Jetons

1 Concepts généralisés

1.1 Horloge logique

Heure virtuelle donnée à tout évènement, tel qu'un ordre strict existe.

1.2 Mutex

Un seul processus à la fois ne peut être dans l'état SC.

1.3 Lamport

- Si ma requête est **le plus ancien message** que j'aie reçu, je peux entrer en SC
- Je suis responsable de maintenir une idée de l'**état des autres**.

1.4 Carvalho et Roucairol

- Si **j'ai les jeton de tout le monde**, je peux entrer en SC
- Je ne suis responsable que de **donner mon jeton** si je n'en ai pas besoin

1.5 Ricart et Agrawala

- Si **tout le monde est OK** que je le fasse, je peux entrer en SC
- Je ne suis responsable que de dire quand **c'est bon pour moi** qu'un autre soit en SC

1.6 Raymond

- Si je **possède le jeton**, je peux entrer en SC
- Je dois maintenir **mon parent** et les nœuds qui **me voient comme parent**
- Je suis responsable de router les requêtes **vers l'owner**

2 Mutex par jeton unique

Algorithme de Raymond.

2.1 Propriétés

On cherche un protocole de **transmission** assurant *unicité* et *progrès*.

2.1.1 Unicité (correctness)

Assurer que le jeton ne sera pas dupliqué ni perdu

2.1.2 Transmission

Gérer le transit du jeton d'un demandeur à un autre

2.1.3 Progrès

Assurer que tout demandeur aura le jeton **un jour**

2.2 Fonctionnement

2.2.1 Système de requêtes

- `REQ` : donne-moi le jeton
- `OK` : Tiens

2.2.2 Trouver où se trouve le jeton

Les liens de l'arbre entre les nœuds sont **dirigés** vers le voisin le plus proche du jeton.

Les requêtes sont transmises via ces liens.

2.2.3 Se souvenir d'où vient une requête

Une file FIFO est utilisée pour se rappeler des demandes entrantes.

2.3 Algorithme

2.3.1 App veut entrer en SC

Envoi d'un REQ à moi-même

2.3.2 Réception d'un REQ

Push l'envoyeur dans ma file (**le voisin requêteur**)

Si je n'ai pas le jeton
- Si je n'ai pas de REQ en attente : j'envoie un `REQ` à mon parent

Si j'ai le jeton
- Décider de quoi faire du jeton

2.3.3 App sort de SC / Réception d'un OK

Décider de quoi faire du jeton

2.3.4 Décider quoi faire du jeton

Si ma file est vide, rien faire

Sinon, pop p la tête de file
- Si p = self, j'entre en SC
- Sinon, je passe le jeton à p, mon parent devient p et si ma file d'attente contient q alors j'envoie un `REQ` à p

2.4 Effets de bord

2.4.1 Messages OK

La condition nécessaire pour envoyer un OK est :

$$\text{OWNER} = \text{self} \wedge \neg\text{inSC} \wedge Q \neq \varnothing \wedge \text{head}(Q) \neq \text{self}$$

La dernière condition ne survient que dans la situation où un OK vient tout juste d'être reçu, dans ce cas si *self* est en tête de queue le processus peut d'abord entrer en section critique avant de donner le jeton au suivant, sinon elle n'entrerait jamais en SC.

2.4.2 Messages REQ

La condition nécessaire pour pouvoir envoyer un REQ au owner est que *self* le veut le token (pour elle-même ou pour un de ses enfants). Il faut aussi éviter d'envoyer plusieurs fois un REQ au parent.

$$\text{OWNER} = \text{self} \wedge Q \neq \varnothing \wedge \neg\text{sentREQ}$$

sentREQ permet d'éviter que *self* soit présent plusieurs fois dans la queue du parent, ce qui assure que la quantité de nœuds présents dans la file d'un processus ne dépasse jamais le nombre d'enfants de ce processus. On parlera alors qu'il y aura toujours une chaine bien maintenue de nœuds entre un processus qui souhaite entrer en SC et celui qui possède le jeton et la gestion de l'algorithme prouve qu'il n'y a pas de possibilité qu'un nœud soit oublié suite à sa demande.

