Chapter 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.

Chapter 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**