2.4.3 Croisement de messages

Si des messages se croisent, ce n'est pas un problème. Si une demande de X arrive à Y avant le OK de X, alors sa demande est mise en file d'attente dans Y. Comme Y aura d'abord envoyé un REQ à X pour avoir le jeton, cela implique que Y ne peut pas envoyer d'autres REQ à son parent, X. Quand le message OK arrivera à Y, Y pourra entrer en SC ou passer le jeton plus loin (sachant que dans ce cas X ne sera forcément pas en tête).

2.4.4 Variation « greedy »

Il est possible pour un système qui présente fréquemment une charge élevée de configurer un nœud pour qu'il se place en tête de file au lieu de la queue s'il souhaite entrer en section critique. Cela a l'avantage que si le jeton passe par lui-même dans un cas où un enfant l'a demandé avant, alors sa section critique sera exécutée immédiatement avant de passer le jeton plus loin.

En termes de propriétés, cette variation amène une pénalité de progrès où cela pourrait prendre plus de temps pour qu'une demande d'entrée en SC soit autorisée, mais elle le sera quand même *un jour*.

2.5 Gestion de pannes

En cas de panne d'un nœud qui cause la perte des informations de l'algorithme, X, après un délai suffisamment long pour s'assurer que tous les messages précédemment envoyés soient reçus, pourra envoyer un message RESTART à ses voisins.

En cas de réception d'un REQ de Y, X ajoute Y dans sa file. Si X reçoit un OK, il devient propriétaire du jeton. Si X veut entrer en SC, alors X s'ajoute en fin de file.

Sur la base des informations de chaque voisin, les voisins de X peuvent envoyer un message en réponse avec les informations qui peuvent être déduites.

En cas de pannes durant le processus de récupération, il se peut que des messages de réponses de l'ancienne panne soit mélangés. Dans ce cas il est possible d'identifier les messages de récupération avec un identifiant unique qui permettra de les séparer.

En cas de pannes concurrentes entre deux nœuds, il est possible de récupérer l'état seulement s'ils n'étaient pas adjacents.

Chapitre 4 — Diffusion

1 Élections

Étant donné un groupe de processus :

— Chacun a une **aptitude**, dynamique.
— On veut **élire** celui qui a la meilleure aptitude.

Utilisable pour regénérer un jeton d'un mutex après une perte, load balancing ou manager/workers.

1.1 Propriétés

1.1.1 Sureté

Un seul processus doit être élu à la fois

1.1.2 Progrès

Il doit y avoir un élu un jour

1.1.3 Validité

L'élu doit être le participant correct à la plus grande aptitude

1.1.4 Résilience

— Un processus en panne ne doit pas être élu, même s'il redémarre
— En cas de partition réseau, un élu unique doit exister par partition

2 Algorithme du Bully

Partant du principe que le système est **synchrone**, pour tout message M il existe un délai d'envoi de T maximum et qu'il n'y a pas de panne serveur.

2.1 Fonctionnement

Algorithme naïf.

2.1.1 Une élection commence

1. Je broadcast mon aptitude
2. J'attends $2T$ pour tout recevoir
3. Je détermine l'élu

2.1.2 Réception d'une aptitude

1. Si je ne suis pas déjà en élection : j'entre en élection

2.2 Gestion de pannes

Si tous les messages sont envoyés de manière atomique, alors le danger est que si un processus va gagner une élection et qu'il tombe en panne le système complet sera sans élu (résilience cassée).

Un détecteur de panne *un jour* suffit pour redémarrer une élection si une panne est suspectée.

3 Algorithme de Chang et Roberts

Partant du principe que le système est **asynchrone**, pour tout message M le délai d'envoi est non borné.

3.1 Fonctionnement

3.1.1 Demande d'élection

Si je ne suis pas en cours d'élection
— J'entre en élection
— J'envoie mon aptitude et mon ID
Si je suis en cours d'élection
— Je refuse la demande

3.1.2 Réception d'une aptitude

Si je ne suis pas en cours d'élection
— J'entre en élection
— Je compare l'aptitude reçue avec la mienne
— Je propage la plus grande, avec l'ID associé
Si je suis en cours d'élection
— Si c'est ma demande, j'envoie le résultat
— Sinon, je compare l'aptitude reçue avec la mienne et je propage celle reçue si et seulement si elle est plus grande

3.1.3 Réception d'un résultat

Si ce n'est pas moi
— Je note, je propage, et je sors d'élection
Si c'est moi
— Je suis l'élu et je sors d'élection

3.2 Variations

3.2.1 Demandes concurrentes

Si Y recoit le message de X avant que Y n'émette son message, et que $X > Y$, alors Y ne peut pas être le nœud avec la plus grande aptitude, donc Y n'a pas besoin d'envoyer son propre message et n'aura qu'à propager le message de X.

En conséquence, si Y reçoit une demande de X avec $X <= Y$, le message peut être ignoré si Y est déjà en cours d'élection, car un message d'aptitude pour Y est déjà en train de circuler.

3.3 Gestion de pannes

Dans un système synchrone, où la durée de transit maximale est bornée par une constante T et que les durées de traitement sont négligeables. En partant du principe que les pannes peuvent être récupérables.

On intègre un système de gestion de l'anneau sous forme de buffer circulaire avec le protocole suivant :

3.3.1 Fonctionnement de l'anneau

3.3.1.1 Demande d'envoie au suivant

— Incrémenter t
— Envoyer message et t au n-ième suivant dans l'anneau
— Lancer un timer, dont le timeout demandera l'envoi du mesasge au $(n+1)$-ième suivant.

3.3.1.2 Réception de m et t_i

— Notifier de la réception
— Répondre ACK avec t_i

3.3.1.3 Réception de ACK avec t_i

Annuler le timeout associé à t_i

3.3.2 Propriétés de l'anneau

Complet et correct. Si le système est asynchrone, ce n'est plus correct car il se peut qu'un nœud soit continuellement ignoré alors qu'il est juste plus lent.

3.3.3 Fonctionnement de l'électeur

La phase d'annonce peut ne jamais se terminer si le nœud qui doit être élu est celui qui tombe en panne. Il faut alors ajouter dans les messages d'annonce les aptitudes pour chaque nœud parcouru, et dans les messages de réponses l'id du nœud élu et tous ceux qui l'ont accepté.

3.3.4 Fonctionnement de l'électeur

3.3.4.1 Demande d'élection

Si je ne suis pas en cours d'élection
— J'entre en élection
— J'envoie une annonce $[(self, apt)]$
Si je suis en cours d'élection
— Je refuse la demande

3.3.4.2 Réception d'une annonce

Si je suis dans la liste de l'annonce
— Je calcule l'élu d'après la liste
— J'envoie un résultat $(lu, [self])$
— Je sors d'élection
Sinon
— J'ajoute $(self, apt)$ dans la liste
— J'envoie la liste au suivant
— J'entre en élection

3.3.4.3 Réception d'un résultat

Si je suis dans la liste : j'ignore (c'est mon résultat et tout le monde l'a vu)
Sinon, si je ne suis pas en élection et l'élu reçu est différent du mien
— Démarrer une nouvelle élection (recommencer, il y a un désaccord)
Sinon, si je suis en élection ou que l'élu reçu est le mien
— Je note le nouvel élu
— Je m'ajoute à la liste
— J'envoie un résultat avec l'élu et cette liste
— Je sors d'élection

4 Sondes et Échos

Processus en deux phases :

1. La **diffusion** propage l'information avec des **sondes**
2. La **contraction** renvoie une information à la source avec des **echos** ***agrégés***

Une **invariante**

$$\#\text{sondes envoyées} = \#\text{échos reçus}$$

Utilisable dans l'agrégation d'informations, le partage d'information ou la synchronisation de processus.

4.1 Fonctionnement en arbre

Nous sommes dans un réseau acyclique.

4.1.1 Envoi

Source envoie une sonde à ses enfants

4.1.2 Réception d'une sonde

Je la propage à mes enfants
Si je n'ai pas d'enfants : je réponds avec un écho

4.1.3 Réception d'un echo

— Si j'ai tous les echos, je répond avec un echo à mon parent
— Si j'ai tous les echos et que je suis la source, j'ai fini.
— Sinon, j'attends

4.2 Fonctionnement en graphe

Dans un graphe contenant potentiellement des cycles on ajoutera deux vérifications :

1. Réception d'une sonde : si je connais déjà la sonde, j'arrête d'attendre l'écho de ce voisin
2. Réception d'un echo : je réponds quand j'ai tous les echos *attendus*

4.3 Fonctionnement avec plusieurs sources

Chaque sonde doit être marquée d'un identifiant unique de façon globale au système et cela permettra de les traiter indépendemment.

4.3.1 Protocole de GUID – Snowflake

Construction d'un ID selon le numéro de processus et un ID localement unique.

4.4 Cas d'utilisation

4.4.1 Topologie d'un réseau en arbre

Lors de la propagation d'une sonde, on ne fait rien. Par contre, au moment où on retourne un echo, on fait l'union des dictionnaires reçus de la part des enfants et on s'y ajoute soi-même et on envoie ça au parent.

4.4.2 Arbre de recouvrement d'un réseau cyclique

On envoie les sondes. Ensuite, pour chaque echo qui est attendu, on le note comme étant dans l'arbre de recouvrement avant d'envoyer un echo sans information au parent.

Chapitre 5 — Battements

1 Synchronisation

Un **algorithme synchrone** est un algorithme où tous les systèmes avancent ensemble.
Ne pas confondre avec un **système synchrone** qui parle d'une borne T pour la durée de transit de messages.

1.1 Requirements

Un **signal** est reçu pour déclencher un nouveau **battement** (pulse).
Dans chaque battement, un processus peut faire des calculs, envoyer/recevoir **jusqu'à un** message *par voisin*, le tout **en fonction des battements précédents**.
Un processus est **prêt** quand tous ses messages ont été reçus.

1.1.1 Remarques

On dira donc que tout événement d'un battement k ne dépend que des battements $< k$. Qu'il est possible de ne pas envoyer de messages pendant un battement, et qu'il est *impossible* d'envoyer *plusieurs* messages au même voisin durant un même battement.
L'ordre des événements est évident : E a eu lieu avant $F \Leftrightarrow$ Le battement de E est plus ancien que celui de F. C'est un **ordre partiel**, deux événements du même battement n'ont pas d'ordre.

1.1.2 Approches

1.1.2.1 Centralisée

Un **contrôleur** est responsable d'envoyer les signaux à tous les processus au bon moment.

1.1.2.2 Décentralisée

Les processus doivent communiquer pour savoir quand ils peuvent déclencher un signal localement.
Deux problématiques à répondre :
1. Savoir quand je suis prêt
2. Savoir quand je peux lancer le battement suivant

1.2 α-synchronizer

Complexité temporelle faible, beaucoup de messages.

1.2.1 Stratégie

Question 1 : demander un ACK pour chaque message envoyé, dès que tout est acquitté je suis prêt
Question 2 : informer mes voisins quand je suis prêt, quand ils le sont tous je peux lancer le battement suivant

1.2.2 Problèmes éventuels

Si un processus me dit *prêt*, **tous ses messages** suivants appartiendront au prochain battement. Si je ne suis pas encore au battement suivant, je les buffer et les traiterai au moment venu.

1.3 β-synchronizer

Complexité temporelle haute, peu de messages.

1.3.1 Changements

On suppose un arbre intégré sur le réseau. Un **leader** est élu et devient la racine de l'arbre. Chaque nœud **représente** lui et *ses enfants*.

1.3.2 Stratégie

Question 1 : tous mes messages et *ceux de mes enfants* ont été reçus
Question 2 : quand la racine est prête, car elle représente l'arbre entier
Comment lancer le prochain battement ? La racine envoie un **signal** à travers l'arbre.

1.3.3 Remarques

Battement déclenché pas au même instant partout.
L'arbre est construit sur un ensemble de connexions pas nécessairement identique à celui qui serait utilisé par l'algorithme synchrone.
Un processus qui n'a pas de messages à envoyer est prêt dès la fin de ses calculs locaux.
C'est un système centralisé, mais le centre peut changer dynamiquement en reconstruisant l'arbre recouvrant.
Algorithme de sondes et échos sur arbre, signal = sonde, echo = prêt. Attente du traitement pour envoyer echo.

2 Algorithmes par battements

2.1 Découvrir la topologie du réseau

Objectif : trouver qui est connecté à qui.
Initialement, chaque processus connait ses voisins directs, et le nombre total de processus, n.
En fin d'algorithme, chaque processus doit connaitre aussi les voisins de chaque processus du système.

2.1.1 Fonctionnement

On stocke un dictionnaire qui représente pour chaque processus un sous-ensemble de processus initialisé à la liste des voisins immédiats du processus initiant la découverte.
À chaque battement, je mets à jour mon dictionnaire avec ceux reçus au battement précédent, j'envoie ensuite ce dictionnaire à tous mes voisins.
Je m'arrête une fois que mon dictionnaire connaît l'entièreté des voisins, donc il a une taille n. Ne pas oublier d'envoyer une dernière fois le dictionnaire pour que mes voisins aient la dernière version de l'information.

2.2 Multiplication distribuée de matrices (Cannon)

Soit A et B, deux matrices de taille $n \times n$. On veut $C = AB$ et on a n^2 processus.
On arrange les processus en grille dont les extrémités sont connectées en opposé.

2.2.1 Stratégie

Processus en (i, j) calculera $C_{ij} = \sum_{k \in [1,n]} A_{ik} \cdot B_{kj}$.
A chaque battement k (i, j) possède A_{ik} et B_{kj}, il les multiple pour calculer C_{ij} et le passe à ceux qui en ont besoin.
C_{ij} est donc calculé progressivement, en k battements.

2.2.1.1 Valeurs initiales

(i, j) a A_{ik} et B_{kj} tels que $k = i + j \mod n$
On shift A_{ij} de i vers la gauche et B_{ij} de j vers le haut.

2.2.1.2 Valeurs à chaque battement

On passe à $k' = k + 1 \mod n$, simplement on envoie la valeur de A vers la gauche et la valeur de B vers le haut.

2.2.2 Généralisation

En général, le nombre de processus disponibles est inférieur au nombre d'éléments dans la matrice. On peut donc remplacer les éléments de la matrice par des sous-matrices, pour faire en sorte que chaque processeur traite plus de valeurs. La multiplication scalaire et l'addition deviennent donc devenir une multiplication et addition séquentielles. La hauteur et la largeur des sous-matrices sera $N = {}^{n}/_{\sqrt{p}}$.

Chapitre 6 — Consensus

1 Décision partagée

Pour un consensus, chaque processus **propose** une valeur et à la fin chaque processus doit **décider** la même valeur, peu importe laquelle.

1.1 Suppositions

Contraintes classiques : pas de perte, pas de duplication, pas de réordonancement.
Toute panne est **permanente** et le système est (partiellement) synchrone (borne T pas nécessairement connue).
Les durées de traitement sont négligeables.
Le réseau est une clique, tous connectés à tous.
Ainsi, nous utiliserons un détecteur de pannes parfait.

1.2 Propriétés

1.2.1 Terminaison

Tout processus correct **décide** d'une valeur, *un jour*.

1.2.2 Validité

Si un processus décide v, alors un des processus a proposé v.

1.2.3 Unicité

Aucun processus ne décide deux fois.

1.2.4 Accord

Tous deux processus décident la même valeur.

1.3 Déroulement

1.3.1 Initialement

Ma liste ne contient que ma proposition. J'entre en round 1.

1.3.2 Quand j'entre en round x

J'envoie ma liste de propositions avec l'ID de round.

1.3.3 Quand je reçois un message pour le round x

Si je ne suis pas encore en round x, je repousse sa réception.
Sinon, je mets à jour ma liste de propositions.

1.3.4 Quand j'ai reçu un message de tous ceux dont je n'ai pas détecté la panne

Si j'ai reçu, ce round, de la part des mêmes qu'au round précédent, je décide parmi mes propositions (de manière déterministe) et j'envoie ma décision à tous les processus en ligne, puis je m'arrête.
Sinon, je passe au round suivant.

1.3.5 Quand je reçois une décision

Je propage, et je m'arrête.